Handbook of Scientific Signal Processing

December 2025 edition

An illustrated handbook with a free software stack and spreadsheet templates to download.

A retirement project by

Tom O'Haver
Professor Emeritus
Department of Chemistry and Biochemistry



orcid.org/0000-0001-9045-1603

Online access to the latest versions

Web address: http://bit.ly/1NLOlLR

Interactive Matlab Tools: http://bit.ly/1r7oN7b

Software download links: http://tinyurl.com/cey8rwh

Animated examples: https://terpconnect.umd.edu/~toh/spectrum/ToolsZoo.html

If you are reading this book on an Internet-connected computer or tablet, you can tap, click or Ctrl-Click on any of the page numbers in the text to jump directly to that page. If you download the Microsoft Word 365 version of the book, the GIF animations will run automatically. (In most free PDF viewers, however, you must click on the GIF links to view the animations). You can also click on the https addresses, on the names of downloadable software or on graphics to view, enlarge, or download those items.

Have a question or suggestion? E-mail me at toh@umd.edu

Acknowledgements. Thanks to my graduate students, many of whom worked with the techniques described within, to M. Farooq Wahab for his many contributions and for many fruitful discussions, to *Baldassarre Cesarano* for his close reading and typographical correction of this text, to *Dr. Raphael Attié* of NASA/Goddard Space Flight Center for corrections, to *Diederick* of The University of Hong Kong for code contributions, to *Yuri Kalambet* of Ampersand, Ltd. for many suggestions, corrections, and ideas, and to the many email correspondents who have made suggestions, asked questions, caught errors, sent examples of their data, and who have shown me new areas of application that have broadened the scope of this work.

Table of Contents

Introduction 10
Signal arithmetic 14
Signal arithmetic in Spreadsheets 16
Signal arithmetic and plotting in Matlab 17
Importing data into Matlab/Octave and Python. 20
Matlab Versions. 21
Math in Python 22
GNU Octave 22
Spreadsheet or Matlab/Python? 23
Signals and noise 24
Detection limit 28
Frequency distribution of random noise 27
Dependence on signal amplitude 28
The probability distribution of random noise 32
Representing random noise in Spreadsheets 34
Random functions in Matlab and Python 35
The difference between scripts and functions 36
Live scripts 37
User-defined functions related to signals and noise. 39
Numerical experiments: the role of simulation and modeling. 41
Smoothing 42
Smoothing algorithms 42
Noise reduction 44
Effect of the frequency distribution of noise 44
End effects and the lost points problem 45
Examples of smoothing 45
The problem with smoothing 46
Optimization of smoothing 48

When should you smooth a signal? 49
When should you NOT smooth a signal? 50
Dealing with spikes and outliers. 50
Ensemble Averaging 51
Condensing oversampled signals 52
Smoothing in spreadsheets 53
Smoothing in Matlab and Octave 55
Segmented smoothing 55
Other smoothing functions. 56
Other noise-reduction functions. 57
Live script for smoothing 59
Smoothing performance comparison 61
Differentiation 63
Basic Properties of Derivative Signals 64
Applications of Differentiation 66
Peak detection 68
Derivative Spectroscopy 69
Trace Analysis 70
Derivatives and Noise: The Importance of Smoothing 72
Differentiation in Spreadsheets 75
Differentiation in Matlab and Python 75
Peak Sharpening 79
Even derivative sharpening 79
Constant-area first-derivative symmetrization ("de-tailing") 83
The Power Law Method 85
Peak Sharpening for Excel and Calc Spreadsheets 87
Peak Sharpening for Matlab and Octave 88
Harmonic analysis and the Fourier Transform. 94
Software details 103
Matlab and Octave 103
Time-segmented Fourier power spectrum. 104
Observing Frequency Spectra with iSignal 106
Frequency visualization. 106
Signal enhancement 107
Showing that the Fourier spectrum of a Gaussian is also a Gaussian 108
Fourier Convolution 109

Simple whole-number convolution vectors 110
Software details for convolution 111
Multiple sequential convolution 113
Fourier Deconvolution 114
Applications of deconvolution 114
Fourier Self-deconvolution 117
Computer software for deconvolution 118
Matlab and Octave 118
Noise reduction in deconvoluted signals 121
Excess noise reduction by denominator addition (regularization) 122
Deconvolution for peak area measurements 123
Multiple sequential deconvolution 124
Segmented deconvolution 125
Live script Self-deconvolution Tool 126
Interactive deconvolution with iSignal 127
Fourier Filter 128
Computer software for Fourier Filtering 129
Wavelets and wavelet denoising 133
Visualization and analysis 134
Wavelet denoising 136
Integration and peak area measurement 140
Dealing with overlapping peaks 142
Peak area measurement in spreadsheets 144
Using sharpening for overlapping peak area measurements 145
Peak area measurement using Matlab and Octave 146
Automatic multiple peak detection 148
Area measurement by iterative curve fitting 153
Correction for background/baseline 153
Asymmetrical peaks and peak broadening: perpendicular drop vs curve fitting 156
Curve fitting A: Linear Least-squares 162
Examples of polynomial fits 162
Reliability of curve fitting results 167
Algebraic Propagation of errors 167
Monte Carlo simulation 169
The Bootstrap method 170
Comparison of error prediction methods. 171

Effect of the number of data points on least-squares fit precision 171
Transforming non-linear relationships 172
Simple fitting of Gaussian and Lorentzian peaks by data transformation 173
Math and software details for linear least squares 177
Spreadsheets for linear least squares 178
Application to analytical calibration and measurement 180
Matlab and Octave 181
Fitting Single Gaussian and Lorentzian peaks 186
Curve fitting B: Multicomponent Spectroscopy 188
Classical Least-squares (CLS) multivariate calibration 188
Inverse Least-squares (ILS) calibration 190
Computer software for multiwavelength spectroscopy 191
Spreadsheets 191
Matlab/Octave and Python 193
Classical Least Squares in Python 198
Curve fitting C: Non-linear Iterative Curve Fitting 199
Spreadsheets and stand-alone programs 201
Matlab and Octave 203
Fitting peaks 204
Simplified general-purpose peak-fitting function 205
Variable shape types 206
Peak Fitting Functions for Matlab and Octave 207
Accuracy and precision of peak parameters 210
a. Model errors. 210
b. Background correction 219
c. Random noise in the signal. 221
d. Iterative fitting errors 224
A difficult case. 225
Fitting signals that are subject to exponential broadening. 228
The Effect of Smoothing before least-squares analysis 233
Peak Finding and Measurement 234
Simple peak detection 235
Gaussian peak measurement 236
Optimization of peak finding 239
How does ‘findpeaksG’ differ from ‘max’ in Matlab or ‘findpeaks’ in the Signal Processing Toolkit? 240

Accuracy of the measurements of peaks 241
Peak finding combined with iterative curve fitting. 242
Comparison of peak finding functions 245
Peak start and end 248
Using the peak table 251
Demo scripts 252
Peak Identification 252
Live Script Peak detection tool 253
iPeak: Keypress-operated interactive peak detector 255
iPeak keyboard Controls (version 8.1): 268
iPeak Demo functions 269
Spreadsheet Peak Finder Templates 273
Hyperlinear Quantitative Absorption Spectrophotometry 276
Background 278
Spreadsheet implementation 282
Matlab/Octave implementation: The fitM.m function 283
Demo function for Octave or Matlab 284
TFitDemo.m: Interactive demo for the Tfit method 285
Statistics of methods compared (TFitStats.m, for Matlab or Octave) 287
Comparison of analytical curves (TFitCalDemo.m, for Matlab or Octave) 287
Application to a three-component mixture 288
Case Studies and Numerical Experiments. 291
Can smoothed noise be mistaken for an actual signal? 291
Signal or Noise? 291
Buried treasure 295
The Battle Rounds: a comparison of methods 298
Ensemble averaging patterns in a continuous signal 301
Harmonic Analysis of the Doppler Effect 303
Measuring spikes 304
Fourier deconvolution vs curve fitting (they are not the same) 306
Digitization noise - can adding noise really help? 308
How low can you go? Performance with very low signal-to-noise ratios. 310
Signal processing in the search for extraterrestrial intelligence 312
Why measure peak area rather than peak height? 314
Using macros to extend the capability of spreadsheets 315
Random walks and baseline correction 318

Modulation and synchronous detection. 320
Measuring a buried peak 323
Signal and Noise in the Stock Market (updated) 326
Measuring signal-to-noise ratio in complex signals 330
Dealing with wide-ranging signals: segmented processing 333
Measurement Calibration 337
Numerical precision of computer software 340
Miniaturized signal processing: Raspberry Pi and Adafruit 344
Batch processing 345
Real-time signal processing 347
Peak sharpening 351
Dealing with variable data arrays in spreadsheets 353
Illuminating the invisible: Computer simulation of instruments 355
Who uses this book, its web site, documents, and software? 358
The Law of Large Numbers 361
Spectroscopy and chromatography combined: time-resolved Classical Least-squares 363
The mystery peak challenge 366
Developing Matlab Live Scripts and Apps 368
Matlab Live Scripts 369
Table of keypress-operated interactive functions and corresponding Live Scripts 370
Matlab Apps 370
Using real-signal modeling to determine measurement accuracy. 374
Signal processing software details. 376
Interactive smoothing, differentiation, and signal analysis (iSignal) 376
Keyboard-operated interactive Fourier filter 391
Live Script Fourier Filter 396
Matlab/Octave Peak Fitters 397
Matlab/Octave command-line function: peakfit.m 397
Examples 401
How do you find the correct input arguments for peakfit? 413
Working with the fitting results matrix "FitResults" 413
Demonstration script for peakfit.m 413
Fitting peaks in multi-column data 414
Dealing with complex signals with lots of peaks 415
Automatically finding and Fitting Peaks 416
Keypress operated Interactive Peak Fitter (ipf.m) 417

ipf keyboard controls (Version 13.4): Obtained by pressing the K key 419
Practical examples with real experimental data: 421
Operating instructions for ipf.m (version 13.4). 424
Demoipf.m 432
Execution time of peak fitting and other signal processing tasks 433
Iterative Curve Fitting Hints and Tips 434
Extracting the equations for the best-fit models 436
How to add a new peak shape to peakfit.m, ipf.m, iPeak, or iSignal 437
Which to use? peakfit, ipf, findpeaks…, iPeak, or iSignal? 438
Live Script Peak Fitter tool 440
Artificial Intelligence and Signal Processing 442
AI as a programmer's assistant. 442
Python: a free, open-source and powerful 448
Sliding average signal smoothing 448
Fourier transform and (de)convolution 450
Classical Least Squares 450
Peak Detection 451
Iterative least-squares fitting 451
Using AI to develop GUI applications in Python. 454
ChatGPT + Python in Canvas 456
Connecting Matlab and Python 458
Worksheets for Analytical Calibration Curves 461
Background 465
Fill-in-the-blanks worksheets for several different calibration methods 465
Comparison of calibration methods 469
Instructions for using the calibration templates 470
Frequently Asked Questions (taken from emails and search engine queries) 472
Catalog of signal processing functions, scripts, and spreadsheet templates 479
Peak shape functions (for Matlab and Octave) 479
Signal Arithmetic 481
Signals and Noise 482
Smoothing 485
Differentiation and peak sharpening 486
Harmonic Analysis 488
Fourier convolution and deconvolution 489
Fourier Filter 490

Wavelets and wavelet denoising 491
Peak area measurement 492
Linear Least-squares 493
Peak Finding and Measurement 495
Multicomponent Spectroscopy 501
Non-linear iterative curve fitting and peak fitting 502
Keystroke-operated interactive functions 506
Hyperlinear Quantitative Absorption Spectrophotometry 506
MAT files (for Matlab and Octave) and Text files (.txt) 507
Spreadsheets (for Excel or OpenOffice Calc) 507
Afterword 511
How this book came to be. 511
Who needs this software? 512
Organization 512
Methodology 512
Influence of the Internet 513
Writing 513
Software platform selection criteria 514
Outcomes 515
Impact 516
The Future 516
References 516
Publications that cite the use of my book, programs and/or documentation 521

Introduction

Interfacing measurement instrumentation to small computers for the purpose of online data acquisition is standard practice in the modern science laboratory. Scientists use computers for data acquisition, data processing, and storage, using digital computer-based numerical methods. Techniques covered in this book can transform signals into more useful forms, detect and measure peaks, reduce noise, improve the resolution of overlapping peaks, compensate for instrumental artifacts, test hypotheses, optimize measurement strategies, diagnose measurement difficulties, and visualize and decompose complex signals into their component parts. These techniques can often make difficult measurements easier by extracting more information from the available data. Many of these techniques employ laborious mathematical procedures that were not even practical before the advent of computerized instrumentation. It is important for you to appreciate the abilities, *as well as the limitations*, of these techniques. In recent decades, computer storage and digital processing have

become far less costly and literally millions of times more capable, reducing the cost of raw data and making complex computer-based signal processing techniques both more practical and necessary. Approximations and shortcuts that were once necessitated by mathematical convenience are no longer needed (e.g. pages 141, 199, 276). The introduction of artificial intelligence (AI) is revolutionizing the field. I take this up on page 442. It is not just the growth of computers: there are now new materials, new instruments, new fabrication techniques, new automation capabilities. We have lasers, fiber optics, superconductors, super-magnets, holograms, quantum technology, nanotechnology, and now chatbots that can write in perfect English and even write code in several computer languages (though this book is human-written). Sensors are smaller, cheaper, and faster than ever before. We can measure over a wider range of speeds, temperatures, pressures, and locations. People carry smartphones and fitness trackers everywhere they go, recording their heart rate, etc., creating new kinds of data sets that we never had before. The increasing complexity and volume of data requires advanced skills and tools to analyze and derive meaningful insights. Kate Keahey, a Senior Scientist at Argonne National Laboratory, writes that "Software is a vital part of the research landscape, and most researchers will benefit from understanding its possibilities, limitations and the requirements for building it".

This book covers only basic topics related to one-dimensional signals, not two-dimensional data such as images. It uses a *pragmatic* approach and is limited to mathematics only up to the elementary aspects of calculus, statistics, and matrix math. I use logical arguments, analogies, and hundreds of graphics and animations to explain ideas, rather than lots of formal mathematics. Data processing without math? Not really! Math is essential, just as it is for the technology of cell phones, GPS, digital photography, the Web, computer games, and modern cars. But you can get started using these tools without understanding all the underlying math and software details. Seeing it working makes it more likely that you will want to understand *how* it works. Nevertheless, in the end, it is not enough just to know how to operate the software, any more than knowing how to use a word processor or a MIDI sequencer makes you a good author or musician. I will get you *started* with things that work. It is up to you to decide if a deep dive into advanced topics becomes necessary for your purposes.

Why do I title this document "signal processing" rather than "data processing"? By "signal" I mean the x,y numerical time-series data recorded by scientific instruments, where x may be time or another quantity like energy or wavelength, as in the various forms of spectroscopy. This is sometimes called “squiggly line” data. I don’t deal so much with categorical data. In other words, I am oriented to data that you would plot in a spreadsheet using the scatter chart type rather than bar or pie charts.

Some of the examples here come from my own areas of research in analytical chemistry, but most come from a wide range of application areas. Over 800 journal papers, theses, and patents have cited my book and software, covering fields from academia, industry, environmental, medical, engineering, earth science, space, military, financial, agriculture, communications, and even music and speech science. Hundreds of readers have sent suggestions and experimental data from their own work that have helped shape my writing and software development. Much effort has gone into making this document concise and understandable. It has been very positively reviewed by many readers.

At the present time, this work does not cover image processing, pattern recognition, or factor analysis. For these topics and for a more rigorous treatment of the underlying mathematics of the topics I do cover, refer to the extensive literature on signal processing and on statistics and chemometrics.

Throughout this work, a wide range of applications and connections are described, some potentially intriguing, such as stock market investing (page 326), human cognitive biases (page 361), the failure of a NASA spacecraft (page 72), cosmic rays from outer space (page 50), adding one kind of noise to reduce another (page 308), studying beach erosion by wind-blown sand (page 304), coding with AI *(page 442),* expanding the classical limits of measurement in spectroscopy (page 276), the intelligibility of digitized speech (page 99 and 381), low-cost miniature computers (page 344), and an easy way to create interactive GUI apps (page 37). The citations list (page 521 in the PDF) is evidence of a truly mind-boggling range of applications.

This site makes considerable use of **Matlab**, a high-performance commercial and proprietary numerical computing environment and "fourth generation" programming language that is widely used in research (14, 17, 19, 20), **Octave**, a free Matlab alternative that runs almost all of the programs and examples in this book, and **Python**, a powerful but free and open-source language. There is a good reason why Matlab and Python have become so popular in science and engineering. They are powerful, fast, and relatively easy to learn. A very important aspect of both languages is the concept of functions, which are self-contained modules of code that accomplish a specific task. Functions usually "take in" data, process it, and "return" a result. (A trivial example is a=sqrt(b), which takes the value of b, computes its square root, and assigns it to the variable a). Once a function is written, it can be used again and again. Functions can also be "called" from the inside of other functions. Matlab and Python come with built-in functions for doing data processing tasks like matrix math, filtering, Fourier transforms, convolution and deconvolution, multi-linear regression, and optimization. *You can write your own custom functions* to use in your future programming projects, and you can download form their collection of thousands of useful user-contributed functions. Matlab has available a large number of add-ons called *toolboxes* created by experts in various fields for performing specialized mathematical tasks, including parallel computing, symbolic math (page 462) and to libraries written in C, C++, Java, Fortran, and Python and it's extensible to model-based design for dynamic and embedded systems. A companion piece called Simulink is a graphical programming environment for modeling, simulating and analyzing multidomain dynamical systems. Python also has an incredible array of free add-on libraries.

Most of the techniques covered in this work can also be performed in common spreadsheets such as Microsoft *Excel* or OpenOffice/LibreOffice *Calc* (11, 22, 23). The latter can be downloaded without cost from their web sites, https://sourceforge.net/projects/octave/ and https://www.libreoffice.org/.

You can download all of my Matlab/Octave or Python scripts and functions, and the spreadsheet templates, from http://tinyurl.com/cey8rwh at no cost. They have received extraordinarily positive feedback from users. (If you try to run one of my scripts or functions and it gives you a "missing function" error, look for the missing item from http://tinyurl.com/cey8rwh, download it into your Matlab/Octave search path. Type “help path” for more information about the search path).

If you do not know Matlab, read page 17 and following for a quick start-up. Matlab is specifically suited to numerical methods, matrix manipulations, plotting of functions and data, creation of algorithms and user interfaces, rapid prototyping, and deployment to portable devices such as tablets - essentially the needs of numerical computing by scientists and engineers. Matlab is loosely and dynamically typed, is less well-structured in a formal sense than other languages, and it tends to be more favored by scientists and engineers and less well-liked by computer scientists and professional

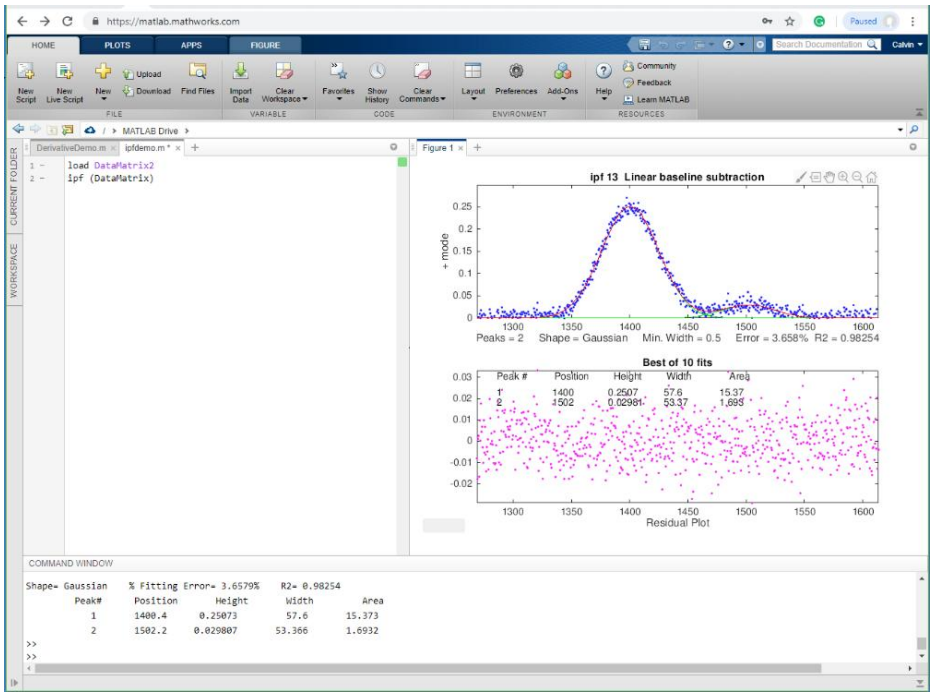
programmers. Matlab's streamlined integrated workflow is one of its biggest advantages. It combines coding, visualization, debugging, and data analysis in a single, cohesive environment, making it ideal for:

1. Rapid Prototyping: Quickly write code, visualize results, and iterate without needing external tools.
2. Data Analysis and Visualization: Built-in tools and straightforward plotting functions make it simple to analyze and visualize data.
3. Integrated Debugging: Debugging in Matlab is intuitive, with tools to inspect variables, step through code, and view results in real time.
4. Toolboxes: Prebuilt, specialized toolboxes for signal processing, machine learning, and more save time and reduce complexity.
5. Ease of Use: Matlab's language and interface are designed to prioritize user-friendly workflows, with everything accessible through the Command Window, Editor, and figures.

While languages like Python offer unmatched flexibility and scalability, Matlab 's simplicity and focus make it particularly effective for researchers, engineers, and scientists who value an environment where everything just works without needing to manage external packages or dependencies.

Python is different in many details, is harder to install, and requires the installation of several add-on "packages", but it has the *great advantage of being free*. I have used the Anaconda distribution and Spyder. Note: you can use an AI chatbot to convert Matlab to Python or vice versa (page 442).

There are several versions of Matlab, including stand-alone low-cost student and home versions, fully functional versions that run in a web browser (see graphic below), and apps that run on iPads and iPhones. See https://www.mathworks.com/pricing-licensing.html for prices and restrictions in their use.



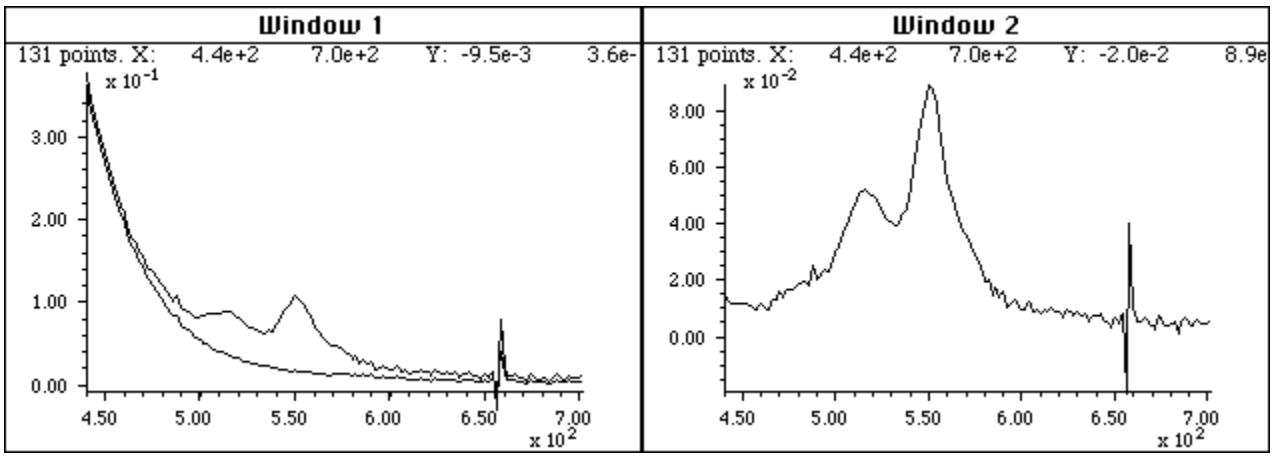
Matlab Online running my interactive peak fitter, ipf.m (page 417) in a Windows PC browser

There are alternatives to Matlab: *Octave*, which is essentially a Matlab clone. There is also Scilab, FreeMat, Julia, and Sage, which are mostly or somewhat compatible with the MATLAB language. For a discussion of other possibilities, see http://www.dspguru.com/dsp/links/matlab-clones.

If you are reading this book *online*, on an Internet-connected computer, you can click or Ctrl-click on any of the http Web addresses or on the names of downloadable software or animations to view or download that item. For a complete list of all my software, see page 479 or http://tinyurl.com/cey8rwh.

Signal arithmetic

The most basic signal processing operations are those that involve simple signal arithmetic: point-by-point addition, subtraction, multiplication, or division of two signals or of one signal and a constant. Despite their mathematical simplicity, these operations can be very useful. For example, in the left part of the figure below (Window 1) the top curve is the optical absorption spectrum of an extract of a sample of oil shale, a kind of rock that is a source of petroleum.



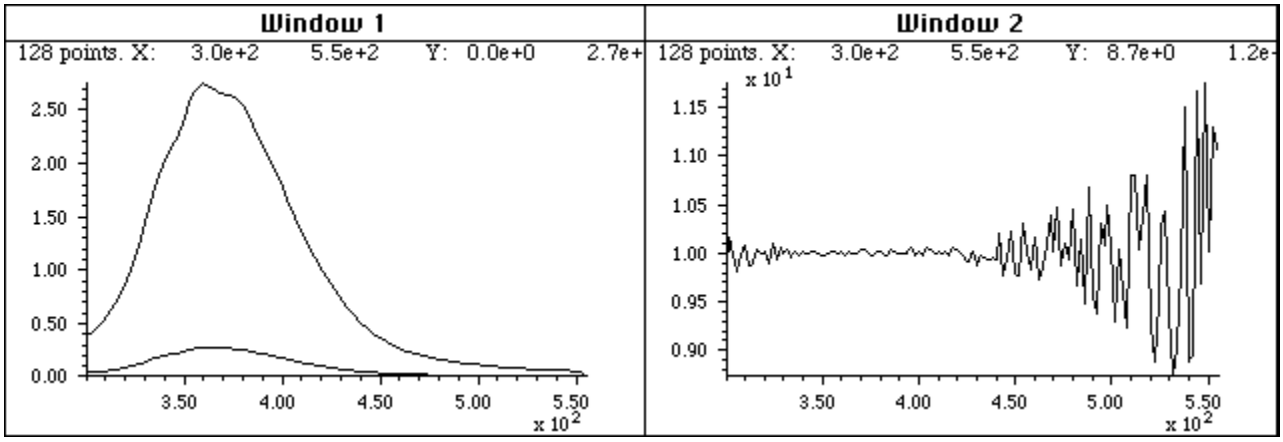
A simple point-by-point subtraction of two signals allows the background (bottom curve on the left) to be subtracted from a complex sample (top curve on the left), resulting in a clearer picture of what is really in the sample (right). (X-axis = wavelength in nm; Y-axis = absorbance).

This optical spectrum exhibits two absorption bands, at 515 nm and 550 nm. These peaks are due to a class of molecular fossils of chlorophyll called *porphyrins*, which are used as "geomarkers" in oil exploration. These bands are superimposed on a background absorption caused by the extracting solvents and by non-porphyrin compounds in the shale. The bottom curve is the spectrum of an extract of a non-porphyrin-bearing shale, showing only the background absorption. To obtain the spectrum of the shale extract without the background, the background (bottom curve) is simply subtracted from the sample spectrum (top curve). The difference is shown in the right in Window 2 (note the change in the Y-axis scale). In this case, the removal of the background is not perfect, because the background spectrum is measured on a separate shale sample. However, it works well enough so that you can see the two bands more clearly and it is easier to measure precisely their absorbances and wavelengths. (Thanks to the late Prof. David Freeman of the Univ. of Maryland for the spectra of shale oil extracts).

In this example and the one below, I am assuming that the two signals in Window 1 have the *same x-axis values* - in other words, that both spectra have been digitized at the same set of wavelengths.

Subtracting or dividing two spectra would not be valid if two spectra were digitized over different wavelength ranges or with different intervals between adjacent points. The x-axis values must match up point for point. In practice, this is very often the case with data sets acquired within one experiment on one instrument, but you must be careful if you change the instrument's settings or if you combine data from two experiments or two instruments. It is possible to use the mathematical technique of *interpolation* to change the sampling rate (x-axis interval) or to equalize unequally spaced x-axis intervals of signals. The results are usually only approximate but often close enough in practice. Excel can perform calculations using the forecast function. Matlab and Octave have built-in functions for interpolation, including interp1.m, see example1 (graphic) and example2 (graphic).

Sometimes one needs to know whether two signals have the same shape, for example comparing the signal of an unknown to a stored reference signal. Most likely the amplitudes of the two signals will be different. Therefore, a direct overlay or subtraction of the two signals will not be useful. One possibility is to compute the point-by-point *ratio* of the two signals. If they have the same shape, the ratio will be constant. For example, examine this figure:



Do the two signals on the left have the same shape? They certainly do not look the same, but that may simply be because one is much weaker than the other one. The ratio of the two signals, shown in the right part (Window 2), is relatively constant from 300 to 440 nm, with a value of 10 +/- 0.2. This means that the shape of these two signals is very nearly identical over this x-axis range.

The left part (Window 1) shows two superimposed signals, one of which is much weaker than the other. But do they have the same shape? It is hard to tell. It's much clearer if you look at the *ratio* of the two signals, shown in the right part (Window 2), which is relatively constant from x=300 to 440, with a value of 10 +/- 0.2. This means that the shape of these two signals is the same, within about +/-2 %, over this x-axis range, and that the top curve is about 10 times more intense than the bottom one. Above x=440 the ratio is not even approximately constant. This is caused by *noise*, which is the subject of the next section (page 24).

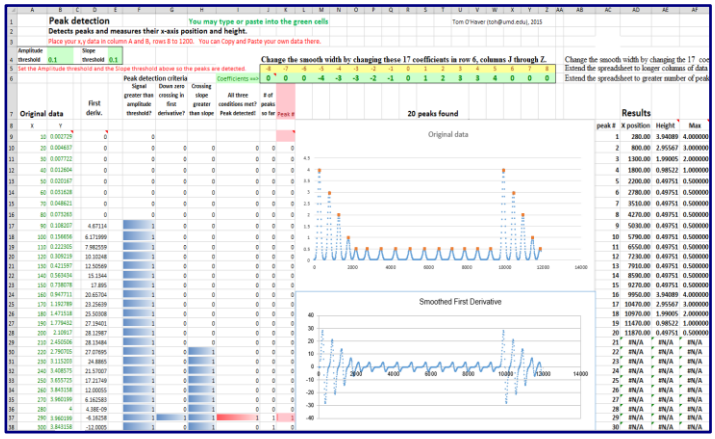
When you divide two vectors point by point, *even a single zero* in the denominator vector will stop the program with a **division by zero error**. A vanishingly small but finite number in the denominator will not stop the program but will generate a huge number in the result. Both problems can usually be avoided by adding a small non-zero constant to the denominator or by applying a small amount of smoothing (page 40) of the denominator or by using the Matlab/Octave function rmz.m (**rem**ove **z**eros)

which replaces zeros with the nearest non-zero numbers. The related function rmnan.m removes NaNs ("Not a Number") and Infs ("Infinite") from vectors, replacing with neighboring real finite numbers.

On-line calculations and plotting. *Wolfram Alpha* is a free Web site and a smartphone app that is an extremely useful computational tool and information source, including capabilities for symbolic mathematics, plotting, vector and matrix manipulations, statistics and data analysis, and many other topics (from the makers of *Mathematica*, page 462). *Statpages.org* can perform a huge range of statistical calculations and tests. There are several Web sites that specialize in plotting data, including Plotly and Grapher. All of these require a reliable Internet connection, and they can be useful when you are working on a mobile device or computer that does not have the required software installed. In the PDF version of this book, you can **Ctrl-Click** on these links to open them in your browser.

Signal arithmetic in Spreadsheets

Popular spreadsheets, such as *Excel* or *Open Office Calc*, are aimed mainly at business and financial applications, but still have built-in functions for many common math operations, named variables, x,y plotting, text formatting, matrix math, etc. Cells can contain numerical values, text, mathematical expressions, or references to other cells. You can represent a spectrum as a row or column of cells. You



can represent a set of spectra as a rectangular block of cells. You can assign your own names to individual cells or to ranges of cells and then refer to them in mathematical expression by name. You can copy mathematical expressions across a range of cells, with the cell references changing or not as desired. You can make plots of various types (including the all-important *x-y* or *scatter* graph) by menu selection. For a nice video demonstration, see this YouTube video: http://www.youtube.com/watch?v=nTlkkbQWpVk. Both *Excel* and *Calc* offer a "form design" capability with a full set of user interface objects such as buttons, menus, sliders, and text boxes. You can use these to create attractive graphical user interfaces for end-user applications, such as ones I have created for teaching analytical chemistry courses on http://terpconnect.umd.edu/~toh/models/. The latest versions of both Excel and OpenOffice *Calc* can open and save either spreadsheet file formats (.xls and .ods, respectively). Simple spreadsheets in either format are compatible with the other program. However, there are small differences in the way that certain operations are interpreted, and for that reason I supply most of my spreadsheets in .xls (for *Excel*) *and* in .ods (for *Calc*) formats. Google "Differences between the Open-Document Spreadsheet (.ods) format and the Excel (.xlsx) format". Basically, *Calc* can do almost everything *Excel* can do, but *Calc* is free to download and is more Windows-standard in terms of look-and-feel. Excel is more "Microsoft-y" and is often faster than *Calc*. If you have access to *Excel*, I recommend using that. Both *Excel* and *Calc* are often updated.

If you are working on a tablet or smartphone, you could use the Excel mobile app, Numbers for iPad, or several other mobile spreadsheets. These apps can do basic tasks but do not have the fancier capabilities of the regular computer versions. By saving their data in the "cloud" (e.g., iCloud or

SkyDrive), these apps automatically sync changes in both directions between mobile devices and desktop or laptop computers, making them useful for field data entry. When in doubt, ask an AI *(page 442)*. Its knowledge of nerdy technical details is astounding.

Signal arithmetic and plotting in Matlab

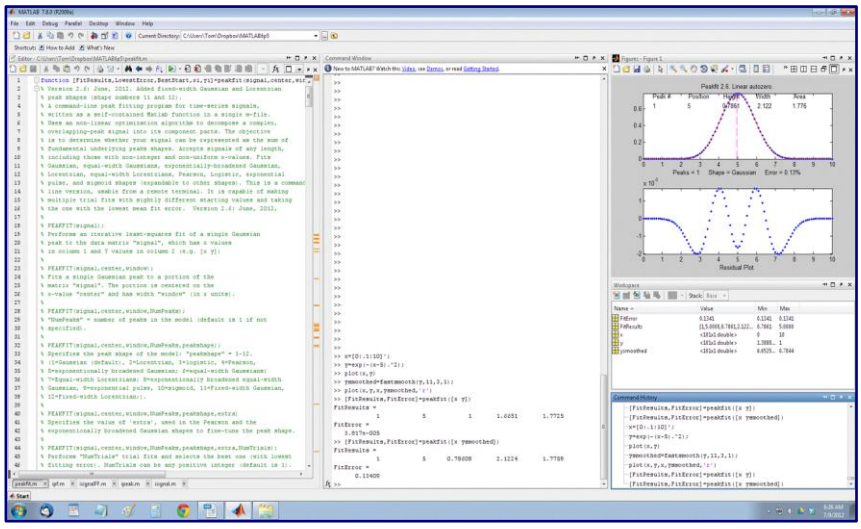
In Matlab (and in its GNU clone *Octave*) or in Python, arithmetic is much like any other language: e.g. (a+b)/c. In Matlab and in Python (page 22), a single variable can represent either a single "scalar" value, a *vector* of values (such as a spectrum or a chromatogram), a *matrix* (a rectangular array of values, such as a set of spectra), or a set of *multiple* matrices. *All the standard math operations and functions adjust to match*. This greatly facilitates mathematical operations on signal waveforms. The subtraction of two signals **a** and **b**, as on page 14, can be performed simply by writing **a-b**. Likewise, the ratio of two signals in Matlab, as on page 15, is "**(a./b)".** So, "**./**" means divide point-by-point and ".*" means multiply point-by-point. The * by itself means *matrix* multiplication, which you can use to perform repeated multiplications without using loops. For example, if x is a vector.

```
A=[1:100]'*x;
```

creates a matrix **A** in which each column is x multiplied by the numbers 1, 2,...100. It is equivalent to writing a "for" loop like this, but more compact to write and faster to execute:

```
for n=1:100;
  A(:,n)=n.*x;
end
```

Plotting data. If you have signal amplitudes in the variable **y**, you can plot it just by typing "**plot(y)**". And if you also have a vector `t` of the same length containing the times at which each value of **y** was obtained, you can plot **y** vs **t** by typing "**plot(t,y)**". Two signals **y** and **z** can be plotted on the same time axis for comparison by typing "**plot(t,y,t,z)**". (Matlab automatically assigns different colors to each line. You can control the color and line style by adding additional symbols. For example, "**plot(x,y,'r.',x,z,'b-')**" will plot **y** vs **x** with red dots and **z** vs **x** with a blue line. You can divide up one figure window into multiple smaller plots by placing



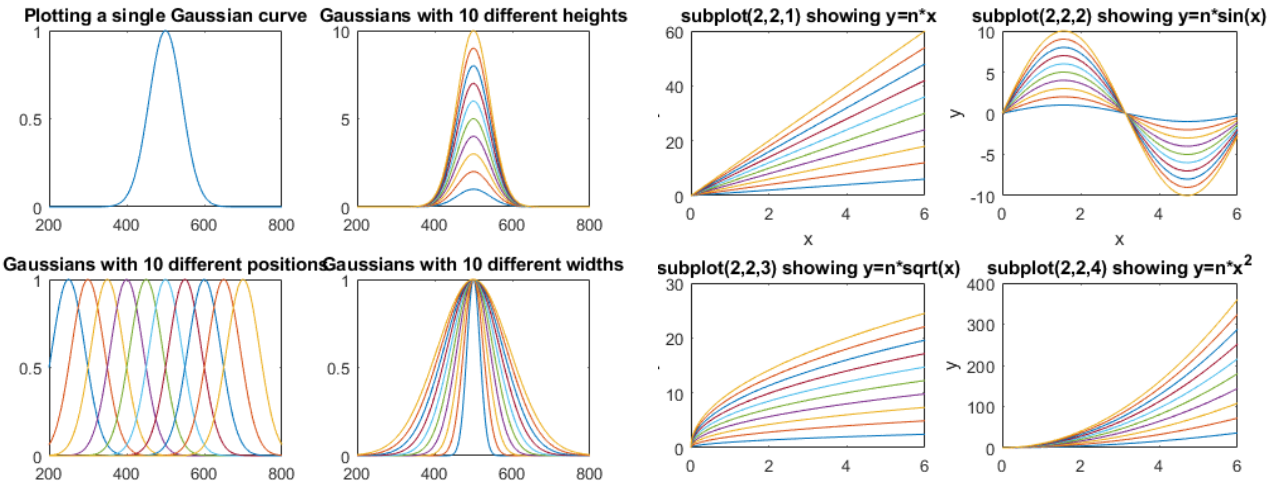
subplot(m,n,p) before the plot command to plot in the p[th] section of an m-by-n grid of plots. (If you are reading this online, you can click here for an example of a 2x2 subplot. You can also select, copy, and paste, or select, drag and drop, any of the single-line or multi-line code examples into the Matlab or Octave editor or directly into the command line and press **Enter** to execute it immediately). In Matlab, type "help plot" for more plotting options. In Python, "**import matplotlib.pyplot as plt**" does Matlab-like plotting.

For **publication-quality** graphs, click on a Figure window, then click **File > Export setup**, choose the size, resolution, color, fonts, etc., then click **Export** and select the file format (e.g., TIF, eps, etc.). You can also use PlotPub , a downloadable library that is free, easy to use, allows great flexibility in choosing graph details, and creates great-looking graphs within Matlab that can be exported in EPS, PDF, PNG and TIFF with adjustable resolution. Here is an example (script, graphic).

The function **`max(y)`** returns the maximum value of **`y`** and **`min(y)`** returns the minimum. Individual elements in a vector are referred to by *index number*. For example, **`t(10)`** is the 10th element in vector **t**, and **`t(10:20)`** is the vector of values of **t** from the 10th to the 20th entries. You can find the index number of the entry closest to a given value in a vector by using my val2ind.m function. For example, **`t(val2ind(y,max(y)))`** returns the time of the maximum y, and **`t(val2ind(t,550):val2ind(t,560))`** is the vector of values of **t** between 550 and 560 (assuming **t** contains values within that range). The *units* of the time data in the **t** vector could be anything - microseconds, milliseconds, hours, any time units.

A Matlab variable can also be a *matrix*, a set of vectors of the same length combined into a rectangular array. For example, intensity readings of 10 different optical spectra, each taken at the same set of 100 wavelengths, could be combined into the 10x100 matrix S. `S(3,:)` would be the third of those spectra and `S(5,40)` would be the intensity at the 40th wavelength of the 5th spectrum. The Matlab scripts plotting.m (left) and plotting2.m (right) show how to plot multiple signals using matrices and subplots.

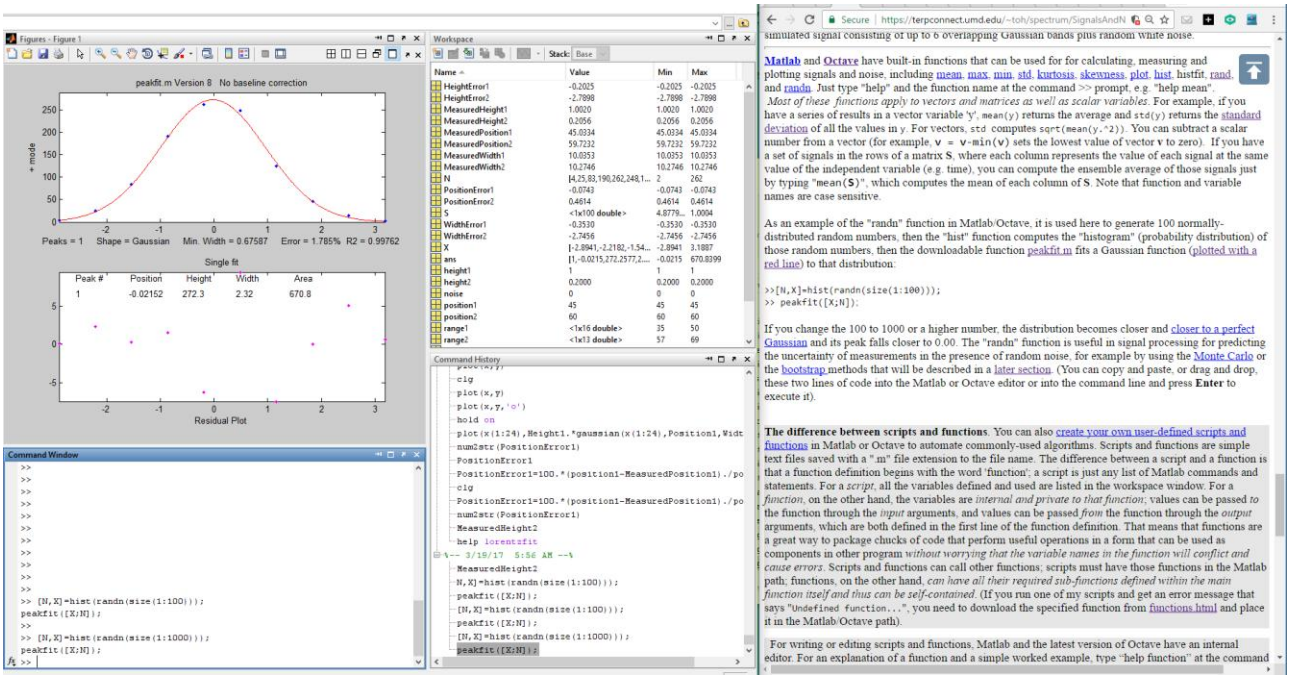
See TimeTrial.txt for details. It will help if you pre-allocate memory space for the **A** matrix by adding the statement `A=zeros(100,100)` before the loop. Even then, the matrix notation is faster than the loop.



In Matlab/Octave, "/" is not the same as "\". Typing "b\a" will compute the "matrix left divide", in effect the weighted average ratio of the amplitudes of the two vectors (a type of least-squares best-fit solution). The point here is that *Matlab does not require you to deal with vectors and matrices as collections of numbers*. It knows when you are dealing with matrices, or when the result of a calculation will be a matrix, and it adjusts calculations accordingly. See https://www.mathworks.com/help/matlab/matlab_prog/array-vs-matrix-operations.html.

Probably the most common errors you'll make in Matlab/Octave are punctuation errors, such as mixing up periods, commas, colons, and semicolons, or parentheses, square brackets, and curly brackets. Type "help punct" at the Matlab prompt and *read the help file* until you fall asleep. *Little things can mean a lot* in Matlab. Another common error is getting the rows and columns of vectors and matrices mixed up. (Full disclosure: I *still* make all these kinds of mistakes all the time). Click for a text file that gives examples of common vector and matrix operations and errors in Matlab and Octave. If you are new to this, I recommend that you read this file and play around with the examples there. Writing Matlab is a trial-and-error process, with the emphasis on *error*. Start simply, get it to work, then expand it in steps.

There are many code examples in this text that you can Copy and Paste and modify into the Matlab/ Octave command line, which is a great way to learn. In the PDF version of this book, you can select, copy, and paste, or select, drag and drop, any of the single-line or multi-line code examples into the Matlab or Octave editor or directly into the command line and press **Enter** to execute it immediately). This is especially convenient if you run Matlab and read my web site or book on the same computer. Position the windows so that Matlab shares the screen with this website (e.g. Matlab on the left and web browser on the right as shown below). Or, even better, some desktop computers have *two* monitor



outputs, so you can use two monitors simultaneously to expand the desktop horizontally.

Hint: *If you try to run one of my scripts or functions and it gives you a "missing function" error, look for the missing item from* http://tinyurl.com/cey8rwh*, download it into the* search path*, and try again.*

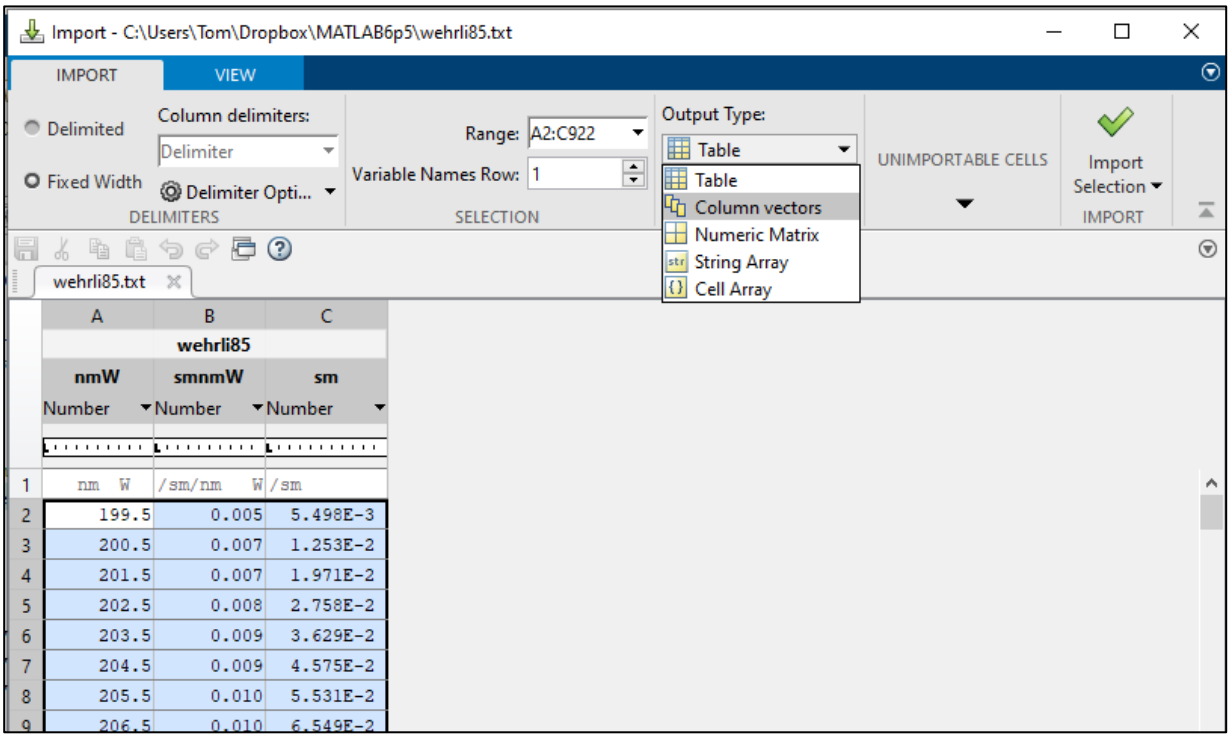
One thing that you will notice about Matlab is that the *very first time* you execute a script or function, and *only* the first time, there is a small delay before execution, while Matlab compiles the code into machine language. However, that only happens the *first* time. After that, the execution starts instantly. (For the fastest execution, the separately available "Matlab Compiler" lets you share programs as *stand-alone* applications, separate from the Matlab environment. "Matlab Compiler SDK" lets you build C/C++ shared libraries, Microsoft .NET assemblies, Java classes, and Python packages from Matlab programs). You can even do some *real-time* plotting in Matlab/Octave. See page 347.

Importing and cleaning data

You can import your data into Matlab by using the “Import Data” and “Clean Data” buttons in the Home tab or the xlsread or importdata functions on the command line or in a script. Data can be imported from plain text files (.txt), CSV files (comma separated values), from several image and sound formats, or from spreadsheets. For example, the following lines will read the first two columns of the csv file “Sample_5.0ppm.csv” in the current folder and assign them to the vectors x5 and y5, i.e. the independent and dependent variables, respectively:

```
mydata=xlsread('Sample_5.0ppm.csv');
x5=mydata(:,1);
y5=mydata(:,2);
```

Data cleaning functions in Matlab include fixing missing and outlier data, smoothing, normalizing, and stacking and unstacking tables (reference). The script "xlsreadDemo.m" provides a simple example of reading a multi-column spreadsheet “xlsx” file. For more complex spreadsheets, Matlab has a very useful *Import Wizard* (click **File > Import Data**) that gives you a preview into the data file, parses the data file looking for columns and rows of numeric data and their labels, and gives you a chance to select and re-label variables and to choose to import them as vectors, matrices, or tables. You can even click on the little arrow next to “Import selection” and *Matlab will write you a script that will perform those operations*, which you can modify for other file types and formats.



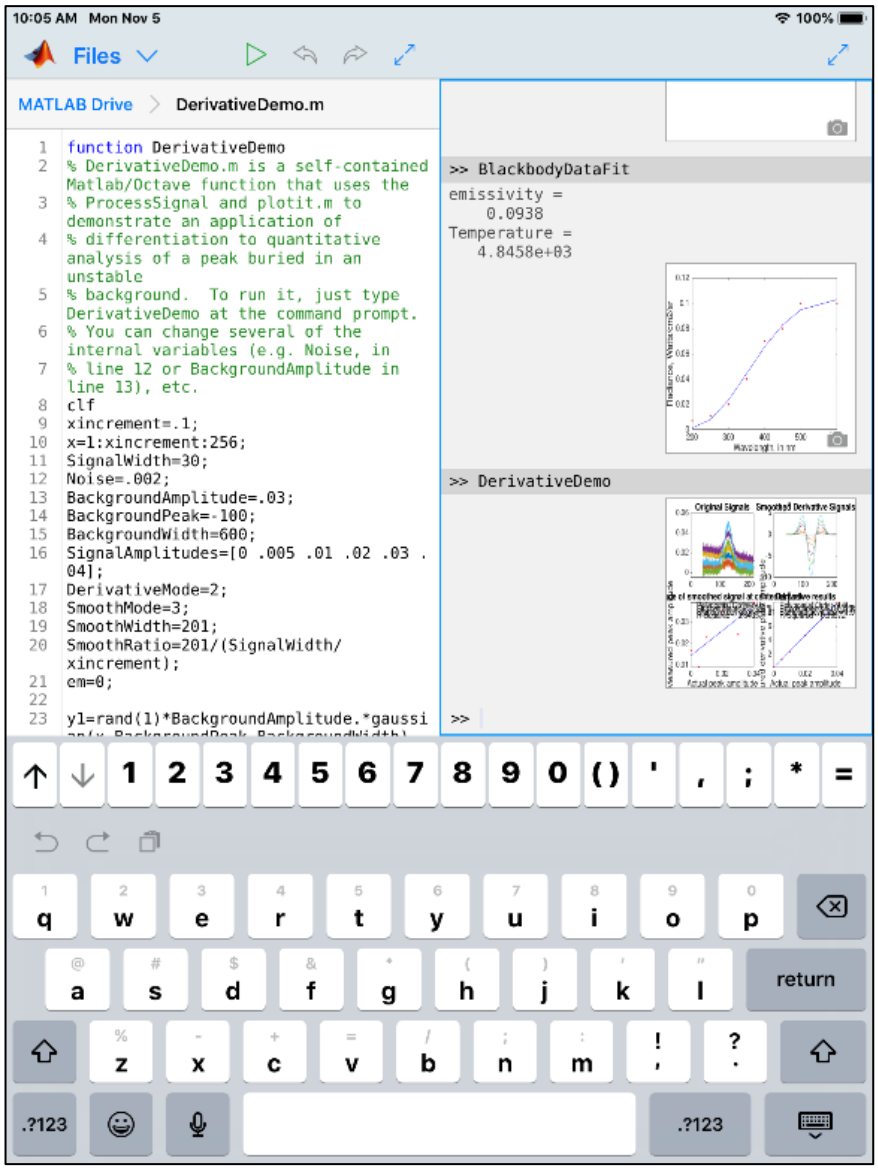
JCAMP-DX is a standard file form for exchanging infrared spectra and related chemical and physical information between spectrometer data systems of different manufactures. Matlab’s jcampread function can import such data. For example, see ReadJcampExample.m. It is also possible to import *approximate* data from graphical line plots or *printed* graphs by using the built-in "ginput" function that obtains numerical data from the coordinates of mouse clicks, or by using more automated applications

such as "Data Thief 3" or *Figure Digitizer* in the Matlab File Exchange. Obviously, the results will not be as accurate as having access to the original data in a numerical data file. Matlab R2013a or newer can read the sensors on your iPhone or Android phone via Wi-Fi. To read the analog output signals of older analog instruments, you need an analog-to-digital converter, an Arduino microcontroller board, or a USB voltmeter. Mathworks has separate data acquisition toolbox for Matlab. **Note**: The addition, subtraction, multiplication, or division of two digital signals requires that they have the *same number of data points*. If necessary, you can remove some points from the longer signal or add some points to the shorter one (usually zeros, which is called "zero filling").

Python can import data in text, CSV, JSON, Matlab, and several other formats, using the V*ariable Explorer* panel in the *Spyder* desktop, or through the separately downloadable Pandas Data Analysis package.

Matlab Versions.

The standard *commercial* version of Matlab is expensive (over $2000) but there are *student* and *home* versions that cost much less (as little as $49 for a basic student version) and that have all the capabilities to perform any of the methods detailed in this book *at comparable execution speeds*. There is also Matlab Online, which *runs in a common web browser* (see the graphic on 13). You do not even need a regular computer: there is a free Matlab Mobile app that runs a Matlab interface on Internet-connected iPhones and iPads (illustrated on the right). This requires only a basic student license and uses all the standard functions, plus any of my functions or scripts (or any of your own) *that you have previously up-loaded* to your account on the Matlab cloud. All these versions have computational speeds that are mostly within a factor of 2 of each other, as shown by TimeTrial.txt.

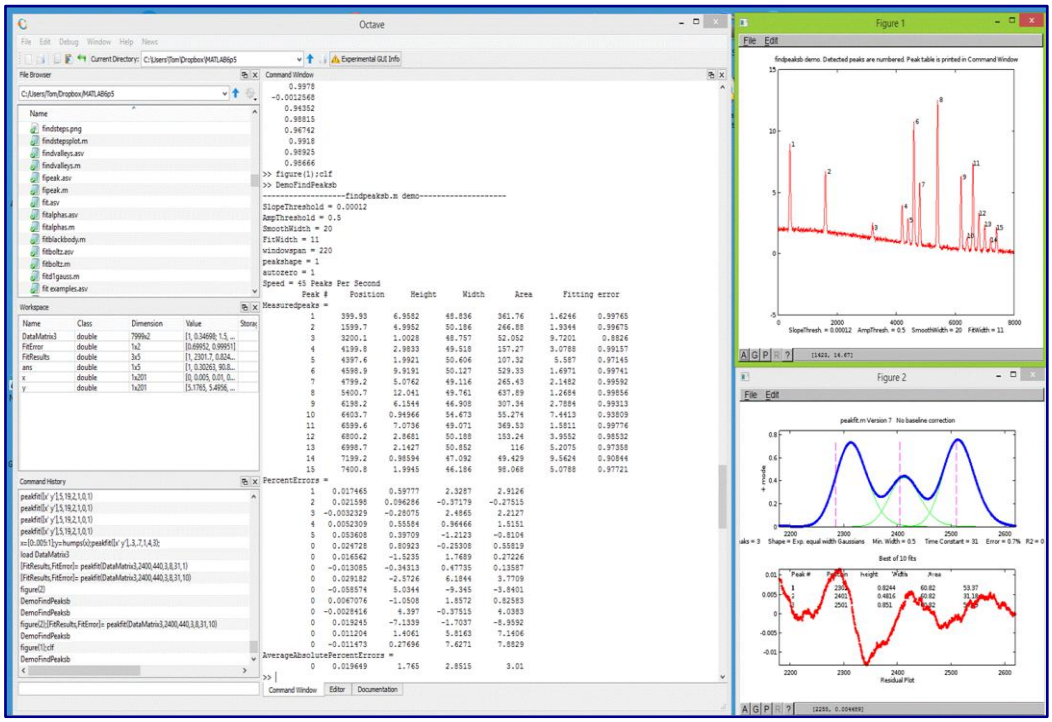


Math in Python

In Python, after importing numpy as np, the basic math functions are very similar to those in Matlab: **`len(d), np.sum(d), np.mean(d), np.std(d), np.sqrt(d), max(d), min(d)`**. Exponentiation in notated as ** rather than ^ as in Matlab. See page 439 for **other examples**.

GNU Octave

Octave is a free alternative to Matlab that is "mostly compatible". DspGURU says that Octave is "...a mature high-quality Matlab clone. It has the highest degree of Matlab compatibility of all the clones." Everything I said above about Matlab also works in Octave. In fact, *the most recent versions of almost all my Matlab functions, scripts, demos, and examples in this document will work in the latest version of Octave without change.* The keystroke-operated interactive functions *iPeak* (page 253), *iSignal* (page 376), *ipf.m* (page 416) and *ifilter.m*, require *separate versions* for Octave, which use different keys for pan and zoom. There is an FAQ that may help in porting Matlab programs to Octave. See "Differences



Between Octave & Matlab". There are Windows, Mac, and Unix versions of Octave. The Windows version can be downloaded from Octave Forge. There is lots of help online: Google "GNU Octave" or see the YouTube videos for help. For signal processing applications specifically, Google "signal processing octave".

Octave Version 10.3.0 is now available for download. The documentation is online (see https://www.octave.org). Almost all my scripts and functions run on Octave. However, it is still computationally slower on average than the latest Matlab version, depending on the task (specific comparisons for several different signal processing tasks are in TimeTrial.txt). Bottom line: Matlab is better, but if you cannot afford Matlab, Octave provides most of the functionality for 0% of the cost. (Note: the older Octave 3.6 can even run on a Raspberry Pi, a low-cost miniature computer described on page 344).

Spreadsheet or Matlab/Python?

For signal processing, computer languages like Matlab/Octave or Python are faster and more powerful than using a spreadsheet, but it is likely that spreadsheets are more commonly installed on science workers' computers than Matlab, Octave or Python. For one thing, spreadsheets are easier to get started with, and they offer flexible presentation and user interface design. Spreadsheets are better for simple manual data entry. You can easily deploy them on portable devices such as smartphones and tablets (e.g. using *Google Sheets*, i*Cloud Numbers* or the *Excel* app). *Spreadsheets are concrete and more low-level, showing every single value explicitly in a cell.* In contrast, Matlab/Octave and Python are more high-level and abstract, because a single variable can be a number, a vector, or a matrix, and each punctuation or function can do so much. This is very powerful, but it is harder to master *at first*. In Matlab and Octave, functions and script files ("m-files") are just plain text files with an ".m" extension (or ".py" in the case of Python), so *those files can be opened and inspected using any text editor, even on devices that do not have those programs installed*, which facilitates the translation of its scripts and functions into other languages. In addition, user-defined functions can call *other* built-in or user-defined functions, which in turn can call other functions, and so on, allowing you to *build up very complex high-level functions in layers*. Fortunately, Matlab and Python can easily analyze Excel ".xls" and ".xlsx" files and import the rows and columns into vector/matrix variables.

Using the analogy of electronic circuits, spreadsheets are like *discrete component* electronics, where every resistor, capacitor, inductor, and transistor is a discrete, macroscopic entity that you can see and manipulate directly. A function-based programming language like Matlab/Octave is more like *micro-electronics*, where the functions (the "m-files" that begin with "function...") are the "chips", which condense complex operations into one package *with documented inputs and outputs* (the function's input and output *arguments*) that you can connect to other functions, but which *hide the internal details* (unless you care to look at the code, which you always can do). For example, the "555 timer" is an 8-pin timer, pulse generator and oscillator chip introduced way back in 1972, which is still in use today and has become the most popular integrated circuit ever manufactured. Almost all electronics are now done with chips, because *it is easier to understand the relatively small number of inputs and outputs* of a chip than to deal with the greater number of internal components. Much of Matlab/Octave is written in Matlab/Octave itself, using more basic functions to build more complex ones. You can write new functions of your own that essentially *extend the language in whatever direction you need* (page 36).

The bottom line is that spreadsheets are easier at first, but for more complex tasks, Python or Matlab/Octave/ is computationally faster, can handle much larger data sets, and can do more with less effort. This is demonstrated by the comparison of both platforms for *multicomponent spectroscopy*, covered on page 188 (RegressionDemo.xls versus the Matlab/Octave CLS.m). Even more dramatic are the different approaches to *finding and measuring peaks* in signals, which is covered in the section beginning on page 234 (i.e. a 250Kbyte spreadsheet versus a 7Kbyte Matlab script that does the same thing but is *50 times faster*). If you have large quantities of data and you need to run it through a multi-step customized process automatically, hands-off, and as quickly as possible, then Matlab/Python is a great way to go. Moreover, you can even write a script in Matlab that will *automate the hands-off processing of volumes of data* stored in separate data files on your computer, as shown by the example on page 345.

Spreadsheets, Matlab/Octave, and Python programs have a huge advantage over commercial end-user programs and compiled freeware programs. You can *inspect and modify them* to customize the routines for specific needs. Simple changes are easy to make with little or no knowledge of programming. For example, you could easily change the labels, titles, colors, or line style of the graphs in Matlab or Octave programs for your own purposes: use **Find...** to search for "title(", "label(" or "plot(". My Matlab code contains *comments that indicate places where you can make specific changes*: search for the word "change". *I invite you to modify my scripts and functions as you wish.* The software license embedded in the comments of my Matlab/Octave code is very liberal.

Signals and noise

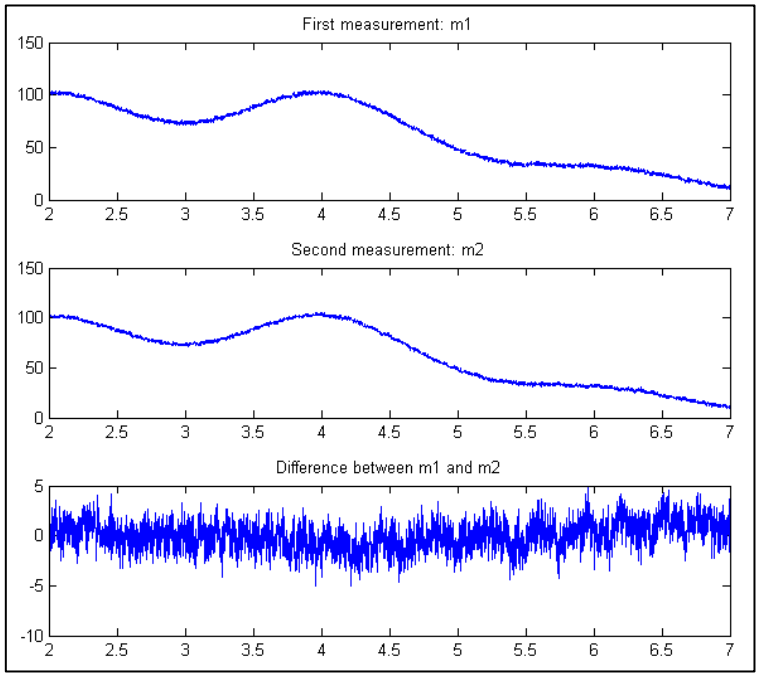
Experimental measurements are never perfect, even with sophisticated modern instruments. Two main types of measurement errors are recognized: (a) *systematic error,* in which every measurement is consistently less than or greater than the correct value by a certain percentage or amount, and (b) *random error,* in which there are unpredictable variations in the measured signal from moment to moment or from measurement to measurement. This latter type of error is often called *noise*, by analogy to acoustic noise. There are many sources of noise in physical measurements, such as building vibrations, air currents, electric power fluctuations, stray radiation from nearby electrical equipment, static electricity, interference from radio and TV transmissions, turbulence in the flow of gases or liquids, random thermal motion of molecules, background radiation from natural radioactive elements, the basic quantum nature of matter and energy itself, digitization noise (the rounding of numbers to a fixed number of digits), and "cosmic rays" from outer space (seriously). Then, of course, there is the ever-present "human error", which can be a major factor anytime humans are involved in operating, adjusting, recording, calibrating, or controlling instruments and in preparing samples for measurement. If random error is present, then a set of repeat measurements, "d" ("d" standing for *data*), will yield results that are not all the same but rather vary or scatter around some average value, which is the sum of the values divided by the number of data values in "d": `sum(d)./length(d)` or simply `mean(d)` in Matlab/Octave notation. The most common way to measure the amount of variation or dispersion of a set of data values is to compute the *standard deviation*, "std", which is the square root of the sum of the squares of the deviations from the average divided by one less than the number of data points: `sqrt(sum((d-mean(d)).^2)./ (length(d)-1))`, in Matlab/Octave notation. These are most easily calculated by the built-in functions mean(d) and std(d), where d is the data vector. A basic fact of random variables is that when they combine, you must calculate the results *statistically*. For example, when two random variables are added, the standard deviation of the sum is the "quadratic sum" (the square root of the sum of the squares) of the standard deviations of the individual variables. In Matlab, the function **randn(1,n)** returns **n** random numbers with a standard deviation of 1. Therefore:

$$\lim_{n\to\infty}(std(randn\,(1,n))) = 1$$

$$\lim_{n\to\infty}\Big(std\big(randn(1,n) + randn(1,n)\big)\Big) = sqrt(2)$$

This is demonstrated by the series of Matlab/Octave commands at this link. Try it.

The term 'signal' can have two meanings. In the more general sense, it can mean the *entire* data recording, including the noise and other artifacts, as in the expression "raw signal", before processing is applied. But it can also mean only the *desirable* or *important* part of the data, the *true underlying signal* that you seek to measure, as in the expression "signal-to-noise ratio". A fundamental problem in signal measurement is distinguishing the true underlying signal from the noise. For example, suppose you want to measure the average of the signal over a certain time or the height of a peak or the area under a peak that occurs in the data. In the absorption spectrum in the right-hand half of the figure on page 14, the "important" parts of the data are probably the absorption peaks located at 520 and 550 nm. The height or the position of either of those peaks might be considered the signal, depending on the application. In this example, the height of the largest peak is about 0.08 absorbance units. But how to measure the noise? In the exceptional case that you have a physical system *and* a measuring instrument which are *both* completely stable (*except* for the random noise), an easy way to isolate and measure the noise is to *record two signals m1 and m2 of the same physical system*. If you subtract those two recordings, the signal part will be cancelled. Then the standard deviation of the noise in the original signals is given by sqrt((std(m1-m2)^2)/2), where "sqrt" is the square root and "std" is the standard deviation. (The derivation of this expression is based on the rules for mathematical error propagation and is worked out in https://terpconnect.umd.edu/~toh/spectrum/Derivation.txt). The Matlab/Octave script "SubtractTwoMeasurements.m" demonstrates this process quantitatively and graphically above.



But suppose that the measurements are not that reproducible or that you had only *one* recording of that spectrum and no other data. In that case, you could try to estimate the noise in that single recording, based on the *assumption* that the visible *short-term fluctuations* in the signal - the little random wiggles superimposed on the smooth signal - are *noise* and not part of the true underlying signal. That depends on some knowledge of the origin of the signal and the possible forms it might take. The examples in the previous section (page 14) are the optical absorption spectra of liquid solutions over the wavelength range of 450 nm to 700 nm. These solutions ordinarily exhibit broad smooth peaks with a width of the order of 10 to 100 nm, so those little wiggles must be *noise*. In this case, those fluctuations have a standard deviation of about 0.001. Often the best way to measure the noise is to locate a region of the signal on the baseline where the signal is flat and to compute the standard deviation in that region. This is easy to do with a computer if the signal is digitized.

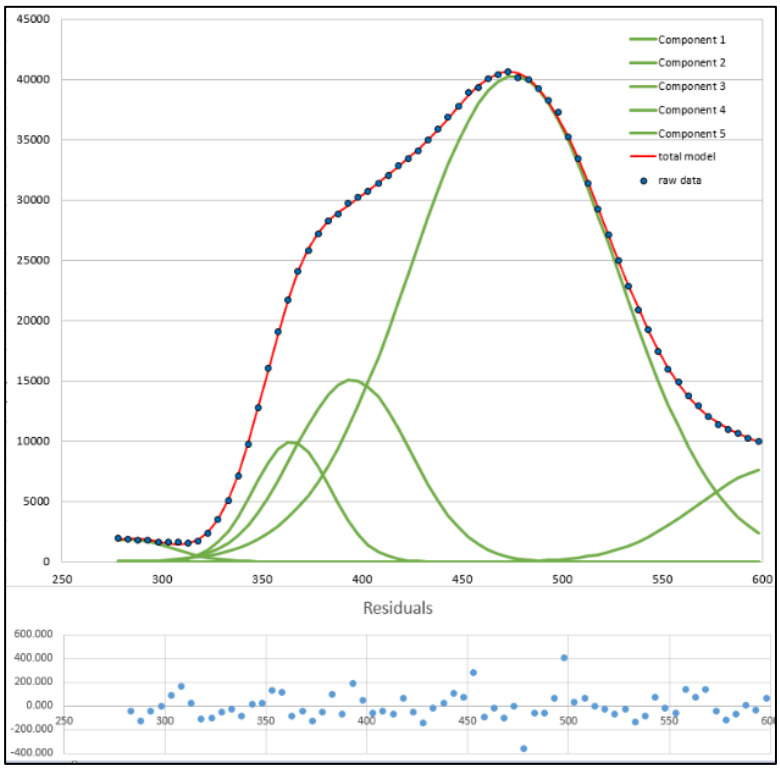
It is important to appreciate that the standard deviations calculated from a small set of measurements can be much higher or much lower than the actual standard deviate on of a larger number of

measurements. For example, the Matlab/Octave function `randn(1,n)`, where *n* is an integer, returns *n* random numbers that have *on average* a mean of zero and a standard deviation of 1.00 if *n* is large. (In Python, the random function is **`np.random.rand(n)`**). But if *n* is small, the standard deviations will be different each time you evaluate that function. For example, if n=5, the standard deviation **`std(randn(1,5))`** might vary randomly from 0.5 to 2 or even more. This is the Law of Large Numbers (page 361). It is the unavoidable nature of small sets of random numbers that their standard deviation is only a *very rough approximation* to the real underlying "population" standard deviation.

A quick but approximate way to estimate the amplitude of noise visually is the *peak-to-peak* range, which is the difference between the highest and the lowest values in a region where the signal is flat. The peak-to-peak range of *n*=100 normally-distributed random numbers is about 5 times the standard deviation, as can be proved by running this line of Matlab/Octave code several times: **`n=100; rn=randn(1,n); (max(rn)-min(rn))/std(rn).`** For example, the data on the right half of the figure page 30 has a peak in the center with a height of about 1.3. The peak-to-peak noise on the baseline is also about 1.0, so the standard deviation of the noise is about 1/5th of that, or 0.2. *However, that ratio varies with the logarithm of n* and is closer to 3 when *n* = 10 and to 9 when *n* = 100000. In contrast, the standard deviation becomes closer and closer to the true value as *n* increases. It is better to compute the standard deviation if possible.

In addition to the *standard* deviation, it is also possible (but not usual) to measure the *mean absolute* deviation ("mad"). The standard deviation is larger than the mean absolute deviation because the standard deviation weights the large deviation more heavily. For a normally-distributed random variable, the mean absolute deviation is on average 80% of the standard deviation: mad=0.8*std.

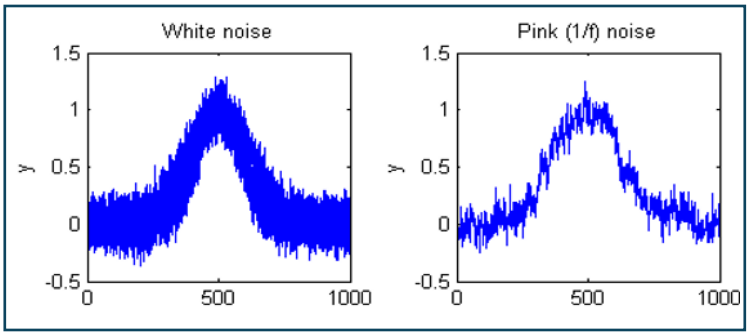
The *quality* of a signal is often expressed quantitatively as the *signal-to-noise ratio* (S/N ratio or SNR), which is the ratio of the true underlying signal amplitude (e.g., the average amplitude or the peak height) to the standard deviation of the noise. Thus, the S/N ratio of the spectrum in the figure on page 14 is about 0.08/0.001 = 80, and the signal on page 30 has an S/N ratio of 1.0/0.2 = 5. So, you would say that the quality of the first one is better because it has a greater S/N ratio. Measuring the S/N ratio is much easier if the noise can be measured separately, in the absence of a signal. Depending on the type of experiment, it may be possible to acquire readings of the noise alone, for example on a segment of the baseline before or after the occurrence of the signal. However, if the magnitude of the noise depends on the level of the signal, then the experimenter must try to produce a



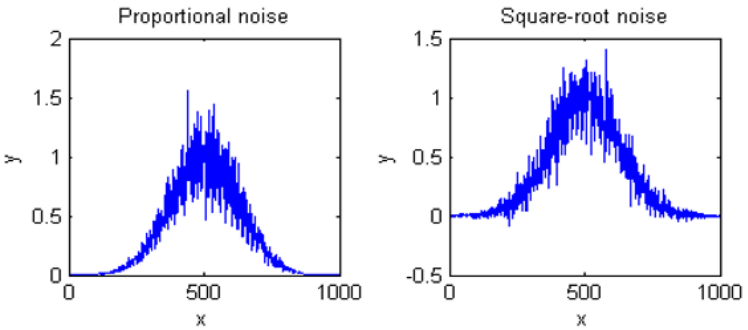
constant signal level to allow measurement of the noise on the signal. In some cases, you can use "iterative curve fitting" (page 199) to model the signal accurately by means of a smooth mathematical function (such as a polynomial or the weighted sum of a number of simple peak shape functions). The noise can then be isolated by subtracting the model from the un-smoothed experimental signal. For example, the graph at the bottom of the previous page shows a *real-data experimental signal* (dark blue dots) whose noise is barely visible and that never goes all the way to the baseline (which would have allowed easy noise measurement). But the signal can be approximated by fitting a model (red line) consisting in this case of 5 overlapping Gaussian peak functions (page 201). The difference between the raw data and the best-fit model, shown in light blue, is the noise in the data, which in this case has a standard deviation of about 100, resulting in a signal-to-noise ratio of 400. Signal-to-noise ratios of scientific signals can sometimes be even higher. The UV detector used in liquid chromatography, can achieve S/N ratios approaching 10000, which allows using signal processing techniques that are very sensitive to noise, such as peak sharpening (page 79) and Fourier deconvolution (page 114).

Frequency distribution of random noise

Sometimes the signal and the noise can be partly distinguished based on frequency components: for example, the signal may contain mostly low-frequency components and the noise may be located at higher frequencies or spread out over a much wider frequency range. This is the basis of filtering and smoothing (page 40). In the figures above, the peak itself contains mostly low-frequency components, whereas the noise is (apparently) random and distributed over a much wider frequency range. The frequency of noise is characterized by its frequency spectrum, often described in terms of noise *color*. *White noise* is random and has equal power over the range of frequencies. It derives its name from *white light*, which has equal brightness at all wavelengths in the visible region. The noise in the previous example signals and in the left half of the figure on the right is *white*. In the acoustical domain, white noise sounds like a *hiss*. In measurement science, white noise is very common. For example, quantization noise, Johnson-Nyquist (thermal) noise, photon noise, and the noise made by single-point spikes all have white frequency distributions, and all have in common their origin in discrete quantized instantaneous events, such as the flow of individual electrons or photons.



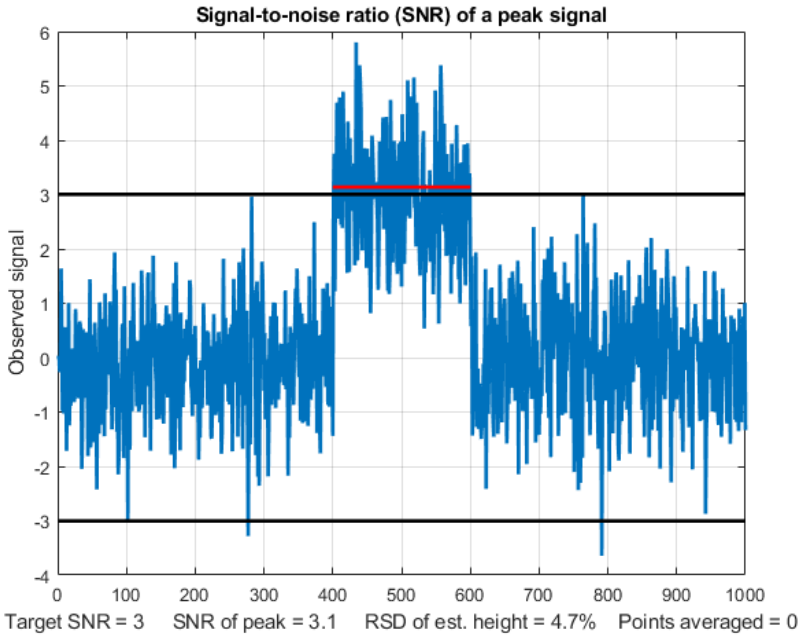
A noise that has a more low-frequency character, that is, that has more power at low frequencies than at high frequencies, is often called "pink noise". In the acoustical domain, pink noise sounds more like a *roar*. (A commonly-encountered sub-species of pink noise is "1/f noise", where the noise power is inversely proportional to frequency, illustrated in the upper right quadrant of the figure on the right). Pink noise is more



troublesome that white noise because a *given standard deviation of pink noise has a greater effect on the accuracy of most measurements than the same standard deviation of white noise* (as demonstrated by the Matlab/Octave function noisetest.m, which generated the figure on the right). Moreover, the application of smoothing and low-pass filtering (page 40) to reduce noise is more effective for white noise than for pink noise. When pink noise is present, it is sometimes beneficial to apply modulation techniques, for example, optical chopping or wavelength modulation in optical measurements, to convert direct-current (DC) signals into alternating current (AC) signals, thereby increasing the frequency of the signal to a frequency region where the noise is lower. In such cases, it is common to use a lock-in amplifier, or the digital equivalent thereof, to measure the amplitude of the signal. Another type of low-frequency weighted noise is *Brownian* noise, named after the botanist Robert Brown. It is also called "red noise" (by analogy to pink noise) or "random walk", which has a noise power that is inversely proportional to the *square* of frequency. This type of noise can occur in experimental signals and can seriously interfere with accurate signal measurements. See page 318: *Random walks and baseline correction*.

Conversely, noise that has more power at *high* frequencies is called “blue” noise. This type of noise is less commonly encountered in experimental work, but it can occur in *processed* signals that have been subject to some sort of differentiation process (page 63) or that have been deconvoluted from some blurring or broadening process (page 114). Blue noise is *easier* to reduce by smoothing (page 27), and it has less effect on least-squares fits than the equivalent amount of white noise.

Detection limit

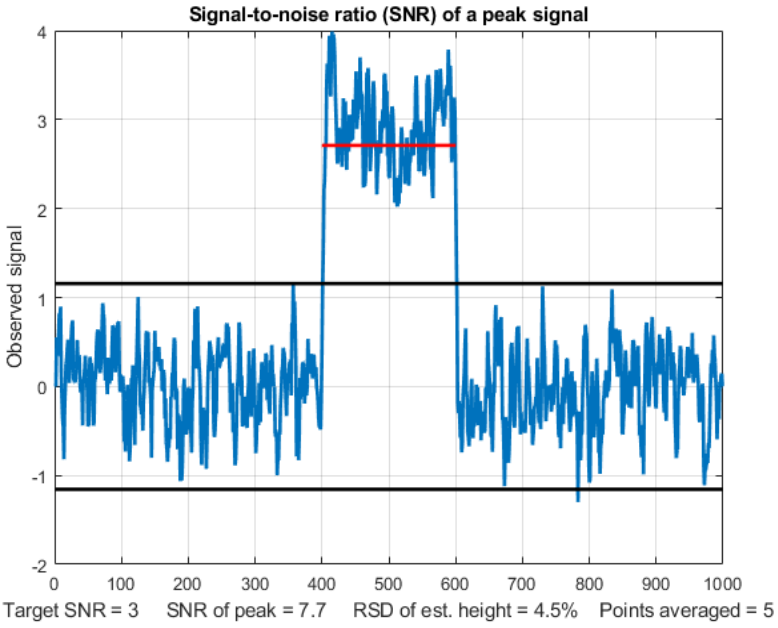


The "detection limit" is defined as the smallest signal that you can reliably detect in the presence of noise. In quantitative chemical analysis, it is usually defined as the concentration that produces the smallest detectable signal (Reference 92). A signal that is below the detection limit cannot be reliably detected. That is, if the measurement is repeated, the signal will often be "lost in the noise" and reported as zero. A signal above the detection limit will be reliably detected and will seldom or never be reported as zero. The most common value of signal-to-noise ratio for reliable detection is 3. This is illustrated in the figure on the left (created by the Matlab/ Octave script SNRdemo.m). This figure shows a noisy signal in the form of a rectangular pulse. I define the "signal" as the average signal magnitude during the pulse, indicated by the **red line**, which is about 3. I define the "noise" as the standard deviation of the random noise on the baseline before and after the pulse, which is about 1.0, roughly 1/5 of the peak-to-peak baseline noise (black lines). Therefore, the signal-

to-noise ratio (SNR) in this case is about 3, which is a common definition of SNR at the detection limit. This means that signals lower than this should be reported as "undetectable".

But there is a problem. The signal here is clearly detectable by eye. In fact, it should be possible to visually detect *lower* signals than this. How can this be? The answer is "averaging". When you look at this signal, you are *unconsciously estimating the average of the data points* on the signal pulse and on the baseline, and your detection ability is enhanced by this visual averaging. Without that averaging, looking only at *individual* data points in the signal, only about half those individual points would meet the SNR=3 criterion. You can see in the graphic above that several points on the signal peak are *lower* than some of the higher data points on the baseline. But this is not a problem in practice, because any properly written software will include averaging that duplicates the visual averaging that we all do.

In the script SNRdemo.m, the number of points averaged is controlled by the variable "AveragePoints" in line 7. If you set that to 5, the result (shown below on the left) shows that all the signal points (each of which is now the average of 5 raw data points) are above the highest baseline points. This graphic more closely represents *how you judge* a signal like that in the previous graphic, which has a clear separation of signal and baseline. The SNR of the peak has improved from 3.1 to 7.7 and *the detection limit will be correspondingly reduced*. As a rule of thumb, for the most common type of random noise, the noise decreases by roughly the square root of the number of points averaged (in this case, sqrt(5)=2.2). Higher values will further improve the SNR and reduce the relative standard deviation of the average signal, but the *response time* – which is the time it takes for the signal to reach the average value - will become slower and slower as the number of points averaged increases. This is shown by another graphic, with 100 points averaged. With a much lower signal equal to 1.0, the raw signal is not reliably detectable visually, but with a 100 point average, the signal precision is good. Digital averaging beats visual averaging in this case. Similar behavior would be observed if the signal were a rounded peak rather than a rectangle.



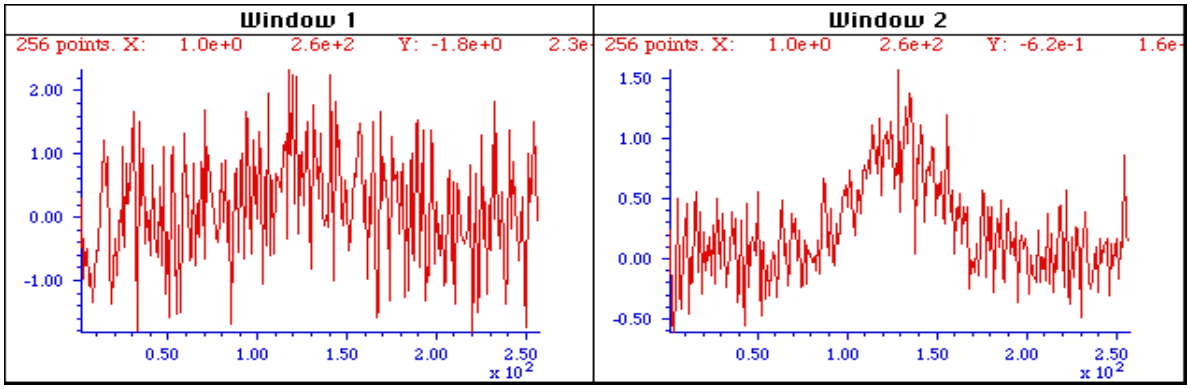
In SNRdemo.m, the noise is constant and independent of the signal amplitude, which is often the case. In the variant SNRdemoHetero.m, the noise in the signal is directly proportional to the signal level or to its square root, and as a result the detection limit depends on the constant baseline noise (graphic). See page 28. In the variant SNRdemoArea.m, it is the peak *area* that is measured rather than the peak height, which results in the SNR being improved by the square root of the width of the peak (graphic).

An example of a practical application of a signal like that illustrated in the figures above would be to

turn on a warning light or buzzer if the signal ever exceeds a threshold value of 1.5. This would not work if you used the *raw unaveraged* signal on the previous page. There is no threshold value that would never be exceeded by the baseline but always exceeded by the signal. Only the *averaged* signal would reliably turn on the alarm above the threshold of 1.5 and never activate it below 1.5.
You will also hear the term "Limit of determination", which is the lowest signal or concentration that achieves a minimum acceptable precision, defined as the relative standard deviation of the signal amplitude. The limit of determination is defined at a much higher signal-to-noise ratio, say 10 or 20, depending on the requirements of your applications. Averaging such as done here is the simplest form of "smoothing", which is covered in the next chapter (page 42).

Ensemble averaging

One key thing that really distinguishes signal from noise is that random noise is not the same from one measurement of the signal to the next, whereas the genuine signal is (ideally) reproducible. So, if the signal can be measured more than once, use can be made of this fact by measuring the signal repeatedly, as fast as is practical, and *adding up* all the measurements point-by-point, then dividing by the number of signals averaged. This is called *ensemble averaging*, and it is one of the most powerful methods for improving signals, when it can be applied. For this to work properly, the noise must be random, and the signal must occur at the same time in each repeat. Look at the example in this figure.



Window 1 (left) is a single measurement of a very noisy signal. There is a broad peak near the center of this signal, but it is difficult to measure its position, width, and height accurately because the S/N ratio is very poor. Window 2 (right) is the average of 9 repeated measurements of this signal, clearly showing the peak emerging from the noise. The expected improvement in S/N ratio is 3 (the square root of 9). Often it is possible to average hundreds of measurements, resulting in much more substantial improvement. The S/N ratio in the resulting average signal in this example is about 5.

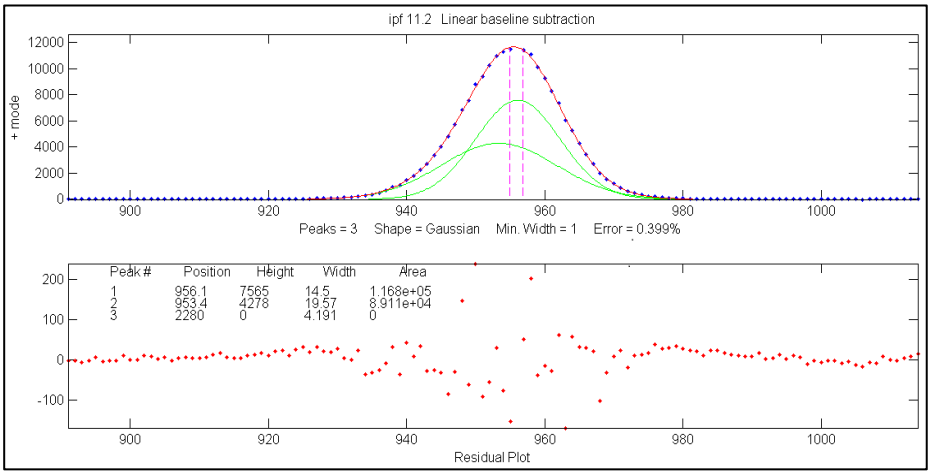
The Matlab/Octave script EnsembleAverageDemo.m demonstrates the technique graphically for an ensemble of 500 signals. (If you are reading this online, click for graphic). Other examples are shown in the video animation at these links, EnsembleAverage1.wmv or EnsembleAverageDemo.gif, which shows the ensemble averaging of 1000 repeats of a signal, improving the S/N ratio by about 30 times. You can also reduce *digitization noise* by ensemble averaging, *but only if small amounts of random noise are present in, or added to, the signal*. See page 308.

Visual animation of ensemble averaging. This crude 17-second video (EnsembleAverage1.wmv) demonstrates the ensemble averaging of 1000 repeats of a signal with a very poor S/N ratio. The signal itself consists of three peaks located at x = 50, 100, and 150, with peak heights 1, 2, and 3 units. These

signal peaks are buried in random noise whose standard deviation is 10. Thus, the S/N ratio of the smallest peaks is 0.1, which is far too low to even *see* a signal, much less measure it. The video shows the accumulating average signal as 1000 measurements of the signal are performed. At the end of the run, the noise is reduced (on average) by the square root of 1000 (about 32), so that the S/N ratio of the smallest peaks ends up being about 3, just enough to detect the presence of a peak reliably. If you are reading this online, click here to download a brief video (2 MByte) in WMV format.

Dependence on signal amplitude

Noise can also be characterized by the way it varies with the signal amplitude. Constant "background" noise is independent of the signal amplitude. Or the noise may increase with signal amplitude, which is a behavior that is often observed in emission spectroscopy, mass spectroscopy and in the frequency spectra of signals. The technical names for these two types of behaviors are *homoscedastic* and *heteroscedastic*, respectively. One way to observe this is to select a segment of signal over which the signal amplitude varies widely, fit the signal to a polynomial or to a multiple peak model (page 204), and observe how the residuals vary with signal amplitude. The experimental signal shown on the left, in the top panel, shows little visible noise. The difference between that signal (the dark blue dots) and the best-fit model (the red line) from a least-squares curve-fitting operation (page 199) is shown in the bottom panel, leaving the noise easily visible (red dots). Clearly, the noise *increases* with signal amplitude in this case. In other cases, the noise might increase with the square root of the signal, or it might be independent of the signal amplitude as in the example on page 26.



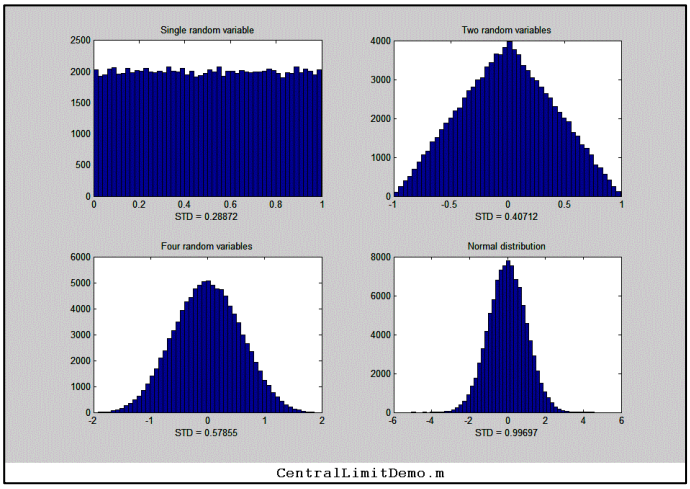
Often, there is a mix of noises with different behaviors. In optical spectroscopy, three fundamental types of noise are recognized, based on their origin and on how they vary with light intensity: *photon noise*, *detector noise*, and *flicker (fluctuation) noise*. Photon noise (often the limiting noise in instruments that use photomultiplier detectors) is *white* and is proportional to the *square root* of light intensity. Detector noise (often the limiting noise in instruments that use solid-state photodiode detectors) is *independent* of the light intensity and therefore the detector SNR is directly proportional to the light intensity. Flicker noise, caused by light source instability, vibration, sample cell positioning errors, sample turbulence, light scattering by suspended particles, dust, bubbles, etc., is directly proportional to the light intensity (and is usually *pink* rather than *white*), so the flicker S/N ratio is not decreased by increasing the light intensity. In practice, the total noise observed is likely to be some contribution of all three types of amplitude dependence, as well as a mixture of white and pink noises.

Only in a very few special cases is it possible to eliminate noise completely, so usually, you must be satisfied by increasing the S/N ratio as much as possible. The key in any experimental system is to understand the possible sources of noise, break down the system into its parts and measure the noise

generated by each part separately, then seek to reduce or compensate for as much of each noise source as possible. For example, in optical spectroscopy, source flicker noise can often be reduced or eliminated by using in feedback stabilization, choosing a better light source, using an internal standard, or specialized instrument designs such as double-beam, dual-wavelength, derivative, and wavelength modulation (page 320). The effect of photon noise and detector noise can be reduced by increasing the light intensity at the detector, and electronic noise can sometimes be reduced by cooling or upgrading the detector and/or electronics. Fixed *pattern noise* in array detectors can be corrected in software. *Photon noise* can be predicted from first principles, as is done in these spreadsheets that simulate the photon noise limited signal-to-noise behavior of ultraviolet-visible spectrophotometry, fluorescence spectroscopy, and atomic emission spectroscopy.

The probability-distribution of random noise

Another property that distinguishes random noise is its *probability distribution*, the function that describes the probability of a random variable falling within a certain range of values. In physical measurements, the most common distribution is called a *normal curve* (also called as a "bell" or "haystack" curve) and is described by a *Gaussian* function, y=e^(-(x-*mu*)^2 / (2**sigma*^2)) / (sqrt(2**mu*)**sigma*), where *mu* is the mean (average) value and *sigma* (σ) is the standard deviation. In this distribution, the most common noise errors are small (that is, close to the *mean*) and the errors become less common the greater their deviation from the mean. So why is this distribution so common? The noise observed in physical measurements is often the balanced sum of many unobserved random events, each of which has some unknown probability distribution related to, for example, the kinetic properties of gases or liquids or to the quantum mechanical behavior of fundamental particles such as photons or electrons. But when many such events combine to form the overall variability of an observed quantity, the resulting probability distribution is almost always *normal*, that is, described by a Gaussian function. This common observation is summed up in the *Central Limit Theorem*.



A simulation can demonstrate how this behavior arises naturally. In the example on the left, you start with a set of 100,000 *uniformly distributed* random numbers that have an equal chance of having any value between certain limits - between 0 and +1 in this case (like the "rand" function in most spreadsheets and in Matlab/ Octave). The graph in the upper left of the figure shows the probability distribution, called a "histogram", of that random variable, which divides the 100,000 numbers into 50 equally-spaced "bins" of about 2000 numbers each. Next, you combine two such sets of independent, uniformly distributed random variables (subtracting them so that the average is centered at zero). The result (shown in the graph in the upper right in the figure) has a *triangular* distribution between -1 and +1, with the highest point at zero. That's because there are many ways for the difference between two random numbers to be small, but only *one* way for the difference to be 1 or to -1 (that happens only if one number is exactly zero *and* the other is exactly

1or -1). Next, if you combine *four* independent random variables (lower left),the resulting distribution now has a total range of -2 to +2, but it is even *less* likely that the result be near 2 or -2 and many *more* ways for the result to be small, so the distribution is narrower and more rounded, and is already starting to look like a normal Gaussian distribution (generated by using the "randn" function and shown for reference in the lower right). If you combine ever more independent uniform random variables, the combined probability distribution becomes closer and closer to the Gaussian, shown for comparison in the bottom right. The important point is that *the emerging Gaussian distribution that you observe here is not forced by prior assumption; rather, it arises naturally*. (You can download a Matlab script for this simulation from http://terpconnect.umd.edu/~toh/spectrum/CentralLimitDemo.m).

Remarkably, *the distribution of individual events hardly matters at all*. You could modify the individual distributions in this simulation by substituting the *rand* function by modified versions such as sqrt(rand), sin(rand), rand^2, log(rand), etc., to obtain other *radically non-normal* individual distributions. But it seems that no matter what the distribution of the single random variable might be, by the time you combine even as few as four of them, the resulting distribution is already visually close to normal. Real-world *macroscopic* observations in science are usually the result of *millions* or *billions* of individual *microscopic* events, so whatever the probability distributions of the *individual* events, the *combined* macroscopic observations often approach a normal distribution nearly perfectly. It is on this common adherence to normal distributions that the common statistical procedures are based. The use of the mean, standard deviation σ, least-squares fits, confidence intervals, etc., are all based on the *assumption* of a normal distribution. But it's usually a very good assumption.

Even so, experimental errors and noise are not *always* normal. Sometimes there are very large errors that fall well beyond the "normal" range. They are called "outliers", and they can have a very large effect on the standard deviation. In such cases, you can calculate the "interquartile range" (IQR), defined as the difference between the upper and lower quartiles (i.e. `prctile(n,75)-prctile(n,25)`), instead of the standard deviation, because *the interquartile range is not affected by a few outliers*. For a *normal* distribution, the interquartile range is equal to 1.34896 times the standard deviation. A quick way to check the distribution of a large set of random numbers is to compute both the standard deviation and the interquartile range. If their ratio is roughly 1.35, the distribution is probably normal. If the standard deviation is *much* larger, the data set probably contains outliers and the standard deviation *without* the outliers can be better estimated by dividing the interquartile range by 1.34896.

The importance of the normal distribution is that if you know the standard deviation of some measured value, then you can predict the likelihood that your measurement might be in error by a certain amount. About 68% of values drawn from a normal distribution are within one σ away from the mean. 95% of the values lie within 2σ, and 99.7% are within 3σ. This is known as the 3-sigma rule. But the real practical problem is this: *standard deviations are hard to measure accurately unless you have large numbers of samples.* See "*The Law of Large Numbers"* (page 361). Note: the standard deviation is usually given the symbol "σ", called "sigma".

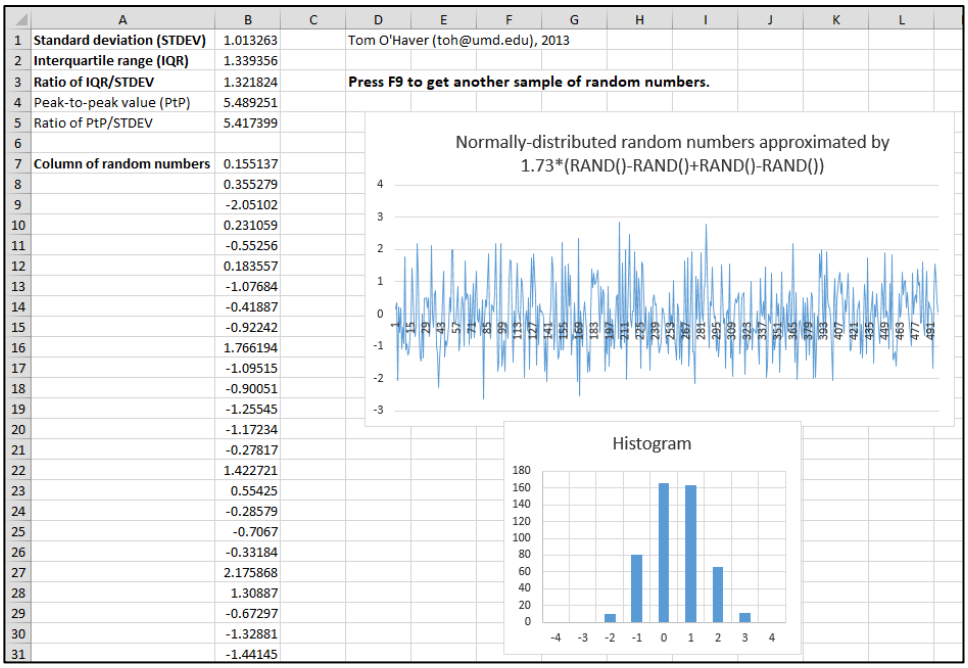
The three characteristics of noise discussed in the paragraphs above - the frequency distribution, the amplitude distribution, and the signal dependence - are mutually independent. Noise may in principle have any combination of those properties.

Representing random noise in Spreadsheets

Popular spreadsheets**,** such as *Excel* or *Open Office Calc*, have built-in functions that can be used for calculating, measuring and plotting signals and noise. For example, the cell formula for one point on a **Gaussian** peak is **`amplitude*EXP(-1*((x-position)/(0.60056120439323*width))^2)`**, where 'amplitude' is the maximum peak height, 'position' is the location of the maximum on the x-axis, 'width' is the full width at half-maximum (FWHM) of the peak (which is equal to *sigma* times 2.355), and 'x' is the value of the independent variable at that point. The cell formula for a **Lorentzian** peak is **`amplitude/(1+((x-position)/(0.5*width))^2)`**. Other useful functions include AVERAGE, MAX, MIN, STDEV, VAR, RAND, and QUARTILE. Most spreadsheets have only a *uniformly-distributed* random number function (RAND) and not a *normally-distributed* random number function, but it is much more realistic to simulate errors that are normally-distributed. But don't worry, you can use the Central Limit Theorem to create approximately normally-distributed random numbers by combining several RAND functions, For example, the odd-looking expression **SQRT(3)*(RAND()-RAND()+RAND()-RAND())** creates nearly normal random numbers with a mean of zero, a standard deviation very close to 1, and a maximum range of ±4. I use this trick in spreadsheet models that simulate the operation of analytical instruments. (The expression **`SQRT(2)*(RAND()-RAND()+RAND()-RAND()+RAND()-RAND())`** works similarly but has a larger maximum range). To create random numbers with a standard deviation other than 1, simply *multiply* by that number. To create random numbers with an average other than zero, simply *add* that number.

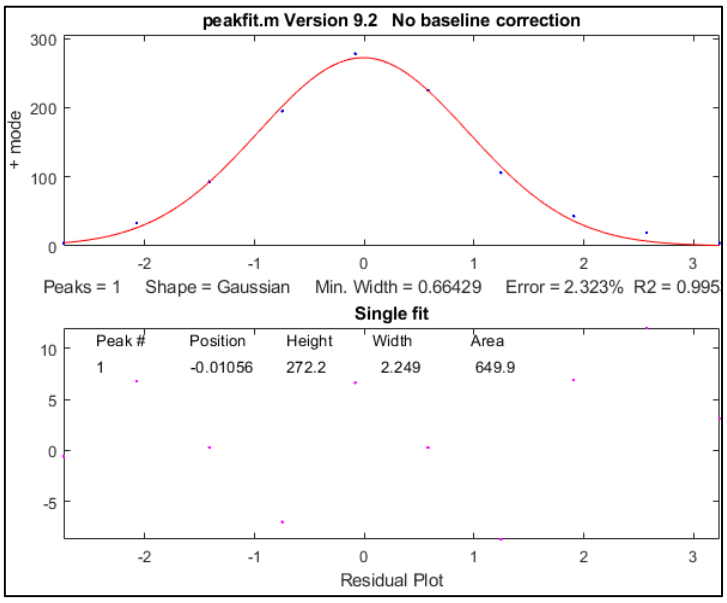
The *interquartile range* (IQR) can be calculated in a spreadsheet by subtracting the third quartile from the first (e.g., **QUARTILE(B7: B504,3) - QUARTILE(B7: B504,1)**).

The spreadsheets RandomNumbers.xls, for Excel, and RandomNumbers.ods, for OpenOffice, (screen image below), and the Matlab/Octave script RANDtoRANDN.m, all demonstrate these facts. The same technique is used in the spreadsheet SimulatedSignal6Gaussian.xlsx, which computes and plots a simulated signal consisting of up to 6 overlapping Gaussian bands plus random white noise.



Random functions in Matlab and Python

Matlab and **Octave** have built-in functions that can be used for calculating, measuring and plotting signals and noise, including mean, max, min, std, kurtosis, skewness, plot, histogram, rand, and randn. Just type "help" and the function name at the command prompt, e.g., "help mean". *Most of these functions apply to vectors and matrices as well as scalar variables*. For example, if you have a series of results in a vector variable 'y', `mean(y)` returns the average and `std(y)` returns the standard deviation of all the values in `y`. For vectors, `std` computes `sqrt(mean(y.^2))`. You can subtract a scalar number from a vector (for example, **`v = v-min(v)`** sets the lowest value of vector **v** to zero). If you have a set of signals in the rows of a matrix **S**, where each column represents the value of each signal at the same value of the independent variable (e.g., time), you can compute the ensemble average of those signals just by typing "`mean(S)`", which computes the mean of each column of **S**. Note that function and variable names are case-sensitive. (You can open the code for any function by selecting its name and selecting "open..").



The "randn" function in Matlab/Octave generates normally-distributed random numbers with a mean of zero and a standard deviation of 1: e.g., randn(1,100) returns a vector of 100 such numbers. (In Python, after importing "numpy" as "np", the syntax is similar: **`np.random.randn(100)`**).

In the following example, "rand" is used to generate 100 random numbers, then Matlab's "histogram" function computes the *histogram* (probability distribution) of those random numbers, then my **peakfit.m** function (page 396, download link) fits a Gaussian function (plotted with a red line) to that distribution.

```
[N,X]=histogram(randn(size(1:100)));
peakfit([X;N]);
```

If you change the 100 here to 1000 or to an even higher number, the distribution of those numbers becomes *closer and closer* to a perfect Gaussian and its peak falls closer to 0.00. Here is an MP4 animation that demonstrates the gradual emergence of a Gaussian normal distribution as the number of "randn" samples increases from 2 to 1000. Note how many samples it takes before the normal distribution is well-formed. The "randn" function is useful in signal processing for predicting the uncertainty of measurements in the presence of random noise, for example by using the Monte Carlo or the bootstrap methods that will be described in a later section (pages 169, 170). (Note: In the PDF version of this book, you can select, copy, and paste, or select, drag, and drop, any of the single-line or multi-line code examples into the Matlab or Octave editor or directly into the command line and press **Enter** to execute it immediately.

The difference between scripts and functions

If you find that you are writing the same series of commands repeatedly, consider writing a *script* or a *function* that will save your code to the computer so you can use it again easily without the danger of typographical errors or clumsy copying and pasting. It is extremely handy to create your own user-defined scripts and functions in Matlab or Python to automate commonly used algorithms.

In Matlab, scripts and functions are just simple text files saved with the ".m" file. The difference between a script and a function is that a function definition begins with the word 'function'. A script is just any list of Matlab commands and statements. For a *script*, all the variables defined and used are listed in the workspace window and shared with other scripts. For a *function*, on the other hand, the variables are *internal and private to that function*. Values can be passed *to* the function through the *input* variables (called "*arguments"*), and values can be passed *from* the function through the *output* variables, which are both defined in the first line of the function definition.

```
[output variables] = FunctionName(input variables)
```

That means that functions are a great way to package chunks of code that perform useful operations in a form that can be used as components in *other* scripts and functions *without worrying that the internal variable names within the function will conflict and cause errors*. When you write a function, you can save it to the computer, and it can be used just like the built-in functions that came with Matlab. Or you can upload it to your Matlab account, where it can be used on a tablet or smartphone. Here is a very simple example: a function that calculates the relative standard deviation of a vector x, rsd.m:

```
function relstddev=rsd(x)
   relstddev=std(x)./mean(x);
```

In Python, the same function would be coded like so:

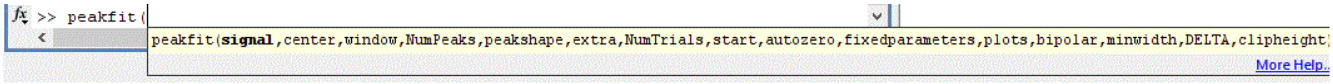
```
def rsd(x):
    return np.std(x)/np.mean(x)
```

(The indentation is critical in Python. It's not required in Matlab but is nevertheless recommended for clarity when reading code). Scripts and functions can call other functions. In older versions of Matlab, scripts must have those functions stored in the Matlab search path. Functions, on the other hand, *can have their required sub-functions defined within the main function itself and thus can be self-contained*. If you write a script or function that calls one or more of your custom functions, and you send it to someone else, be sure to include all the custom functions that it calls. (It is best to make all your functions *self-contained* with all required sub-functions included). If one of my scripts gives an error message that says, "`Undefined function...`", you need to download the specified function from http://tinyurl.com/cey8rwh and place it in the Matlab/Octave search path. **Note**: in Matlab R2016b or later, you CAN include functions within scripts. Just place them at the end of the script and add an additional "end" statement to each function added. See https://www.mathworks.com/help/matlab/matlab_prog/local-functions-in-scripts.html.

To get an explanation of a function, type "help FunctionName" at the command prompt, where FunctionName is the name of the function, or, in Python, "help(FunctionName)". For writing or editing scripts and functions, Matlab, the latest version of Octave, and Python/Spyder all have internal editors. When you are writing your own functions or scripts, you should always add lots of "comment lines",

beginning with the character **%** (or # in Python) that explains what is going on. *You will be glad you did later*. The first group of comment lines, up to the first blank line that does not begin with a %, are used as the "help file" for that script or function. Typing “help FunctionName” displays those comment lines for that function or script in the command window, just as it does for the built-in functions and scripts. It’s also a great idea to add one or more examples of operations that users can copy and paste into the command line. This will make your scripts and functions much easier to understand and use, both by other people and by yourself in the future. *Resist the temptation to skip this*. As you develop custom functions for your own work, you will be developing a “toolkit” that will become very useful to your co-workers, or even to yourself in the future, *if you use comments liberally*. I say this from personal experience. (I did not always follow my own advice).

Matlab has a very handy helper for functions: when you type a function name into the Matlab editor, if you *pause for a moment* after typing the open parenthesis immediately after the function name, Matlab will display a pop-up listing all the possible input variables as a reminder. *This works even for downloaded functions and for any new functions that you yourself create*! It is especially handy when the function has so many possible input variables that it is hard to remember all of them. The popup *stays on the screen as you type*, highlighting each variable in turn, to remind you where you are:

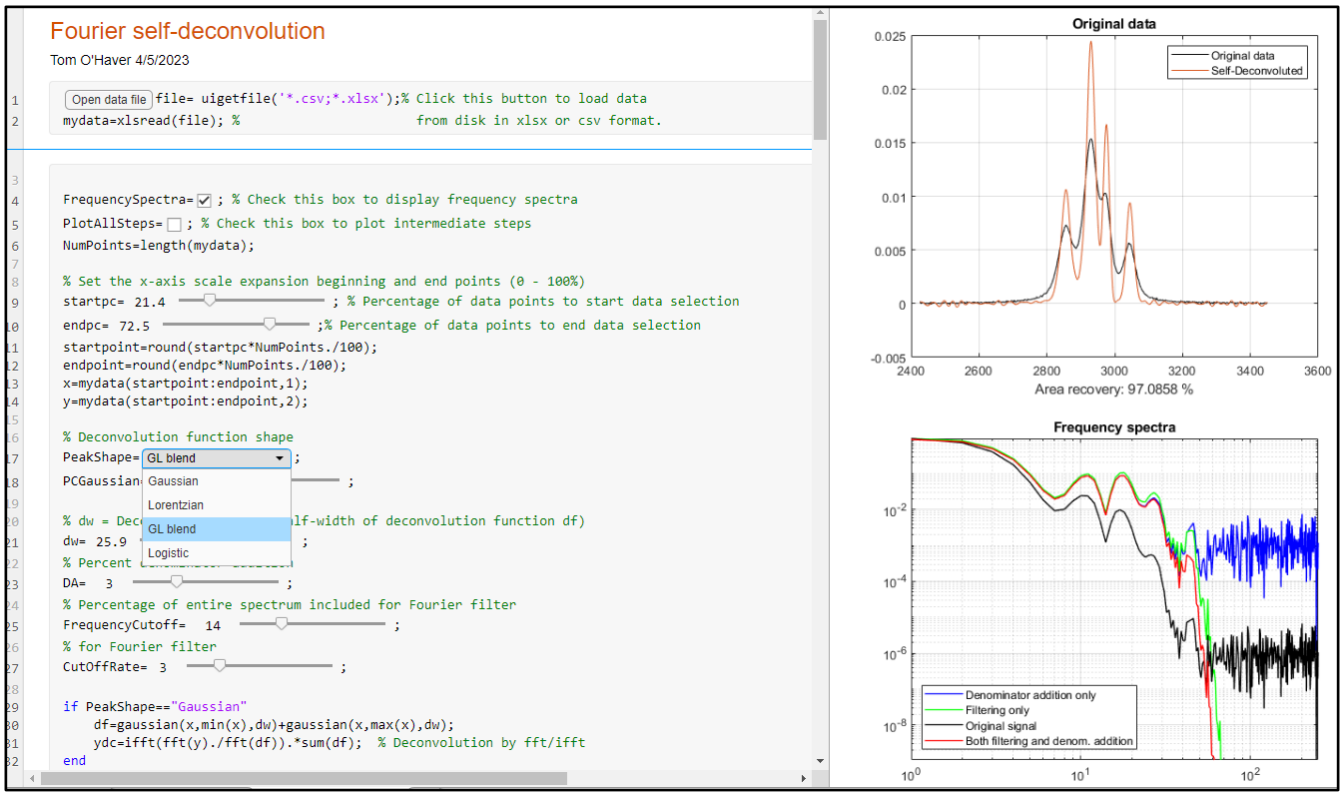


This feature is easily overlooked, but it is very handy. Clicking on the little “More Help...” link on the right displays the help for that function in a separate window. Note: Octave does not have this feature.

Interactive environments: Live scripts

Matlab, Python and Mathematica all have *interactive alternatives* to conventional scripts. Live Scripts in Matlab are interactive documents that combine code, output, formatted text, and interactive controllers in a single environment called the Live Editor. (Live Scripts were available in MATLAB R2016b and later). See page 368. Python has Jupyter Notebooks which are used to create an interactive narrative around your code. Mathematica (page 462) has built-in interactivity using the *manipulate* function. These make it easy to create sharable interactive documents with graphical user interface devices such as pull-down menus, check boxes, and sliders to adjust numerical values interactively. In Matlab, you can simply open a conventional (.m) script in the Live Editor and insert the interface devices directly into the script where the numbers in assignment statements would have gone. When you save it, it becomes an .mlx file.

Live scripts make it especially easy for Matlab users to create custom tools for interactive exploration. An example of a portion of a Live Script is shown on the next page. This example shows four types of interactive controllers. Line 1 shows a button that opens a ***file browser*** that allows you to navigate to a specific file, in this case a data file that you want to process. Lines 5 and 6 show ***checkboxes***, which are used to enable or disable optional sections of code. Several lines show numeric ***sliders***, which are used to control continuous variables. Line 17 shows a ***drop-down menu*** that allows multiple choices, shown pulled down in the screenshot below.



Live Scripts produce graphic output in small windows on the right side of the Live editor window, where you can copy, pan and zoom and export to png files as usual using the mouse. You can also convert any Live Script graphic into a standard figure window (by clicking its upper right corner), which can then be exported to other graphic formats, expanded to full screen, printed, etc.

Other examples of Live Scripts include the versatile **data smoothing tool** shown on page 59, a tool for **differentiation** (page 78), the **self-deconvolution script** shown above (page 126) and a **peak detection tool** (page 253). These Live scripts are surprisingly easy to create within the Matlab environment by modifying a conventional script and a **peak fitting tool** on page 440. See page 368 for more details about developing Matlab Live Scripts and Apps. See page 370 for a table listing of Live Script tools and their corresponding keypress-driven functions.

User-defined functions related to signals and noise.

Here are some examples of user-defined functions that I have created for my signal processing toolkit. These are not built-in functions. You must download them and put them in the Matlab path to use them.

Data plotting: plotit.m, an easy-to-use function for plotting and fitting x,y data in matrices or in separate vectors. For handling *very large* signals more easily, plotxrange.m (**`[xx,yy,irange] = plotxrange(x,y,x1,x2)`**) extracts and plots values of vectors x,y only for x values between specified values of x. Segplot.m (**`[s,xx,yy] = segplot(x,y,NumSegs,seg)`**) divides signals into "NumSegs" equal-length segments and plots segments marked by vertical lines, each labeled with a small segment number at the bottom, and returns a vector 's' of segment indexes and the subset xx,yy, of values in the segment number 'seg'.

Peak shapes. The code examples in this book often uses Gaussian peaks, but you can exchange that for any other shape. Here are links to several Matlab functions for peak shapes commonly encountered in science: Gaussian, Lorentzian, lognormal, Pearson 5, exponentially-broadened Gaussian, exponentially-broadened Lorentzian, exponential pulse, sech2, sigmoid, Gaussian/Lorentzian blend, bifurcated Gaussian, bifurcated Lorentzian), Voigt profile, triangular. See page 479 and the following for a more complete list. The self-contained script Sech2ShapeComparison.m compares Gaussian, Lorentzian, and sech2 pulse shapes, showing the sech2 pulse in intermediate between Gaussian and Lorentzian (graphic).

peakfunction.m, a function that generates any of those peak types specified by number.

ShapeDemo demonstrates the 16 basic peak shapes graphically, showing the variable-shape peaks as multiple lines. (Graphic on page 424)

Noise generators. There are several functions for simulating different types of random noise (white noise, pink noise, blue noise, proportional noise, and square root noise).

Miscellaneous Matlab functions:

stdev.m, computes the standard deviation and rsd.m the *relative* standard deviation.

PercentDifference.m, simply calculates the percent difference between two variables.

IQrange.m computes the interquartile range (explained above).

halfwidth.m for measuring the full width at half maximum of smooth peaks of any shape.

ExpBroaden.m applies exponential broadening to any time-series vector.

rmnan.m removes "not-a-number" entries from vectors, which is useful for cleaning up real data files; rmz.m removes zeros from vectors, replacing with nearest non-zero numbers.

val2ind.m returns the index and the value of the element of vector x that is closest to a particular value. This is a simple function that is more useful than you might imagine. Search this document for “val2ind” to find several examples of the practical use of this function.

These functions are useful in modeling and simulating analytical signals and testing measurement techniques (page 41). In the PDF version of this book, you can click or ctrl-click on these links to inspect the code or you can right-click and select "Save link as..." to download them to your computer.

Once you have downloaded those functions and placed them in the search path, you can use them just like any other built-in function. For example, you can plot a Gaussian peak with white noise by typing **`x=[1:256]; y=gaussian(x,128,64) + whitenoise(x); plot(x,y).`** The script plotting.m, shown in the figure on page 18, uses the gaussian.m function to demonstrate the distinction between the *height*, *position*, and *width* of a Gaussian curve. The script SignalGenerator.m calls several of these downloadable functions to create and plot a realistic computer-generated signal with multiple peaks on a variable baseline plus variable random noise. You might try to modify the variables in the indicated places to make it look like your type of data. All these functions will work in the latest version of Octave without change. For a complete list of my downloadable functions and scripts developed for this project, see page 479 or on the Web at http://tinyurl.com/cey8rwh.

The Matlab/Octave function noisetest.m demonstrates the appearance and effect of different noise types. It plots Gaussian peaks with four different types of added noise: constant white noise, constant pink (1/f) noise, proportional white noise, and square root white noise, then fits a Gaussian to each noisy data set and computes the average and the standard deviation of the peak height, position, width, and area for each noise type. Type "help noisetest" at the command prompt. My Matlab/Octave script SubtractTwoMeasurements.m (page 25) demonstrates the technique of subtracting two separate measurements of a waveform to extract the random noise (but it works only if the signal is stable, except for the noise).

iSignal (page 376) is one of a group of multi-purpose downloadable Matlab modules I have developed that combine many of the techniques covered here. *iSignal* can plot signals with keypress-operated pan and zoom controls, measure signal and noise amplitudes in selected regions of the signal and compute the S/N ratio of peaks. It is operated by simple key presses. Other capabilities of iSignal include smoothing (page 40), differentiation, peak sharpening and de-tailing, deconvolution, least-squares peak measurement, etc.

Others in this group of interactive functions include ***iPeak*****,** page 255, which focuses on peak detection, and ***ipf.m***, page 419, which focuses on iterative curve fitting. These functions are ideal for initial explorations of complex signals because they make it easy to select operations and adjust the controls by simple key presses. These work even if you run Matlab Online in a web browser, but they do not work on Matlab Mobile. Note that the Octave versions, ipfoctave.m, ipeakoctave.m, isignaloctave.m, and ifilteroctave.m, use the < and > keys (with and without shift) for pan and zoom.

For signals that contain repetitive waveform patterns occurring in one continuous signal, with nominally the same shape except for noise, the interactive peak detector function *iPeak* (page 253), has an ensemble averaging function (**Shift-E**) can compute the average of all the repeating waveforms. It works by detecting a single reference peak in each repeat waveform to synchronize the repeats (and therefore does not require that the repeats be equally spaced or synchronized to an external reference signal). To use this function, first adjust the peak detection controls to detect *only one peak in each repeat pattern*, zoom in to isolate any one of those repeat patterns, and then press **Shift-E**. The average waveform is displayed in Figure 2 and saved as “EnsembleAverage.mat” in the current directory. See iPeakEnsembleAverageDemo.m for a demonstration. See page 330: *Measuring the Signal-to-Noise Ratio of Complex Signals* for more examples of the signal-to-ratio in Matlab/Octave computations.

Numerical experiments: the role of simulation and modeling.

A simulation is an imitation of the operation of a real-world process or system over time. Simulations require the use of *models*, which represent the important characteristics or behaviors of the selected system or process, whereas the *simulation* represents the evolution of the model over time. The Wikipedia article on simulation lists 27 widely different areas where simulation and modeling are applied. In the context of scientific measurement, simulations of measurement instruments (page 355) or of signal processing techniques have been widely applied. A simulated signal can be synthesized using mathematical models for signal shapes (page 479) combined with appropriate types of simulated random noise (page 24), both based on the common characteristics of real signals.

It is important to realize that a simulated signal is not a "fake" signal, because it is *not intended to deceive*. Rather, you can use simulated signals to test the accuracy and precision of a proposed processing technique, using simulated data whose true underlying parameters are known (which is not the case for real signals). Moreover, you can test the robustness and reproducibility of a proposed technique by creating multiple signals with the same underlying signal parameters but with imperfections added, such random noise, non-zero and shifting baselines, interfering peaks, shape distortion, etc. For example, the script CreateSimulatedSignal.m shows how to create a realistic model of a multi-peak signal that is based on the measured characteristics of an experimental signal. You will find many applications of this idea, e.g., on pages 310 and 337.

Simulated signals that you create should always have at least small amounts of random noise, even if it is not visible in plots. This is not only because all real signals have noise but also because some processing techniques will yield *misleadingly impressive performance* if applied to purely computer-generated signals that are limited only by the numerical resolution of the computer (page 340). Examples of such techniques are higher-order differentiation (page 72), peak sharpening (page 79), and deconvolution (page 114).

Simulation is also applicable in more sophisticated cases. On page 363, I describe a published commercial technical report that contained a detailed example of a practical application of liquid chromatography with a diode array detector to separate three similar chemical isomers. With that information I was able to create realistic a *data-based* simulation of the data obtained in that experiment, which allowed me to "repeat" the experiment numerically with different amounts of noise and with different chromatographic and spectroscopic parameters, in order to explore the limits of applicability of that method to other potentially more challenging applications.

Smoothing

In many experiments in science, the true signal amplitudes (y-axis values) change rather smoothly as a function of the x-axis values, whereas many kinds of noise are seen as rapid, random changes in amplitude from point to point within the signal. In the latter situation it may be useful in some cases to attempt to reduce the noise by a process called *smoothing*. In smoothing, the data points of a signal are modified so that individual points that are *higher* than the immediately adjacent points (presumably because of noise) are reduced, and points that are *lower* than the adjacent points are increased. This naturally leads to a smoother signal (and a slower step response to signal changes). If the true underlying signal is smooth, then the true signal will not be much distorted by smoothing, but the high-frequency noise will be reduced. In terms of the frequency components of a signal, a smoothing operation acts as a low-pass filter, reducing the high-frequency components and passing the low-frequency components with little change. If the signal and the noise is measured for all frequencies, then the signal-to-noise ratio will be improved by smoothing, by an amount that depends on the frequency distribution of the noise. (Smoothing can be contrasted to *wavelet denoising*, pages 132 and 61, which also reduces noise but does not necessarily make the signal completely smooth).

Smoothing algorithms

The simplest smoothing algorithms are based on the "*shift and multiply*" technique, in which a group of adjacent points in the original data is multiplied point-by-point by a set of numbers (coefficients) that defines the smooth shape, the products are added up and divided by the sum of the coefficients, which becomes one point of smoothed data, then the set of coefficients is shifted one point along the original data and the process is repeated. The simplest smoothing algorithm is the *rectangular boxcar* or *unweighted sliding-average smooth*. It simply replaces each point in the signal with the average of *m* adjacent points, where *m* is a positive integer called the *smooth width*. For example, for a 3-point smooth (*m* = 3):

$$S_j = \frac{Y_{j-1} + Y_j + Y_{j+1}}{3}$$

This is evaluated for j = 2 to *n*-1, where S_j is the jth point in the smoothed signal, Y_j is the jth point in the original signal, and *n* is the total number of points in the signal. Most spreadsheets and programming languages have a “mean” or “average” function which can do this work quickly, so S_j=mean($y_{j-w/2}$:$y_{j+w/2}$). Similar smooth operations can be constructed for any desired smooth width, *m*. Usually *m* is an odd number. If the noise in the data is "white noise" (that is, evenly distributed over all frequencies) and its standard deviation is *D*, then the standard deviation of the noise remaining in the signal after the first pass of an unweighted sliding-average smooth will be approximately D over the square root of *m* (*D*/sqrt(*m*)), where *m* is the smooth width. Despite its simplicity, this smooth is actually *optimum* for the common problem of reducing white noise while keeping the *sharpest step response* (click here for a logical proof). The response to a step change is, in fact, *linear*, so this filter has the advantage of responding completely with no residual effect within its *response time* (which is equal to the smooth width divided by the sampling rate). Smoothing can be performed either *during* data acquisition, by programming the digitizer to measure and to average multiple readings and save

only the average, or *after* data acquisition ("post-run"), by storing all the acquired data in memory and smoothing the stored data. The latter requires more memory but is more flexible.

The *triangular smooth* is like the rectangular smooth, above, except that it implements a *weighted* smoothing function. For a 5-point smooth ($m = 5$):

$$S_j = \frac{Y_{j-2} + 2Y_{j-1} + 3Y_j + 2Y_{j+1} + Y_{j+2}}{9}$$

for j = 3 to n-2, and similarly for other smooth widths (see the spreadsheet UnitGainSmooths.xls). In both of these cases, the integer in the denominator is the *sum of the coefficients* in the numerator, which results in a "unit-gain" smooth that has no effect on the signal where it is a straight line and which preserves the area under peaks.

It is often useful to apply a smoothing operation more than once, that is, to smooth an already smoothed signal, to build longer and more complicated smooths. For example, the 5-point triangular smooth above is equivalent to two passes of a 3-point rectangular smooth. *Three* passes of a 3-point rectangular smooth result in a 7-point *haystack* smooth, also called a "p-spline", for which the coefficients are in the ratio 1:3:6:7:6:3:1. The general rule is that *n* passes of a *w*-width smooth results in a combined smooth width of *n***w*-*n*+1. For example, 3 passes of a 17-point smooth results in a 49-point smooth. These multi-pass smooths are more effective at reducing high-frequency noise in the signal than a rectangular smooth, but they exhibit a slower step response.

In all these smooths, the width of the smooth *m* is chosen to be an odd integer, so that the smooth coefficients are symmetrically balanced around the central point, which is important because it *preserves the x-axis position of peaks and other features* in the smoothed signal. (This is especially critical for analytical and spectroscopic applications because the peak positions are often important measurement objectives.

We are assuming here that the x-axis interval of the signal is uniform, that is, that the difference between the x-axis values of adjacent points is the same throughout the signal. This is also assumed in many of the other signal-processing techniques described in this book, and it is a very common (but not necessary) characteristic of signals that are acquired by automated and computerized equipment.

More advanced algorithms. The *Savitzky-Golay* smooth (ref 97) is based on the least-squares fitting of polynomials to segments of the data. The algorithm is discussed on Wikipedia. Compared to the sliding-average smooths of the same width, the Savitzky-Golay smooth is less effective at reducing noise, but more effective at retaining the shape of the original signal. It is capable of differentiation as well as smoothing. The algorithm is more complex, and the computational times may be greater than the smooth types discussed above, but with modern computers, the difference is insignificant. Code in various languages is widely available online. See page 60. My interactive iSignal function (page 376) has a Savitzky-Golay option. Another advanced algorithm is the wavelet-based *denoise* function (see page 136) which attempts to distinguish signal from noise by analyzing the frequency structure of the signal.

The shape of any smoothing algorithm can be determined by applying that smooth to a *delta function*, a signal consisting of all zeros except for one point, as demonstrated by the simple Matlab/Octave script DeltaTest.m. The result is called the *impulse response function*.

Noise reduction

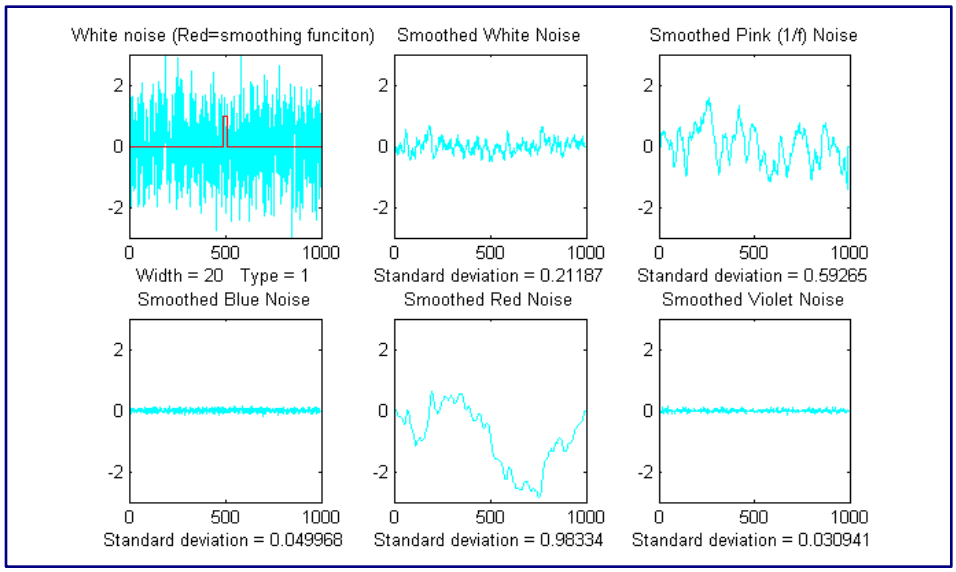
Smoothing usually reduces the noise in a signal. If the noise is "white" (that is, evenly distributed over all frequencies) and its standard deviation is *D*, then the standard deviation of the noise remaining in the signal after one pass of a rectangular smooth will be approximately *D*/sqrt(*m*), where *m* is the smooth width. If a triangular smooth is used instead, the noise will be slightly less, about *D**0.8/sqrt(*m*). Smoothing operations can be applied more than once: that is, a previously smoothed signal can be smoothed again. In some cases, this can be useful if there is a great deal of *high*-frequency noise in the signal. However, the noise reduction for *white* noise is less in each successive smooth. For example, *three* passes of a rectangular smooth reduce white noise by a factor of approximately *D**0.7/sqrt(*m*), only a slight improvement over two passes. For a spreadsheet demonstration, see VariableSmoothNoiseReduction.xlsx.

Effect of the frequency distribution of noise

The frequency distribution of noise, designated by noise "color" (page 24), substantially affects the ability of smoothing to reduce noise. The Matlab/ Octave function "NoiseColorTest.m" compares the effect of a 20-point boxcar (unweighted sliding average) smooth on the standard deviation of white, pink, red, and blue noise, all of which have an original unsmoothed standard deviation of 1.0. Because smoothing is a low-pass filter process, it affects low-frequency (pink and red) noise *less*, and effects high-frequency (blue and violet) noise *more*, than it does white noise.

Original unsmoothed noise	1
Smoothed **white** noise	0.1
Smoothed **pink** noise	0.55
Smoothed **blue** noise	0.01
Smoothed **red** (random walk) noise	0.98

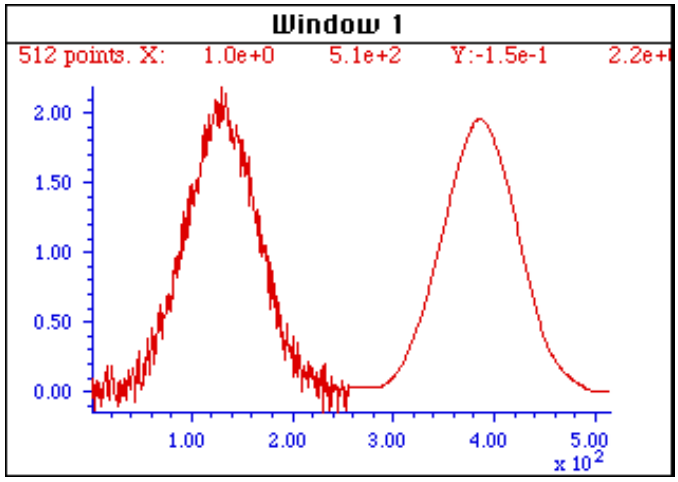
Note that the computation of standard deviation is independent of the order of the data and thus of its frequency distribution. Sorting a set of data does not change its standard deviation. The standard deviation of a sine wave is independent of its frequency. Smoothing, however, changes both the frequency distribution and standard deviation of a data set.



End effects and the lost points problem

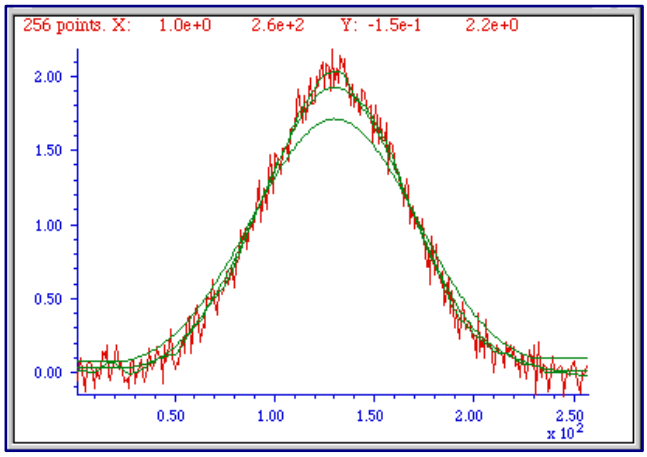
In the equations above, the 3-point rectangular smooth is defined only for j = 2 to n-1. There is not enough data in the signal to define a complete 3-point smooth for the first point in the signal (j = 1) or for the last point (j = n), because there are no data points before the first point or after the last point. (Similarly, a 5-point smooth is defined only for j = 3 to n-2, and therefore a smooth operation cannot be calculated for the first two points or for the last two points). In general, for an *m*-width smooth, there will be (*m*-1)/2 points at the beginning of the signal and (*m*-1)/2 points at the end of the signal for which a complete *m*-width smooth cannot be calculated the usual way. What to do? There are two approaches. One is to accept the loss of points and trim off those points or replace them with zeros in the smooth signal. (That's the approach taken in most of the figures in this book). The other approach is to use *progressively smaller smooths* at the ends of the signal, for example to use smooth widths of 2, 3, 5, 7... points for signal points 1, 2, 3, and 4..., and for points n, n-1, n-2, n-3..., respectively. The latter approach may be preferable if the edges of the signal contain critical information, but it increases execution time. My Matlab/Octave *fastsmooth* function (page 485) can utilize either of these two methods. An alternative approach is to pad the edges with a mirror image of the data itself, which is commonly done in smoothing two-dimensional (image) data.

Examples of smoothing

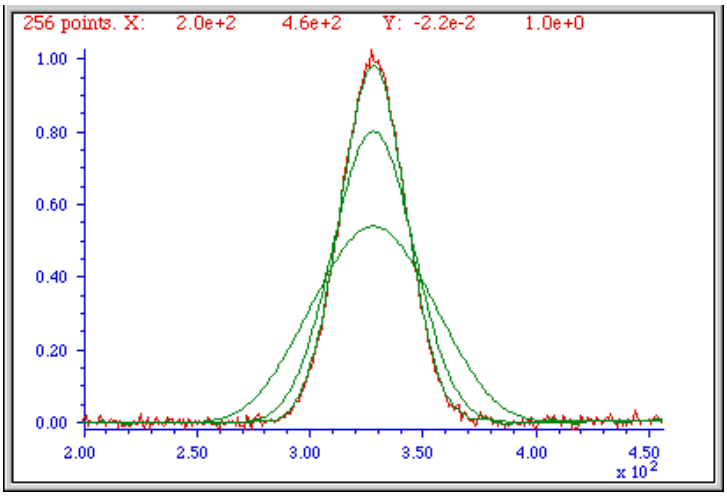


The figure below shows a simple example of smoothing. The left half of this signal is a noisy peak. The right half is the same peak after undergoing a triangular smoothing algorithm. The noise is greatly reduced while the peak itself is hardly changed. The reduced noise allows the signal characteristics (peak position, height, width, area, etc.) to be measured more accurately by visual inspection*, but it does not improve measurements made by least-squares methods.* See below.

The larger the smooth width, the greater the noise reduction, but also the greater the possibility that the signal will be *distorted* by the smoothing operation. The optimum choice of smooth width depends upon the width and shape of the signal and the digitization interval. For peak-type signals, the critical factor is the *smooth ratio*, the ratio between the smooth width *m* and the number of points in the half-width of the peak. In general, increasing the smoothing ratio improves the signal-to-noise ratio but causes a reduction in amplitude and an increase in the width of the peak. Be aware that the smooth width can be expressed in two different ways: (a) as the number of data points or (b) as the x-axis interval (for spectroscopic data usually in nm or in frequency units). The two are simply related: the number of data points is simply the x-axis interval times the increment between adjacent x-axis values. The *smooth ratio* is the same in either case.

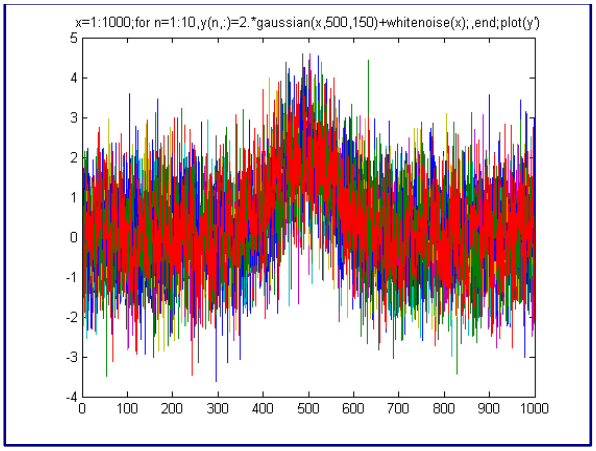


The figures here show examples of the effect of three different smooth widths on noisy Gaussian-shaped peaks. In the figure on the left, the peak has a true height of 2.0 and there are 80 points in the half-width of the peak. The red line is the original unsmoothed peak. The three superimposed green lines are the results of smoothing this peak with a triangular smooth of width (from top to bottom) 7, 25, and 51 points. Because the peak width is 80 points, the *smooth ratios* of these three smooths are 7/80 = 0.09, 25/80 = 0.31, and 51/80 = 0.64, respectively. As the smooth width increases, the noise is progressively reduced but the peak height also is reduced slightly. For the largest smooth, the peak *width* is noticeably increased. In the figure on the right, the original peak (in red) has a true height of 1.0 and a half-width of 33 points. (It is also less noisy than the example above.) The three superimposed green lines are the results of the *same* three triangular smooths of width 7, 25, and 51 points. But because the peak width, in this case, is only 33 points, the *smooth ratios* of these three smooths are *larger* - 0.21, 0.76, and 1.55, respectively. You can see that the peak distortion effect (reduction of peak height and increase in peak width) is greater for the narrower peak because the smooth ratios are higher. Smooth ratios of greater than 1.0 are seldom used because of excessive peak distortion. Note that even in the worst case, the peak positions are not affected (assuming that the original peaks were symmetrical and not overlapped by other peaks). If retaining the shape of the peak is more important than optimizing the signal-to-noise ratio, the Savitzky-Golay has the advantage over sliding-average smooths. In all cases, the total signal remains unchanged. If the peak widths vary substantially, an adaptive or *segmented* smooth, which allows the smooth width to vary across the signal, may be used (page 333). In this context, "segmented" means that the signal is divided into segments with a different smooth applied to each segment.

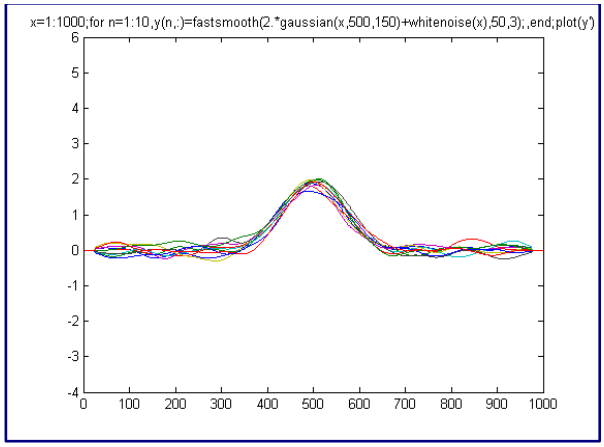


The problem with smoothing

Smoothing *is often less beneficial than you might think*. It is important to understand that smoothing results such as illustrated in the figures above could be viewed as *deceptively impressive* because they employ a *single sample* of a noisy signal that is smoothed to different degrees. This causes the viewer to underestimate the contribution of *low-frequency* noise, which is hard to estimate visually because there are *so few low-frequency cycles* in the signal record. This problem can be visualized by recording a few independent samples of a noisy signal consisting of a single peak, as illustrated in the two figures below.




```
x=1:1000;
for n=1:10,
y(n,:)=2.*gaussian(x,500,150)…
+whitenoise(x);
end
plot(x,y)
```




```
x=1:1000;
for n=1:10,
y(n,:)=2.*gaussian(x,500,150)+whitenoise(
x);
 y(n,:)=fastsmooth(y(n,:),50,3);
end
plot(x,y)
```

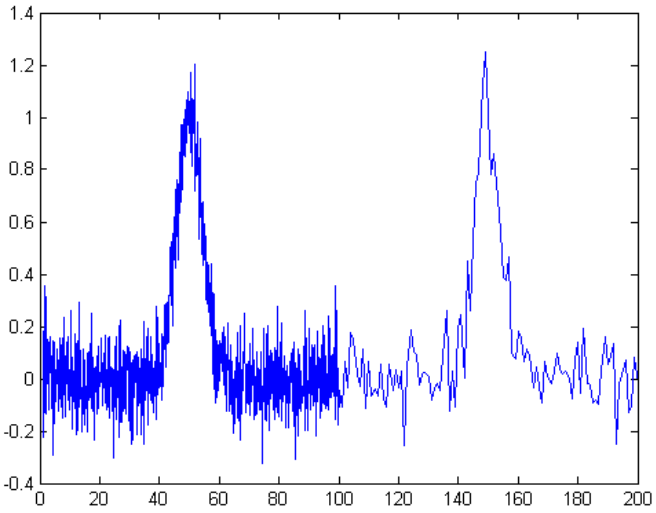
These figures show ten superimposed plots with the same peak but with independent white noise, each plotted with a different line color, unsmoothed on the left and smoothed on the right. Clearly, the noise reduction is substantial, but close inspection of the different colored smoothed signals on the right shows that there is still variation in peak position, height, and width between the 10 samples, which is caused by the low-frequency noise remaining in the smoothed signals. Without the noise, each peak would have a peak height of 2, peak center at 500, and a width of 150. *Just because a signal looks smooth does not mean there is no noise.* Low-frequency noise remaining in the signals after smoothing can still interfere with the precise measurement of peak position, height, and width.

(The generating scripts below each figure require that the functions gaussian.m, whitenoise.m, and fastsmooth.m be downloaded from http://tinyurl.com/cey8rwh.)

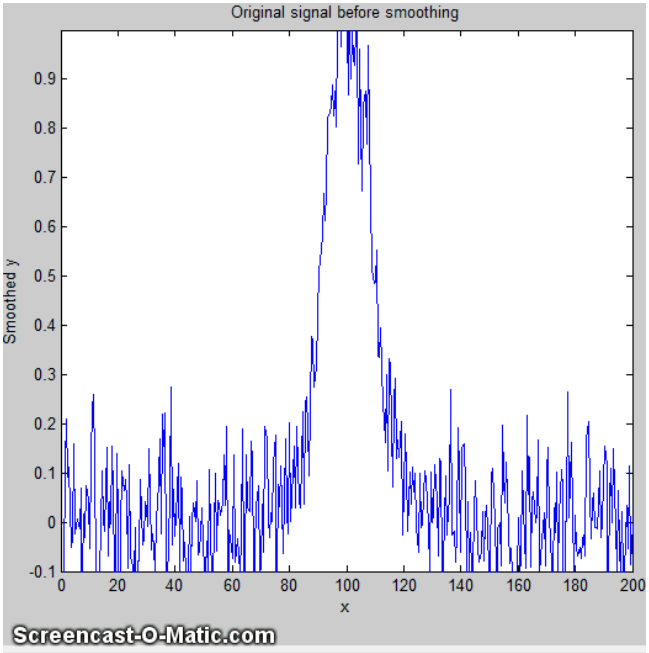
It should be clear that smoothing can seldom eliminate noise *completely*, because most noise is spread out over a range of frequencies and smoothing simply reduces the noise in *part* of its frequency range. Only for some very specific types of noise (e.g., discrete frequency sine-wave noise or single-point spikes) is there hope of anything close to complete noise elimination. Smoothing *does* make the signal smoother, and *it does* reduce the standard deviation of the noise, but whether that makes for a *better measurement* or not depends on the situation. And do not assume that just because a little smoothing is good that more will necessarily be better. Smoothing is like cursing. Sometimes you need it - but you should do it judiciously.

The figure on the right below is another example of a signal that illustrates some of these principles. The signal consists of two Gaussian peaks, one located at x=50 and the second at x=150. Both peaks have a peak height of 1.0 and a peak half-width of 10, and the same normally-distributed random white noise with a standard deviation of 0.1 has been added to the entire signal. The *x-axis sampling interval*,

however, is different for the two peaks: it is 0.1 for the first peak from x=0 to 100) and 1.0 for the second peak (from x=100 to 200). This means that the first peak is characterized by *ten times more data points* than the second peak. It may *look* like the first peak is noisier than the second, but that is just an illusion. The signal-to-noise ratio for both peaks is 10. The second peak looks less noisy only because there are fewer noise samples there and *we tend to underestimate the dispersion of small samples*. The result of this is that when the signal is smoothed, the *second peak* is much more likely to be distorted by the smooth operation, becoming shorter and wider than the first peak. The first peak can tolerate a much wider smooth width, resulting in a greater degree of noise reduction. Similarly, if both peaks are measured with the least-squares curve fitting method (page 199), the fit of the first peak is more stable with the noise and the measured parameters of that peak will be about *3 times more accurate* than the second peak, because there are 10 times more data points in that peak, and the measurement precision improves roughly with the square root of the number of data points if the noise is white. You can download this data file, "udx", in TXT format or in Matlab MAT format.



Optimization of smoothing



As smooth width increases, the smoothing ratio increases, noise is reduced quickly at first, then more slowly, and the peak height is also reduced, slowly at first, then more quickly. The *noise reduction* depends on the smooth width, the smooth type (e.g., rectangular, triangular, etc.), and the noise color, but the *peak height reduction* also depends on the peak width. The result is that the signal-to-noise (defined as the ratio of the peak height of the standard deviation of the noise) increases quickly at first, then reaches a maximum. This is illustrated by the animation on the left, which shows the result of smoothing a *Gaussian peak plus white noise* (produced by this Matlab/Octave script). The maximum improvement in the signal-to-noise ratio depends on the number of points in the peak: the more points in the peak, the greater smooth widths can be employed and the greater the noise reduction. This figure also illustrates that most of the noise reduction is due to *high-frequency* components of the noise, whereas much of the *low*-frequency noise remains in the signal even as it is smoothed.

Which is the best smooth ratio? It depends on the purpose of the peak measurement. If the ultimate objective of the measurement is to measure the peak height or width, then smooth ratios below 0.2 should be used and the *Savitzky-Golay* (or wavelet denoise: see page 136) smooth is preferred. But if the objective of the measurement is to measure the peak position (x-axis value of the peak), larger smooth ratios can be employed if desired, because smoothing has little effect on the peak position (unless peak is asymmetrical or the increase in peak width is so much that it causes adjacent peaks to overlap). If the peak is actually formed of two underlying peaks that overlap so much that they appear to be one peak, then curve fitting is the only way to measure the parameters of the underlying peaks. Unfortunately, the optimum signal-to-noise ratio corresponds to a smooth ratio that significantly distorts the peak, which is why curve fitting the unsmoothed data is often the preferred method for measuring peaks position, height, and width. The peak *area* is not changed by a properly constructed smoothing operation unless it changes your estimate of the beginning and the ending of the peak.

In *quantitative chemical analysis* applications based on calibration by standard samples, the peak height reduction caused by smoothing is not so important. If the *same* signal processing operations are applied to the samples and to the standards, the peak height reduction of the standard signals will be *the same* as that of the sample signals and the effect will *be cancelled out* exactly. In such cases, smooth widths from 0.5 to 1.0 can be used if necessary, to further improve the signal-to-noise ratio, as shown in the figure on the previous page (for a simple sliding-average rectangular smooth function). In practical analytical chemistry, absolute peak height measurements are seldom required. Calibration against standard solutions is the rule. (Remember: the objective of quantitative analysis is not to measure a signal but rather to measure the concentration of the unknown.) It is very important, however, to apply *the same* signal processing steps to the standard signals as to the sample signals, otherwise a large systematic error will result.

For a more detailed comparison of all four smoothing types considered above, see page 60.

When should you smooth a signal?

There are four reasons to smooth a signal:

(a) for cosmetic reasons, to prepare a nicer-looking or more dramatic graphic of a signal for visual inspection or publication, especially in order to emphasize *long-term* behavior over *short-term*, or

(b) If the signal contains mostly *high-frequency* ("blue") noise, which can look bad but has less effect on the low-frequency signal components (e.g. the positions, heights, widths, and areas of peaks) than white noise, or

(c) if the signal will be subsequently analyzed by a method that would be degraded by the presence of too much noise in the signal, for example, if the heights of peaks are to be determined *visually or graphically* or by using the MAX function, of the widths of peaks is measured by the halfwidth function, or if the location of maxima, minima, or inflection points in the signal is to be determined automatically by detecting zero-crossings in derivatives of the signal. Optimization of the amount and type of smoothing is important in these cases (see page 44). Generally, if a computer is available to make quantitative measurements, it is better to use least-squares methods on the *unsmoothed* data, rather than graphical estimates on smoothed

data. If a commercial instrument has the option to smooth the data for you, it is best to disable the smoothing and record and save the *unsmoothed* data. You can always smooth it yourself later for visual presentation and it will be better to use the unsmoothed data for a least-squares fitting or other processing that you may want to do later. Smoothing can be used to *locate peaks,* but it should not be used to *measure peaks*.

(d) The formal limit of detection and limit of quantification of an analytical method (references 91, 92) may be improved by smoothing or averaging, depending on the method of signal measurement, as was described on page 27 and demonstrated by the Matlab/Octave script SNRdemo.m.

You must use care in the design of algorithms that employ smoothing. For example, in a popular technique for peak finding and measurement discussed later (page 234), peaks are located by detecting downward zero-crossings in the *smoothed* first derivative, but the position, height, and width of each peak is determined by least-squares curve-fitting (page 173) of a segment of original *unsmoothed* data in the vicinity of the zero-crossing (page 236), rather than simply taking the maximum of the smoothed data. That way, even if heavy smoothing is necessary to provide reliable discrimination against noise peaks, the peak parameters extracted by curve fitting are not distorted by the smoothing function.

When should you NOT smooth a signal?

One common situation where you should ***not*** smooth signals is prior to statistical procedures such as least-squares curve-fitting. There are several reasons (reference 43).

(a) Smoothing will not significantly improve the accuracy of parameter measurement by least-squares measurements between separate independent signal samples.

(b) All smoothing algorithms are at least slightly "lossy", entailing at least some change in signal shape and amplitude.

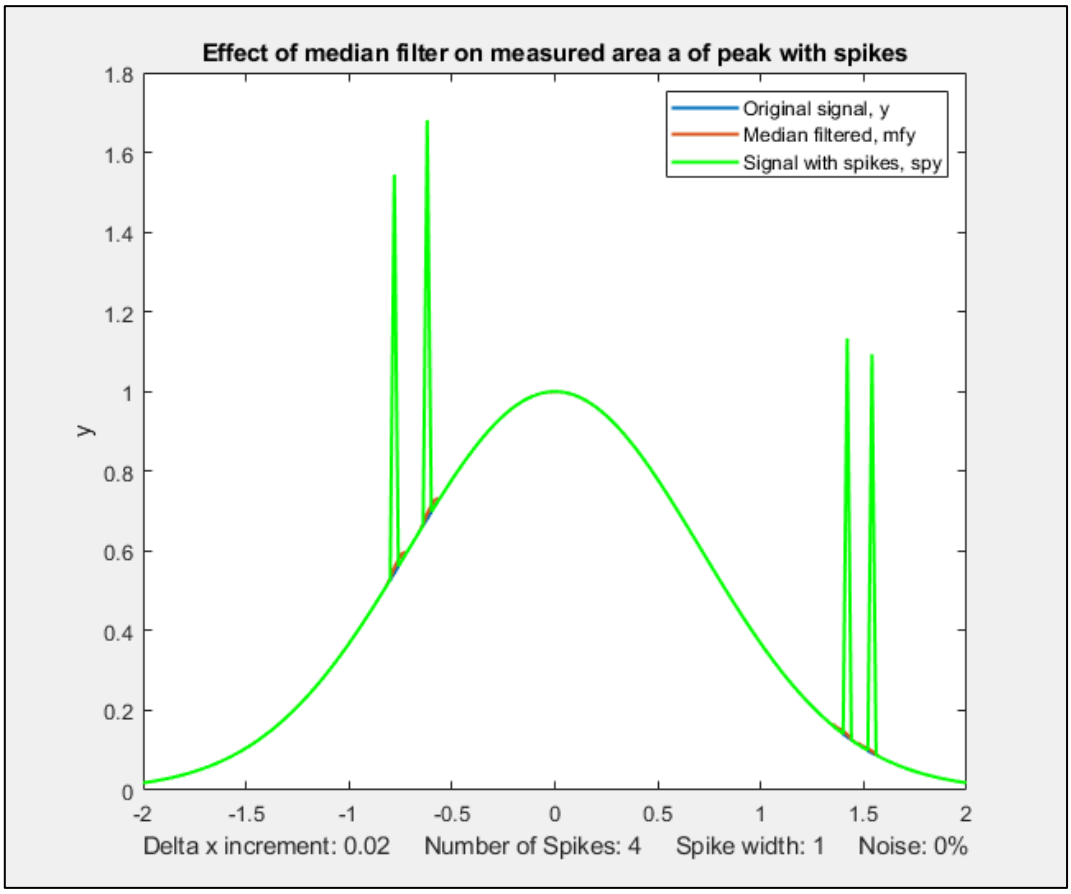
(c) It is harder to evaluate the fit by inspecting the residuals if the data are smoothed, because *smoothed noise may be mistaken for an actual signal*.

(d) Smoothing the signal will seriously underestimate the parameter errors predicted by the algebraic propagation-of-error calculations and by the *bootstrap* method (page 170). Even a visual estimate of the quality of the signal is compromised by smoothing, which makes the signal *look better than it really is*.

Dealing with spikes and outliers.

Sometimes signals are contaminated with very tall, narrow “spikes” or "outliers" occurring at random intervals and with random amplitudes, but with widths of only one or a few points. For example, optical spectroscopy using photomultiplier tube detectors is subject to spikes caused by “cosmic rays” from outer space passing through the front window of the detector, creating a pulse of Cherenkov radiation. It not only looks ugly, but it also upsets the assumptions of least-squares computations because it is not *normally distributed* random noise. This type of interference is difficult to eliminate using the above smoothing methods without distorting the signal. However, a “median” filter, which replaces each point in the signal with the median (rather than the *average*) of *m* adjacent points, can

eliminate narrow spikes, with little change in the signal, if the width of the spikes is only one or a few points and equal to or less than *m*. See http://en.wikipedia.org/wiki/Median_filter. The script "TestSpikefilters.m" demonstrates the median filter in action, removing the effect of narrow spikes:




```
PercentAreaErrorBefore =4.5%
PercentAreaErrorMedian =0.16%
PercentAreaErrorInterp =0.004%
```

For another example, see page 295.

A different approach to spike elimination is used by my killspikes.m function. It locates and eliminates the spikes by "patching over them" using linear interpolation from the signal points immediately before and after the spike. See page 57 for details.

Unlike conventional smooths, these functions can be profitably applied *prior* to least-squares fitting functions. (On the other hand, if the *spikes themselves* are the signal of interest, and the other components of the signal are interfering with their measurement, see page 304).

Ensemble Averaging

Another way to reduce noise in repeatable signals, such as the set of ten unsmoothed signals on page 47, is simply to compute their average, called *ensemble averaging*, which can be performed in this case very simply by the Matlab/Octave code **plot(x, mean(y)).** The result shows a reduction in white noise by about sqrt(10)=3.2. This improves the signal-to-noise ratio enough to see that there is a single peak with Gaussian shape, which can then be measured by curve fitting (covered in a later section, page 199)

using the Matlab/Octave code **peakfit([x; mean(y)],0,0,1)**, with the result showing excellent agreement with the position (500), height (2), and width (150) of the Gaussian peak created in the third line of the generating script (on page 47). A huge advantage of ensemble averaging is that the *noise at all frequencies is reduced*, not just the *high*-frequency noise as in smoothing. This is a big advantage if either the signal or the baseline drift.

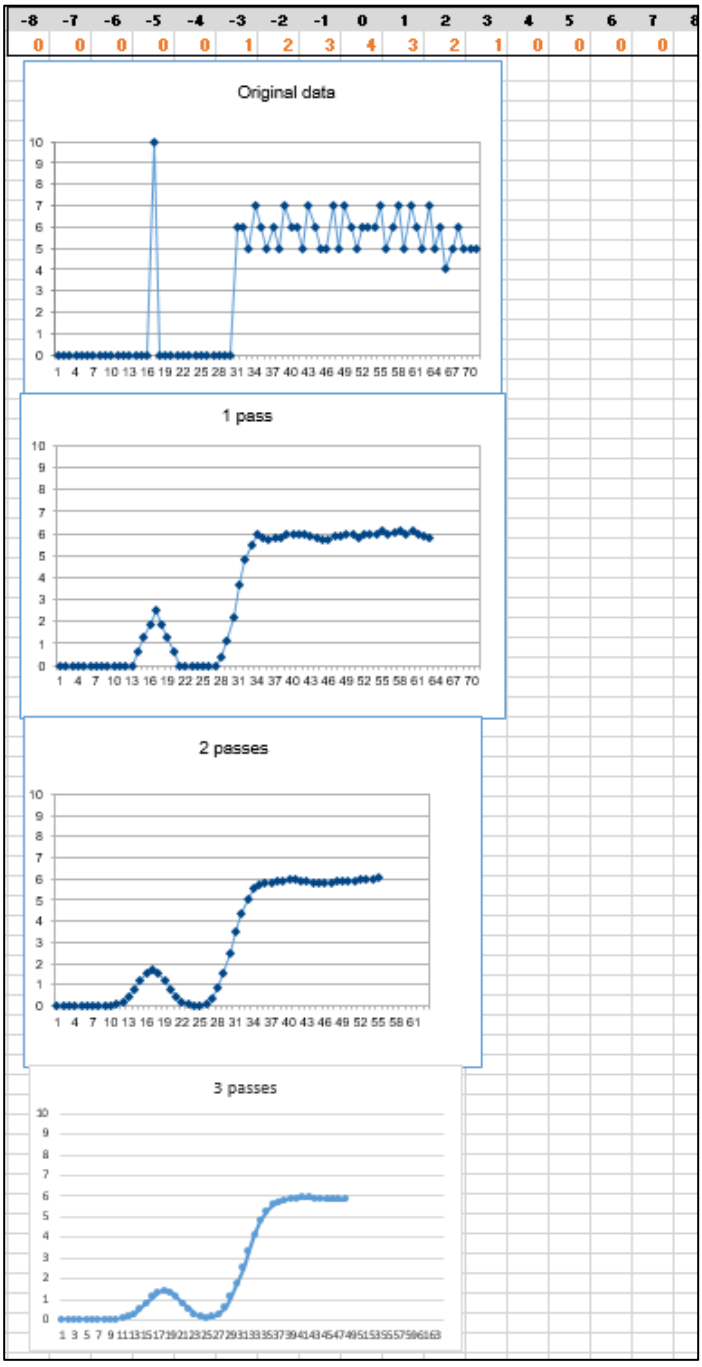
Condensing oversampled signals

Sometimes signals are recorded more densely (that is, with higher sampling frequency or with smaller x-axis intervals) than necessary to capture all the important features of the signal. This results in larger-than-necessary data sizes, which slows down signal processing procedures and may tax storage capacity. To correct this, oversampled signals can be reduced in size either by eliminating data points (say, dropping every second point or every third point) or by replacing groups of adjacent points by their *averages,* which is often called *bunching*. Bunching has the advantage of *using* rather than *discarding* data points, and it acts like smoothing to provide some measure of noise reduction. If the noise in the original signal is white, and the signal is condensed by averaging every "*n"* points, the noise is reduced in the condensed signal by the square root of *n*, but with *no change* in the frequency distribution of the remaining noise. The Matlab/Octave script testcondense.m demonstrates the effect of boxcar averaging using the condense.m function to reduce noise without changing the noise color. Shows that the boxcar reduces the measured noise, removing the high-frequency components but has little effect on the peak parameters. Least-squares curve-fitting on the condensed data is faster and results in a lower fitting error, but *no more accurate measurement* of peak parameters. If you find yourself resorting to very large smooth widths, consider using the condense function first.

Video Demonstration. This 18-second, three MByte video (Smooth3.wmv) demonstrates the effect of triangular smoothing on a single Gaussian peak with a peak height of 1.0 and a peak width of 200. The initial white noise amplitude is 0.3, giving an initial signal-to-noise ratio of about 3.3. An attempt to measure the peak amplitude and peak width of the noisy signal, shown at the bottom of the video, are initially seriously inaccurate because of the noise. As the smooth width increases, however, the signal-to-noise ratio and the accuracy of the measurements of peak amplitudes and peak widths are both improved. However, above a smooth width of about 40 (smooth ratio 0.2), the smoothing function causes the peak to be shorter than 1.0 and wider than 200, *even though the signal-to-noise ratio continues to improve* as the smooth width is increased.

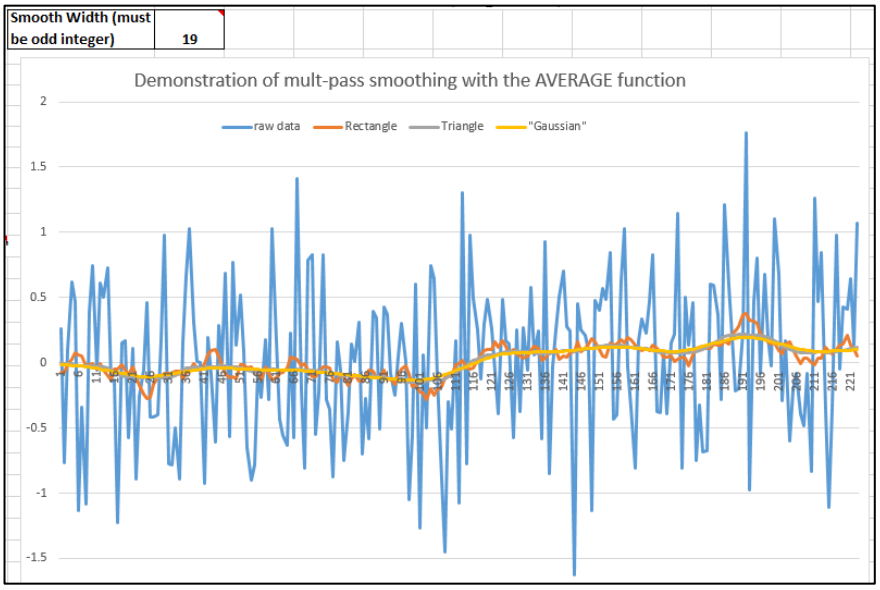
Smoothing in spreadsheets

Smoothing can be done in spreadsheets using the "shift and multiply" technique described above. In the demonstration spreadsheets smoothing.ods and smoothing.xls (screen image) the set of multiplying coefficients is contained in the formulas that calculate the values of each cell of the smoothed data in columns C and E. Column C performs a 7-point *rectangular* smooth (1 1 1 1 1 1 1). Column E performs a 7-point *triangular* smooth (1 2 3 4 3 2 1), applied to the data in column A. You can type in (or Copy and Paste) any data you like into column A, and you can extend the spreadsheet to longer columns of data by dragging the last row of columns A, C, and E down as needed. But to change the smooth width, you would have to change the equations in columns C or E and copy the changes down the entire column. It is common practice to divide the results by the sum of the coefficients so that the net gain is unity and the area under the curve of the smoothed signal is preserved. The spreadsheets UnitGainSmooths.xls and UnitGainSmooths.ods (screen image) contain a collection of unit-gain convolution coefficients for a rectangular, triangular, and p-spline smooth of width 3 to 29 in both vertical (column) and horizontal (row) forma Copy and Paste these into your own spreadsheets.



The spreadsheets MultipleSmoothing.xls and MultipleSmoothing.ods (screen image on the left) demonstrate another method in which the coefficients are contained in a group of 17 adjacent cells (in row 5, columns I through Y), making it easier to change the *smooth shape* and width (up to a *maximum* of 17) just by changing those 17 cells. (To make a smaller smooth operation, just insert zeros for the unused coefficients. In this example, a 7-point triangular smooth is defined in columns N - T and the rest of the coefficients are zeros). In this spreadsheet, the smooth is applied *three times* in succession in columns C, E, and G, resulting in an effective maximum smooth width of n*w-n+1 = 49 points applied to column G. A disadvantage of the above technique for smoothing in spreadsheets is that it is cumbersome to expand them to very large smooth widths.

A more flexible and powerful technique, especially for very large and variable smooth widths, is to use the built-in spreadsheet function AVERAGE, which by itself is equivalent to a rectangular smooth, but if applied two or three times in succession, generates triangle and P-spline-shaped smooths. It is best used in conjunction with the INDIRECT function (page 353) to control a dynamic range of values. This is demonstrated in the spreadsheet VariableSmooth.xlsx in which the data in column A are smoothed by three successive applications of AVERAGE, in columns B, C, and D, each with a smooth width specified in a single cell F3. If *w* is the smooth width, which can be *any odd positive number*, the resulting smooth in column D has a *total* width of n**w*-n+1 = 3**w*-2 points. The cell formula of the smooth operations (`=AVERAGE(INDIRECT("A"&ROW(A17)-($F$3-1)/2&":A"&ROW(A17)+($F$3-1)/2))`) uses the INDIRECT function to apply the AVERAGE function to the data in the rows from *w*/2 rows *above* to *w*/2 rows *below* the current row, where the smooth width *w* is in cell F3. If you Copy and Paste this formula to your own spreadsheets, you must manually *change all references to column "A"* to the column that contains the data to be smoothed in your spreadsheet and change all references to "$F$3" to the location of the smooth width in your spreadsheet. Then when you drag-copy down to cover all your data points, the row cell references will take care of themselves.



The example in the graphic above shows smoothing applied to a DC (direct current) signal with a step change occurring at x=111. Without smoothing (blue line) the step is almost invisible. As an application example, the smoothed signal might be used to trigger an alarm whenever it exceeds a value of 0.2, warning that something has occurred, whereas the raw unsmoothed signal would be completely unsuitable for that purpose.

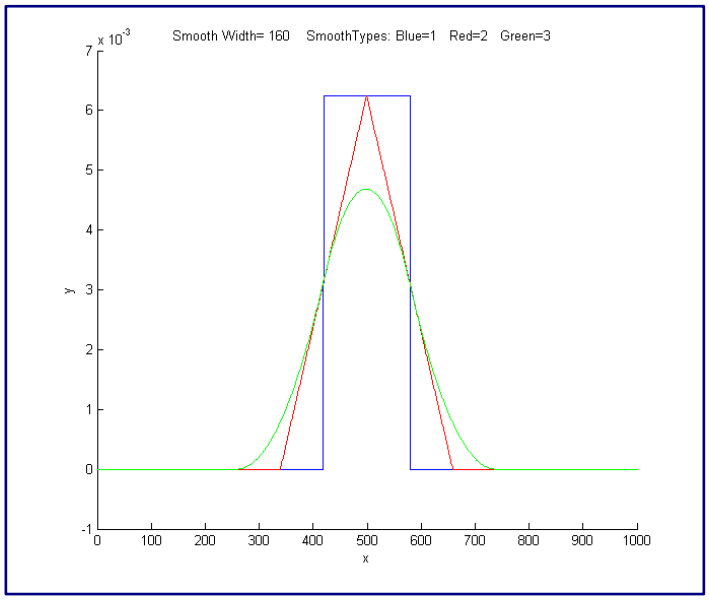
Another set of spreadsheets that uses this same AVERAGE(INDIRECT()) technique is SegmentedSmoothTemplate.xlsx, a *segmented* multiple-width data smoothing spreadsheet template that can apply individually specified different smooth widths to different regions of the signal. This is especially useful if the widths or the noise level of the peaks vary substantially across the signal. In this version, there are 20 segments. Similar templates could be constructed with any number of segments.

SegmentedSmoothExample.xlsx is an example with data (graphic). Note that the plot is conveniently lined up with the columns containing the smooth widths for each segment. A related sheet, GradientSmoothTemplate.xlsx or GradientSmoothExample2.xlsx (graphic), performs a *gradient* smooth, linearly increasing (or decreasing) in smooth width across the entire signal, given only the starting and ending values, and automatically generating as many segments and different smooth

widths as are necessary. (It also enforces the restriction, in column C, that each smooth width must be an odd number, to prevent an x-axis shift in the smoothed data).

Smoothing in Matlab and Octave

The built-in "smooth" function, in both Matlab and Python, implements several different types of smoothing (type "help smooth" at the command prompt).



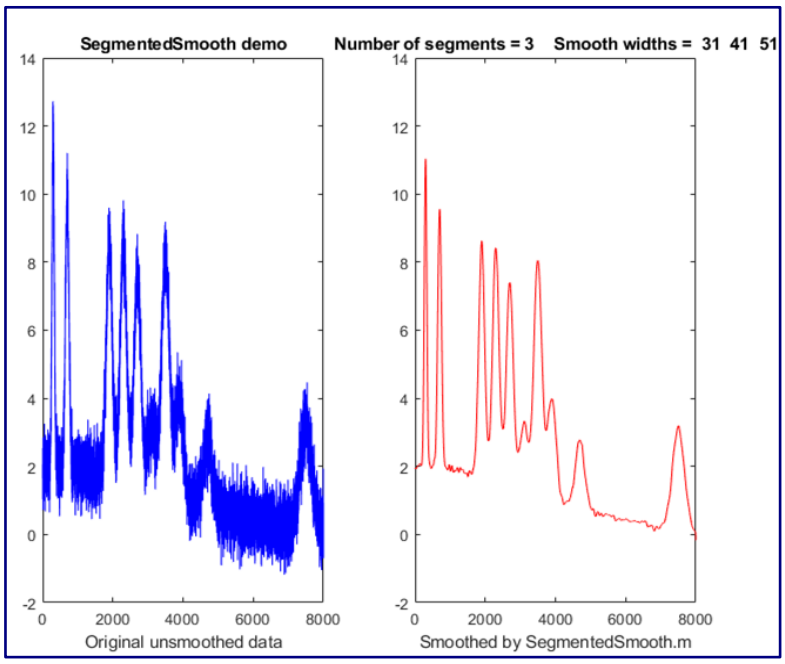
My Matlab function fastsmooth implements shift-and-multiply type smooths using a faster *recursive algorithm*. It is a Matlab function of the form **s**=fastsmooth(a,w, type, edge). The argument "a" is the input signal vector, "w" is the smooth width (a positive integer), "type" determines the smooth type: type=1 gives a rectangular (sliding-average or boxcar) smooth,type=2 gives a triangular smooth, equivalent to two passes of a sliding average, and type=3 gives a "p-spline" smooth, equivalent to three passes of a sliding average. These shapes are compared in the figure on the left. (See page 60 for a comparison of these smoothing modes). The argument "edge" controls how the "edges" of the signal (the first w/2 points and the last w/2 points) are handled. If edge=0, the edges are zero. (In this mode the elapsed time is independent of the smooth width. This gives the fastest execution time). If edge=1, the edges are smoothed with progressively smaller smooths the closer to the end. (In this mode the execution time increases with increasing smooth widths). The smoothed signal is returned as the vector "s". (You can leave off the last two input arguments: fastsmooth(Y,w,type) smooths with edge=0 and fastsmooth(Y,w) smooths with type=1 and edge=0). Compared to convolution-based smooth algorithms, fastsmooth uses a simple recursive algorithm that typically gives faster execution times for large smooth widths. It can smooth a 1,000,000-point signal with a 1,000-point sliding average in less than 0.1 seconds on a standard Windows PC. Here's a simple example of fastsmooth demonstrating the effect on white noise (graphic).

```
x=1:100;
y=randn(size(x));
plot(x,y,x, fastsmooth(y,5,3,1),'r')
xlabel('Blue: white noise.    Red: smoothed white noise.')
```

Segmented smoothing

SegmentedSmooth.m is a *segmented* version of fastsmooth. *The syntax is the same as* fastsmooth.m, except that the second input argument "smoothwidths" can be a *vector*: `SmoothY = SegmentedSmooth (Y, smoothwidths, type, ends)`. The function divides Y into several equal-length regions defined by the length of the vector 'smoothwidths', then smooths each region with a smooth of type 'type' and width defined by the elements of *vector* 'smoothwidths'. In the graphic

example on the next page, `smoothwidths=[31 52 91]`, which divides up the signal into three equal regions and smooths the first region with smoothwidth 31, the second with smoothwidth 51, and the last with smoothwidth 91. *You may use any number of smooth widths and any sequences of smooth widths*, just by how you define the vector "smoothwidths". No other change is needed. Type "help SegmentedSmooth" for other examples.



DemoSegmentedSmooth.m is a demonstration script that shows the operation with different signals consisting of noisy variable-width peaks that become progressively wider (figure on the right). If the peak widths increase or decrease regularly across the signal, you can calculate the smoothwidths vector by giving only the number of segments ("NumSegments"), the first value, "startw", and the last value, "endw", like so:

```
wstep=(endw-startw)/NumSegments;
smoothwidths=startw:wstep:endw;
```

Other smoothing functions.

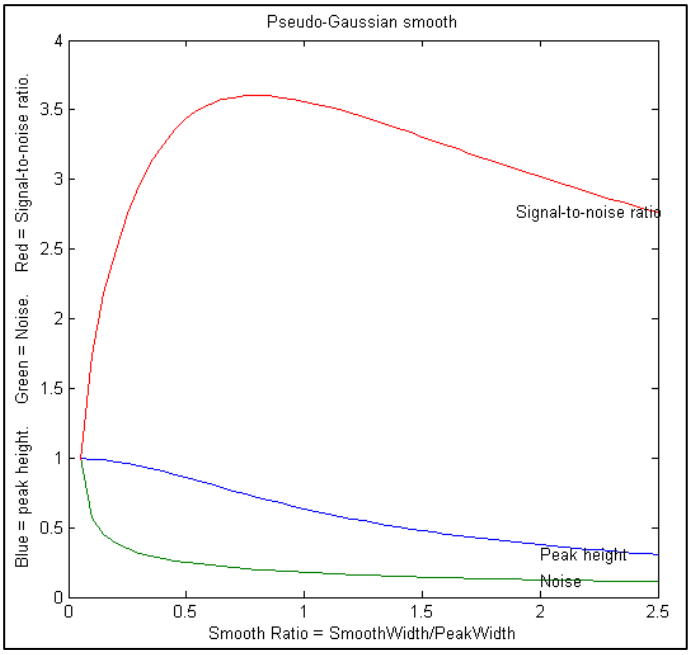
Diederick has published a Savitzky-Golay smooth function in Matlab, which you can download from the Matlab File Exchange. It is included (with attribution) in my iSignal function (page 376). Greg Pittam has published a modification of my fastsmooth function that tolerates NaNs ("Not a Number") in the data file (nanfastsmooth(Y,w,type,tol)) and another version for smoothing "angle" data that repeats every 360° or 2 π radians (nanfastsmoothAngle(Y,w,type,tol)).

Most signal processing operations, including smoothing, will fail in the raw data set contains a NaN ("not a number") or Inf ("infinity"). In such cases you can remove them from a data vector by using my downloadable rmnan.m function, which removes NaNs and Infs from vectors, replacing with neighboring real numbers. The example below creates a data vector v that contains two Infs and returns a clean vector of the same length:

```
>> v=[1 2 3 4 Inf 6 7 Inf 9];
>> rmnan(v)

ans =
   1     2     3     4     4     6     7     7     9
```

SmoothWidthTest.m is a demonstration script that uses the fastsmooth function to demonstrate the effect of smoothing on peak height, noise, and signal-to-noise ratio of a peak. You can change the peak shape in line 7, the smooth type in line 8, and the noise in line 9. A typical result for a Gaussian peak with white noise smoothed with a p-spline (pseudo-Gaussian) smooth is shown on the left. Here, as it is for most peak shapes, the optimal signal-to-noise ratio occurs at a smooth ratio of about 0.8. However, that optimum corresponds to a *significant reduction in peak height*, which could be a problem. A smooth width of about *half* the width of the original unsmoothed peak produces less distortion of the peak but still achieves good noise reduction. SmoothVsCurvefit.m is a similar script but is also compares curve fitting (page 199) as an alternative method to measure the peak height *without smoothing*.



This effect is explored more completely by the code below, which shows an experiment in Matlab or Octave that creates a Gaussian peak, smooths it, compares the smoothed and unsmoothed version, then uses the max(), halfwidth(), and trapz() functions to print out the *peak height, halfwidth, and area*. (Note: "max" and "trapz" are both built-in functions in Matlab and Octave, but you must download halfwidth.m. To learn more about these functions, type "help" followed by the function name).

```
x=[0:.1:10]';
y=exp(-(x-5).^2);
plot(x,y)
ysmoothed=fastsmooth(y,11,3,1);
plot(x,y,x, ysmoothed, 'r')
disp([max(y) halfwidth(x,y,5) trapz(x,y)])
disp([max(ysmoothed) halfwidth(x,ysmoothed,5)  trapz(x, ysmoothed)])
```

```
      max         halfwidth      Area
        1          1.6662       1.7725
  0.78442          2.1327       1.7725
```

These results show that smoothing *reduces* the peak height (from 1 to 0.784) and *increases* the peak width (from 1.66 to 2.13) but has *no observable effect* on the peak area if you measure the *total area* under the broadened peak. Smoothing is useful if the peak height, position, or width are measured by simple methods, but *there is no need to smooth the data* if the noise is white and these peak parameters are measured by least-squares methods, because the least-squares results obtained on the unsmoothed data will be more accurate than the slightly distorted smoothed signal (see page 233).

Other noise-reduction functions.

The Matlab/Octave user-defined function condense.m, *condense(y,n)*, returns a condensed version of *y* in which each group of *n* points is replaced by its average, reducing the length of *y* by the factor *n*. (For *x,y* data sets, use this function on **both** independent variable *x* **and** dependent variable *y* so that the

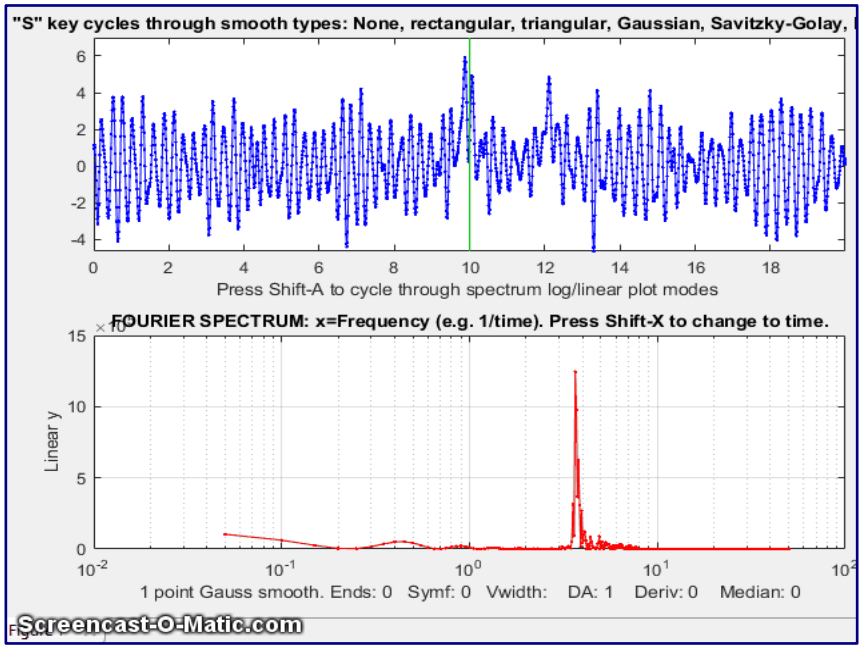
features of *y* will appear at the same *x* values). Random white noise in the signal is reduced by sqrt(*n*) but the noise color is unchanged.

The Matlab/Octave user-defined function medianfilter.m, **`medianfilter(y,w)`**, performs a median-based filter operation that replaces each value of *y* with the median of *w* adjacent points (which must be a positive integer). killspikes.m is a threshold-based filter for eliminating narrow spike artifacts. The syntax is `fy= killspikes(x, y, threshold, width)`. Each time it finds a positive or negative jump in the data between y(n) and y(n+1) that exceeds "threshold", it replaces the next "width" points of data with a linearly interpolated segment spanning x(n) to x(n+width+1). The script TestSpikefilters compares both spike filters on a Gaussian with spikes and shows how accurately they recover the original peak area.

ProcessSignal is a Matlab/Octave command-line function that performs smoothing and differentiation on the time-series data set x,y (column or row vectors). It can employ all the types of smoothing described above. Type "help ProcessSignal" at the command line. This function returns the processed signal as a vector that has the same shape as x, regardless of the shape of y. The syntax is Processed = ProcessSignal(x, y, DerivativeMode, w, type, ends, Sharpen, factor1, factor2, Symize, Symfactor, SlewRate, MedianWidth).

Real-time smoothing in Matlab is discussed on page 347. Smoothing in *Python* is described on page 448. A *Live Script* (page 37) for smoothing is covered on page 59.

iSignal (page 376) is an interactive keystroke-operated function for Matlab that includes smoothing for time-series signals using *all the algorithms discussed above*, including the Savitzky-Golay smooth, the segmented smooth, a median filter, and a condense function. Simple keystrokes allow you to adjust any of the smoothing parameters continuously while observing the effect on your signal instantly, making it easy to observe how different types and amounts of smoothing effect noise and signal, such as the height, width, and areas of peaks. Other functions of iSignal include differentiation, peak sharpening, interpolation, least-squares peak measurement, and a frequency spectrum mode that shows how smoothing and other functions can change the frequency spectrum of your signals. The simple script "iSignalDeltaTest" demonstrates the frequency response of iSignal's smoothing functions by applying them to a single-point spike, allowing you to change the smooth type and width to see how the frequency response changes. (View the code here or download the ZIP file with sample data for testing). The Octave version is isignaloctave.m, which has different keys for pan and zoom.



You try it: Here's an experiment you can try using *iSignal*. This uses a previously recorded example of a very noisy signal with lots of high-frequency (blue) noise *totally obscuring a perfectly good peak* in the center at x=150, height=1e-4, and SNR=90. First, download iSignal.m and NoisySignal.mat into the Matlab search path, then execute these statements:

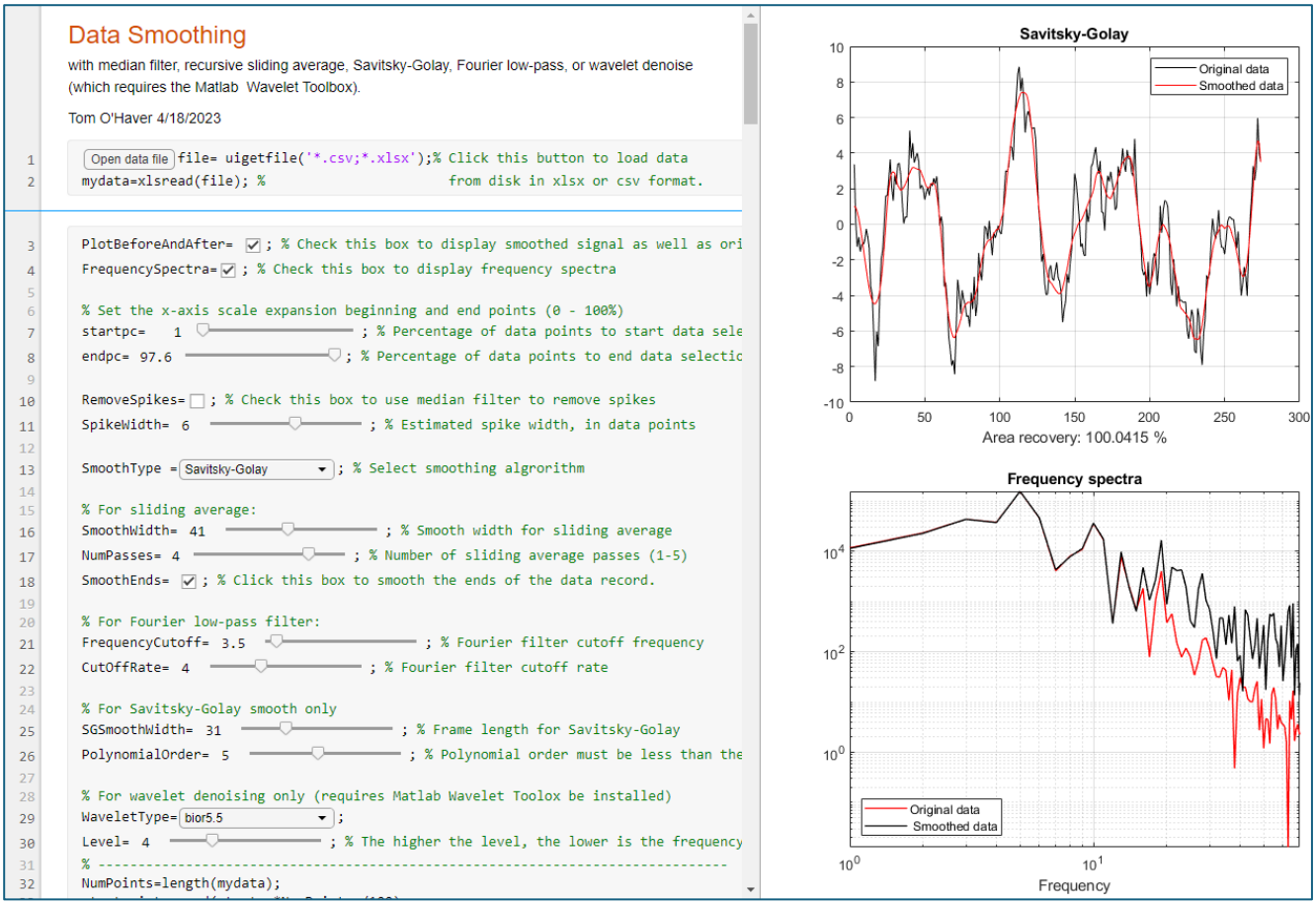
```
>> load NoisySignal
>> isignal(x,y);
```

Use the **A** and **Z** keys to increase and decrease the smooth width, and the **S** key to cycle through the available smooth types. Hint: use the "p-spline" smooth and keep increasing the smooth width until the peak becomes visible. (Unfortunately, iSignal does not currently work in *Octave*, but it does work in a Web browser using *Matlab Online*. See https://www.mathworks.com/products/matlab-online.html).

Note: If you are reading this online, you can right-click on any of the m-file links on this site and select **Save Link As...** to download them to your computer for use within Matlab.

Live script for smoothing

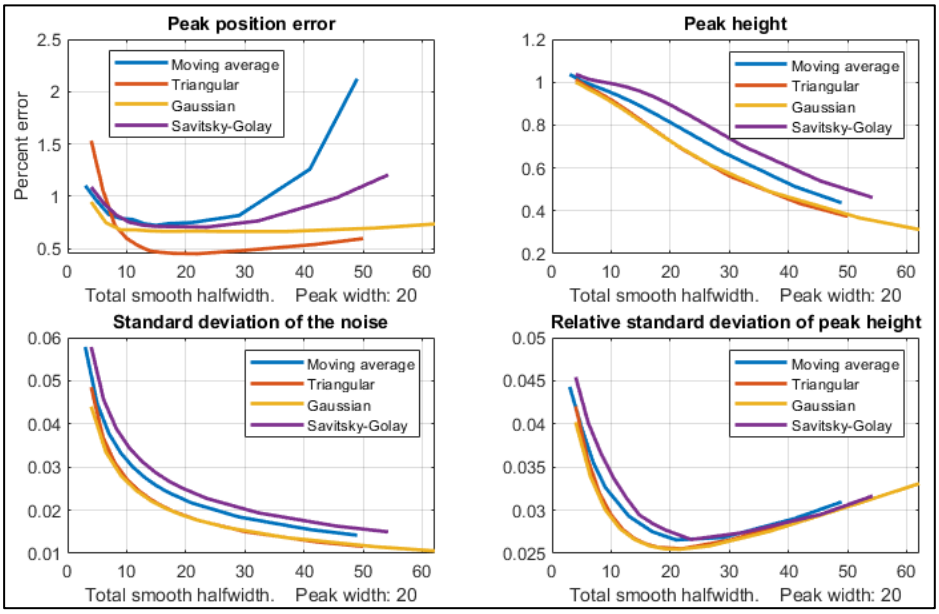
Here is an interactive *Matlab Live Script* for performing several types of smoothing applied to experimental data stored on disk (page 368, download link: DataSmoothing.mlx). It can perform spike removal, sliding average smooths with up to 5 passes, Savitsky-Golay and Fourier low-pass filtering (page 128), and wavelet denoising (page 133, which requires the Matlab Wavelet Toolkit). Clicking the **Open data file** button in line 1 opens a file browser, allowing you to navigate to your data file (in .csv or .xlsx format. The script assumes that your x,y data are in the first two columns). All the variables and settings appear in the Matlab workspace as usual. The finished smoothed data are in the vector "sy".



The script has several interactive controls. The **startpc** and **endpc** sliders in lines 7 and 8 allow you to select which portion of the data range to process, from 0% to 100% of the total range of the data file. The **RemoveSpikes** checkbox applies a median filter (page 50) to remove sharp narrow spikes. **SmoothType** drop-down menu in line 13 selects the smoothing algorithm - each has one or more controls specific to that smooth type in lines 16 to 30. The first choice is the recursive sliding average (fastsmooth.m) algorithm (page 42). The smooth width and number of passes are controlled by the sliders in lines 16 and 17. The other controls are explained in the accompanying comment lines (in green). Fourier filtering, Savitsky-Golay and wavelet denoising are topics that will be explained in later sections. The **PlotBeforeAndAfter** checkbox in line 3 gives you the option of plotting the original signal (in black) along with the processed signal (in red). The **FrequencySpectra** checkbox in line 4 allows you to show the *frequency spectrum* of the original and/or processed signals (page 94). **Note**: to view the graphic plots to the right of the code, as shown above, right-click on the empty space on the right and select "Disable synchronous scrolling". Note: you can double-click any of the sliders to change their ranges if the initial range is insufficient. See page 370 for other interactive tools.

Smoothing performance comparison

The Matlab/Octave function "MultiPeakOptimization.m" is a self-contained function that compares the performance of four types of linear smooth operations: (1) sliding-average rectangular, (2) triangular, (3) p-spline (equivalent to three passes of a sliding-average), and (4) Savitzky-Golay. These are the four smooth types discussed above, corresponding to the four values of the “SmoothMode” input



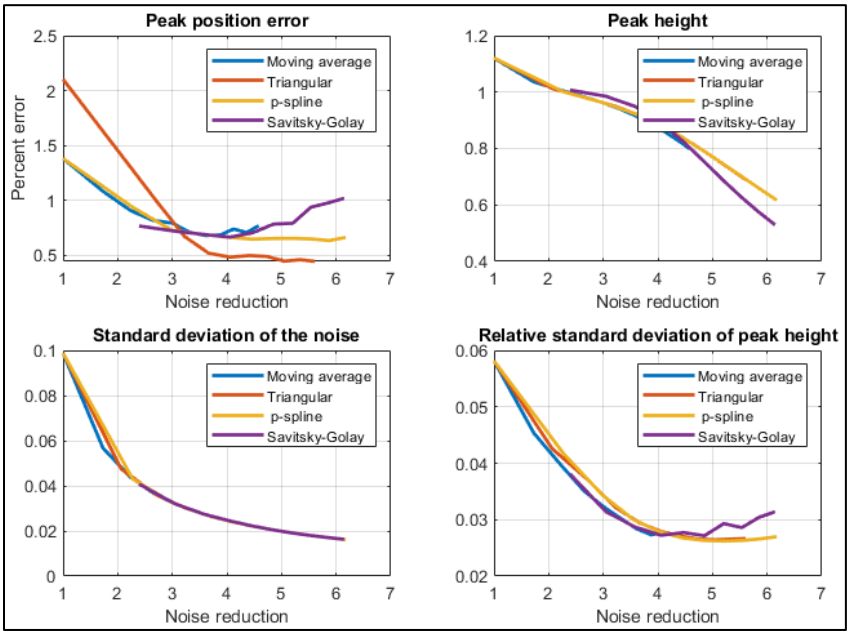
argument of the ProcessSignal and the interactive iSignal functions. These four smooth operations are applied to a 18000-point signal consisting of 181 Gaussian peaks all with a height of 1.0 and a FWHM (full-width at half-maximum) of 20 points (“wid”, line 10), which are all separated by an x-value of 160.01 (line 16), plus added noise consisting of normally-distributed random white noise with a mean of zero and a standard deviation of “Noise” (line 20). The x-axis peak position and y-axis height of each smoothed peak is determined by the height and position of the maximum single signal point for each peak. The relative standard deviation of the measured peak heights is recorded as a function of “total smooth width”, *tsw*, which is defined as the halfwidth of the impulse response of each smooth type. The results are shown in the figure below for a peak halfwidth of 20 and a noise standard deviation of 0.2 (i.e., 20% of the peak height).

The four quadrants of the graph are: (upper left) peak position error expressed as a percentage of the peak separation, (upper right) the mean peak height of the smoothed peaks, (lower left) the standard deviation of the smoothed noise, and (lower right) the relative standard deviation of the measured peak heights. The different smooth types are indicated by color: blue - sliding-average; red - triangular; yellow - p-spline, and purple - Savitzky-Golay.

These results show that the results of these different smooth types are quite similar but that, the Savitzky-Golay smooth gives the smallest reduction in peak height but the smallest reduction in noise amplitude, compared to the other methods. All these smoothing methods result in similar improvements in the standard deviation of the peak height (bottom right panel) and in the peak position error (upper left panel). Moreover, in all cases, the optimum performance is achieved when the total smooth width is approximately equal to the halfwidth of the peak. The conclusions are the same for a

Lorentzian peak, as demonstrated by a similar function "MultiPeakOptimizationLorentzian.m", graphic, the difference being that the peak height reduction is greater for the Lorentzian. For applications where the shape of the signal must be preserved as much as possible, the Savitzky-Golay smooth is a good choice. In peak detection applications (page 68), on the other hand, where the purpose of smoothing is to reduce the noise in the derivative signal, the retention of the shape of that derivative is less important because peak parameters are determined by least-squares fitting. Therefore, the triangular or p-spline smooth is well suited to this purpose and can be faster for very large smooth widths.

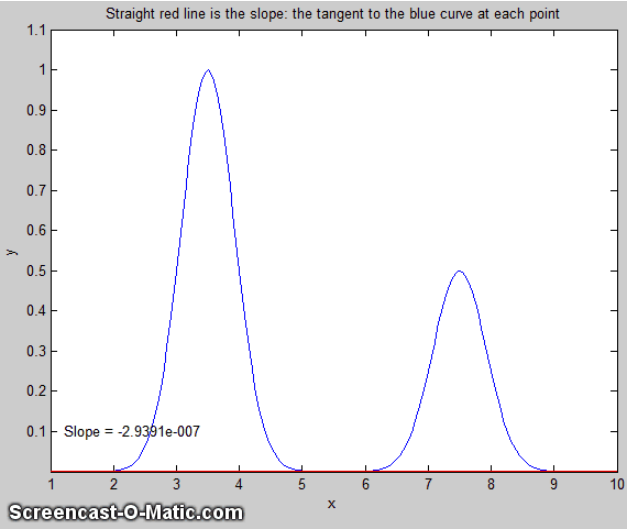
The differences between these methods are even less when the abscissas in the above graphs are changed from *total smooth bandwidth* to *white noise reduction factor*, defined as the square root of the reciprocal of the sum of the square of the impulse response function, as shown below.



An important detail is that these results apply only of the noise in the signal is *white* (page 27). If you smooth a signal that has been differentiated, for example, the second derivative of a Gaussian peak with white noise (graphic), high-frequency content of both the signal and the noise are greatly enhanced, and these results will be different, showing much poorer relative performance for the simple moving average (graphic). The Savitzky-Golay smooth remains superior in this case also. A more sophisticated method of noise reduction, called *wavelet denoising*, will be introduced on page 132.

Differentiation

The symbolic differentiation of functions is a topic that is introduced in all elementary Calculus courses. The numerical differentiation of digitized signals is an application of this concept that has many uses in analytical signal processing. The first derivative of a signal is the rate of change of y with x, that is, dy/dx, which we interpret as the *slope* of the tangent to the signal at each point, as illustrated by the animation shown on the left by this script. (If the animation does not show, click this link). The simplest algorithm for computing the first derivative is called a "finite difference" method:



$$Y_j' = \frac{Y_{j+1} - Y_j}{X_{j+1} - X_j} = \frac{Y_{j+1} - Y_j}{\Delta X} \qquad X_j' = \frac{X_{j+1} + X_j}{2}$$

(for 1< j <n-1).

where X'_j and Y'_j are the X and Y values of the jth point of the derivative, n = number of points in the signal, and ΔX is the difference between the X values of adjacent data points. A commonly used variation of this algorithm computes the *average* slope between three adjacent points:

$$Y_j' = \frac{Y_{j+1} - Y_{j-1}}{2\Delta X} \qquad X_j' = X_j$$ (for 2 < j <n-1).

This is called a *central-difference* method. Its advantage is that it does not produce a shift in the x-axis position of the derivative. It is also possible to compute *gap-segment* derivatives in which the x-axis interval between the points in the above expressions is greater than one: for example, Y_{j-2} and Y_{j+2}, or Y_{j-3} and Y_{j+3}, etc. This is equivalent to applying a sliding-average (rectangular) smooth (page 40) in addition to the derivative.

The *second derivative* is the derivative of the derivative: it is a measure of the *curvature* of the signal, that is, the rate of change of the slope of the signal. It can be calculated by applying the first derivative calculation twice in succession. The simplest algorithm for direct computation of the second derivative in one step is:

$$Y_j'' = \frac{Y_{j+1} - 2Y_j + Y_{j-1}}{\Delta X^2} \qquad X_j' = X_j$$ (for 2 < j <n-1).

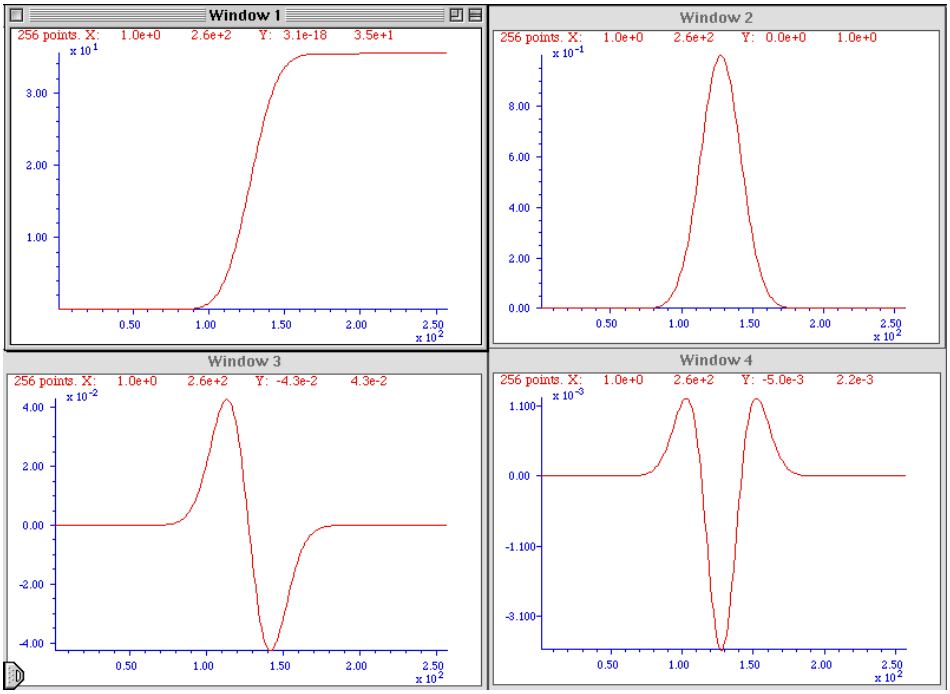
Similarly, higher derivative orders can be computed using the appropriate sequence of coefficients: for example, +1, -2, +2, -1 for the third derivative and +1, -4, +6, -4, +1 for the 4th derivative, although these derivatives can also be computed simply by taking successive lower order derivatives. The first derivative we interpret as the *slope* of the original signal at each point and the second derivative as the *curvature*. Where the signal curvature is concave-*down*, the second derivative is *negative*, and where

the signal is concave-*up*, the second derivative is *positive*. For higher derivatives, we have no single-word labels, at least in English. Each derivative is just the rate of change of the one before it.

The *Savitzky-Golay* smooth (page 40) can also be used as a differentiation algorithm with the appropriate choice of input arguments. It neatly combines differentiation and smoothing into one algorithm.

The *accuracy* of numerical differentiation is demonstrated by the Matlab/Octave script GaussianDerivatives.m (graphic link), which compares the exact *analytical* expressions for the derivatives of a Gaussian (readily obtained from Wolfram Alpha) to the *numerical* values obtained by the expressions above, demonstrating that the shape and amplitude of the derivatives are an exact match as long as the sampling interval is not too coarse. It also demonstrates that you can obtain the numerical n^{th} derivative exactly by applying "*n*" successive first differentiations. Ultimately, the numerical precision limitation of the computer could be a limitation, but only in some extreme cases, as demonstrated on page 340. (An alternative differentiation method based on the *Fourier Transform*, page 94, can calculate any derivative order but has not been used much in practice. See reference 88).

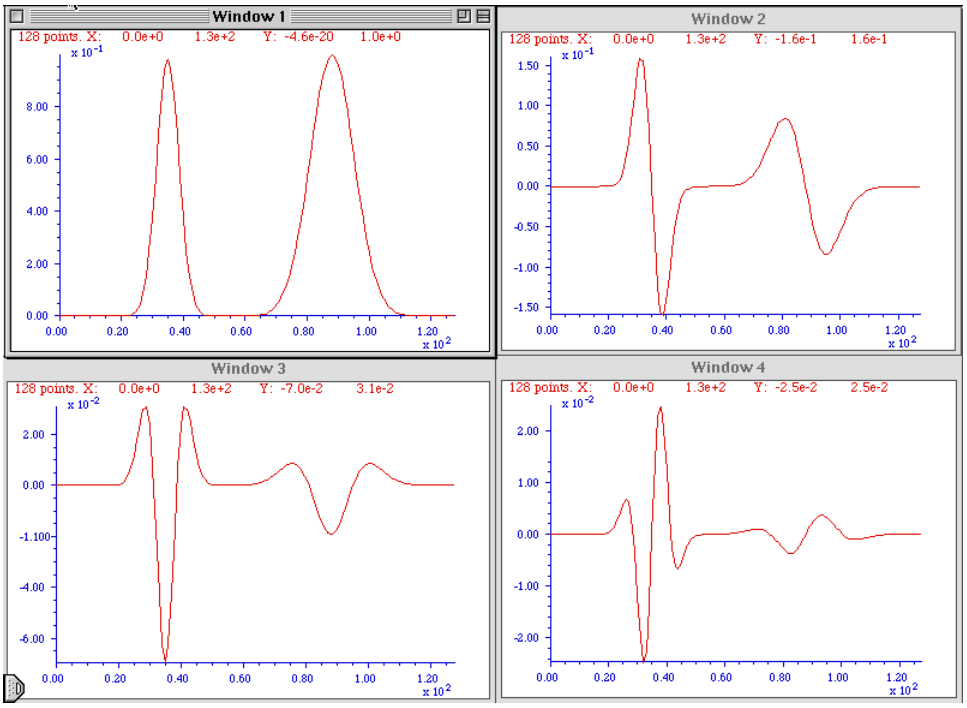
Basic Properties of Derivative Signals



The figure on the left shows the results of the successive differentiation of a computer-generated Gaussian peak. The signal in each of the four windows is the first derivative of the one before it. That is, Window 2 is the first derivative of Window 1, Window 3 is the first derivative of Window 2, Window 3 is the *second* derivative of Window 1, and so on. You can predict the shape of each signal by recalling that the derivative is simply the slope of the original signal: where a signal slopes *up*, its derivative is positive. Where a signal slopes *down*, its derivative is negative. Where a signal is flat, its derivative is zero. (Matlab/Octave code for this figure.)

The sigmoidal signal shown in Window 1 has an *inflection point* (the point where the slope is maximum) at the center of the x-axis range. This corresponds to the *maximum* in its first derivative (Window 2) and to the *zero-crossing* (point where the signal crosses the x-axis going either from positive to negative or *vice versa*) in the second derivative in Window 3. This behavior can be useful for precisely locating the inflection point in a sigmoid signal, by computing the location of the zero-crossing in its second derivative. Similarly, the location of the maximum in a peak-type signal can be

computed precisely by computing the location of the zero-crossing in its first derivative. Different peak shapes have different derivatives shapes: the Matlab/Octave function DerivativeShapeDemo.m demonstrates the first derivative forms of 16 different model peak shapes (graphic on page 424). Any smooth peak shape with a single maximum has sequential derivatives that exhibit a series of *alternating maxima and minima, the total number of which is one more than the derivative order.* The even-order derivatives have a maximum or a minimum at the peak center, and the odd-order derivatives have a zero-crossing at the peak center (Matlab/Octave code). You can also see here that the numerical magnitude of the derivatives (y-axis values) is much less than the original signal because derivatives are the *differences* between adjacent y values, divided by the independent variable increment. (It is the same reason the odometer in your car usually displays a much larger number than the speedometer (unless your car is very new, and you drive very fast). The speedometer is essentially the first derivative of the odometer).



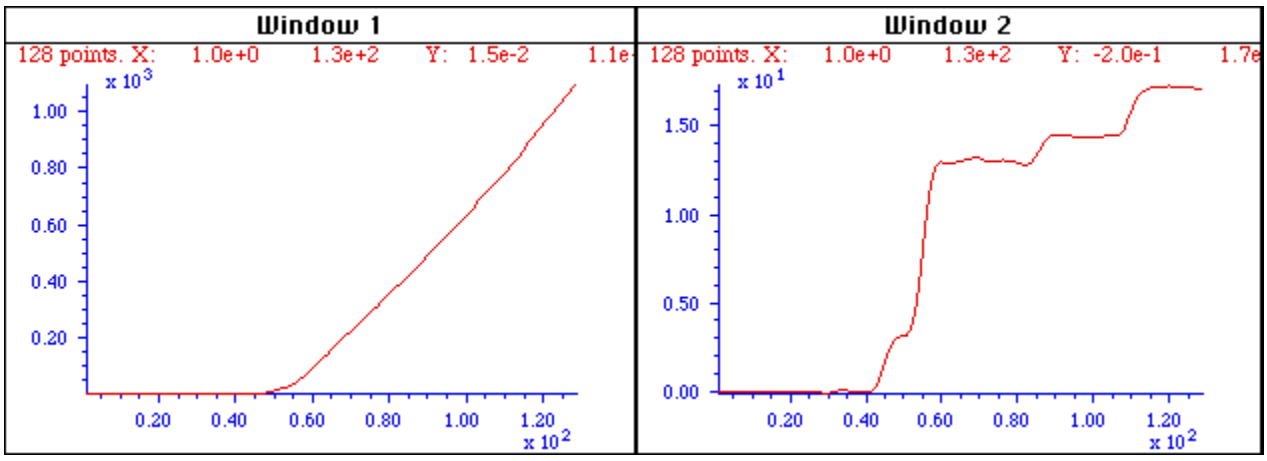
An important property of the differentiation of peak-type signals is the effect of the peak *width* on the amplitude of derivatives. The figure on the left shows the results of the successive differentiation of two computer-generated Gaussian bands. The two bands have the same amplitude (peak height) but one of them is exactly twice the width of the other. As you can see, the *wider* peak has a *smaller* derivative amplitude, and the effect becomes more noticeable at higher derivative orders. In general, the amplitude of the nth derivative of a peak is inversely proportional to the nth power of its width, for signals having the same shape and amplitude. Thus, differentiation in effect discriminates against wider peaks and the higher the order of differentiation the greater the discrimination. This behavior can be useful in quantitative analytical applications for detecting peaks that are superimposed on and obscured by stronger but broader background peaks. (Matlab/Octave code for this figure). The amplitude of a derivative of a peak also depends on the *shape* of the peak and is directly proportional to its peak *height*. Gaussian and Lorentzian peak shapes have slightly different first and second derivative shapes and amplitudes. The amplitude of the nth derivative of a Gaussian peak of height H and width W can be estimated by the empirical equation H*(10^(0.027*n^2+n*0.45-0.31)).*W^(-n), where W is the full width at half maximum (FWHM) measured in the number of x,y data points.

Although differentiation completely changes the shape of *peak-type* signals, a *periodic signal* like a

sine wave signal behaves very differently. The derivative of a sine wave of frequency *f* is a *phase-shifted* sine wave, or cosine wave, of the *same frequency* and with an amplitude that is proportional to *f*, as can be demonstrated in Wolfram Alpha. The derivative of a periodic signal containing several sine components of different frequency will *still contain those same frequencies*, but with altered amplitudes and phases. For this reason, when you take the derivative of a music or speech signal, the music or speech is still completely recognizable, but with the high frequencies increased in amplitude compared to the low frequencies, and as a result, it sounds "thin" or "tinny". See page 394 for an example.

Applications of Differentiation

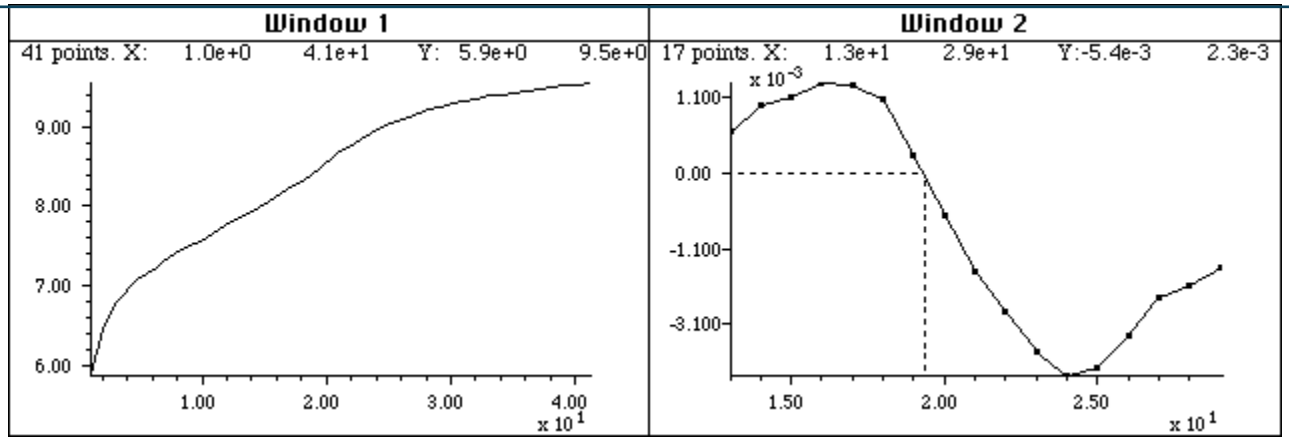
A simple example of the application of the differentiation of experimental signals is shown in the figure below. This signal is typical of the type of signal recorded in amperometric titrations and some kinds of thermal analysis and kinetic experiments: a series of straight-line segments of different slope. The objective is to determine how many segments there are, where the breaks between them fall, and the slopes of each segment. This is difficult to do from the raw data because the slope differences are small, and the resolution of the computer screen display is the limiting factor. The task is much simpler if the first derivative (slope) of the signal is calculated (below right). Each segment is now clearly seen as a separate step whose height (y-axis value) is the slope. The y-axis now takes on the units of dy/dx. Note that in this example the steps in the derivative signal are not completely flat, indicating that the line segments in the original signal were not perfectly straight. This is most likely due to random noise in the original signal. Although this noise was not particularly evident in the original signal, it is more noticeable in the derivative.



The signal on the left seems to be a more-or-less straight line, but its numerically calculated ***derivative*** *(dx/dy), plotted on the right, shows that the line in fact has several approximately straight-line segments with distinctly different slopes and with well-defined breaks between each segment.*

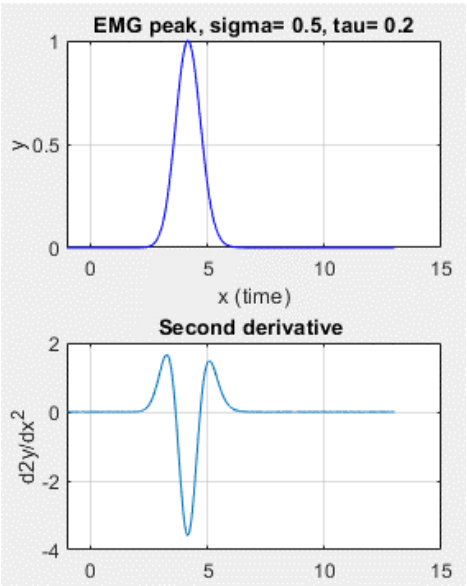
It is commonly observed that differentiation degrades the signal-to-noise ratio unless the differentiation algorithm includes smoothing (page 40) that is carefully optimized for each application. Numerical algorithms for differentiation are as numerous as for smoothing and must be carefully chosen to control signal-to-noise ratio degradation (page 72).

A classic use of second differentiation in chemical analysis is in the location of endpoints in potentiometric titration. In most titrations, the titration curve has a sigmoidal shape and the inflection point, the point where the slope is maximum and the curvature is zero, indicates the endpoint. The first derivative of the titration curve will, therefore, exhibit a *maximum* at the inflection point, and the second derivative will exhibit a *zero-crossing* at that point. Maxima and zero crossings are usually much easier to locate precisely than inflection points.



*The signal on the left is the pH titration curve of a very weak acid with a strong base, with volume in mL on the X-axis and pH on the Y-axis. The endpoint is the point of the greatest slope. This is also an inflection point, where the curvature of the signal is zero. With a weak acid such as this, it is difficult to locate this point precisely from the original titration curve. The endpoint is much more easily located in the **second derivative**, shown on the right, as the zero-crossing.*

The figure above shows a pH titration curve of a very weak acid with a strong base, with volume in mL on the X-axis and pH on the Y-axis. The volumetric equivalence point (the "theoretical" endpoint) is 20 mL. The endpoint is the point of the greatest slope. This is also an inflection point, where the curvature of the signal is zero. With a weak acid such as this, it is difficult to locate this point precisely from the original titration curve. The second derivative of the curve is shown in Window 2 on the right. The zero-crossing of the second derivative corresponds to the endpoint and is much more precisely measurable. Note that in the second derivative plot, both the x-axis and the y-axis scales have been expanded to show the zero-crossing point more clearly. The dotted lines show that the zero-crossing falls at about 19.4 mL, close to the theoretical value of 20 mL.



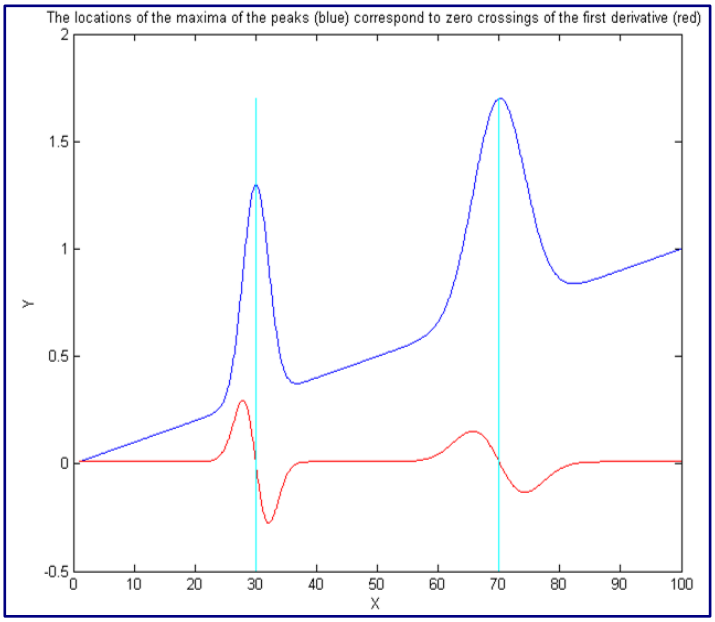
Derivatives can also be used to detect unexpected asymmetry in otherwise symmetrical peaks. For example, pure Gaussian peaks are symmetrical, but the real peaks encountered in science are sometimes asymmetrical. If the degree of broadening is small, it can be difficult to detect visually, and that is where differentiation can help. The Matlab/Octave script DerivativeEMGDemo.m (graphic) shows the 1st through 5th derivatives of a slightly asymmetrical Gaussians. The second derivative most clearly shows unequal positive peaks that would be expected to be equal for a purely symmetrical peak. The higher derivatives offer no

clear advantage and are more susceptible to white noise in the signal. Another example of asymmetry occurs if a Gaussian peak is heavily overlapped by a smaller peak with a slightly different peak center. The script DerivativePeakOverlapDemo (graphic) shows the 1st through 5th derivatives of *two* overlapping Gaussians where the second peak is so small and so close that it is impossible to discern visually, but again the *second derivative shows the asymmetry clearly* by comparing the heights of the two positive peaks. DerivativePeakOverlap.m detects the minimum extent of peak overlap by the first and second derivatives, looking for the point at which two peaks are visible. For each trial separation, it prints out the separation, resolution, and the number of peaks detected in the first and second derivatives.

Derivatives can even be used to *correct* peak asymmetry, by adding a weighted portion of the first derivative to the original peak, as described in the section on peak sharpening on page 83.

Peak detection

A very common use of differentiation is in the detection of peaks in a signal, especially to automatically determine the number of peaks and their locations. It is clear from the basic properties described in the previous section that the first derivative of a peak has a downward-going zero-crossing at the peak maximum, which can be used to locate the x-value of the peak, as shown on the right (script). If there is *no noise* in the signal, then any data point that has lower values on both sides of it will be a peak maximum. But there is always at least a little noise in real experimental signals, and that will cause many false zero-crossings simply due to the noise. To avoid this problem, one popular technique smooths the first derivative of the signal first, before looking for downward-going zero-crossings, and then takes only those zero-crossings whose slope exceeds a certain predetermined minimum (called the "slope threshold") at a point where the original signal amplitude exceeds a certain minimum (called the "amplitude threshold"). By carefully adjusting the smooth width, slope threshold, and amplitude threshold, it is possible to detect only the desired peaks over a wide range of peak widths and ignore peaks that are too small, too wide, or too narrow. Moreover, because smoothing can distort peak signals, reducing peak heights, and increasing peak widths (page 40), this technique can be extended to measure the position, height, and width of each peak by least-squares curve-fitting of a segment of *original unsmoothed signal near the top of the peak* (where the signal-to-noise ratio is usually the best). Thus, even if heavy smoothing is necessary to provide reliable discrimination against noise, the peak parameters extracted by curve fitting are not distorted, and the effect of random noise in the signal is reduced by curve fitting over multiple data points in the peak. This technique has been implemented in Matlab/Octave (page 75) and in spreadsheets (page 75).



Peak detection algorithms like this are widely applied in spectroscopy, biomedical research, environmental monitoring, financial analysis, image processing, neuroscience, physics and material science, chemistry and chromatography, speech, and audio processing (reference 100).

Derivative Spectroscopy

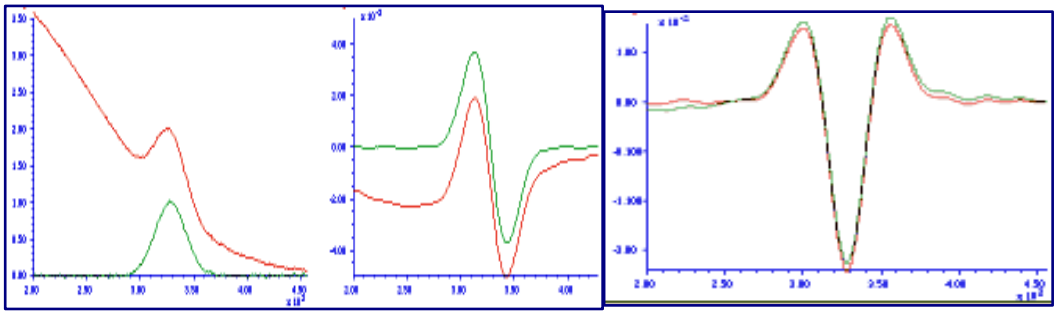
In spectroscopy, the differentiation of spectra is a widely used technique, particularly in infra-red, u.v.-visible absorption, fluorescence, and reflectance spectrophotometry, referred to as derivative spectroscopy. Derivative methods have been used in analytical spectroscopy for three main purposes.

(**a**) Spectral discrimination, as a qualitative fingerprinting technique to accentuate small structural differences between nearly identical spectra.

(**b**) Spectral resolution enhancement (peak sharpening), as a technique for increasing the apparent resolution of overlapping spectral bands in order to more easily determine the number of bands and their wavelengths.

(**c**) Quantitative analysis, as a technique for the correction for irrelevant background absorption and to facilitate multicomponent analysis. (Because differentiation is a linear technique, the amplitude of a derivative is proportional to the amplitude of the original signal, which allows quantitative analysis applications employing any of the standard calibration techniques (page 465). Most commercial spectrophotometers now have a built-in derivative capability. Some instruments are designed to measure the spectral derivatives optically, using dual-wavelength or wavelength modulation designs.

Because the amplitude of the nth derivative of a peak-shaped signal is inversely proportional to the nth power of the width of the peak, differentiation may be employed as a general way to discriminate against broad spectral features in favor of narrow components. This is the basis for the application of differentiation as a method of correction for background signals in quantitative spectrophotometric analysis. Very often in the practical applications of spectrophotometry to the analysis of complex samples, the spectral bands of the analyte (i.e., the compound to be measured) are superimposed on a broad, gradually curved background. Background signals of this type can be reduced by differentiation.



This idea is illustrated by the figure on the left, which shows a simulated UV spectrum (absorbance vs wavelength in nm), with the green curve representing the spectrum of the pure analyte and the red line representing the spectrum of a mixture containing the analyte plus other components that give rise to the large sloping "background" absorption. The first derivatives of these two signals are shown in the center. You can see that the difference between the pure analyte spectrum (green) and the mixture spectrum (red) is reduced. This effect is considerably enhanced in the second derivative, shown on the right. In this case, the spectra of the pure analyte and of the mixture are almost identical. For this technique to work, it is

necessary that the background absorption be broader (that is, have lower curvature) than the analyte spectral peak, but this turns out to be a rather common situation. Because of their greater discrimination against broad background, second (and sometimes even higher order) derivatives are often used for such purposes. See DerivativeDemo.m for a Matlab/Octave.

It is sometimes (mistakenly) said that differentiation "increases the sensitivity" of analysis. You can see how it would be tempting to say something like that by inspecting the three figures above. It does seem that the signal amplitude of the derivatives is greater than that of the original analyte signal (at least graphically). However, it is not valid to compare the amplitudes of signals and their derivatives because they *have different units*. The y-axis units of the original spectrum are *absorbance*. The units of the first derivative are *absorbance per nm*, and the units of the second derivative are *absorbance per nm*^{2}. You cannot compare absorbance to absorbance per nm any more than you can compare miles to miles per hour. (It is meaningless, for instance, to say that a speed of 30 miles per hour is greater than a distance of 20 miles.) You *can*, however, compare the *signal-to-background ratio* and the *signal-to-noise ratio*. For instance, in the above example, it would be valid to say that the signal-to-background ratio is better (higher) in the derivatives.

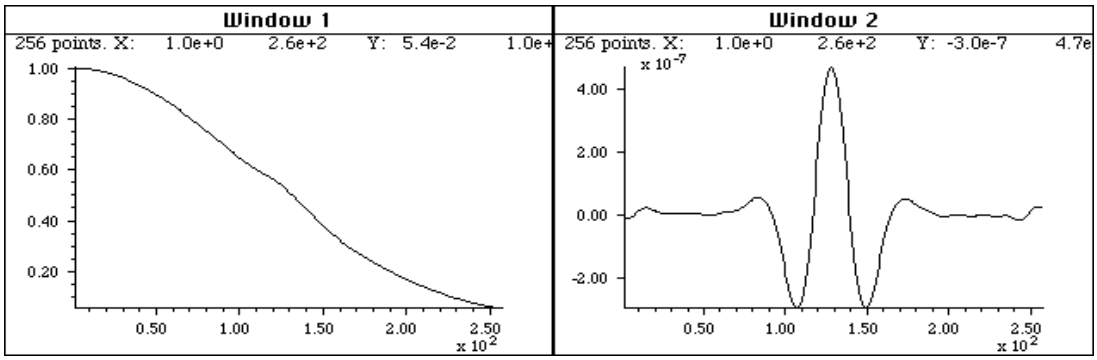
Loosely speaking, the opposite of differentiation is integration, so if you take the first derivative of a signal, you might expect to be able to regenerate the original (zeroth derivative) by integration. However, there is a catch. The constant term in the original signal (like a flat baseline) is completely lost in differentiation. Integration cannot restore it. So strictly speaking, differentiation represents a net *loss* of information, and therefore differentiation should only be used in situations where the constant term in the original signal is not of interest.

There are several ways to measure the amplitude of a derivative spectrum for quantitative chemical analysis: the absolute value of the derivative at a specific wavelength, the value of a specific feature (such as a maximum), or the difference between a maximum and a minimum. Another widely used technique is the zero-crossing measurement - taking readings derivative amplitude at the wavelength where an interfering peak crosses the zero on the y (amplitude) axis. In all these cases, it is important to measure the standards and the unknown samples in the same way. Also, because the amplitude of a derivative of a peak depends strongly on its width, it is important to control environmental factors that might change spectral peak width subtlety, such as temperature.

Trace Analysis

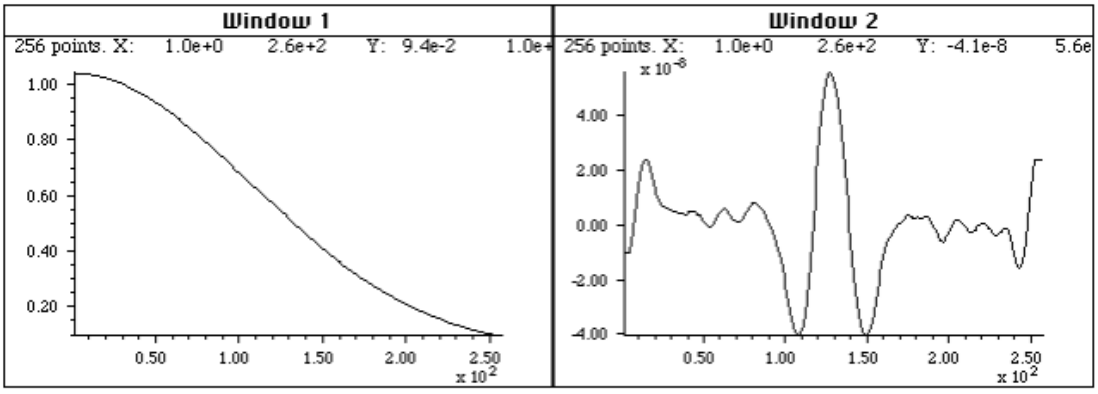
One of the widest uses of the derivative signal processing technique in practical analytical work is in the measurement of small (“trace”) amounts of substances in the presence of large amounts of potentially interfering materials. In such applications, it is common that the analytical signals are weak, noisy, and superimposed on large background signals. Measurement precision is often degraded by sample-to-sample baseline shifts due to non-specific broadband absorption, non-reproducible cuvette (sample cell) positioning, dirt or fingerprints on the cuvette walls, imperfect cuvette transmission matching, and solution turbidity. Baseline shifts from these sources are usually either wavelength-independent (light blockage caused by bubbles or large suspended particles) or exhibit a weak wavelength dependence (small-particle turbidity). Therefore, you can expect that differentiation will in general help to discriminate relevant absorption from these sources of baseline shift. An obvious

benefit of the suppression of broad background by differentiation is that *variations* in the background amplitude from sample to sample are also reduced. This can result in improved precision or measurement in many instances, especially when the analyte signal is small relative to the background or if there is a lot of uncontrolled variability in the background. An example of the improved ability to detect trace component in the presence of strong background interference is shown in this figure:



*The absorption spectrum on the left shows a weak shoulder near the center due to a small concentration of the substance that is to be measured (e.g., the active ingredient in pharmaceutical preparations). The peak is obscured by the strong background caused by other substances in the sample. The **fourth derivative** of this spectrum is shown on the right. The background has been almost completely suppressed and the analyte peak now stands out clearly, facilitating measurement.*

The spectrum on the left shows a weak shoulder near the center (at x=130) due to the analyte. The signal-to-noise ratio is very good in this spectrum, but despite that, the broad, sloping background obscures the peak and makes quantitative measurement very difficult. The fourth derivative of this spectrum is shown on the right. The background has been almost completely suppressed and the analyte peak now stands out clearly, facilitating measurement. An even more dramatic case is shown below. This is essentially the same spectrum as in the figure above, except that the concentration of the analyte is ten times lower. The question is: is there a detectable amount of analyte in this spectrum? This is quite impossible to say from the normal spectrum, but inspection of the fourth derivative (right) shows that the answer is *yes*. Some noise is visually evident here, but nevertheless the signal-to-noise ratio is sufficiently good for a reasonable quantitative measurement.

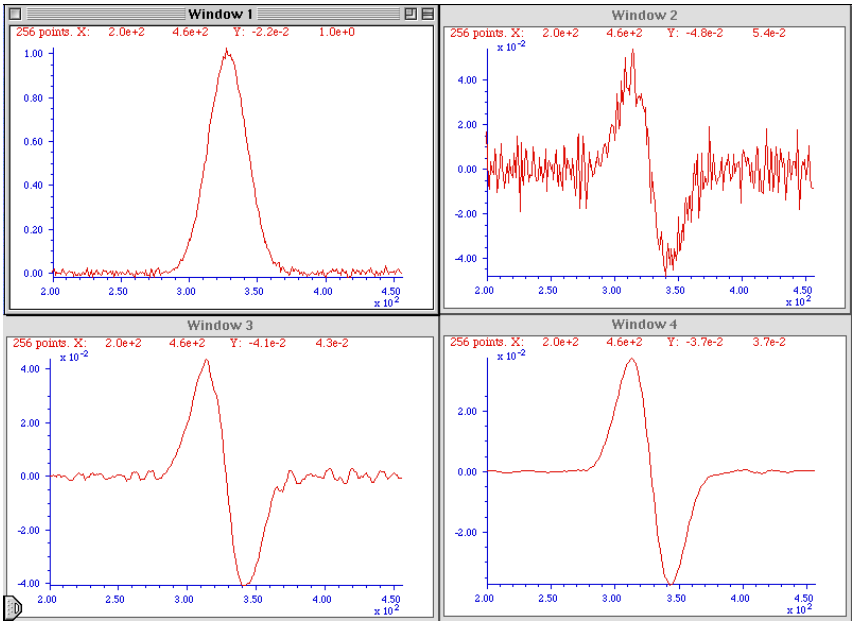


Like the previous figure, but in this case the peak is ten times lower - so weak that it cannot even be seen in the spectrum on the left. The fourth derivative (right) shows that a peak is still there, but much reduced in amplitude (note the smaller y-axis scale) and in signal-to-noise ratio.

This use of signal differentiation has become widely used in quantitative spectroscopy, particularly for quality control in the pharmaceutical industry. In that application, the analyte would typically be the active ingredient in a pharmaceutical preparation and the background interferences might arise from the presence of fillers, emulsifiers, flavoring or coloring agents, buffers, stabilizers, or other excipients. Of course, in trace analysis applications, care must be taken to optimize the signal-to-noise ratio of the instrument as much as possible.

Although it will eventually be shown that more advanced techniques such as curve fitting (page 199) can also perform many of these quantitative measurement tasks quite well (page 298), the derivative techniques have the advantage of conceptual and mathematical simplicity and an easily understood graphical way of presenting data.

Derivatives and Noise: The Importance of Smoothing

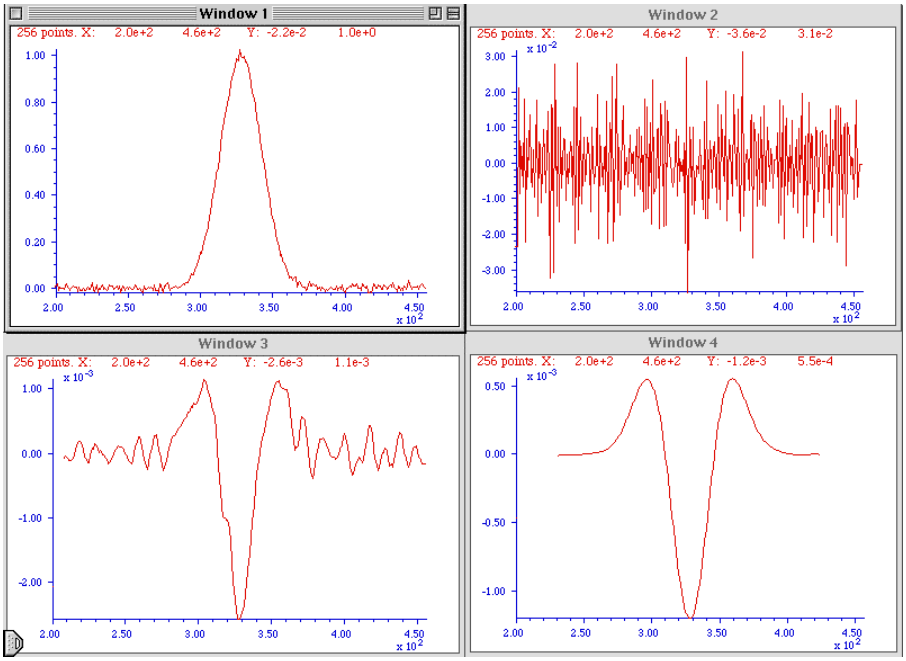


It is often said that "differentiation increases the noise". That is true, but it is not the main problem. In fact, computing the unsmoothed first derivative of a set of random numbers increases its standard deviation by only *the square root of 2*, simply due to the usual propagation of errors of the sum or difference between two numbers. As an example, the standard deviation (std) of the numbers generated by the Matlab/Octave randn function is 1.0 and the standard deviation of its first derivative, **`std(deriv1(randn(size(1:10000))),`** equals about 1.4. But even a little bit of smoothing (page 40) applied to the derivative will reduce this standard deviation greatly, e.g. a 2-point smooth applied by the fastsmooth function, **`std(fastsmooth(deriv1(randn(size(1:10000))),2,3)),`** equals about 0.4.

More important is the fact that the *signal-to-noise ratio* of an *unsmoothed* derivative is almost always much lower (poorer) than that of the original signal, mainly because *the numerical amplitude of the derivative is usually much smaller* (as you can see for yourself in all the examples on this page). But smoothing is *always* used in any practical application to control this problem. With *optimal* smoothing, the signal-to-noise of a derivative can even be *greater* than the unsmoothed original. For the successful application of differentiation in quantitative analytical applications, it is essential to use differentiation in combination with sufficient smoothing, to optimize the signal-to-noise ratio. This is illustrated in the figure on the left. (Matlab code for this figure.) Window 1 shows a Gaussian band with a small amount of added white noise. Windows 2, 3, and 4, show the first derivative of that signal with increasing smooth widths. As you can see, *without enough smoothing, the signal-to-noise ratio of the derivative can be substantially poorer than the original signal*. However, with adequate amounts of smoothing,

the signal-to-noise ratio of the smoothed derivative is much better and can even be visibly better than that of the unsmoothed original.

This effect of smoothing derivatives is even more striking in the *second* derivative, as shown on the right (Matlab/Octave code for this figure). In this case, the signal-to-noise ratio of the unsmoothed second derivative (Window 2) is so poor you cannot even see the signal visually, but the smoothed second derivative looks fine. *Differentiation does not actually add noise to the signal*. If there were no noise at all in the original signal, then the derivatives would also have no noise (exception: see page 340).



What is particularly interesting about the noise in these derivative signals, however, is the noise "color". This noise is not *white.* Rather, it is *blue* - that is, it has much more power at *high* frequencies than white noise. The consequence of this is that the noise in the differentiated signal is easily reduced greatly by *smoothing*, as demonstrated above.

Because sliding-average smoothing and differentiation are both linear operations, it makes no difference whether the smooth operation is applied before or after the differentiation. What is important, however, is the nature of the smooth, its smooth ratio (ratio of the smooth width to the width of the original peak), and the number of times the signal is smoothed. The optimum value of the smooth ratio for derivative signals is approximately 0.5 to 1.0. For a first derivative, *two* applications of a simple rectangular sliding-average smooth (or one application of a triangular smooth) is adequate. For a second derivative, *three* applications of a simple rectangular smooth or two applications of a triangular smooth are adequate. The general rule is this: for the nth derivative, use a smooth that is the equivalent of at least n+1 applications of a rectangular smooth. The Savitzky-Golay method is ideal for computing smoothed derivatives because it combines differentiation with the right kind of smoothing. The Matlab signal processing program *iSignal*, discussed on page 376, uses this approach.

If the peak widths vary substantially across the signal recording - for example, if the peaks get regularly wider as the x-value increases - then it may be helpful to use a segmented smooth (page 333), in which the smooth width is made to vary across the signal.

Smoothing derivative signals usually results in a substantial attenuation of the derivative *amplitude*. In the figure on the right above, the amplitude of the most heavily smoothed derivative (in Window 4) is much less than its less-smoothed version (Window 3). However, this will not be a problem in *quantitative analysis* applications, if the standard (analytical) curve is prepared using the exact same

derivative, smoothing, and measurement procedure as is applied to the unknown samples. Because differentiation and smoothing are both linear techniques, the amplitude of a smoothed derivative is exactly proportional to the amplitude of the original signal, which allows quantitative analysis applications employing any of the standard calibration techniques (page 465). If you apply the *same* signal-processing techniques to the standards as well as to the samples, everything works.

Because of the different kinds and degrees of smoothing that might be incorporated into the computation of digital differentiation of experimental signals, it is difficult to compare the results of different instruments and experiments unless the details of these computations are known. In commercial instruments and software packages, these details may well be hidden. However, if you can obtain both the original (zeroth derivative) signal, as well as the derivative and/or smoothed version from the same instrument or software package, then the technique of Fourier deconvolution, which will be covered later, can be used to discover and duplicate the underlying hidden computations.

Interestingly, neglecting to smooth a derivative was ultimately responsible for the failure of the first spacecraft of NASA's Mariner program on July 22, 1962, which was reported in InfoWorld's "11 infamous software bugs". In his 1968 book "The Promise of Space", Arthur C. Clarke described the mission as "wrecked by the most expensive hyphen in history." The "hyphen" was in fact a superscript bar over the symbol for velocity (the first derivative of position), handwritten in a notebook. An overbar conventionally signifies an *averaging* or *smoothing* function, so the formula *should* have calculated the *smoothed* value of the time derivative of position. *Without* the smoothing function, even minor variations would cause its derivative to be very noisy and to trigger the corrective boosters to kick in prematurely, causing the rocket's flight to become unstable. (To be fair to those pioneering engineers, it was quite early in the history of space exploration).

Video Demonstrations

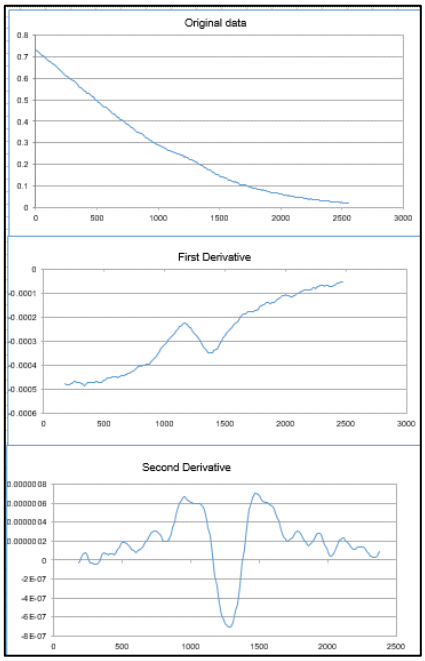
The first 13-second, 1.5 MByte video (SmoothDerivative2.wmv) demonstrates the huge signal-to-noise ratio improvements that are possible when smoothing derivative signals, in this case, a 4th derivative.

The second video, 17-second, 1.1 MByte, (DerivativeBackground2.wmv) demonstrates the measurement of a weak peak buried in a strong sloping background. At the beginning of this brief video, the amplitude (Amp) of the peak is adjusted between 0 and 0.14, but the background is so strong that the changes in peak amplitude, located at x = 500, are hardly visible. Then the fourth derivative (Order=4) is computed, and the scale expansion (Scale) is increased, with a smooth width (Smooth) of 88. Finally, the amplitude (Amp) of the peak is varied again over the same range, but now the changes in the signal are now quite noticeable and easily measured.

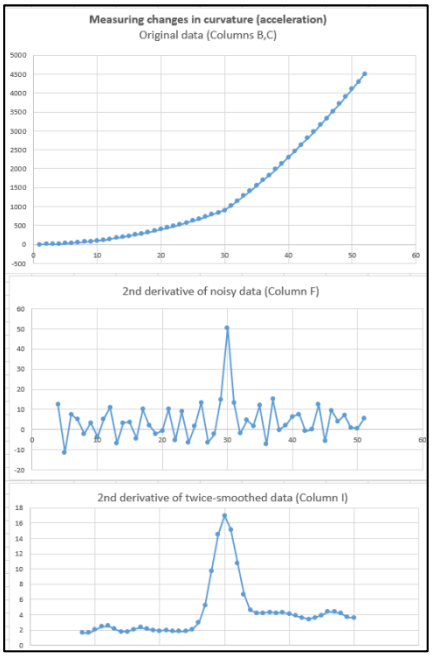
The differentiation of *analog* signals can be performed with a simple operational amplifier circuit. Two or more such circuits can be cascaded to obtain second and higher-order derivatives. The same noise problems described above apply to analog differentiation also, requiring the use of low-pass filter circuits that are analogous to smoothing.

Differentiation in Spreadsheets

Differentiation operations such as described above can readily be performed in spreadsheets such as Excel or OpenOffice Calc. Both the derivative and the required smoothing operations can be performed by the shift-and-multiply method described in the chapter on smoothing (page 40). In principle, it is possible to combine any degree of differentiation and smoothing into one set of shift-and-multiply coefficients (as illustrated here), but it is more flexible and easier to adjust if you compute the derivatives and each stage of smoothing separately in successive columns. This is illustrated by DerivativeSmoothing.ods for OpenOffice Calc and DerivativeSmoothing.xls for Excel, which smooths the data and computes the first derivative of Y (column **B**) with respect to X (column **A**), then applies that smoothing and differentiation process successively to compute the smoothed second and third derivatives. The same smoothing coefficients (in row **5**, columns **K** through **AA**) are applied successively for each stage of differentiation. You can enter any set of numbers here (preferably symmetrical about the center number in column **S**). You can type or paste your own data into columns **A** and **B** (X and Y), rows **8** to **263**.



DerivativeSmoothingWithNoise.xlsx (right) demonstrates the effect of smoothing on the signal-to-noise ratio of derivatives of a weak peak located at x = 1200 on a sloping baseline. It uses the same data as DerivativeSmoothing.xls but adds simulated white noise to the Y data. You can control the amount of added noise (cell **D5**).



Another example of a derivative application is the spreadsheet SecondDerivativeXY2.xlsx (left), which demonstrates locating and measuring changes in the second derivative (a measure of curvature or acceleration) of a time-changing signal. This spreadsheet shows the apparent increase in noise caused by differentiation and the extent to which the noise can be reduced by smoothing (in this case by two passes of a 5-point triangular smooth). The smoothed second derivative shows a large peak the point at which the acceleration changes (at x=30), and the baseline on either side of the peak is distinctly unequal, showing the change in the acceleration before and after the peak (y=2 and 4, respectively). Other examples of differentiation and smoothing by "shift-and-multiply" convolution include MultipleConvolutionFirstDerivativeDemo.xls and MultipleConvolution4thDerivativeDemo.xls

Differentiation in Matlab and Python

Finite difference differentiation functions such as described above can easily be created in Matlab or Octave. Some simple derivative functions for equally-spaced time series data: deriv, a first derivative using the 2-point central-difference method, deriv1, an unsmoothed first derivative using adjacent

differences, deriv2, a second derivative using the 3-point central-difference method, a third derivative deriv3 using a 4-point formula, and deriv4, a 4th derivative using a 5-point formula. Each of these is a simple Matlab function of the form **d=deriv(y)**. The input argument is a signal vector "**y**", and the differentiated signal is returned as the vector "**d**". For data that are *not* equally-spaced on the independent variable (x) axis, there are versions of the first and second derivative functions, derivxy and secderivxy, that take two input arguments (x,y), where **x** and **y** are vectors containing the independent and dependent variables.

SmoothDerivative.m combines differentiation and smoothing. The syntax is SmoothedDeriv = SmoothedDerivative(x,y,DerivativeOrder,w,type,ends) where 'DerivativeOrder' determines the derivative order (0 through 5), 'w' is the smooth width, 'type' determines the smooth mode:

If type=0, the signal is not smoothed
If type=1, rectangular (sliding-average or boxcar)
If type=2, triangular (2 passes of sliding-average)
If type=3, p-spline (3 passes of sliding-average)
If type=4, Savitzky-Golay smooth

'ends' controls how the "ends" of the signal (the first w/2 points and the last w/2 points) are handled: If ends=0, the ends are zeroed: If ends=1, the ends are smoothed with progressively smaller smooths the closer to the end. Type “help SmoothDerivative” for some examples (graphic). An alternative differentiation method based on the Fourier Transform (page 94) can calculate derivatives of any order and also inlcudes smoothing (reference 88).

Peak detection. The simplest code to find peaks in *x,y* data sets simply looks for every *y* value that has lower *y* values on both sides (allpeaks.m). An alternative approach is to use the first derivative to find all the maxima by locating the points of downward zero-crossing, that is, the points at which the first derivative "d" (computed by derivxy.m) passes from positive to negative. In this example, the “sign” function is a built-in function that returns 1 if the element is greater than zero, 0 if it equals zero, and -1 if it is less than zero. The routine prints out the value of x and y at each zero-crossing:

```
d=derivxy(x,y);
for j=1:length(x)-1
  if sign(d(j))>sign(d(j+1))
    disp([x(j) y(j)])
  end
end
```

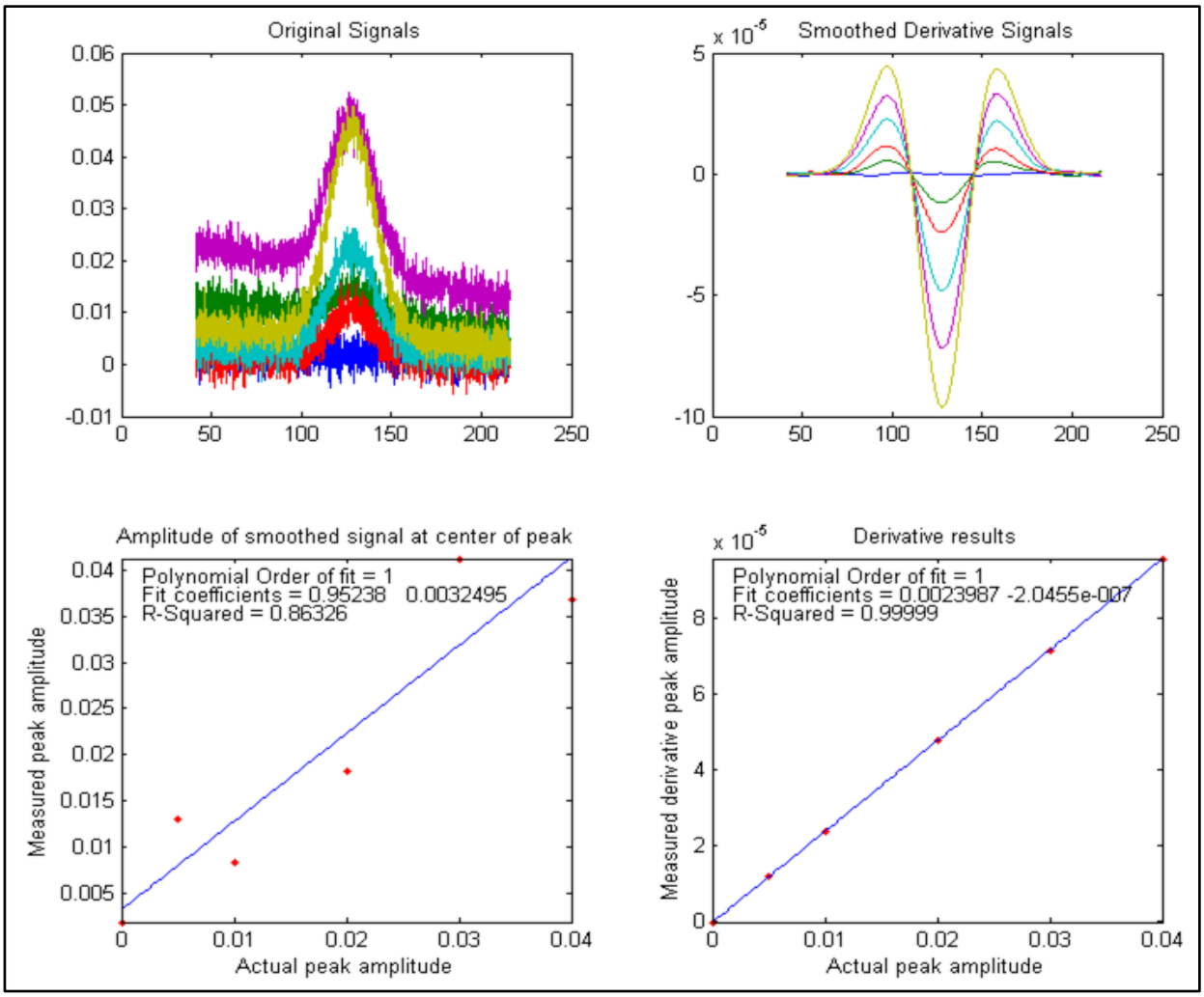
If the data are noisy, many false zero crossings will be reported, but smoothing the data will reduce that. If the data are sparsely sampled, a more accurate value for the peak position (x-axis value at the zero-crossing) can be obtained by interpolating between the point before and the point after the zero-crossing, using the Matlab/Octave “interp1” or “spline” function:

```
interp1([d(j) d(j+1)],[x(j) x(j+1)],0)
```

In Python, you can import a derivative function using “**from scipy.misc import derivative**”

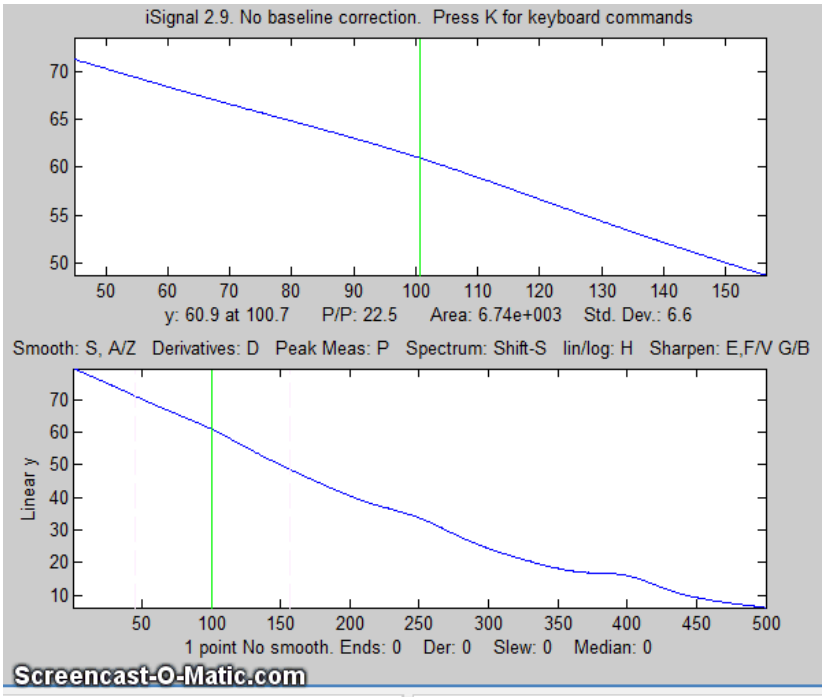
ProcessSignal.m is a Matlab/Octave command-line function that performs smoothing and differentiation on the time-series data set x,y (column or row vectors). Type "help ProcessSignal". It returns the processed signal as a vector that has the same shape as x, regardless of the shape of y. The syntax is **`Processed = ProcessSignal(x, y, DerivativeMode, w, type, ends, Sharpen, factor1, factor2, Symize, Symfactor, SlewRate, MedianWidth)`**

DerivativeDemo.m (below) is a self-contained Matlab/Octave demo function that uses



ProcessSignal.m and plotit.m to demonstrate an application of differentiation to the quantitative analysis of a peak buried in an unstable background (e.g. as in various forms of spectroscopy). The object is to derive a measure of peak amplitude that varies linearly with the actual peak amplitude and is minimally affected by the background and the noise. To run it, just type DerivativeDemo at the command prompt. You can change several of the internal variables (e.g., Noise, BackgroundAmplitude) to make the measurement harder or easier. Note that, even though the magnitude of the derivative seems to be numerically smaller than the original signal (because it has different units), the signal-to-noise ratio of the derivative is better than that of the original signals and is much less affected by the background instability.

iSignal.m (page 376), shown on the left) is an interactive function for Matlab that performs many signal-processing operations that are covered in this book, including differentiation and smoothing for time-series signals, up to the 5th derivative, automatically including the required type of smoothing. Simple keystrokes allow you to adjust the smoothing parameters (smooth type, width, and ends treatment) while observing the effect on your signal dynamically. In the animated GIF example shown here, a series of three peaks at x=100, 250, and 400, with heights in the ratio 1:2:3, are buried in a strong curved background. The smoothed second and fourth derivatives are computed to suppress that background. View the code here or download the ZIP file with sample data for testing. The interactive keypress operation works even if you run Matlab in a web browser, but not on Matlab Mobile or in Octave. (Note: figures like the one above that display "Screencast-O-Matic" in the lower-left are *animated* graphics that can be viewed in a web browser or in Microsoft Word 365 but will not animate in any PDF viewer that I have ever tried.)



As an example of smoothing in iSignal, the following statements generate the 4th derivative of a noisy Gaussian peak and display it in **iSignal**. You will need to download isignal.m, gaussian.m, and deriv4.m before executing the following statements.

```
>> x=[1:.1:300]';
>> y=deriv4(100000.*gaussian(x,150,50)+.1*randn(size(x)));
>> isignal(x,y);
```

The signal is mostly blue noise (because of the differentiated white noise) unless you smooth it considerately. Use the **A** and **Z** keys to increase and decrease the smooth width and the **S** key to cycle through the available smooth types. Hint: use the P-spline smooth and increase the smooth width.

Real-time differentiation in Matlab is discussed on page 347**.**

Live Script for Differentiation

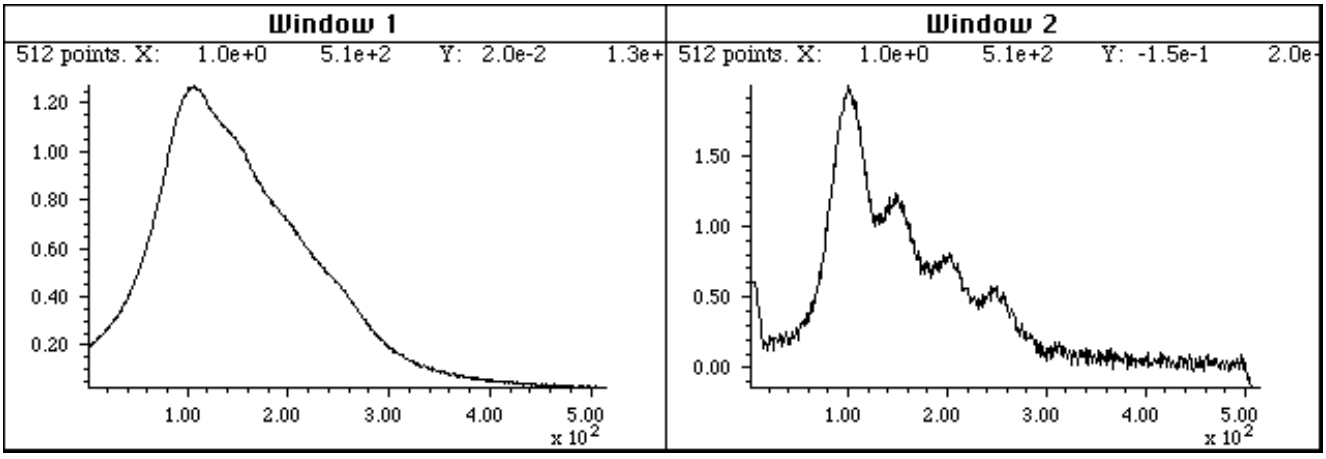
DataDifferentiation.mlx (graphic) is a Live Script for differentiation and smoothing applied to experimental data stored on disk. It is similar to DataSmoothing.mlx discussed on page 59, with the addition of a slider (line 9) to select the derivative order (up to 10). Note: If the "PlotBeforeAndAfter" check box is checked, the derivative (red curve) will be scaled to match the maximum of the original signal (black curve). If that box is not checked, the derivative will be displayed by itself with its *actual*

amplitude. (This is done because the numerical amplitude of derivatives is often orders of magnitude different than the original signals). See page 370 for other interactive tools.

Peak Sharpening

Digital techniques to reduce the width of peaks, called peak sharpening, use algorithms to artificially reduce the widths of overlapping peaks in a signal, with the intent of locating the peak positions of hidden peaks and/or measuring their areas more accurately. The techniques usually employed make the peaks narrower but taller, preserving the areas under the peaks. There are two classes of sharpening techniques: derivative-based (dealt with here) and deconvolution-based (dealt with on page 117).

The figure below shows a simple example: the signal on the left consists of several poorly-resolved (that is, partly overlapping) bands or peaks. The extensive overlap of the bands makes accurate measurement of the peak positions impossible, even though the signal-to-noise ratio is very good. It would be easier to measure the positions of the peaks accurately if they were more completely resolved, that is, if the peaks were narrower.



A peak sharpening algorithm applied to the signal on the left artificially improves the apparent resolution of the peaks. In the resulting signal, right, you can measure the positions of the peaks more accurately, but at the cost of a decrease in signal-to-noise ratio.

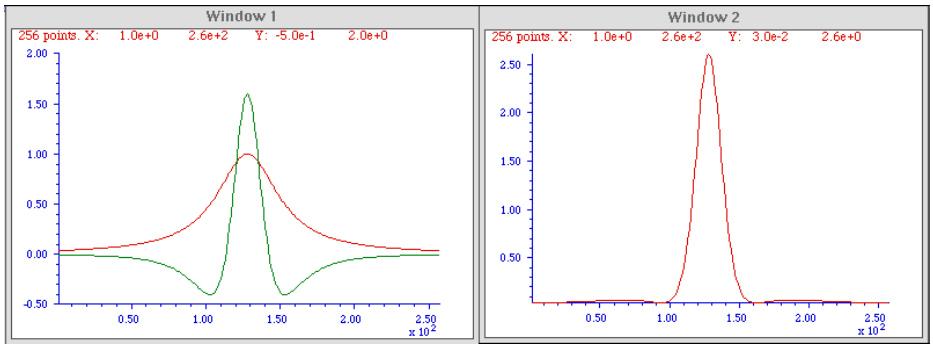
Even derivative sharpening

One of the simplest peak sharpening algorithms computes the weighted sum of the original signal and the negative of its second derivative:

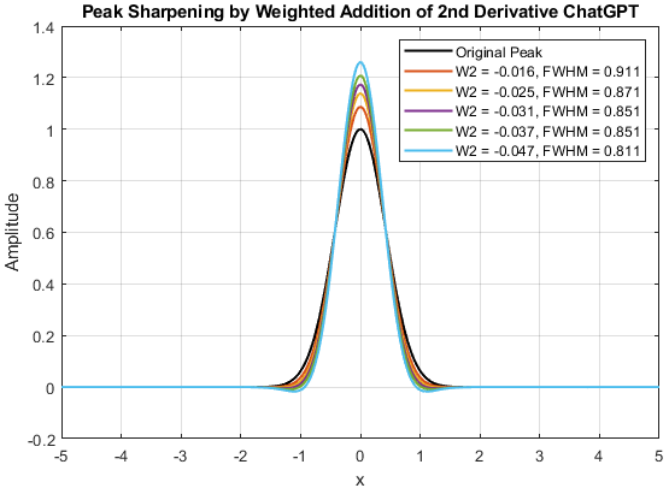
$$R_j = Y_j - k_2 Y''$$

where R_j is the resolution-enhanced signal, Y is the original signal, Y" is the second derivative of Y, and k_2 is a user-selected 2nd derivative weighting factor. It is up to the user to select the weighting factor k_2 which gives the best trade-off between the extent of sharpening, signal-to-noise degradation (described on page 70), and baseline flatness. The optimum choice depends upon the width, shape, and digitization interval of the signal. As an inevitable trade-off, the signal-to-noise ratio is degraded, but this can be moderated by smoothing (page 40). As with all sharpening techniques, this will be *useful only if the overlap of peaks, rather than the signal-to-noise ratio, is the limiting factor*.

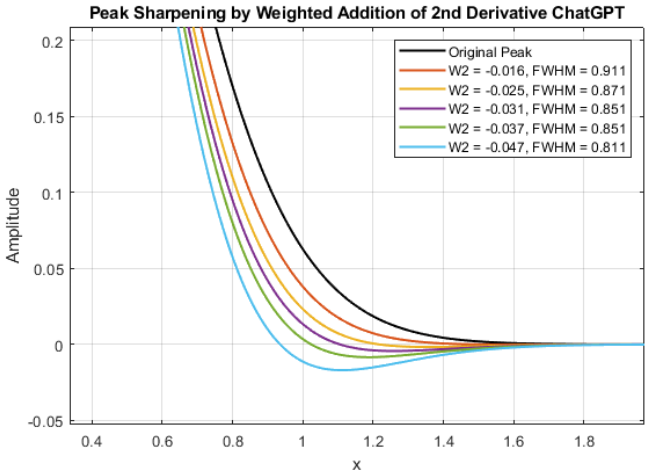
Here is how it works. The figure below shows, in Window 1, a computer-generated peak (with a Lorentzian shape) in red, superimposed on the *negative* of its second derivative in green).



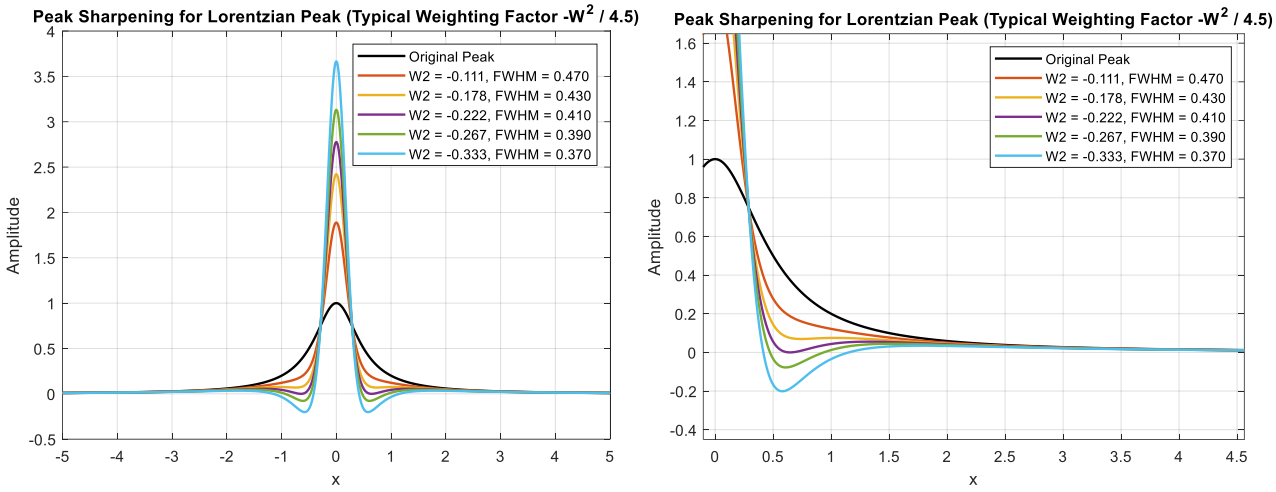
The second derivative is amplified (by multiplying it by an adjustable constant) so that the negative sides of the inverted second derivative (from approximately X = 0 to 100 and from X = 150 to 250) are a rough mirror image of the sides of the original peak over those regions. In this way, when the original peak is added to the inverted second derivative, the two signals will *approximately* cancel out in the two side regions but will reinforce each other in the central region (from X = 100 to 150). The result, shown in Window 2, looks more like a Gaussian with a substantial (about 50%) reduction in the width, and a corresponding increase in height. This effect is most dramatic with Lorentzian-shaped peaks. With Gaussian-shaped peaks, the resolution enhancement is less dramatic (only about 20%).



The optimum value for the derivative weighting factor for second derivative, k_2, is not fixed: it depends on what you consider the best trade-off between peak sharpening and baseline flatness. Higher values result in greater sharpening, too high a value will cause the sides of the sharpened peak to dip below zero. Typically, one "titrates" the k_2 value, staring from low values and increasing by degrees while watching the baseline. A reasonable initial value for a Gaussian peak is $k_2 = W^2/32$, where W is the halfwidth of the peak in x units. With that weighting factor, a Gaussian peak will be reduced in width by about 20%, the baseline will still be visually flat, and the resulting peak will fit a Gaussian model with a percent fitting error of less than 0.3% and an R^2 of 0.9999. Larger values of k_2 will result in a narrower peak, but the baseline will not be so flat. The figure above left shows a computer-generated Gaussian peak with an initial half-width of 1.0 sharpened by five different k_2 values listed in the legend. The baseline undershoot observed when the k_2 values to too large can be more easily seen in the figure on the right, which "zooms in" to the



trailing edge of the peak. For a Lorentzian peak, the initial value of k_2 is $W^2/4.5$. The figures below show a computer-generated Lorentzian peak with an initial half-width of 1.0 sharpened by five different k_2 values listed in the legend.

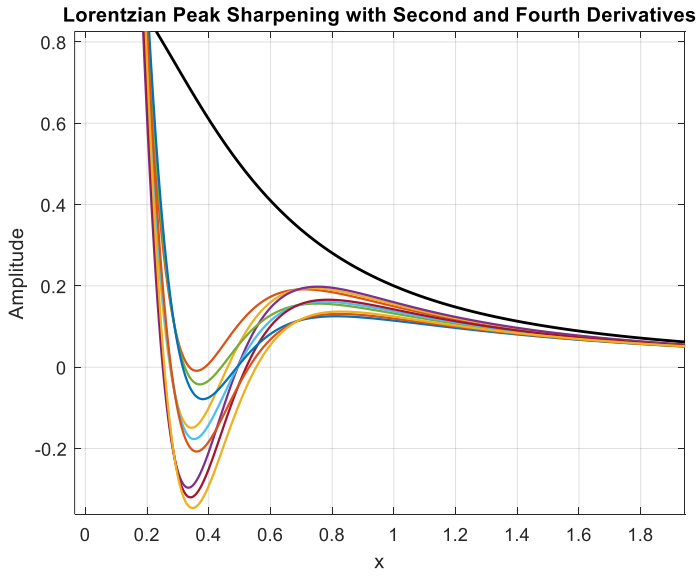


In practice, it's common that experiment peaks are neither pure Gaussian nor pure Lorentzian but rather some blend of the two, so you can expect that the derivative weighting factors and the reduction in sharpened peak width will be somewhere between the Gaussian and Lorentzian values.

This technique has been used in various forms of spectroscopy and chromatography for many years (references 74-76), in the early days using analog electronics. Mathematically, this technique is a simplified version of a converging Taylor series expansion, in which only the even order derivative terms in the expansion are taken and for which their coefficients alternate in sign. The above example is the simplest possible version that includes only the first two terms - the original peak and its negative second derivative. Slightly better results can be obtained in principle by adding a *fourth* derivative term, with two adjustable factors k_2 and k_4:

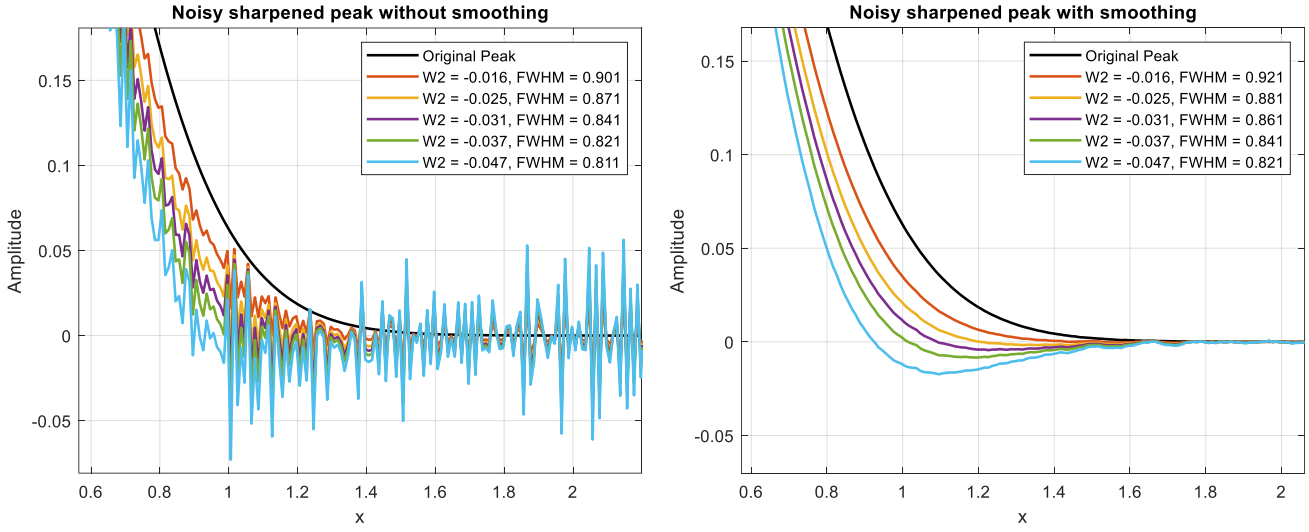
$$R_j = Y_j - k_2 Y'' + k_4 Y''''$$

where Y'' and Y'''' are the 2nd and 4th derivatives of Y. Clearly, the need to optimize *two* interacting weighting factors is a complication. In the graph and table, a Lorentzian peak whose FWHM is 1.00 is subject to three values each of k_2 and k_4 (figure created by this Matlab script).



k_2	k_4	FWHM	Min/Max
0.10	0.00	0.278	-0.26%
-0.10	0.01	0.266	-4.01%
-0.10	0.01	0.258	-7.22%
-0.13	0.00	0.282	-1.20%
-0.13	0.01	0.27	-4.51%
-0.13	0.01	0.258	-7.44%
-0.15	0.00	0.282	-2.10%
-0.15	0.01	0.27	-5.03%
-0.15	0.01	0.262	-7.69%

There are two important limitations of the even-derivative method of peak sharpening. The main one is that differentiation emphasizes the high-frequency noise in the signal (page 72). This means that it is important to smooth or filter the derivatives before adding to the original signal. A sharp cut-off filter is best for that purpose, such as a Fourier low-pass filter (page 128) or a wavelet denoise (page 136). In practice, this type of sharpening technique is limited to signals that have a good signal-to-noise ratio to begin with. An example is shown in the figures below, which are zoomed in to the trailing edge of the peak as before (generated by the Matlab script SecDerSharpNoisySmoothedGaussian.m). The left-hand figure shows the 2nd derivative sharpening of a slightly noisy signal (line 10) without any smoothing at all. As the derivative addition factor increases, the noise is greatly amplified, especially for the highest sharpening (blue line). However, with a modest amount of smoothing (line 35), the noise is greatly reduced, with only a very slight increase in the FWHM. The great improvement is the result of the high-frequency nature of the noise, which is easily reduced by low-pass filtering (page 72). The noise increase caused by the addition of the second derivative is very high-frequency weighted and is thus greatly reduced by even modest amounts of smoothing (page 44).

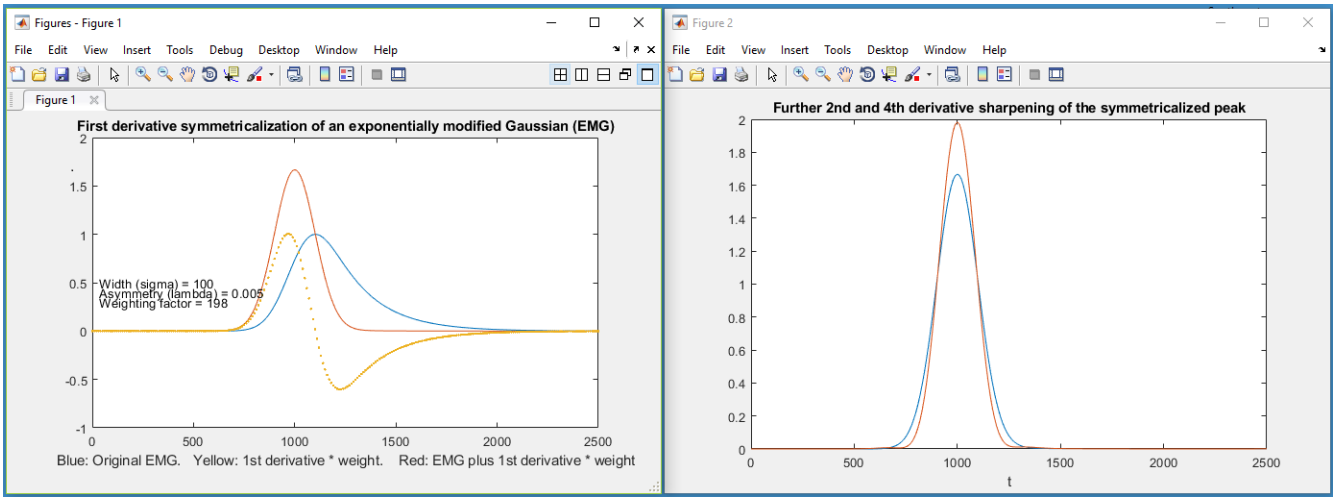


Another limitation is that the k factors for the second and fourth derivatives vary with the width W raised to the 2nd and 4th power respectively, so they can vary over a very large numerical range for peaks of different width. For this reason, if the peak widths vary substantially across the signal, it is useful to use *segmented* or *gradient* versions of this method, as we did previously for smoothing (page 55), so that the sharpening can be optimized for each region of the signal. ("Segmented" means each segment is defined independently and "gradient" means a gradual increase or decrease between specified start and end values).

Constant-area first-derivative symmetrization ("de-tailing")

If the peak is *asymmetrical* - that is, slopes down faster on one side than the other - then the weighted addition (or subtraction) of a *first derivative* term, Y', may be helpful, because the first derivative of a peak is *antisymmetric* (positive on one side and negative on the other). In the graphic example below, on the left, the asymmetrical peak (in blue) tails to the right, and its first derivative, Y', (dotted yellow) has a positive lobe on the left and a broader but smaller negative lobe on the right. When the peak is added to the weighted first derivative, the *positive lobe of the derivative reinforces the leading edge* and the *negative lobe suppresses the trailing edge,* resulting in improved symmetry. (Had the peak sloped to the *left*, the *negative* of its derivative would be added). This is also an old technique, having been used in chromatography since at least 1965 (reference 75, 76), where it has been called "de-tailing", i.e., "removing the tail".

$$S_j = Y_j + k_1 Y'$$



It sometimes happens that peaks which under ideal circumstances are expected to be purely symmetrical are in fact are observed to be asymmetrical, with an exponential "tail" on the leading or trailing edge. For a Gaussian peak this can be described mathematically:

$$f(t) = \frac{\lambda}{2} \exp\left(\frac{\lambda}{2}\left(2\mu + \lambda\sigma^2 - 2t\right)\right) \cdot \operatorname{erfc}\left(\frac{\mu + \lambda\sigma^2 - t}{\sqrt{2}\sigma}\right)$$

where *t* is the independent variable, μ is the center of the Gaussian, σ is its width (standard deviation) and λ is the rate of exponential decay.

Yuri Kalambet (reference 73) has shown that the first-derivative addition technique works perfectly for exponentially broadened peaks of *any shape*, not only for the Gaussian. With the correct first derivative weighting factor, k_1, the result S_j is a symmetrical peak with a half-width substantially less than that of the original (orange line). In fact, it is exactly the underlying peak to which the exponential convolution has been applied (References 70, 71). The first derivative weighting factor k_1 is independent of the peak height and width and is simply equal to the exponential time constant λ, in the above formulation of the EMG. It works perfectly if $k_1 = \lambda$. In practice, λ is not known and so k_1 must be determined

experimentally, which is most easily done for the last peak in a group of peaks (graphic, animation). Put simply, if you get k_1 too high, the result will dip below the baseline after the peak.

It is easy to determine the optimal k_1 value experimentally for an *isolated* peak. Just increase it until the processed signal S_j dips below the baseline after the peak, then reduce it until the baseline is flat, as shown in the GIF animation at this link. Of course, in real application the signal will contain noise, with the result that the symmetrized result will be noisier that the original signal. The first derivative weighting factor, k_1, will have to be estimated by eye and is therefore subject to some uncertainty. If one stage of derivative addition does not do the trick, try one of the double exponential routines described below. Furthermore, this appears to be a general behavior and it works similarly for any other peak shape that is broadened by exponential convolution, such a Lorentzian, and it even works for peaks that are already broadened by a previous exponential convolution (i.e., a double exponential), which can be handled by two successive stages of derivative addition with different *taus*.

If the signal-to-noise ratio of the original signal is high enough, the symmetrized peak Sj resulting from the first-derivative addition procedure can still be further sharpened by the even-derivative techniques described above, assuming that the signal-to-noise ratio of the original is good enough. This is illustrated in the right-hand panel of the previous figure.

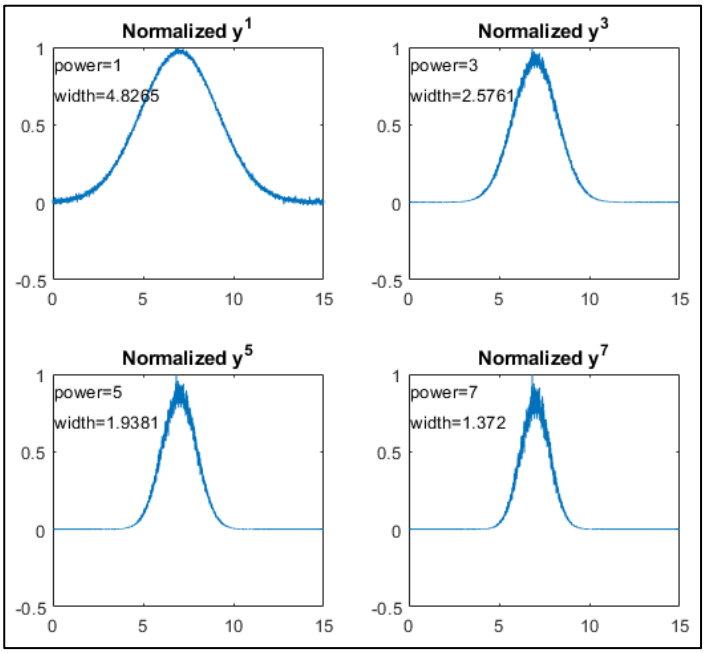
A useful property of *all* of these derivative addition algorithms is that they do not change the *total* area under the peaks, because the *total* area under the curve of *any* derivative of any peak-shaped signal that returns to the baseline is essentially *zero* (the area under the negative lobes cancels the area under the positive lobes). Therefore, these techniques can be helpful in measuring the areas under overlapped peaks (page 132).

However, a remaining problem is that the baseline on either side of the sharpened peak may not be *perfectly flat*, leaving some interference from nearby peaks, even if baseline resolution of adjacent peaks is achieved. For the even-derivative technique applied to a Gaussian peak, about 99.7% (graphic link) of the area of the peak is contained in the central maximum, and for a Lorentzian peak, about 80% of the area of the peak (graphic link) is contained in the central maximum.

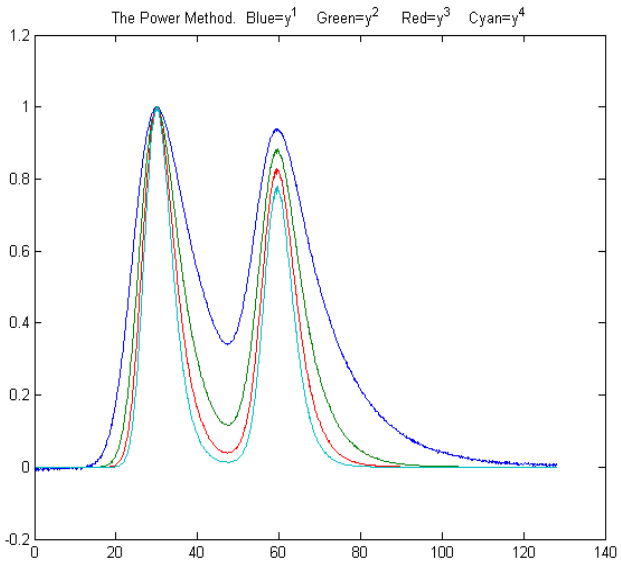
Because differentiation and smoothing are both linear techniques, the superposition principle applies and the amplitude of a symmetrized or sharpened signal is directly proportional to the amplitude of the original signal, which allows standards-based quantitative analysis applications employing any of the standard calibration techniques (page 454). But it is *essential* that you apply the *same signal-processing techniques to the standards as well as to the samples* and measure the signals in the same way.

Peak sharpening can be useful in automated peak detection and measurement (page 234) to increase the ability to detect weak overlapping peaks that appear only as shoulders in the original signal. If you are reading this online, click for an animated example. Peak sharpening can also be useful before measuring the areas (page 145) under overlapping peaks, because it is easier and more accurate to measure the areas of peaks that are more completely separated.

The Power Law Method



A very simple method of peak sharpening involves raising each data point to a power ***n*** greater than 1 (references 61, 63). The effect of this is to change the peak shapes, essentially stretching out the highest center region of the peak to greater amplitudes and placing more weight on the points near the peak, resulting in a *smaller peak width.* For Gaussian peaks specifically, the result is another Gaussian with a width reduced by the square root of the power ***n***. The technique is demonstrated by the Matlab/ Octave script PowerLawDemo.m, shown in the figure on the right, which plots noisy Gaussians raised to the power p=1 to 7, with their peak heights normalized to 1.0, showing that as the power increases, peak width decreases and noise is reduced on the baseline but increased on the peak maximum. Since this process does not move the positions of the peaks, the peak resolution (defined as the ratio of peak separation to peak base width) is increased. For Gaussian peaks, the area under the original peak can be calculated from the area under the normalized power-sharpened curve (reference 63).



In the figure on the left, the blue line shows two slightly overlapping EMG (exponentially modified Gaussian) peaks. The other lines are the result of raising the data to the power of ***n*** = 2, 3, and 4 and normalizing each to a height of 1.00. The results are more nearly Gaussian peak shapes (only because most peak shapes are locally Gaussian near the peak maximum), and the peak widths, measured with the halfwidth.m function, are reduced: 19.2, 12.4, 9.9, and 8.4 units for powers 1 through 4, respectively. This method is independent of, and can be used in conjunction with, the other sharpening methods discussed above. However, for a signal of two overlapping Gaussians, the result of raising the signal to a power is not the same as adding two power-narrowed Gaussians: simply, a^n+b^n is not the same as $(a+b)^n$ for $n>1$. This can be demonstrated graphically by the script PowerPeaks.m (graphic), which curve-fits a two-Gaussian model to the power-raised sum of two overlapping Gaussians. As the power *n* increases, the peaks are narrowed and the valley between them is deepened, but the resulting signal is no longer the sum of two Gaussians unless the resolution is sufficiently high that the two peaks do not overlap significantly.

Some limitations to the power-law method are:

(a) It only works if the peaks of interest make a distinct maximum (it is not effective for side peaks that are so small that they only form *shoulders*; there *must* be a valley between the peaks).

(b) The baseline must be zero for best results.

(c) For noisy signals there is a decrease in signal-to-noise ratio because the smaller width means fewer data points are contributing to the measurement (smoothing, page 40, can help).

Compensating for the non-linearity. Naturally, the power method introduces severe non-linearity into the signal, changing the ratios between peak heights (as is evident in the previous figure) and complicating further processing, especially quantitative measurement calibration. But there is an easy way to compensate for this: after the raw data have been raised to the power n and peaks heights and/or areas have been measured, the resulting peak measures can be simply raised to the power $1/n$, restoring the original linearity (*but not the slope*) of the calibration curves used in quantitative analytical measurements. (This works because the peak area is proportional to the height times width, and peak height of the power transformed peaks is proportional to the nth power of the original height, but the width of the peak is not a function of peak height at constant n, thus the area of the transformed peaks remains proportional to nth power of the original height). The technique is demonstrated quantitatively for two variable overlapping peaks by the Matlab/Octave script PowerLawCalibration-Demo.m (graphic), which takes the nth power of the overlapping-peak signal, measures the areas of the power-narrowed peaks, and then takes the $1/n$ power of the measured areas, constructing and using a calibration curve to convert areas to concentration. Peak areas are measured by perpendicular drop, using the half-way point to mark the boundary between the peaks. The script simulates a mixture signal with concentrations that you can specify in lines 15 and 16. You can change the power and any of the parameters in lines 14-22. The results show that the power method improves the accuracy of the measurements as long as the 4-sigma resolution (the ratio of peak separation to 4 times the standard deviation of the Gaussians) is above about 0.4. It is most accurate when the peaks are roughly equal in width and when the ratio of the two concentrations is not very different from the ratio in the standards from which the calibration curve is constructed. Note that, even when the simulated random noise (in line 22) is zero, the results are not perfect because of the effect of peak overlap on area measurement, which varies depending upon the ratio of two components in the mixture.

The self-contained function PowerMethodDemo.m demonstrates the power method for measuring the area of small shouldering peak that is partly overlapped by a much stronger interfering peak (graphic). It shows the effect of random noise, smoothing, and any uncorrected background under the peaks.

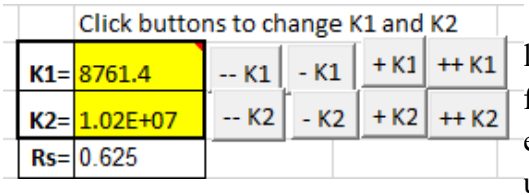
Combining sharpening methods. The power method is independent of, and can be used in conjunction with, the derivative methods discussed above. However, because the power method is non-linear, the *order in which the operations are* performed is important. The *first* step should be the first-derivative symmetrization if the signal is exponentially broadened, the *second* step should be even-derivative sharpening, and the power method should be used *last*. The reason for this order is that the power method depends on whether there is a valley between the peaks *but cannot create one*, whereas the derivative methods *may* be able to create a valley between peaks if the overlap is not too severe. Moreover, when used last, the power method reduces the severity of baseline oscillations that are a

residue of the even-derivative sharpening (particularly noticeable on a Lorentzian peak). The Matlab/Octave scripts SharpenedGaussianDemo2.m (Graphic) and SharpenedLorentzianDemo2.m (Graphic) make this point for Gaussian and Lorentzian peaks respectively, comparing the result of even-derivative sharpening alone with even-derivative sharpening followed by the power method (and preforming the power method two ways, taking the square of the sharpened peak or multiplying it by the original peak). For both the Gaussian and Lorentzian original peak shapes, the final sharpened results are fit to Gaussian models to show the changes in peak parameters. The result is that the combination of methods yields (a) the narrowest final peak and (b) the closest to Gaussian final shape. Of course, the linearity issues of the power method remain, but they can be compensated as before.

Deconvolution. Another signal processing technique that can increase the resolution of overlapping peaks is *deconvolution,* which will be covered on page 114. It is applicable the situations where the original shape of the peaks has been broadened and/or made asymmetrical by some broadening process or function. If the broadening process can be described mathematically or measured separately, then deconvolution from the observed broadened peaks is in principle capable of extracting the shape of the underlying peak.

Peak Sharpening for Excel and Calc Spreadsheets

The even-derivative sharpening method with two derivative terms (2nd and 4th) is available for Excel and Calc in the form of an empty template (PeakSharpeningDeriv.xlsx and .ods) or with example data entered (PeakSharpeningDerivWithData.xlsx and .ods). You can either type in the values of the derivative weighting factors K1 and K2 directly into cells **J3** and **J4**, or you can enter the estimated peak width (FWHM in number of data points) in cell **H4** and the spreadsheet will calculate K1 and K2. There is also a demonstration version with adjustable simulated peaks which you can experiment with (PeakSharpeningDemo.xlsx and PeakSharpeningDemo.ods).



There are also versions that have clickable buttons (detail on left) for convenient interactive adjustment of the K1 and K2 factors by 1% or by 10% for each click. You can type in first estimates for K1 and K2 directly into cells J4 and J5 and then use the buttons to fine-tune the values. If the signal is noisy, adjust the smoothing using the 17 coefficients in row 5 columns **K** through **AA**, just as with the smoothing spreadsheets (page 53). (Note: Unfortunately, these ActiveX buttons do not work in the iPad version of Excel).

There is also a "segmented" template version where the sharpening constants can be specified for each of 20 signal segments (SegmentedPeakSharpeningDeriv.xlsx). For those applications in which the peak widths gradually increase (or decrease) with time, there is also a *gradient* peak sharpening template where you need only set the starting and ending peak widths and the spreadsheet will apply the required sharpening factors K1 and K2. (GradientPeakSharpeningDeriv.xlsx) and an example with data already entered (GradientPeakSharpeningDerivExample.xlsx);

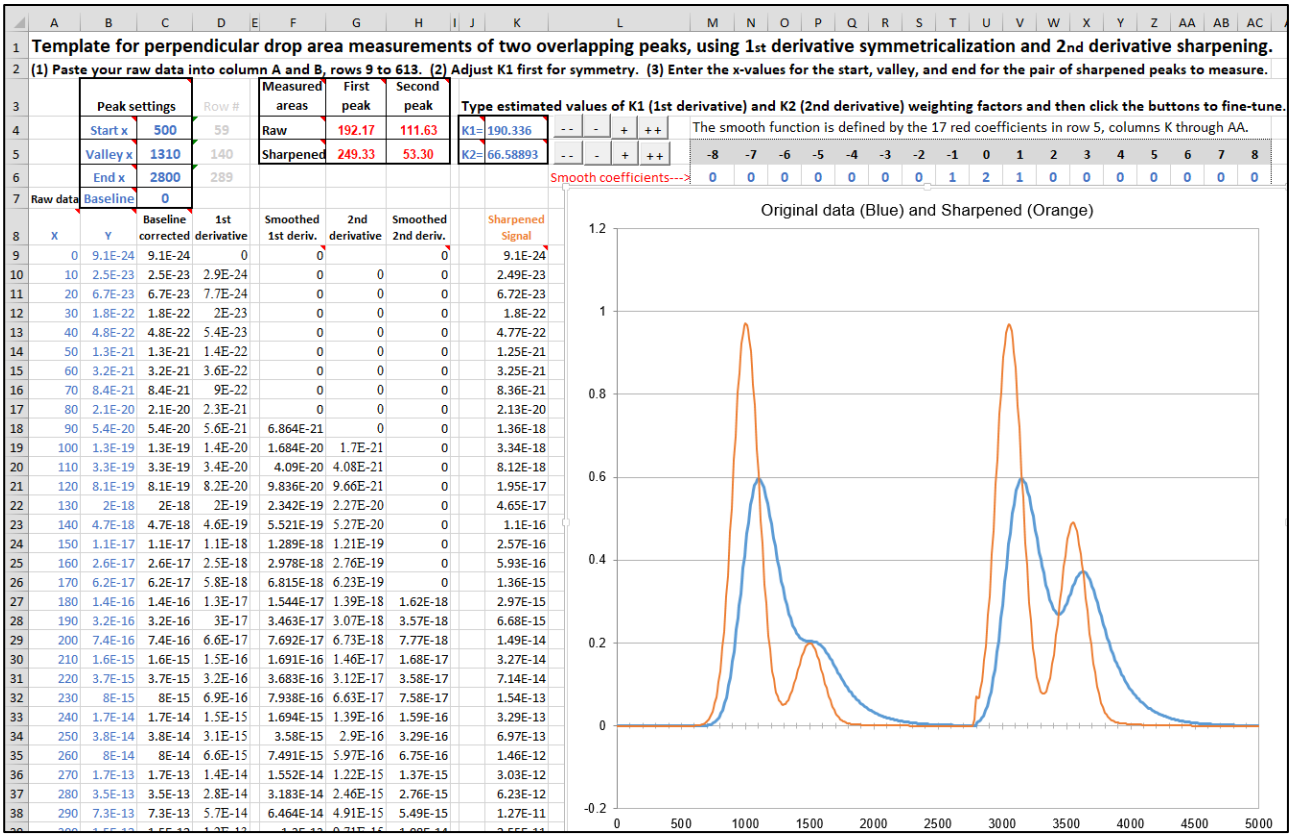
The template PeakSymmetricalizationTemplate.xlsm (screen image on the next page) performs symmetrization of exponentially modified Gaussians (EMG) by the weighted addition of the first derivative. PeakSymmetricalizationExample.xlsm is an example application with sample data already

typed in. It is shown on the next page.

There is also a demo version that allows you to determine the accuracy of the technique by synthesizing overlapping peaks with specified resolution, asymmetry, relative peak height, noise, and baseline: PeakSharpeningAreaMeasurementEMGDemo2.xlsm (graphic). These spreadsheets also allow further second derivative sharpening of the resulting symmetrical peak.

PeakDoubleSymmetrizationExample.xlsm performs the symmetrization of a doubly exponential broadened peak. It has buttons to interactively adjust the two first-derivative weightings. Two variations (1, 2) include data for two overlapping peaks, for which the areas are measured by perpendicular drop.

EffectOfNoiseAndBaselineNormalVsPower.xlsx demonstrates the effect of the power method on area measurements of Gaussian and exponentially broadened Gaussian peaks, including the different effects that random noise and non-zero baseline has on the power sharpening method.



The spreadsheet template "PeakSymmetricalizationTemplate.xlsm" is shown measuring the areas of the first of two pairs of overlapping asymmetrical peaks after applying first derivative symmetrization.

Peak Sharpening for Matlab and Octave

The custom Matlab/Octave function sharpen has the form

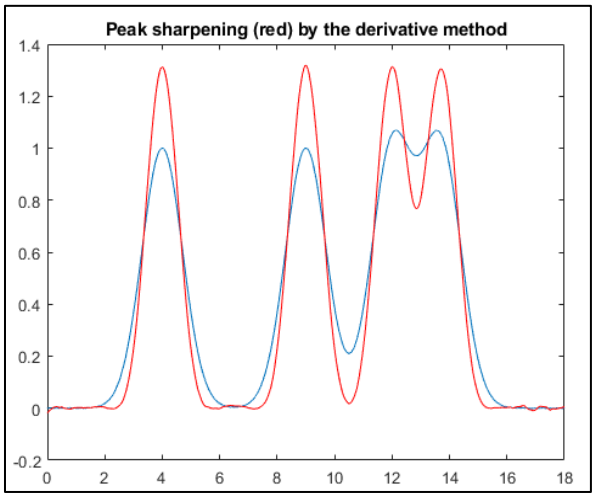
`SharpenedSignal = sharpen(signal,k1,k2,SmoothWidth)`,

where "signal" is the original signal vector, the arguments k2 and k4 are 2nd and 4th derivative weighting factors, and SmoothWidth is the width of the built-in smooth. The resolution-enhanced signal is returned in the vector "SharpenedSignal ". If you are reading this online, you can click on the

link above to inspect the code, or right-click to download for use within Matlab or Octave. The **k** values determine the trade-off between peak sharpness and baseline flatness. The values vary with the peak shape and width and should be adjusted for your own needs. For peaks of Gaussian shape, a reasonable value for k_2 is PeakWidth^2/32 and for k_4 is PeakWidth^4/900 (or PeakWidth^2/8 and PeakWidth^4/700 for Lorentzian peaks), where PeakWidth is the full width at half maximum of the peaks in x units. Because sharpening methods are typically sensitive to random noise in the signal, it is usually necessary to apply a smoothing operation: the Matlab/Octave ProcessSignal.m function allows both sharpening and smoothing to be applied in one function.

Here is a simple Matlab/Octave example that creates a signal consisting of four partly overlapping Gaussian peaks of equal height and width, applies both the derivative sharpening method and the power method, and compares a plot (shown below) comparing the original signal (in blue) to the resolution-enhanced version (in red).

```
x=0:.01:18;
y=exp(-(x-4).^2)+exp(-(x-9).^2)+exp(-(x-12).^2)+exp(-(x-13.7).^2);
y=y+.001.*randn(size(x));
k1=1212;k2=1147420;
SharpenedSignal=ProcessSignal(x,y,0,35,3,0,1,k1,k2,0,0,0,0);
figure(1)
plot(x,y,x,SharpenedSignal,'r')
title('Peak sharpening (red) by the derivative method')
figure(2)
plot(x,y,x,y.^6,'r')
title('Peak sharpening (red) by the power method')
```



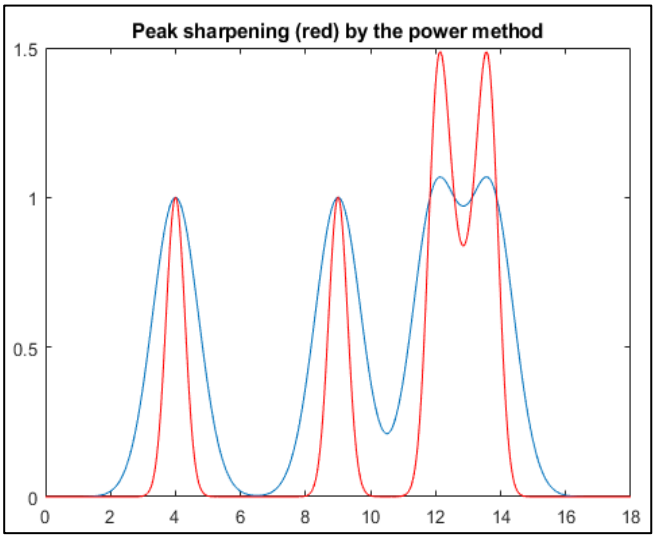
Four overlapping Gaussian peaks of equal height and width.
Blue: Original. Red: After sharpening by the even-derivative method.

SharpenedOverlapDemo.m is a script that attempts to *automatically determines the optimum degree of even-derivative sharpening* that minimizes the errors of measuring peak areas of two overlapping Gaussians by the perpendicular drop method using the autopeaks.m function. It does this by applying different degrees of sharpening and plotting the area errors (percent difference between the true and

measured errors) vs the sharpening factor. It also shows the height of the valley between the peaks as a yellow line. This shows that:

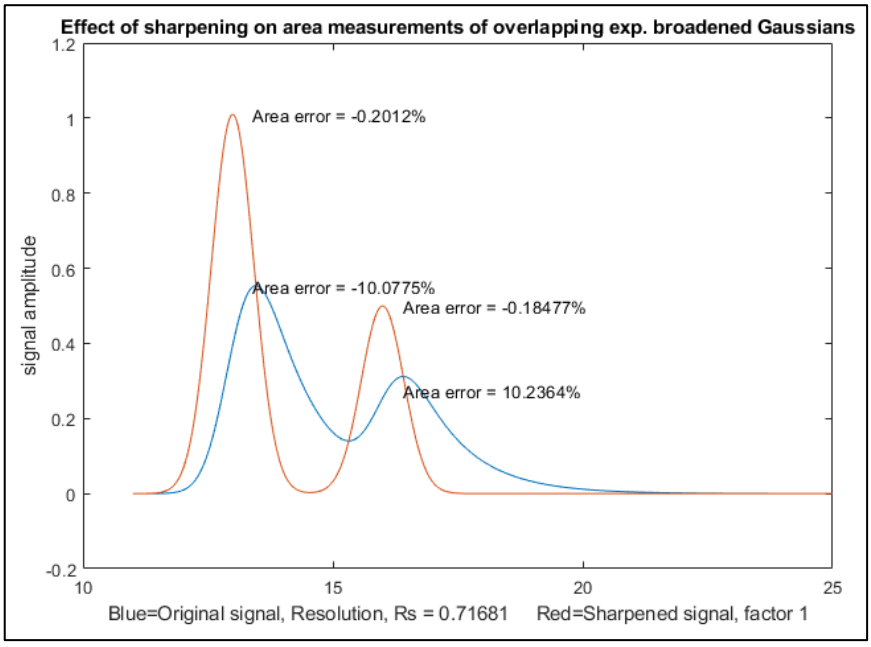
(1) the optimum sharpening factor depends upon the width and separation of the two peaks and on their height ratio,
(2) the degree of sharpening is not overly critical, often exhibiting a broad optimum region,
(3) the optimum for the two peaks is not necessarily the same, and
(4) the optimum for area measurement might not occur at the point where the valley is zero.

(To run this script, you must have gaussian.m, derivxy.m, autopeaks.m, val2ind.m, and halfwidth.m in the Matlab search path. Download these from https://terpconnect.umd.edu/~toh/spectrum/).



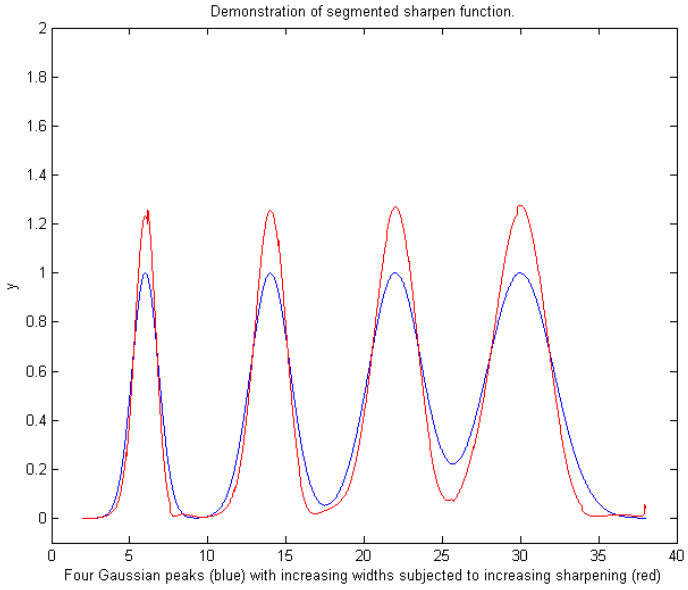
The power method is effective as long as there is a valley between the overlapping peaks, but it introduces non-linearity, which must be corrected later, whereas the derivative method preserves the original peak areas and the ratio between the peak heights. PowerLawCalibrationDemo demonstrates the linearization of the power transform calibration curves for two overlapping peaks by taking the nth power of data, locating the valley between them, measuring the areas by the perpendicular drop method (page 142), and then taking the 1/n power of the measured areas (graphic).

Constant-area symmetrization (de-tailing) of asymmetric peaks by the weighted addition of the first derivative is performed by the function ySym = symmetrize(t,y,factor,smoothwidth,type,ends). "t" and "y" are independent variable and dependent variable vectors. "factor" is the first derivative weighting factor. "smoothwidth", "type", and "ends" are the SegmentedSmooth parameters for the internal smoothing of the derivative. To perform a segmented symmetrization, "factor" and "smoothwidth" can be vectors. In version 2, symmetrize.m smooths only the derivative, not the entire signal. SymmetrizeDemo.m runs all five examples in the symmetrize.m help file, each in a different figure window. If the tau is *not* known, it can be determined *for a single isolated peak* by using

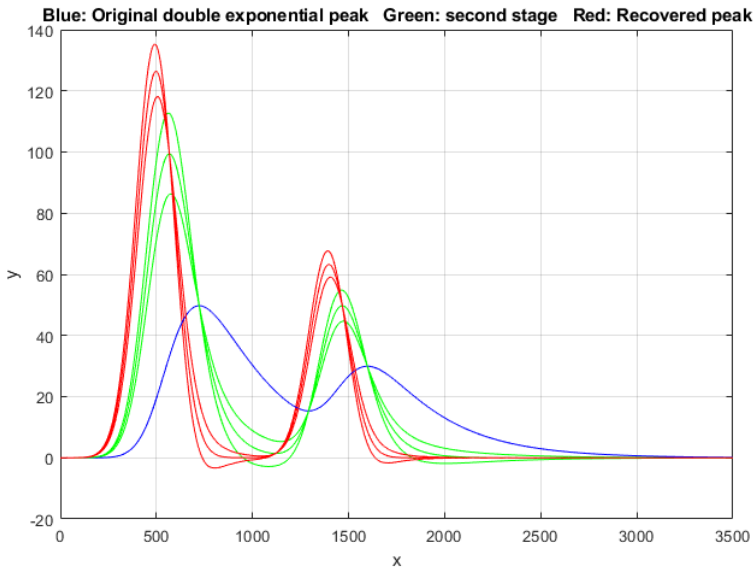


AutoSymmetrize(t, y, SmoothWidth, plots), which finds the value of tau that produces the most symmetrical peak, judged by comparing the slope of the tangents to the leading and trailing edges. In the example shown above, the original peak (blue line) is a mathematically calculated exponentially modified Gaussian with a tau value of 100 and the red line is the output generated by AutoSymmetrize, which estimates the tau to an accuracy of 1%. Type "help AutoSymmetrize". The areas of the two are equal within 0.01%. SymmetizedOverlapDemo.m demonstrates the optimization of the first derivative symmetrization for the area measurement of two overlapping exponentially broadened Gaussians.

Segmented even-derivative peak sharpening. If the peak widths or the noise variance changes substantially across the signal, you can use the *segmented* version SegmentedSharpen.m, for which the input arguments factor1, factor2, and SmoothWidth are *vectors*. The script DemoSegmentedSharpen.m, shown on the right, uses this function to sharpen four Gaussian peaks with gradually increasing peak widths from left to right with increasing degrees of sharpening, showing that the peak width is reduced by 20% to 22% from to the original. DemoSegmentedSharpen2.m shows four peaks of the *same* width sharpened to increasing degrees.



Double exponential symmetrization in Matlab/Octave is performed by the function DEMSymm.m which applies two successive applications of weighted addition of the first derivative, with weighting factors ideally equal to the two taus. The objective is to make the peaks more symmetrical and narrower while preserving the peak area. A three-level plus-and-minus bracketing technique helps you to determine the best values for the two weighting factors. The technique is demonstrated by the script DemoDEMSymm.m and its two variations (1, 2), which creates two overlapping double exponential peaks from Gaussian originals, then calls the function DEMSymm.m to perform the symmetrization. In the example on the left, the middle bracketing line is the optimum value. In summary, if you attempt to symmetrize an asymmetrical peak by weighted first-derivative addition and the result is still asymmetrical, it may be that the remaining asymmetry could

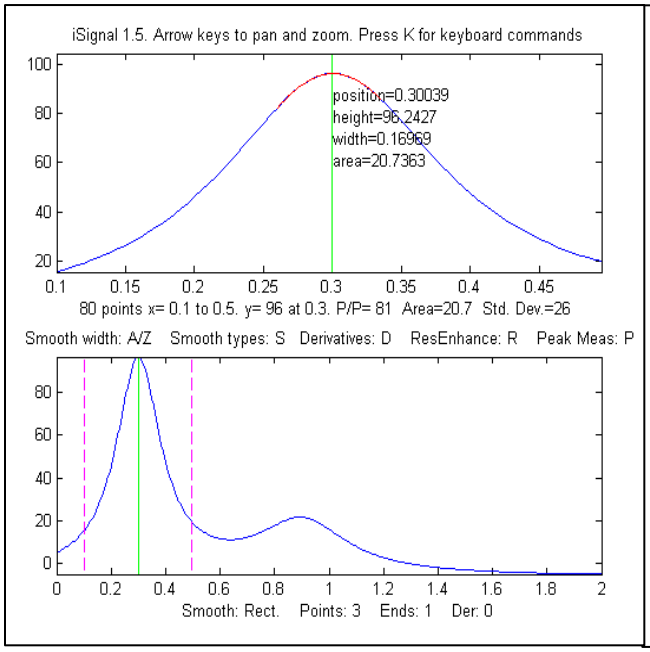


be due to another stage of exponential broadening with a different *tau*, so in that case, the application of DEMSymm.m will likely produce a more symmetrical final result.

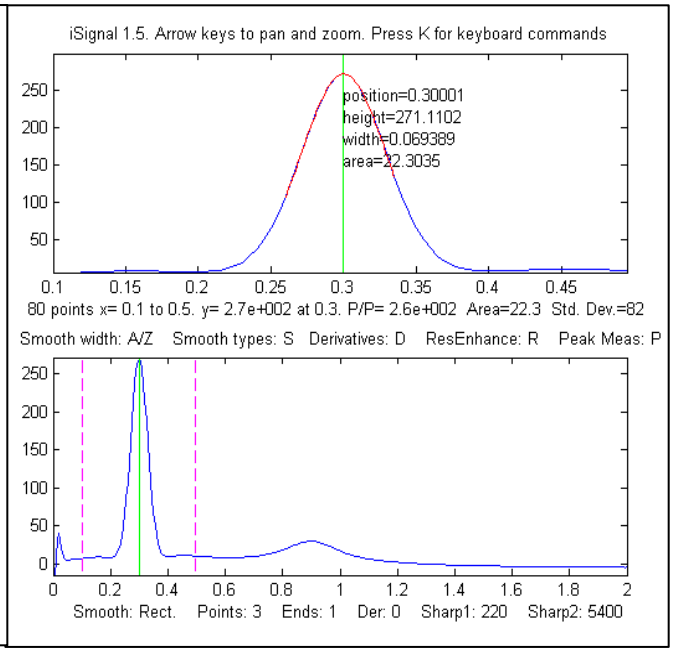
ProcessSignal, a Matlab/Octave command-line function that performs smoothing, differentiation, and peak sharpening on the time-series data set x,y (column or row vectors). Type "help ProcessSignal". It returns the processed signal as a vector that has the same shape as x, regardless of the shape of y.

```
Processed=ProcessSignal(x, y, DerivativeMode, w, type, ends, Sharpen,
factor1, factor2, Symize, Symfactor, SlewRate, MedianWidth)
```

iSignal (Version 8.3, page 376) is a multi-function interactive Matlab function that includes peak sharpening for time-series signals, using *both* the even-derivative method (*sharpen* function) and the first-derivative symmetrization method, with keystrokes that allow you to adjust the derivative weighting factors and the smoothing continuously while observing the effect on your signal dynamically. The **E** key turns the peak sharpening function on and off. View the code here or download the ZIP file with sample data for testing. *iSignal* estimates the sharpening and smoothing settings for Gaussian and for Lorentzian peak shapes using the **Y** and **U** keys, respectively, using the expression given above. Just isolate a single typical peak in the upper window using the pan and zoom keys, press **P** to turn on the peak measurement mode, then press **Y** for Gaussian or **U** for Lorentzian peaks. You can fine-tune the sharpening with the **F/V** and **G/B** keys and the smoothing with the **A/Z** keys. (If your signal has peaks of widely different widths, one setting will not be optimum for all the peaks. In such cases, you can use the segmented sharpen function, SegmentedSharpen.m).

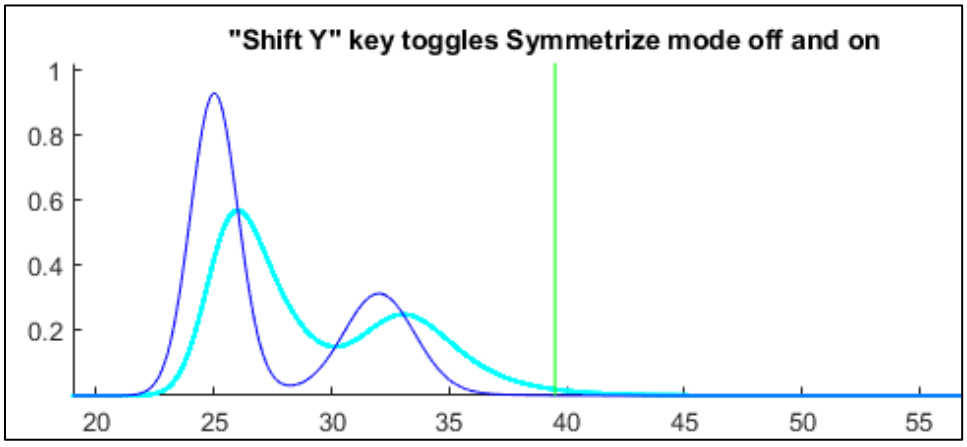


Before peak Sharpening in iSignal



After peak sharpening in iSignal

In *iSignal* and in *iPeak*, the **Shift-Y** key engages the first-derivative symmetrization technique and uses the **1**, **Shift-1**, **2**, and **Shift-2** keys to adjust the weighting factor by 10% or 1% per keypress. The idea is to increase the factor until the baseline after the peak goes negative, then increase it slightly so that it is *as low as possible but not negative*.



iSignal can also use the power transform method (press the ^ key, enter the power *n* (any positive number greater than 1.00) and press **Enter**. To reverse this, simply raise to the 1/*n* power. *iPeak*, (page 253), a Matlab interactive peak detection and measurement program, has a built-in peak sharpening mode that is based on the even derivative technique, as well as the first-derivative symmetrization using the same keystrokes as *iSignal*. See ipeakdemo5 on page 269. The GIF animation demonstrates this in action.

Peak sharpening, both by the even-derivative symmetrization and Fourier self-deconvolution methods, are included as part of the interactive peak detection tool **PeakDetection.mlx** discussed on page 253.

Real-time peak sharpening in Matlab is discussed on page 347.

Harmonic analysis and the Fourier Transform.

Some signals exhibit periodic components that repeat at fixed intervals throughout the signal, like a sine wave. It is often useful to describe the amplitude and frequency of such periodic components exactly. In fact, it is possible to analyze *any* arbitrary set of data into periodic components, whether or not the data appear periodic. Harmonic analysis is conventionally based on the *Fourier transform*, which is a way of expressing a signal as a weighted sum of sine and cosine waves. It can be shown that any arbitrary discretely sampled signal can be described completely by the sum of a finite number of sine and cosine components whose frequencies are 0, 1, 2, 3 ... n/2 times the frequency $f=1/n\Delta x$, where Δx is the interval between adjacent x-axis values and n is the total number of points. The Fourier transform is simply the set of amplitudes of those sine and cosine components (or, which is equivalent mathematically, the frequency and phase of sine components). You could calculate those coefficients yourself simply but laboriously by multiplying the signal point-by-point with each of those sine and cosine components and adding up the products. The famous Cooley-Tukey "Fast Fourier Transform" (FFT) dates from 1965 and is a faster and more efficient algorithm that makes use of the symmetry of the sine and cosine functions and other math shortcuts to get the same result *much* more quickly. The *inverse* Fourier transform (IFT) is a similar algorithm that converts a Fourier transform back into the original signal.

As a mathematical convenience, Fourier transforms are traditionally expressed in terms of "complex numbers", which allows one to combine the sine and cosine (or amplitude and phase) information at each frequency onto a single compact expression, using the identity

$$\cos(2\pi ft) + i\,\sin(2\pi ft) = e^{i2\pi ft}$$

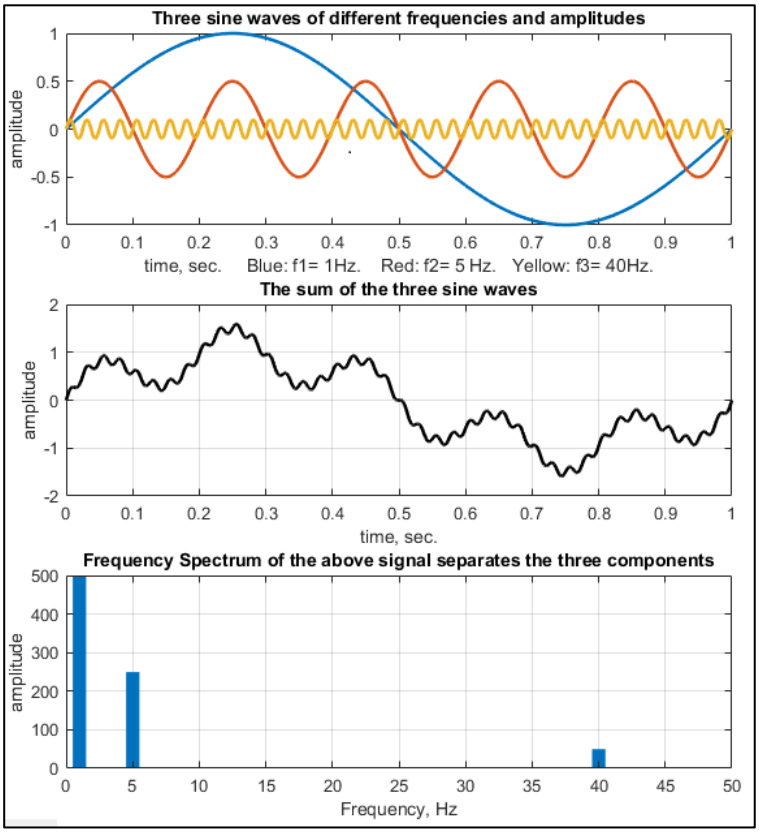
Even for data that are not complex, using the "exp" notation rather than "sin+cos" is surely more *compact and elegant*, and most computer languages can handle complex arithmetic automatically. But this terminology can be misleading because the sine and cosine parts are *equally important*. Just because the two parts are called "real" and "imaginary" in mathematics *does not imply that the first is more significant* than the second. (For a rigorous explanation, see Fourier Transforms by Gary Knott).

The concept of the Fourier transform is involved in two very important modern instrumental methods in chemical analysis. In Fourier transform infrared spectroscopy (FTIR), the Fourier transform of the spectrum is measured directly by the instrument, as the interferogram formed by plotting the detector signal vs mirror displacement in a scanning Michaelson interferometer. In Fourier Transform Nuclear Magnetic Resonance spectroscopy (FTNMR), excitation of the sample by an intense, short pulse of radio-frequency energy produces a free induction decay signal that is the Fourier transform of the resonance spectrum. In both cases, *a computer is used to recover the spectrum* by inverse Fourier transformation of the measured (interferogram or free induction decay) signal.

The *power spectrum* or *frequency spectrum* is a simple way of showing the total amplitude at each of these frequencies. It is calculated as the square root of the sum of the squares of the coefficients of the sine and cosine components. The power spectrum retains the *frequency* information but discards the *phase* information, so that the power spectrum of a sine wave would be the same as that of a cosine wave of the same frequency, even though the complete Fourier transforms of sine and cosine waves are

different in phase. In rare situations where the *phase components* of a signal are the major source of noise (e.g. random shifts in the horizontal x-axis position of the signal), it can be advantageous to base measurement on the power spectrum, which discards the phase information, by ensemble averaging (page 27) the power spectra of repeated signals: this is demonstrated by the Matlab/Octave scripts EnsembleAverageFFT.m and EnsembleAverageFFTGaussian.m.

The Fourier transform is simply the set of amplitudes of those sine and cosine components A time-series signal with *n* points gives a power spectrum with only (n/2)+1 points. The first point is the zero-frequency (constant) component, corresponding to the DC ("direct current") component of the signal. It looks like a straight flat line. The second point corresponds to a frequency of 1/nΔx (whose period is exactly equal to the time duration of the data), the next point to 2/nΔx, the next point to 3/nΔx, etc., where Δx is the interval between adjacent x-axis values and n is the total number of points. The last (highest frequency) point in the power spectrum (n/2)/nΔx=1/2Δx, which is one-half the sampling rate.

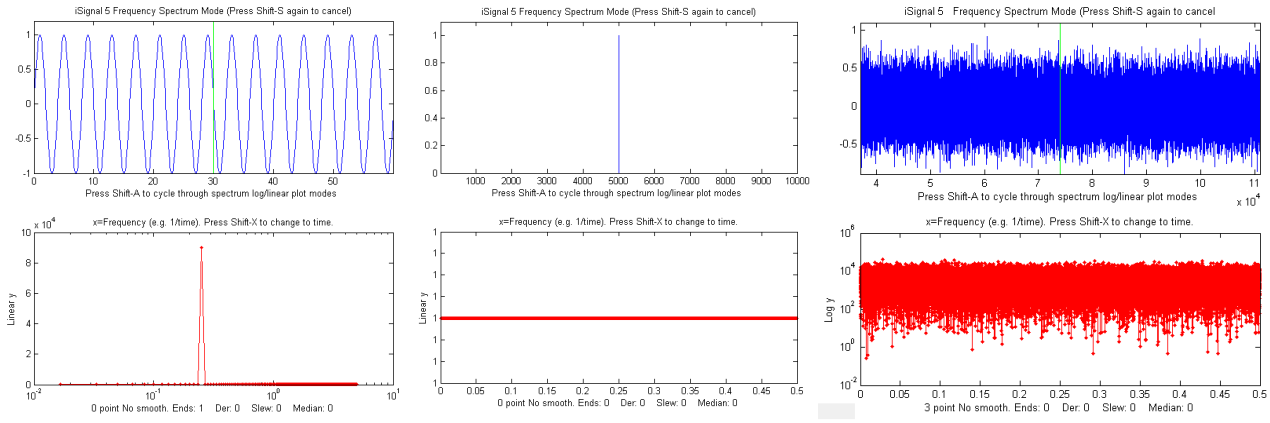


The figure on the left shows a simulated 1000-point signal with one-second duration and a sampling rate of 1000 Hz (middle panel). This signal contains only three sine waves (shown separately in different colors in the top panel), all of which are clearly distinguishable when added up in the signal itself (middle panel). You can even count the cycles of the sine waves to confirm their frequencies. The frequencies all show up at the expected places and with the expected relative amplitudes in the *Fourier amplitude spectrum*, which I have drawn here as a bar graph in the bottom panel(showing frequencies only up to 50 Hz, out of a maximum of 500 Hz). This also works similarly with *cosine* waves, which differ from sine waves only in their *phase* (x-axis shift).

The limits of sampling. The *highest* frequency that can be represented in a discretely sampled waveform is one-half the sampling frequency, which is called the *Nyquist frequency*. In the signal above, the Nyquist frequency is ½ x1000 = 500 Hz. Attempts to digitize analog signals with higher frequencies are "folded back" to lower frequencies, severely distorting the signal. This is called *aliasing*. The frequency resolution, that is, the difference between the frequencies of adjacent points in the calculated frequency spectrum, is simply the reciprocal of the time duration of the signal, 1 Hz.

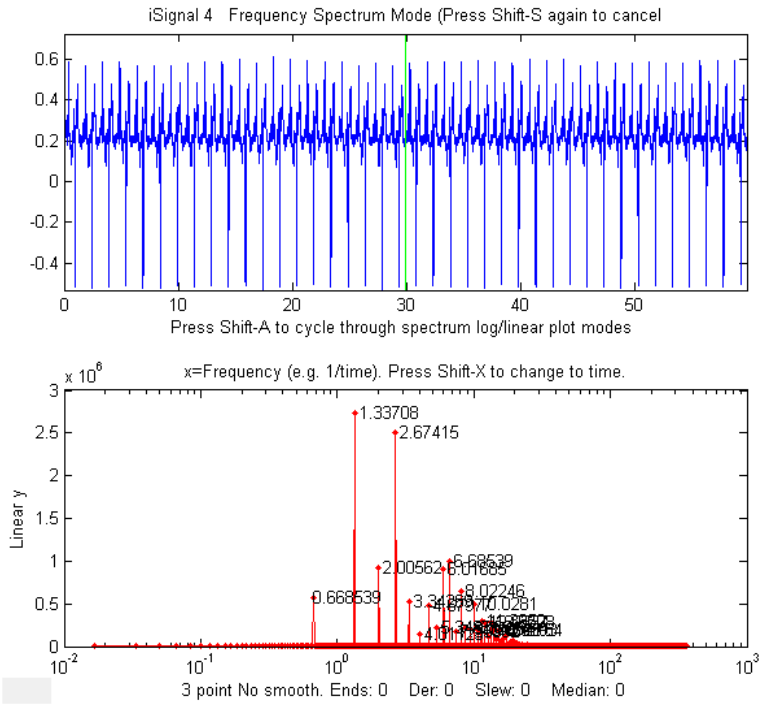
The self-contained Matlab script AliasingDemo.m (graphic) demonstrates the phenomenon of aliasing and frequency folding. It creates a sine wave of a fixed frequency (100 Hz), then samples it repeatedly at gradually decreasing sampling rates, starting at 600 Hz, well above the Nyquist frequency (200 Hz)

and ending at 130 Hz, well below the Nyquist frequency. The running graphic shows that the distortion caused by sampling starts small but increases drastically as the sampling rate approaches 200 Hz, below which the apparent frequency - indicated by the number of peaks counted - decreases.



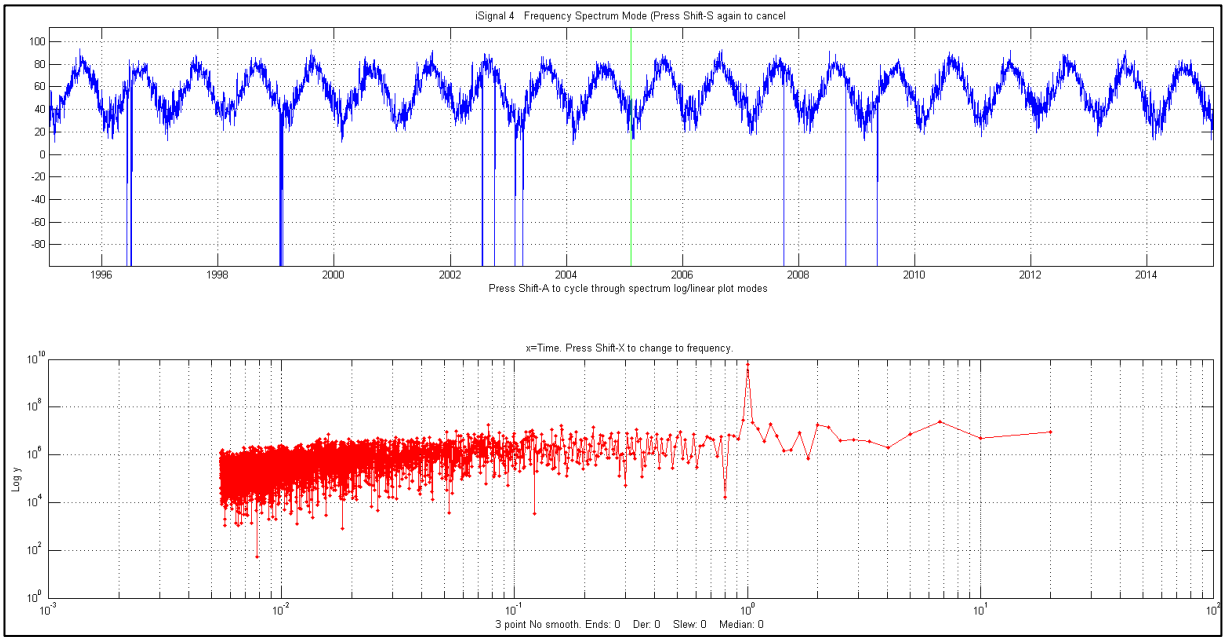
A pure sine or cosine wave that has an exact integral number of cycles within the recorded signal has a single non-zero Fourier component corresponding to its frequency (above left). Conversely, a signal consisting of zeros everywhere except at a single point, called a delta function, has equal Fourier components at all frequencies (previous page, center). Random noise also has a power spectrum that is spread out over a wide frequency range. The noise amplitude distribution depends on the noise color (page 29) with pink noise having more power at low frequencies, blue noise having more power at high frequencies, and white noise having roughly the same power at all frequencies (above right).

Real signal examples. The figure below shows a 60-second recording of a heartbeat, called an electro-cardiograph (ECG), which is an example of a periodic waveform that repeats over time. The figure shows the waveform in blue in the top panel and its frequency spectrum in red in the bottom panel. The smallest repeating unit of the signal is called the *period*, and the reciprocal of that period is called the fundamental frequency. Non-sinusoidal periodic waveforms like this exhibit a series of frequency components that are *multiples of the fundamental frequency*, which are called "harmonics". This spectrum shows a fundamental frequency of 0.6685 Hz, with multiple harmonics at frequencies that are ×2, ×3, ×4..., etc., times the fundamental frequency. The lowest frequency in the spectrum is 0.067 Hz (the reciprocal of the recording time) and the highest is 400 Hz (one-half the sampling rate). The fundamental and the

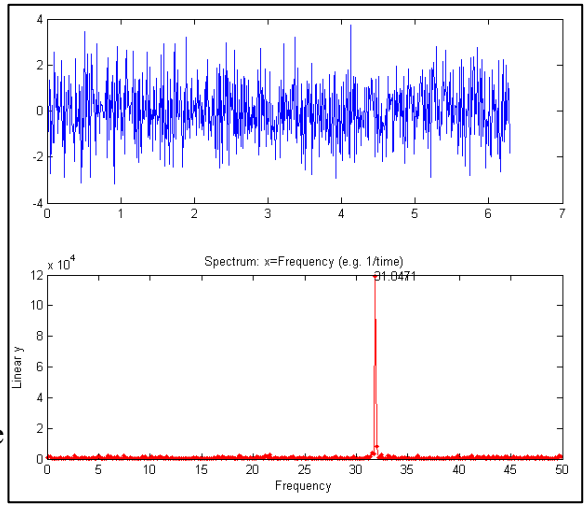


harmonics are *sharp peaks*, and they are labeled with their frequencies on tis graph. The spectrum is qualitatively similar to that for perfectly regular identical peaks (graphic). (Recorded vocal sounds, especially vowels, also have this kind of periodic waveform with harmonics (graphic)). The sharpness of the peaks in the electrocardiogram shows that the amplitude and the frequency are very constant over the 60 second recording interval in this example (which is normal behavior for a healthy heart). Changes in amplitude or frequency over the recording interval will produce *clusters* or *bands* of Fourier components rather than sharp peaks, as in the example on page 303.

Another familiar example of stable periodic oscillation is the seasonal variation in temperature, for example, the average daily temperature measured in New York City between 1995 and 2015, shown in the figure below. This signal exhibits obvious periodicity, except for the sharp negative spikes (which are due to missing data points – perhaps local power outages). *Note the logarithmic scale* on the y-axis of the spectrum in the bottom panel. This spectrum covers a *very* wide range of amplitudes.



In this example, the spectrum in the lower panel, in red, is plotted with *time* (the reciprocal of frequency) on the x-axis. This is called a *periodogram*. Despite the considerable random noise due to local weather variations and missing data, this shows the expected peak at exactly 1 year. That peak is *over 100 times stronger* than the background noise and is very *sharp* because the periodicity is extremely precise (in fact, it is literally *astronomically* precise). In contrast, the random noise is *not* periodic but rather is spread out roughly equally over the entire periodogram.

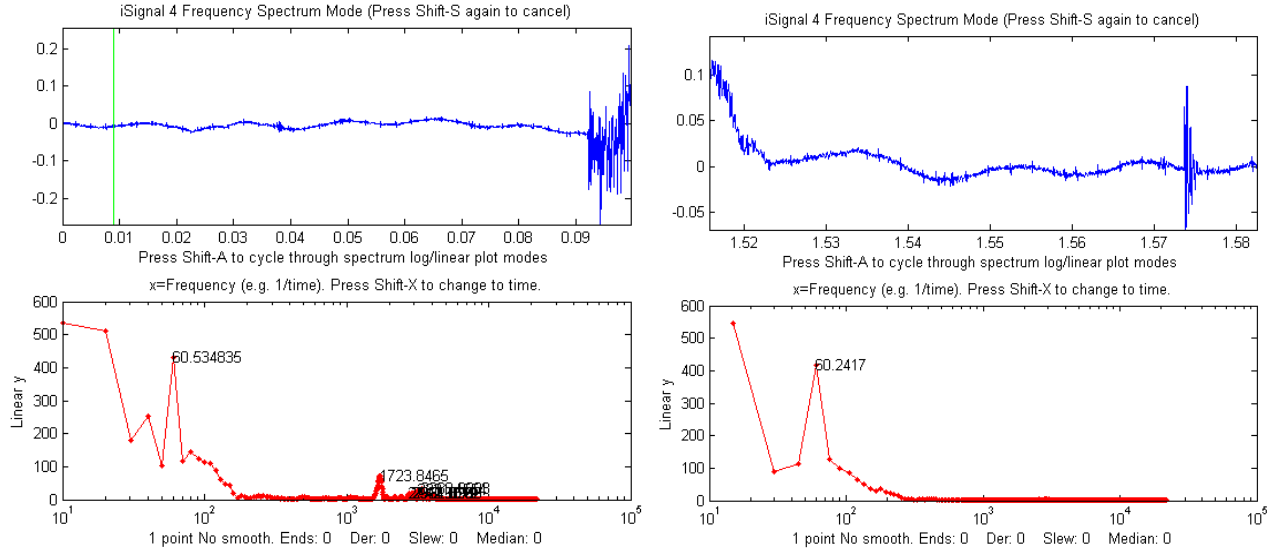


The figure on the right shows some simulated data that demonstrates how hard it is to see a periodic component in the presence of random noise, and yet how easy it is to pick it out in the frequency spectrum. In this example, the signal (top panel) contains an *equal mixture* of random

white noise and a single sine wave. The sine wave is almost completely obscured by the random noise. The frequency spectrum (created using my Matlab/Octave function "PlotFrequencySpectrum") is shown in the bottom panel. The frequency spectrum of the white noise is spread out evenly over the entire spectrum, whereas the sine wave is concentrated into a *single* spectral element, where it stands out clearly. Here is the Matlab/ Octave code that generated that figure. You can Copy and Paste it into Matlab/Octave:

```
x=[0:.01:2*pi]';
y=sin(200*x)+randn(size(x));
subplot(2,1,1);
plot(x,y);
subplot(2,1,2);
PowerSpectrum=PlotFrequencySpectrum(x,y,1,0,1);
```

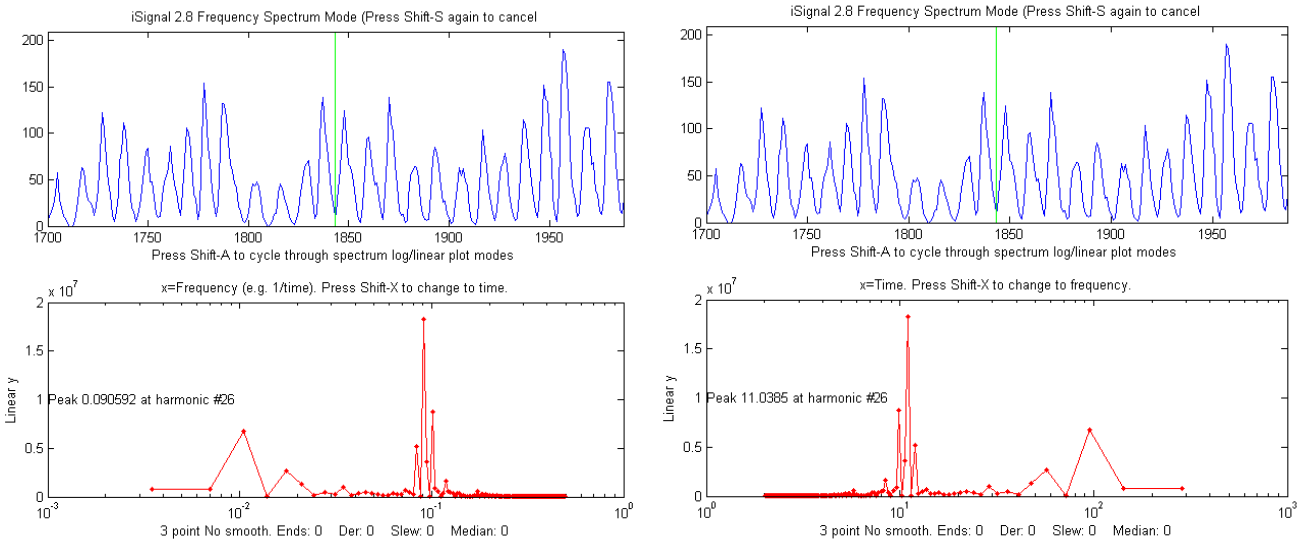
A common practical application is the use of the power spectrum as a diagnostic tool to distinguish between signal and noise components. An example is the AC power-line pickup depicted in the figure below, which has a fundamental frequency of 60 Hz in the USA (why that frequency?) or 50 Hz in many other countries. Again, the sharpness of the peaks in the spectrum shows that the amplitude and the frequency are very constant. Power companies take pains to keep the frequency of the AC very constant to avoid problems between different sections of the power grid. Other examples of signals and their frequency spectra are shown below.



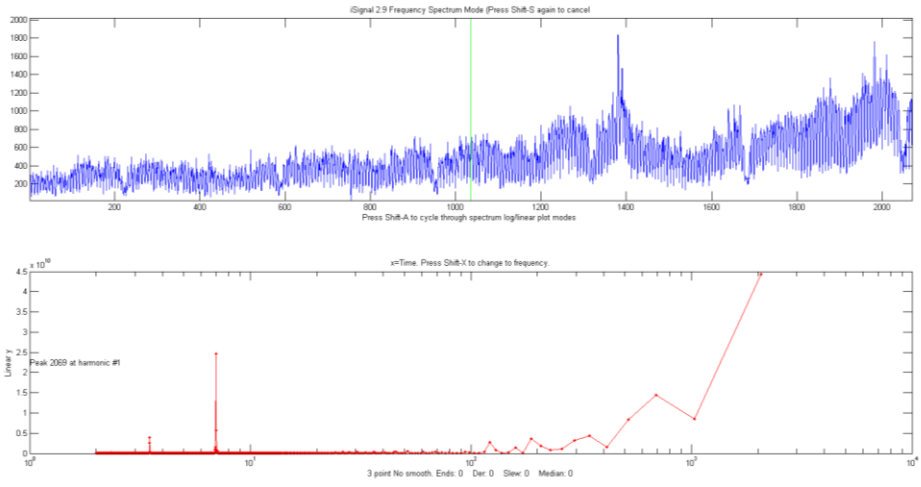
***iSignal**, showing data from an audio recording, zoomed in to the "quiet" period immediately before (left) and after (right) the actual sound. This shows there is a residual sinusoidal oscillation in those periods (x = time in seconds). In the lower panel, the power spectrum of each signal (x = frequency in Hz) shows a strong sharp peak very near 60 Hz, suggesting that this oscillation is caused by stray pick-up from the 60 Hz power line (since it was recorded in the USA. Had the recording been made in Europe, it would be 50 Hz). Improved shielding and grounding of the equipment might reduce this interference. The "before" spectrum, on the left, has a frequency resolution of only 10 Hz (the reciprocal of the recording time of about 0.1 seconds) and it includes only about 6 cycles of the 60 Hz frequency (which is why that peak in the spectrum is the 6th point). To achieve a better resolution you*

would have had to have begun the recording earlier, to achieve a longer recording. The "after" spectrum, on the right, has an even shorter recording time and thus a poorer frequency resolution.

Peak-type signals have power spectra that are concentrated in a range of low frequencies, whereas random noise often spreads out over a much wider frequency range. This is the reason smoothing (low-pass filtering) can make a noisy signal *look* nicer, but also why smoothing does not usually help with quantitative measurement, because most of the peak information is found at *low* frequencies, where low-frequency noise remains unchanged by smoothing (See page 44).



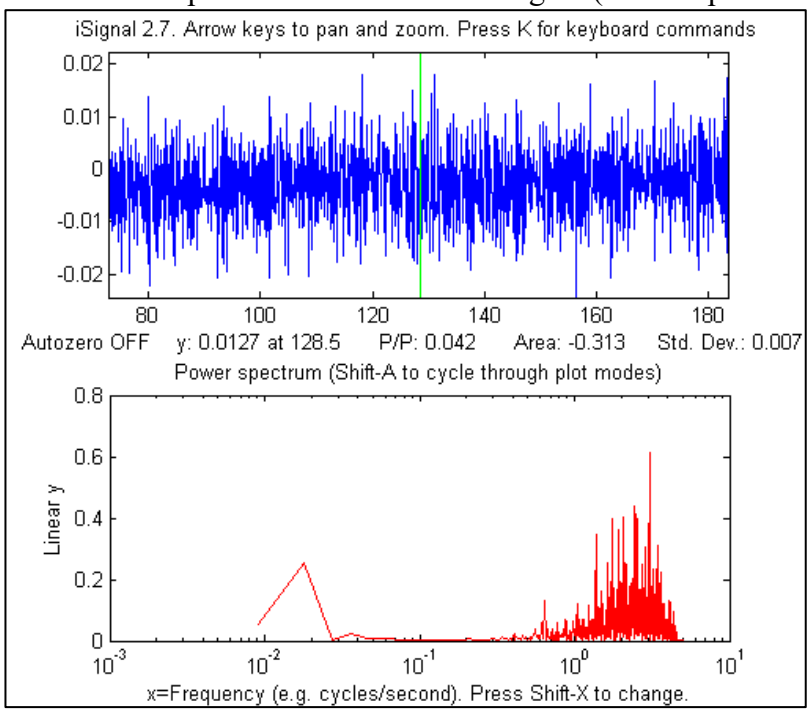
The figures above show a classic example of harmonic analysis. It shows the annual variation in the number of sunspots observed, which have been recorded annually since the year 1700! In this case, the time axis is in *years* (top window). A plot of the power spectrum (bottom window, left) shows a strong peak at 0.09 cycles/year, and the periodogram (right) shows a peak at the well-known 11-year cycle, plus some evidence of a weaker cycle at around a 100-year period. (You can download this data set or the latest yearly sunspot data from NOAA. These frequency spectra are plotted using my Matlab function iSignal (page 376). In this case, the peaks in the spectrum are *not* sharp single peaks, but rather form a *cluster* of Fourier components, because *the amplitude and the frequency are not constant* over the nearly 300-year interval of the data, as is obvious by inspecting the data in the time domain. The strong solar flares observed in 2024 fall on the next predicted sunspot maximum and also on a maximum in the solar radio-frequency emissions, which effects communications and power grids.



An example of a time series with complex multiple periodicities is the world-

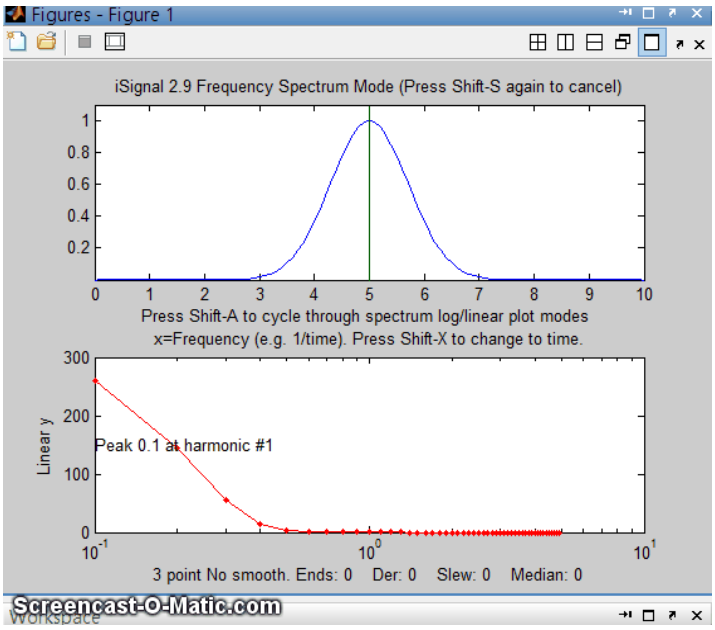
wide daily page views (x=days, y=page views) for this web site over a 2070-day period (about 5.5 years). In the periodogram plot (shown here) you can clearly see sharp peaks at 7 and 3.5 days, corresponding to the first and second harmonics of the expected workday/weekend cycle. It also shows smaller peaks at 365 days (corresponding to a sharp dip each year during the winter holidays in the northern hemisphere) and at 182 days (roughly a half-year), perhaps caused by increased use in the two-per-year semester cycle at universities. The large values at the longest times are caused by the gradual increase in use over that time period, which can be thought of as a very low-frequency component whose period is much longer than the entire data record.

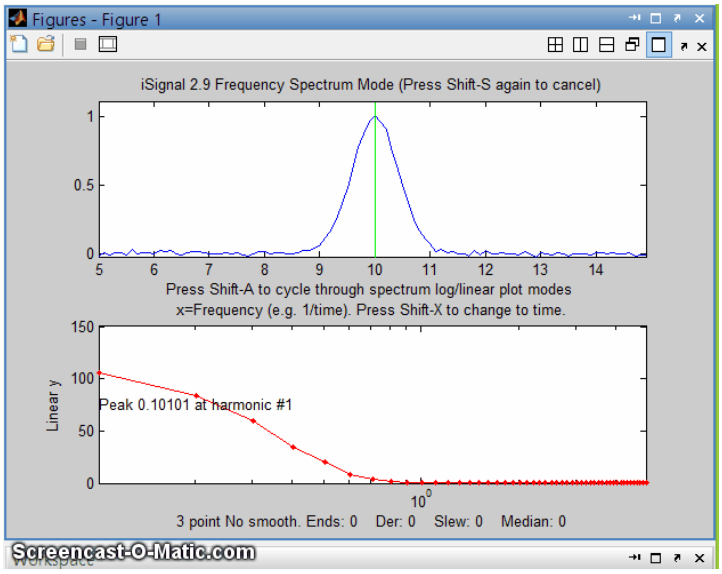
Another example is shown below. The signal (in the top window) contains no visually evident periodic components. It *seems* to be just random noise. However, the frequency spectrum (in the bottom window) shows that there is much more to this signal than meets the eye. There are two major frequency components: one at low frequencies around 0.02 and the other at high frequencies between 0.5 and 5. (If the x-axis units of the signal plot had been *seconds*, the units of the frequency spectrum plot would be *Hz*. Note that the x-axis is logarithmic). In this case, the *lower* frequency component is, in fact, the *signal*, and the frequency component is residual *blue noise* remaining from previous signal processing operations. The two components are fortunately well separated on the frequency axis, suggesting that low-pass filtering (i.e., smoothing, page 42) will be able to remove the noise without distorting the signal.



In all the examples shown above, the signals are time-series signals with *frequency* (or *time*) as the independent variable. More generally, it is possible to compute the Fourier transform and power spectrum of *any* signal, such as an *optical spectrum*, where the independent variable might be wavelength or wavenumber, or an *electrochemical signal*, where the independent variable might be volts, or a *spatial signal*, where the independent variable might be in length units. In such cases, the units of the x-axis of the power spectrum are simply the reciprocal of the units of the x-axis of the original signal (e.g., nm^{-1} for a signal whose x-axis is in nm).

Analysis of the frequency spectra of signals provides another way to understand signal-to-noise ratio, filtering, smoothing, and differentiation. Smoothing is a form of *low-pass* filtering, reducing the high-frequency components of a signal. If a signal consists of smooth features, such as Gaussian peaks, then its spectrum will be concentrated mainly at *low* frequencies. The wider the width of the peak, the more concentrated the frequency spectrum will be at low frequencies. (If the figure below does not animate, click this link). A signal that has white noise (spread out evenly over all frequencies), then smoothing will make the signal look better, because it reduces the high-frequency components of the noise. However, the low-frequency noise will remain in the signal after smoothing, where it will continue to interfere with the measurement of signal parameters such as peak heights, positions, widths, and areas. This can be demonstrated by a least-squares measurement.





Conversely, differentiation is a form of *high-pass* filtering, reducing the *low*-frequency components of a signal and emphasizing any *high*-frequency components present in the signal. A simple computer-generated Gaussian peak shown above (click for GIF animation) has most of its power is concentrated in just a few low frequencies, but as successive orders of differentiation are applied (in the yellow circle), the waveform of the derivative swings from positive to negative like a sine wave, and its frequency spectrum shifts progressively to higher frequencies. This behavior is typical of any signal with smooth peaks. So, the optimum range for signal information of a *differentiated signal* is restricted to a relatively narrow range, with little useful information above and below that range.

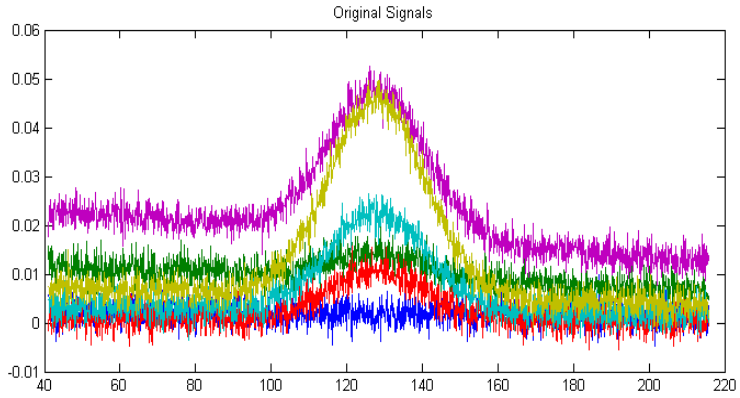
The fact that white noise (page 27) is spread out in the frequency domain roughly equally over all frequencies has a subtle advantage over other noise colors when the signal and the noise cannot be cleanly separated in the time domain. You can more easily estimate the intensity of the noise by observing it in frequency regions where the signal does not interfere, since most signals do not occupy the entire spectrum frequency range. This idea will be used later as a method for estimating the errors of measurements that are based on least-squares curve fitting of noisy data (page 170).

The technique of peak sharpening (page 78) also emphasizes the high-frequency components by adding

a portion of the second and fourth derivatives to the original signal. You can see this clearly in the Matlab/Octave script PeakSharpeningFrequencySpectrum.m, which shows the frequency spectrum of the original and sharpened version of a signal consisting of several peaks (graphic).

SineToDelta.m. A demonstration animation (click for animated graphic) shows the waveform and the power spectrum of a rectangular pulsed sine wave of variable duration (whose power spectrum is a "sinc" function) changing continuously from a pure sine wave at one extreme (where its power spectrum is a delta function) to a single-point pulse at the other extreme (where its power spectrum is a flat line).GaussianSineToDelta.m is similar, except that it shows a *Gaussian* pulsed sine wave, whose power spectrum is a Gaussian function, but which is the same at the two extremes of pulse duration (animated graphic).

Real experimental signals are often contaminated with drift and baseline shift, which are essentially *low-frequency* effects, and random noise, which is usually spread out over *all frequencies*. For these reasons, differentiation is always used in conjunction with smoothing. Working together, smoothing and differentiation act as a kind of frequency-selective *bandpass* filter that optimally passes the band of frequencies containing the differentiated signal information but reduces both the *lower-frequency* effects, such as slowly changing drift and background, as well as the *high-frequency* noise. An example of this can be seen in the DerivativeDemo.m described in a previous section (page 75). In the set of six original signals, shown above in different colors, the random noise occurs mostly in high frequencies, with *many cycles* over the x-axis range, and the baseline shift is a much lower-frequency phenomenon, with only a *small fraction of one cycle* occurring over that range. In contrast, the peak of interest, in the center of the x-range, occupies an *intermediate* frequency range, with *a few cycles* over that range. Therefore, you could predict that a quantitative measure based on differentiation and smoothing might work well, because that emphasizes the intermediate frequencies.



Smoothing and differentiation change the *amplitudes* of the various frequency components of signals, but they do not change or shift the frequencies themselves. An experiment described later (page 394) illustrates this idea by smoothing and differentiating a brief recording of human speech. Interestingly, different degrees of smoothing and differentiation will change the timbre of the voice but has *little effect on the intelligibility*, because the sequence of pitches is not shifted in pitch or time but merely changed in amplitude by smoothing and differentiation. Because of this, recorded speech can survive digitization, transmission over long distances, algorithmic compression, and playback via tiny speakers and headphones without significant loss of intelligibility. Music, on the other hand, suffers greater loss under such circumstances, as you can tell by listening to telephone "hold" music, which often sounds terrible *even though speech over the same connection is very intelligible*, because music has a different frequency structure than speech. Cochlear implants for the hearing impaired have the same limitation (as dramatized in the 2019 movie “The Sound of Metal”).

Software details

In a spreadsheet or computer language, a sine wave can be described by the 'sin' function $y=\sin(2\pi fx+p)$ or $y=\sin(2\pi(1/t)x+p)$, where π is 3.14159...., *f* is *frequency* of the waveform, *t* is the *period* of the waveform, *p* is the *phase*, and *x* is the independent variable (usually time).

There are several Web sites that can compute Fourier transforms interactively (e.g. WolframAlpha). Microsoft Excel has an add-in function that makes it possible to perform Fourier transforms relatively easily: (Click **Tools > Add-Ins... > Analysis Toolpak** > **Fourier Analysis**). See "Excel and Fourier" for details. See "Excellaneous" for an extensive and excellent collection of add-in functions and macros for Excel, courtesy of Dr. Robert deLevie of Bowdoin College. There are several dedicated FFT spectral analysis programs, including **ScopeDSP** (https://iowegian.com/scopedsp/) and **Audacity** (http://sourceforge.net/projects/audacity/). If you are reading this online, you can **Ctrl-Click** these links to open these sites automatically

Matlab and Octave

Matlab and Octave have built-in functions for computing the Fourier transform (fft and ifft). These functions express their results as complex numbers. For example, if you compute the Fourier transform of a simple 3-element vector, you get a 3-element result of complex numbers:

```
y=[0 1 0];
fft(y)
ans = 1.0000    -0.5000-0.8660i    -0.5000+0.8660i
```

where the "i" indicates the "imaginary" part. The first element of the fft is just the sum of elements in y. The *inverse* fft, `ifft([1.0000    -0.5000-0.8660i    -0.5000+0.8660i])`, returns the original vector `[0 1 0]`.

For another example, the fft of [0 1 0 1] is [2 0 -2 0]. In general, the fft of an n-element vector of real numbers returns an n-element vector of real or complex numbers, but only the first n/2+1 elements are unique. The rest of the elements is a mirror image of the first. Operations on individual elements of the fft, such as in Fourier filtering, must take this structure into account.

The frequency spectrum "s" of a signal vector "y" can be computed as `real(sqrt(fft(y) .* conj(fft(s))))`. Here is a simple example where you know the answer in advance, at least qualitatively: an 8-element vector of integers that trace out a *single cycle of a sine wave*:

```
y=[0 7 10 7 0 -7 -10 -7];
s=real(sqrt(fft(y).*conj(fft(y))))
```

The frequency spectrum in this case is `[0 39.9 0 0.201 0 0.201 0 39.9].` \

In **Python**, the syntax is similar: `y=np.array([0, 7, 10, 7, 0, -7, -10,-7])` `s=np.real(np.sqrt(fft.fft(y)*np.conj(fft.fft(y))))`. Again, the first element is the average (which is zero) and elements 2 through 4 are the mirror image of the last 4. The unique elements are the first four, which are the amplitudes of the sine wave components whose frequencies are 0, 1, 2, 3 times the frequency of a sine wave that would just fit a single cycle in the period of the signal. In this case, is the *second* element (39.8) that is the largest by far, which is just what you would

expect for a signal that approximates a single cycle of a *sine* (rather than a *cosine*) wave. Had the signal been *two* cycles of a sine wave, s = [0 10 0 -10 0 10 0 -10], the *third* element would have been the strongest (try it). The highest frequency that can be represented by an 8-element vector is one that has a period equal to 2 elements. It takes a minimum of 4 points to show one cycle, e.g. [0 +1 0 -1].

If you are reading this online, click here for a Matlab script that creates and plots a sine wave and then uses the fft function to calculate and plot the power spectrum. Try different frequencies (third line). Watch what happens when the frequency approaches 50. Hint: the Nyquist frequency is 1/(2*Deltat) = 1/0.02=50. Also, see what happens when you change Deltat (first line), which determines how fine the sine wave is sampled.

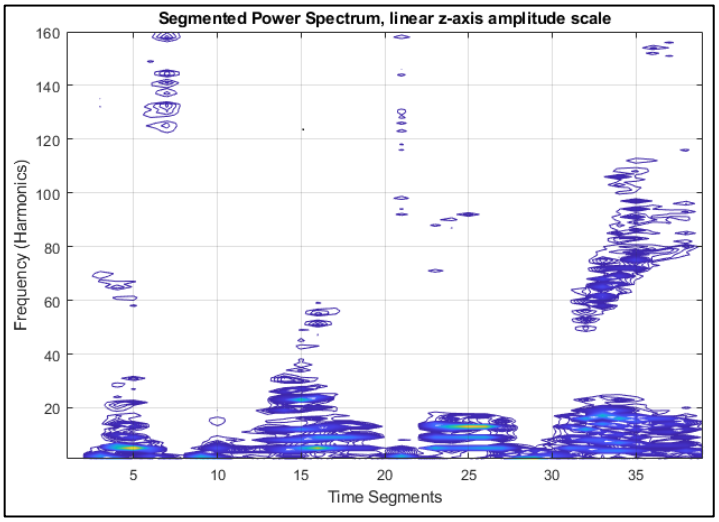
My function FrequencySpectrum.m (syntax **`fs=FrequencySpectrum(x,y)`**) returns real part of the Fourier power spectrum of x,y as a matrix. PlotFrequencySpectrum.m plots frequency spectra and periodograms on linear or log coordinates. Type "help PlotFrequencySpectrum" or try this example:

```
x=[0:.01:2*pi]';
f=25; % Frequency
y=sin(2*pi*f*x)+randn(size(x));
subplot(2,1,1);
plot(x,y);
subplot(2,1,2);
FS=PlotFrequencySpectrum(x,y,1,0,1);
```

The plot of the frequency spectrum FS (**`plotit(FS)`**; graphic) shows a single strong peak at 25. The frequency of the strongest peak in FS is given by **`FS(val2ind(FS(:,2),max(FS(:,2))),1)`**. For some other examples of using FFT, see these examples. A "Slow Fourier Transform" function has also been published, but is *3000 to 7000 times slower* with a 10,000-point data vector. You can prove this by this bit of code that you can copy/paste into the Matlab command line:
`y=cos(.1:.01:100); tic; fft(y); ffttime=toc; tic; sft(y); sfttime=toc; TimeRatio=sfttime/ffttime.`

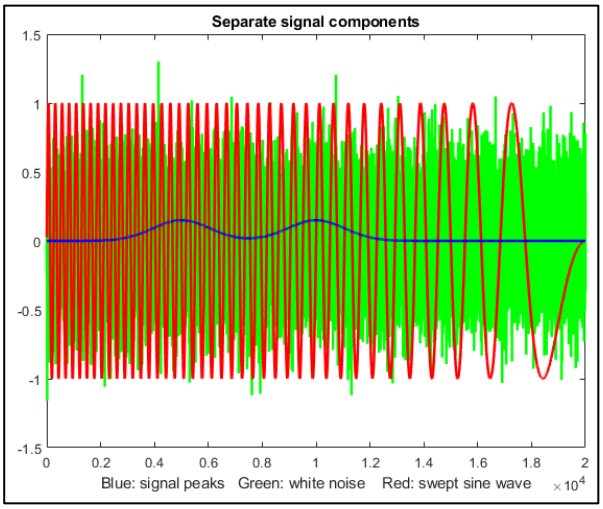
Time-segmented Fourier power spectrum.

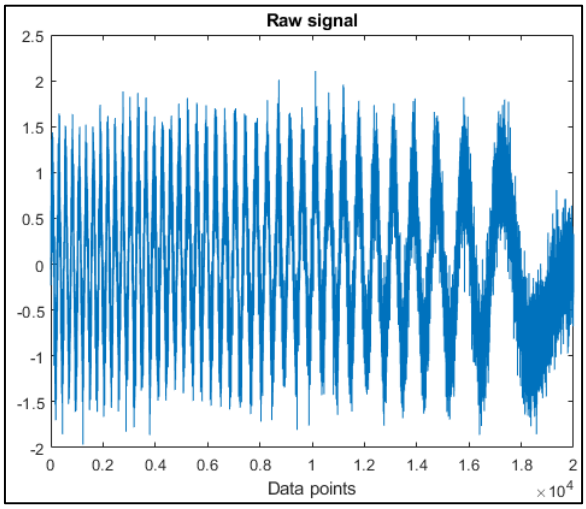
The function PlotSegFreqSpect.m, syntax **`PSM = PlotSegFreqSpect(x, y, NumSegments, MaxHarmonic, logmode)`**, creates and displays a *time-segmented* Fourier power spectrum. It breaks the signal into 'NumSegments' equal-length segments, multiplies each by an apodizing Hanning window, computes the power spectrum of each segment, and plots the magnitude of the first 'MaxHarmonic' Fourier components versus segment number as a contour plot. The function returns the power spectrum matrix (time-frequency-amplitude) as a matrix of size NumSegments x MaxHarmonic. If



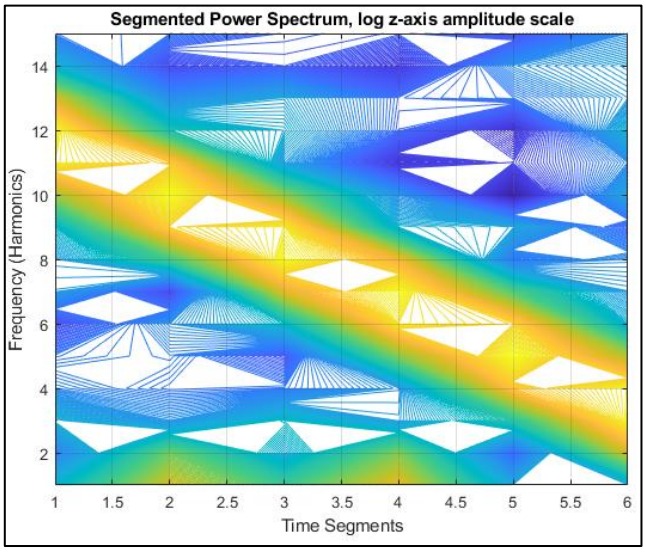
logmode=1, it computes and plots the base10 log of the amplitudes as a contour plot with different colors representing amplitudes (blue=low amplitude and yellow=high). Other examples in the help file include the spectrum of a passing automobile horn, showing the Doppler effect, and of a sample of human speech (or sonogram) shown in example 2 (graphic on previous page, right).

The next example (script) shows a complex signal consisting of three components added together





(below left): two weak Gaussian peaks at x=5000 and 10000 (blue) with height=0.15, a strong swept-frequency sine-wave interference (red), and white noise (green). When you add up all three of those components, the Gaussian peaks are totally buried and are invisible in the raw signal, above right. (I will call this the "buried peaks" signal, and I will use it again later, page 134). Inspection of the PlotSegFreqSpect function (left) reveals the strong diagonal stripe of yellow from the swept sine wave and the blue and white background from the random noise, but you also see two yellow blobs in time segments 2 and 4 at the bottom of the segmented spectrum (left). What that tells us is that there is something around the 2nd and 4th time segments that has higher frequencies than the surrounding area. Once you see that, you can constrain the range of further observation there and can verify the peaks by smoothing to reduce the high frequencies or by curve-fitting to the raw data (introduced on page 173). In fact, the curve fitting results shown below give good values (±10% or better) for the peak positions (true values =5000 and 10000), heights (0.15), and widths (2500), despite the invisibility of the peaks in the raw data.



Peak#	Position	Height	Width	Area
1	5031.4	0.15749	2280.6	382.33
2	10036	0.16136	2407.2	413.46

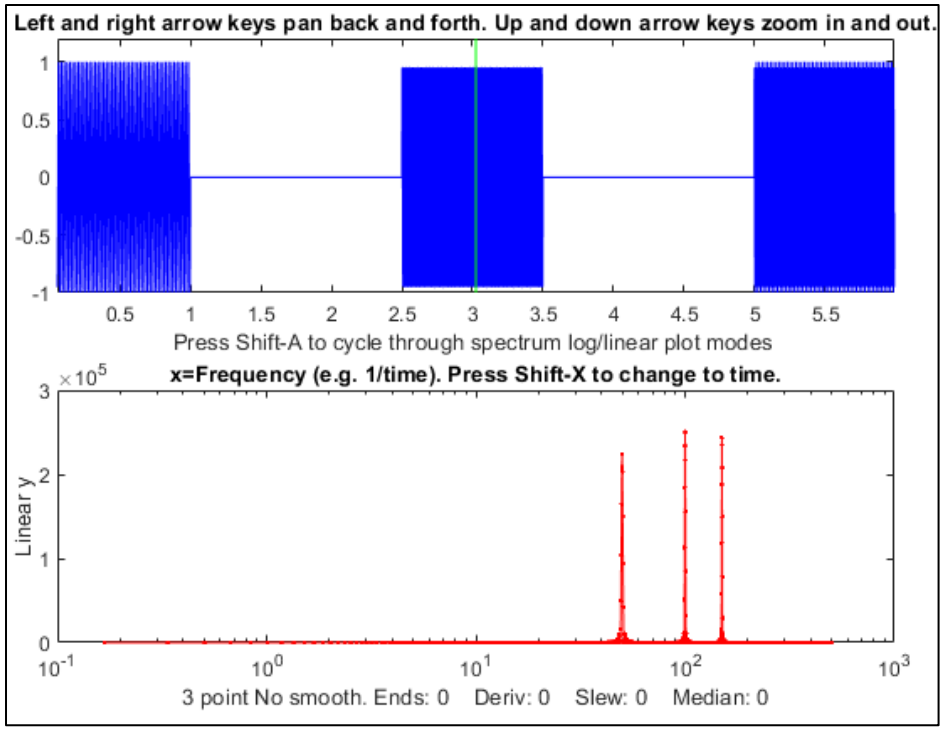
But you must *see* the peaks to know how to try that. *Based on the raw data alone, you might never try*.

Observing Frequency Spectra with iSignal

iSignal (page 376) is a multi-purpose interactive signal processing tool that has a **Frequency Spectrum mode**, toggled on and off by the **Shift-S** key, which computes the frequency spectrum of the segment of the signal displayed in the upper window and displays it in the lower window (in red). You can use the pan and zoom keys to adjust the region of the signal to be viewed or press **Ctrl-A** to select the entire signal. Press **Shift-S** again to return to the normal mode. In the frequency spectrum mode, you can press **Shift-A** to cycle through four plot modes (linear, semilog X, semilog Y, or log-log). Because of the wide range of amplitudes and frequencies exhibited by some signals, the log plot modes often result in a clearer graph that the linear modes. You can also press **Shift-X** to toggle the x-axis between *frequency* and *time*. Details and instructions are on page 376. You can download a ZIP file that contains iSignal.m version 8 and some demos and sample data for testing.

Frequency visualization.

What happens if the frequency content changes with time? Consider, for example, the signal shown in the following figure. The signal (download from SineBursts.mat) consists of three intermittent bursts of sinewaves of three different frequencies, with zeros between the bursts.



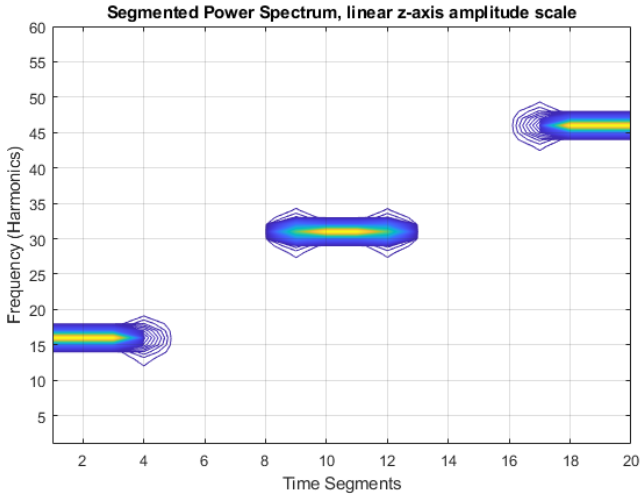
The Matlab function iSignal.m displays a signal (top panel) and its frequency spectrum (bottom panel).

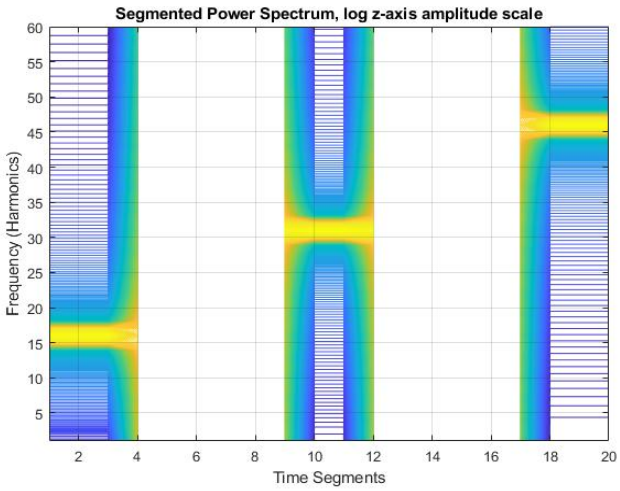
Here the signal is shown in the upper panel in the iSignal.m function (page 376) by typing:

```
load SineBursts
isignal(x,y);
```

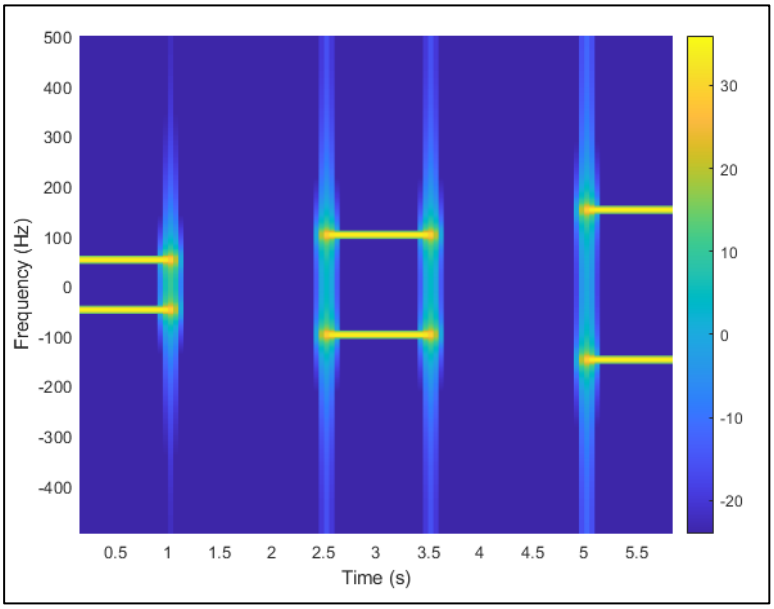
at the Matlab command prompt. By pressing **Shift-S**, its frequency spectrum is displayed in the lower panel, which shows three discreet frequency components. But which one is which? The normal Fourier transform by itself offers no clue, but iSignal allows you to pan and zoom across the signal, using the cursor arrow keys, so you would be able to isolate each. Alternatively, my function

PlotSegFreqSpec.m, just described in the previous section (figures below) is another way to display this signal in a *single static graphic that clearly displays the time and frequency variation of the signal*.



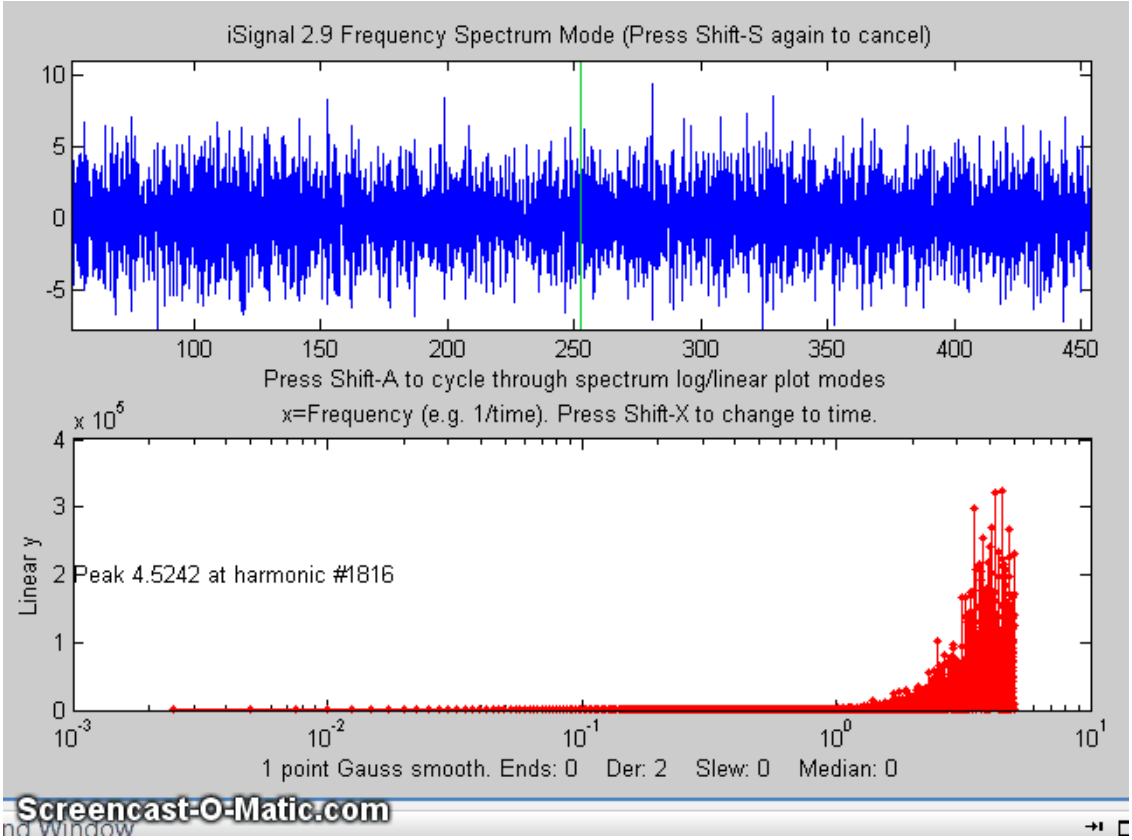


You can do a similar visualization with Matlab's inbuilt "Short Time Fourier Transform" function stft.m (below), which displays both positive and negative frequencies.



Signal enhancement

A very important feature of iSignal is that *all signal processing functions remain active in the frequency spectrum mode* (smooth, derivative, etc.), so you can observe the effect of these functions on the frequency spectrum of the signal immediately. Some signal processing operations may have the side-effect of increasing the effect of random noise or of distorting the signal. The advantage of iSignal is that you can directly observe the trade-off between the desired effect and the side effects while adjusting the signal-processing variables interactively. The figure on the next page shows an example. It shows the effect of increasing the smooth width on the 2nd derivative of a signal containing three weak noisy peaks. Without smoothing, the signal seems to be all random noise, but with enough smoothing, the three weak peaks are clearly visible (in derivative form) and measurable.



The animation above *shows the frequency spectrum mode of iSignal.m, as the smooth width is varied with the **A** and **Z** keys. This shows dramatically how the signal (top panel) and the frequency spectrum (below) are both affected by smooth width.(If you are reading this online, click for* GIF animation.)

The script "iSignalDeltaTest" demonstrates the frequency response of the smoothing and differentiation functions of iSignal by applying them to a delta function. Change the smooth type, smooth width, and derivative order and see how the power spectrum changes.

Showing that the Fourier spectrum of a Gaussian is also a Gaussian

One special (math-nerdy) thing about the *Gaussian* signal shape compared to other shapes is that the Fourier frequency spectrum of a Gaussian is *also* a Gaussian. You can demonstrate this numerically by downloading the gaussian.m and isignal.m functions and executing the following statements:

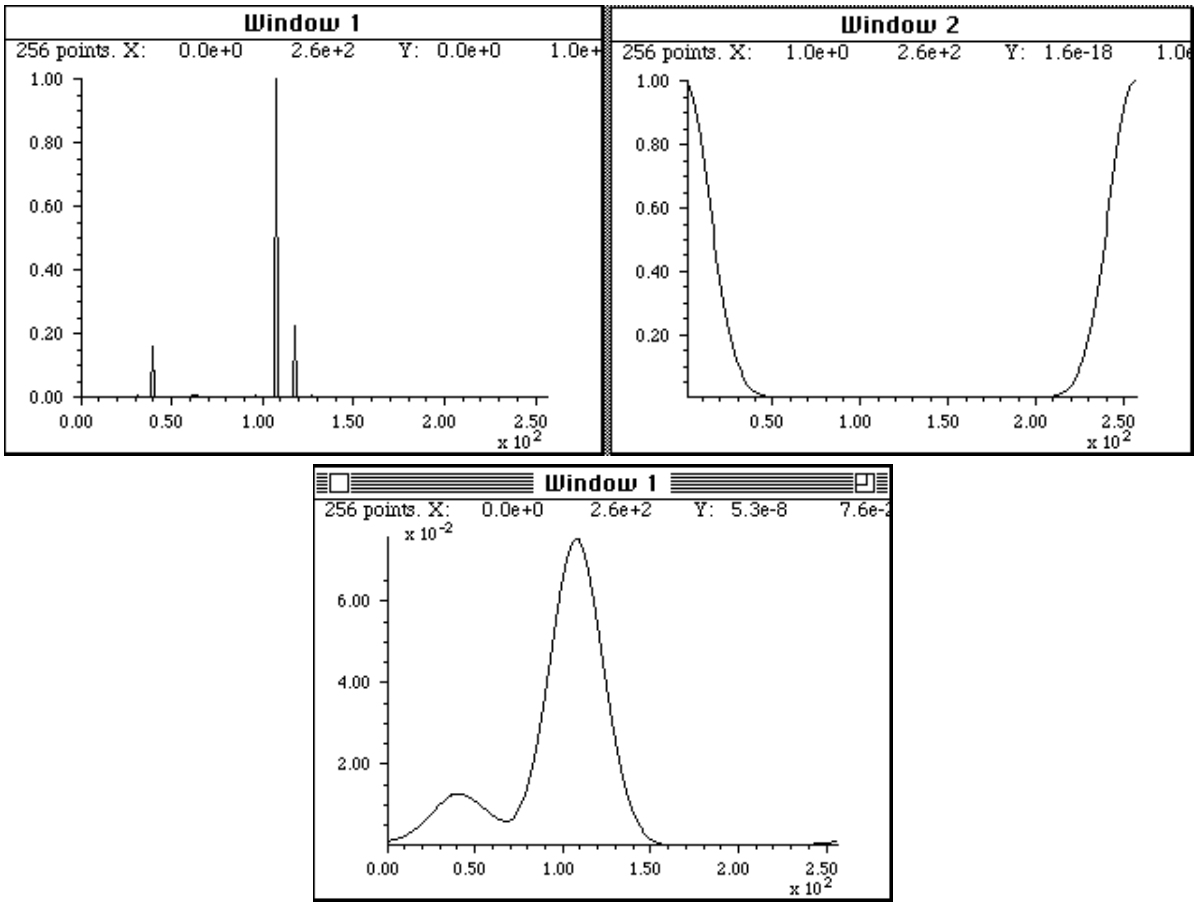
```
x=-100:.2:100;
width=2; y=gaussian(x,0,width);
isignal([x;y],0,400,0,3,0,0,0,10,1000,0,0,1);
```

Click on the figure window, press **Shift-T** to transfer the frequency spectrum to the top panel, then press **Shift-F**, press **Enter** three times, and click on the peak in the upper window. The program computes a least-squares fit of a Gaussian model to the frequency spectrum now in the top panel. The fit is essentially perfect. If you repeat this with Gaussians of *different widths* (e.g., width=1 or 4), you will find that the width of the frequency spectrum peak is *inversely proportional* to the width of the signal peak. In the limit of an infinitely *narrow* peak width, the Gaussian becomes a *delta function,* and its frequency spectrum is flat. In the limit of an infinitely *wide* peak width, the Gaussian becomes a flat

line, and its frequency spectrum is non-zero only at zero frequency.

Fourier Convolution

Convolution is a "shift-and-multiply" operation performed on two signals. It involves multiplying one signal by a delayed or shifted version of another signal, integrating or averaging the product, and repeating the process for different delays. Convolution is a useful process because it describes some effects that occur widely in scientific measurements, such as the influence of a frequency filter on an electrical signal or of the spectral bandpass of a spectrometer on the shape of a recorded optical spectrum, which cause the signal to be spread out in time and reduced in peak amplitude.



Fourier convolution *is used here to determine how the optical spectrum in Window 1 (top left) will appear when scanned with a spectrometer whose slit function (spectral resolution) is described by the Gaussian function in Window 2 (top right). The Gaussian function has already been rotated so that its maximum falls at x=0. The resulting convoluted optical spectrum (bottom center) shows that the two lines near x=110 and 120 will not be resolved but the line at x=40 will be partly resolved. Fourier convolution is used in this way to correct the analytical curve non-linearity caused by spectrometer resolution, in hyperlinear* absorption spectroscopy *(Page 276).*

For large signals, it is common to perform the calculation by point-by-point multiplication of the two signals in the *Fourier* domain. First, the Fourier transform of each signal is obtained. Then the two Fourier transforms are multiplied point-by-point by the rules for complex multiplication and the result

is then inverse Fourier transformed. Fourier transforms are usually expressed in terms of "complex numbers", with real and imaginary parts. If the Fourier transform of the first signal is $a + ib$, and the Fourier transform of the second signal is $c + id$, then the product of the two Fourier transforms is $(a + ib)(c + id) = (ac - bd) + i(bc + ad)$. Although this seems to be a round-about method, it turns out to be faster than the shift-and-multiply algorithm when the number of points in the signal is large. Convolution can be used as a powerful and general algorithm for smoothing and differentiation. Many computer languages will perform this operation automatically when the two quantities divided are complex. In typeset mathematical texts, convolution is often designated by the symbol ∗ (Reference).

Fourier convolution is used as a very general algorithm for the smoothing and differentiation of digital signals, by convoluting the signal with a (usually) small set of numbers representing the convolution vector. Smoothing is performed by convolution with sets of positive numbers, e.g. [1 1 1] for a 3-point boxcar. Convolution with [–1 1] computes a first derivative. Convolution with [1 -2 1] computes a second derivative. Successive convolutions by Conv1 and then Conv2 is equivalent to one convolution with the convolution of Conv1 and Conv2. First differentiation with smoothing is done by using a convolution vector in which the first half of the coefficients is negative, and the second half is positive (e.g. [-1 -2 0 2 1]).

Simple whole-number convolution vectors

Differentiation vectors:

```
[-1 1]          First derivative
[1 -2 1]        Second derivative
[1 -3 3 -1]     Third derivative
[1 -4 6 -4 1]   Fourth derivative
```

Results of successive convolution by two vectors Conv1 and Conv2: (∗ stands for convolution)

Conv1		Conv2	Result	Description
[1 1 1]	∗	[1 1 1]	= [1 2 3 2 1]	Triangular smooth
[1 2 1]	∗	[1 2 1]	= [1 4 6 4 1]	P-spline smooth
[-1 1]	∗	[-1 1]	= [1 -2 1]	2nd derivative
[-1 1]	∗	[1 -2 1]	= [1 -3 3 -1]	3rd derivative
[1 -2 1]	∗	[1 -2 1]	= [1 -4 6 -4 1]	4th derivative
[-1 1]	∗	[1 1 1]	= [1 0 0 -1)	1st derivative gap-segment
[-1 1]	∗	[1 2 1]	= [1 1 -1 -1)	Smoothed 1st derivative
[1 1 -1 -1]	∗	[1 2 1]	= [1 3 2 -2 -3 -1]	Same with more smoothing
[1 -2 1]	∗	[1 2 1]	= [1 0 -2 0 1]	2nd derivative gap-segment
[1 1 1 1]	∗	[1 1 1 1]	= [1 2 3 4 3 2 1]	2 passes of 4-point sliding average

Rectangle ∗ rectangle = triangle or trapezoid, depending on relative widths.
Gaussian ∗ Gaussian = Gaussian of greater width.

Gaussian * Lorentzian = Voigt profile (i.e., something in between Gaussian and Lorentzian, depending on relative widths. Similar (but not identical) to the weighted sum of a Gaussian and Lorentzian).

Software details for convolution

Spreadsheets can perform "shift-and-multiply" convolution for digitally sampled data sets (for example, MultipleConvolution.xlsx, MultipleConvolutionFirstDerivativeDemo.xls (screen shot) and MultipleConvolution4thDerivativeDemo.xls (screen shot) for Excel and MultipleConvolutionOO.ods for Calc), but for larger data sets the performance is slower than Fourier convolution (which is easier done in Matlab/Octave or in Python than in spreadsheets). Spreadsheets, however, show the cell-by-cell "shift-and-multiply" operation more clearly and explicitly.

Matlab/**Octave** have a built-in function for convolution of two vectors: **conv** (numpy.convolve in Python). This function can be used to create filters and smoothing functions, such as sliding-average and triangular smooths:

```
ysmoothed=conv(y,[1 1 1 1 1],'same')./5;
```

This smooths the vector y with a 5-point *unweighted* sliding average (boxcar) smooth.

```
ysmoothed=conv(y,[1 2 3 2 1],'same')./9;
```

This smooths the vector y with a 5-point *triangular* smooth. (The optional argument 'same' returns the central part of the convolution that is the same size as y. If that optional argument is "full", then the length of the result is one less than the sum of the lengths of the two vectors).

Differentiation can be combined with smoothing by using a convolution vector in which the first half of the coefficients is negative, and the second half is positive (e.g. `[-1 0 1]`, `[-2 -1 0 1 2]`, or `[-3 -2 -1 0 1 2 3]`). These compute a first derivative with increasing amounts of smoothing.

The **conv** function in Matlab/Octave can easily be used to combine successive convolution operations, for example, a second differentiation followed by a 3-point triangular smooth:

```
>> conv([1 -2 1],[1 2 1])
ans = 1     0    -2     0     1
```

The next example creates an exponential trailing transfer function (c), which has an effect like a simple RC low-pass filter and applies it to y.

```
c=exp(-(1:length(y))./30);
yc=conv(y,c,'full')./sum(c);
```

In each of the above three examples, the result of the convolution is divided by the sum of the convolution transfer function, to ensure that the convolution has a net gain of 1.000 and thus does not affect the area under the curve of the signal. This makes the mathematical operation closer to the physical convolutions that spread out the signal in time and reduce the peak amplitude but conserve the total energy in the signal, which for a peak-type signal is proportional to the area under the curve.

Alternatively, you could perform the convolution yourself, *without* using the built-in Matlab/Octave "conv" function, by multiplying the Fourier transforms of y and c using the "fft.m" function and then inverse transform the result with the "ifft.m" function. The results are essentially the same and the elapsed time is slightly faster than using the conv function. However, c must be zero-filled to match

the size of yc because the point-by-point multiplication or division of two vectors requires that they have the same length. The "conv" function performs any required zero filling automatically.

```
yc=ifft(fft(y).*fft(c));
```

When using convolution *for smoothing*, it's common to want to keep the *area under the signal* y the same after smoothing. This is easily ensured by dividing by the sum of the members of c:

```
yc=ifft(fft(y).*fft(c))./sum(c);
```

GaussConvDemo.m shows that a Gaussian of unit height convoluted with a Gaussian of the same width is a Gaussian with a height of 1/sqrt(2) and a width of sqrt(2) and of equal area to the original Gaussian. **(**Figure window 2 shows an attempt to recover the original "y" from the convoluted "yc" by using the deconvgauss function). You can optionally add noise in line 9 to show how convolution smooths the noise and how deconvolution restores it. Requires gaussian.m, peakfit.m and deconvgauss.m in the Matlab path.

iSignal (page 376) has a **Shift-V** keypress that displays the menu of Fourier convolution and deconvolution operations that allow you to convolute a Gaussian or exponential function with the signal and asks you for the Gaussian width or the time constant (in X units).

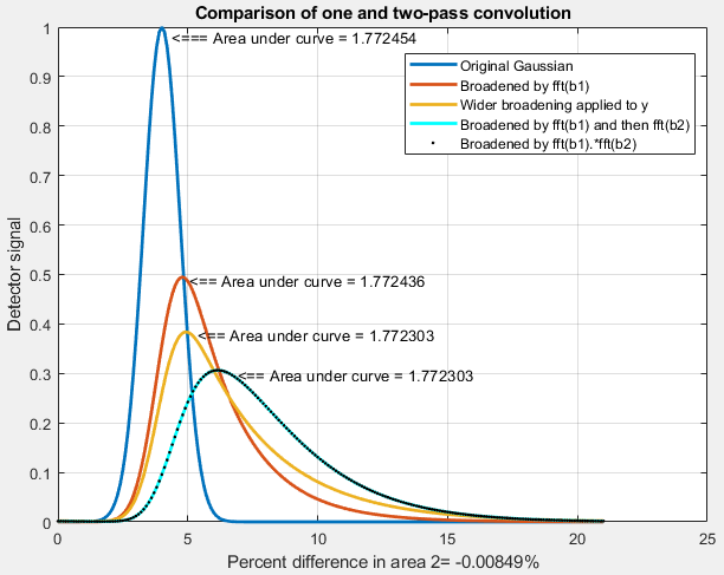
```
Fourier convolution/deconvolution menu
  1. Convolution
  2. Deconvolution
Select mode 1 or 2: 1

Shape of convolution/deconvolution function:
  1. Gaussian
  2. Exponential
Select shape 1 or 2: 2

Enter the exponential time constant:
```

Then you enter the time constant (in x units) and press **Enter**.

Multiple sequential convolution



In the real world, signal broadening mechanisms are not always reducible to a single convolution. Sometimes two or more convolution mechanisms may be in play at the same time. A good example of this occurs in the technique of *twin-column recycling separation process* (TCRSP), a novel chromatography technique in which the injected sample is recycled back to the two columns for obtaining better and better resolution, allowing chromatographers to solve challenging separation problems caused by the partition coefficients for the components being too similar and/or too low column efficiencies [reference 90]. In TCRSP, after the sample is separated by the first column, it flows into the second identical column, and after that separation, switching valves connect it back into the first column. That cycle repeats as many times as required. Each pass through a column increases the separation between the components slightly, so that with a sufficiently large number of cycles, very similar substances can be separated. Chromatographic separations often involve broadening of the peaks by asymmetrical mechanisms (page 140), usually modeled as an exponentially modified Gaussian (EMG). Any broadening that occurs in the first pass will occur repeatedly in the subsequent passes. The net result will be a final peak shape that *cannot be described by a single convolution*. The success of the TCRSP technique depends on the fact that the separation between the components increases faster than the width increase caused by the successive convolutions of broadening mechanisms. But multiple sequential convolutions produce results that differ from a single large convolution. This is demonstrated by the simple example of two sequential exponential convolutions applied to a Gaussian, as shown in the figure on the previous page, generated by a Matlab script. The blue curve is the original Gaussian. The red curve is the result of a single convolution by an exponential function whose time constant *tau* is 2. The cyan curve is the result of two successive convolutions with that same *tau*. The orange curve is an attempt to duplicate that with a single wider convolution of *tau* equal to 3. That attempt fails, resulting in a poor match to the cyan curve. In fact, experiments show that *no single wider exponential convolution can match the result of two (or more) successive convolutions*. The shape is fundamentally different. Multiple exponential convolutions result in a less asymmetrical peak, more shifted to larger x values. On the other hand, a *single* convolution by a function that is the *product* of the Fourier transforms of the two separate functions does work (black dots). With greater numbers of successive convolutions, the peaks become more symmetrical and more Gaussian, as demonstrated by this graphic, generated by this Matlab script. (You can choose the number of convolutions in line 20 of the script).

Fourier Deconvolution

Fourier deconvolution is the converse of Fourier convolution in the sense that division is the converse of multiplication. If you know that ***m*** times ***x*** equals ***n***, where ***m*** and **n** are known but ***x*** is unknown, then ***x*** equals ***n*** divided by ***m***. Similarly, if you know that the vector ***M*** convoluted with the vector ***X*** equals the vector ***N***, where ***M*** and ***N*** are known but ***X*** is unknown, then ***X*** equals ***M*** *deconvoluted* from ***N***.

In practice, the deconvolution of one signal from another is usually performed by point-by-point *division* of the two signals in the Fourier domain, that is, dividing the Fourier transforms of the two signals point-by-point and then inverse-transforming the result. Fourier transforms are usually expressed in terms of complex numbers, with "real" and "imaginary" parts representing the sine and cosine parts. If the Fourier transform of the first signal is $a + ib$, and the Fourier transform of the second signal is $c + id$, then the *ratio* of the two Fourier transforms, by the rules for the division of complex numbers, is

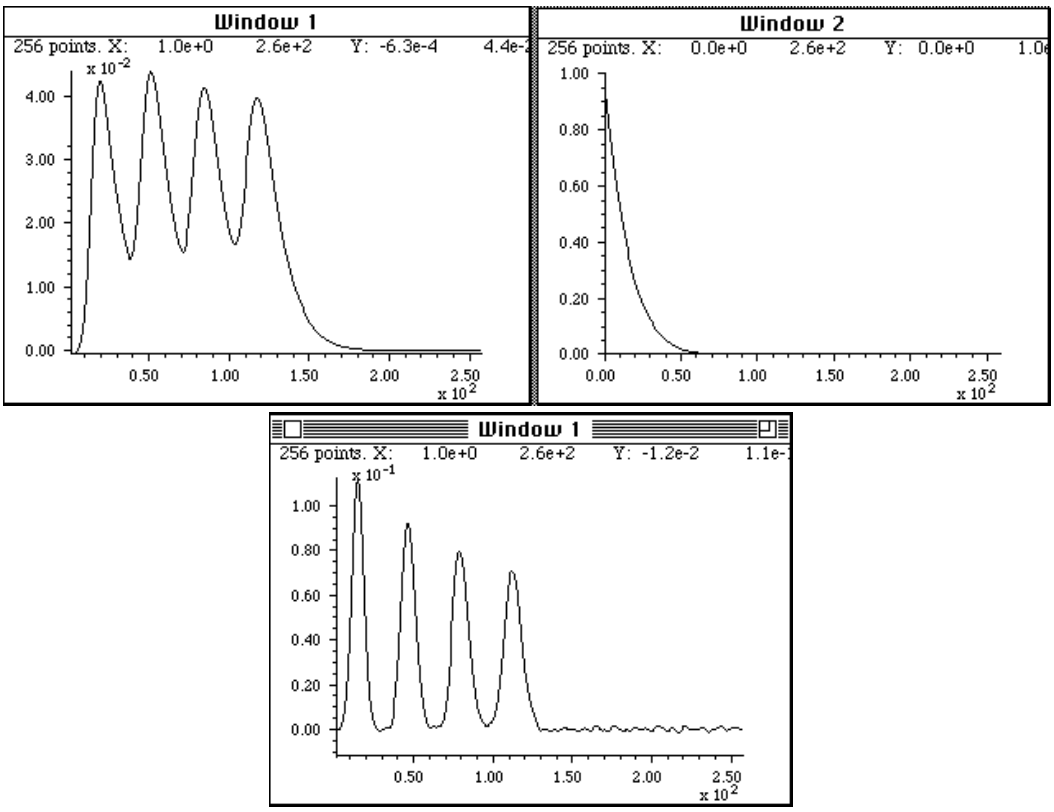
$$\frac{a + ib}{c + id} = \frac{ac + bd}{c^2 + d^2} + i\frac{bc - ad}{c^2 + d^2}$$

Scientific computer languages (such as Fortran, Matlab and Python) will *perform this operation automatically* when two complex numbers are divided.

Note: It is important to realize that the word "deconvolution" can have *two different meanings* in the scientific literature, which can lead to confusion. The Oxford dictionary defines the word as "A process of resolving something into its constituent elements or removing complication in order to clarify it", which in one sense applies to Fourier deconvolution. However, the same word is also sometimes used for the process of resolving or decomposing a set of overlapping signals into their separate additive components by the technique of iterative least-squares curve fitting (page 199) of a proposed model of the signal to the data set. However, that process is conceptually distinct from *Fourier* deconvolution, because in Fourier deconvolution, the underlying peak shape is unknown, but the broadening function is assumed to be known. But in iterative least-squares curve fitting, it is just the reverse: the peak shape is assumed to be known but the width of the broadening process, which determines the width and shape of the peaks in the recorded data, is usually unknown. Thus, the term "spectral deconvolution" is *ambiguous*: it might mean the Fourier deconvolution of a response function from a spectrum, or it might mean the decomposing of a spectrum into its separate additive components. These are different processes. Do not get them confused.

Applications of deconvolution

Fourier deconvolution is often used as a computational way to reverse the result of a convolution occurring in the physical domain, for example, to reverse the signal distortion effect of an electrical filter or of the finite resolution of a spectrometer. In some cases, the physical convolution can be measured experimentally by applying a single spike impulse ("delta") function to the input of the system, then that function can be used as a deconvolution vector. In that application, deconvolution works perfectly only when the signals contain no noise and when the original convolution function is known exactly.



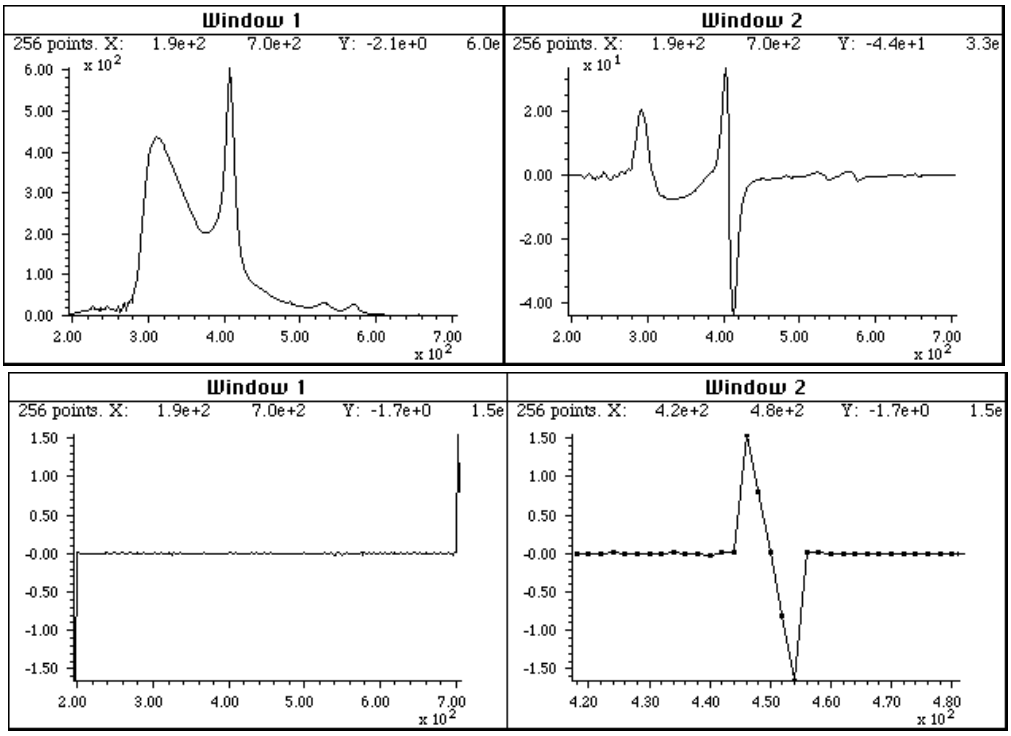
Fourier deconvolution *is used here to remove the distorting influence of an exponential tailing response function from a recorded signal (Window 1, top left) that is the result of a low-pass filter built into the electronics to reduce noise. The response function (Window 2, top right) must be known and is usually either calculated based on some theoretical model or is measured experimentally as the output signal produced by applying an impulse (delta) function to the input of the system. The response function, with its maximum at x=0, is deconvoluted from the original signal. The result (bottom, center) shows a closer approximation to the real shape of the peaks. However, the signal-to-noise ratio is unavoidably degraded compared to the recorded signal, because the Fourier deconvolution operation is simply recovering the original signal before the low-pass filtering, noise and all.*
(If you are reading this online, click for Matlab/Octave script.*)*

Note that this process has an effect that is *visually* similar to derivative peak sharpening (page 78) although the latter requires no specific knowledge of the broadening function that caused the peaks to overlap.

Even if there is no known physical convolution that has broadened the signal, it is possible to use deconvolution as a method of peak sharpening by deconvolution of a model of that peak shape from the signal. That is referred to as “self-deconvolution”, because the shape of the deconvolution function is the same as the shape of the peaks in the signal. Self-deconvolution is a common method of peak sharpening which can be applied to a signal consisting of one or more peaks of a predictable peak shape. The idea is that a noiseless model of the peak shape is deconvoluted from the signal and the width of that model peak is adjusted to provide the desired degree of sharpening.

Deconvolution can also be used to determine the form of an unknown convolution operation that has been previously applied to a signal, by deconvoluting the original and the convoluted signals, as shown

in the following page.



A different application of Fourier deconvolution *is to reveal the nature of an unknown data transformation function that has been applied to a data set by the measurement instrument itself. In this example, the figure in the top left is an ultraviolet-visible absorption spectrum recorded on a commercial photodiode array spectrometer (X-axis: nanometers; Y-axis: milliabsorbance). The figure in the top right is the* first derivative *of that spectrum produced by an (unknown) algorithm in the software supplied with the spectrometer. The objective here is to understand the nature of the* differentiation/ smoothing algorithm *that the instrument's internal software uses. The signal in the bottom left is the surprisingly simple result of deconvoluting the derivative spectrum (top right) from the original spectrum (top left). This, therefore, must be the convolution function used by the differentiation algorithm in the spectrometer's software or its equivalent. Rotating and expanding it on the x-axis makes the function easier to see (bottom right). Expressed in terms of the smallest whole numbers, the convolution series is simply +2, +1, 0, -1, -2, which is a combination of differentiation and smoothing (page 110). This elementary example of "*reverse engineering*" makes it easier to compare results from other instruments or to duplicate these results on other equipment.*

When applying Fourier deconvolution to experimental data, for example, to remove the effect of a known broadening or low-pass filter operator caused by the experimental system, there are *three serious problems* that limit the utility of the method:

(1) A mathematical convolution might not be an accurate model for the convolution occurring in the physical domain.

(2) The width of the convolution - for example, the time constant of a low-pass filter operator or the shape and width of a spectrometer slit function - must be known, or at least adjusted by the user to get the best results.

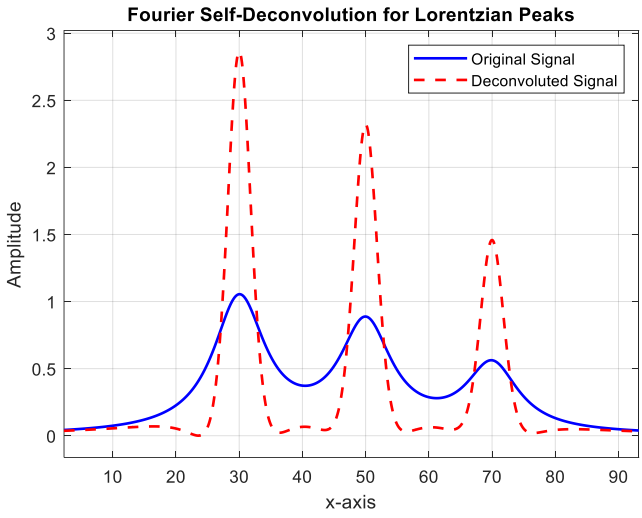
(3) A serious signal-to-noise degradation commonly occurs. Any noise added to the signal by

the system *after* the convolution by the broadening or low-pass filter operator will be greatly amplified when the Fourier transform of the signal is divided by the Fourier transform of the broadening operator, because the high frequency components of the broadening operator (the *denominator* in the division of the Fourier transforms) are typically very small, with some individual components often of the order of 10^{-12} or 10^{-15}, resulting a huge amplification of those particular frequencies in the resulting deconvoluted signal. This can be controlled to some extent by smoothing or filtering to reduce the amplitude of the highest-frequency components.

You can see the amplification of high-frequency noise happening in the first graphic example above on the previous pages. On the other hand, this effect is *not* observed in the second example, because in that case, the noise was present in the original signal, *before* the convolution performed by the spectrometer's derivative algorithm. The high-frequency components of the denominator in the division of the Fourier transforms are typically much *larger* than in the previous example, avoiding the noise amplification and divide-by-zero errors, and the only post-convolution noise comes from numerical round-off errors in the math computations performed by the derivative and smoothing operation, which is always much smaller than the noise in the original experimental signal.

In many cases, the width of the physical convolution is not known exactly, so the deconvolution must be adjusted empirically to yield the best results. Similarly, the width of the final smooth operation must also be adjusted for the best results. The result will seldom be perfect, especially if the original signal is noisy, but it is often a better approximation to the real underlying signal than the recorded data without deconvolution.

Fourier Self-deconvolution

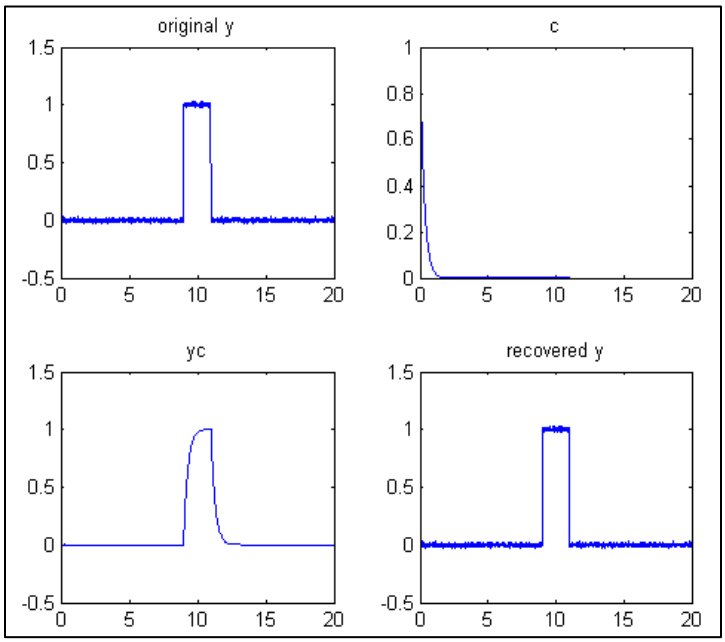


Self-deconvolution is used to describe the deconvolution of a zero-centered model peak of the same shape from a signal containing several peaks, as a method for peak sharpening and to facilitate the measurement of the area of overlapping peaks (page 123). Just like the derivative-based sharpening methods (page 79), the deconvolution algorithm (a) must be adjusted for the width of the peaks, (b) is subject to considerable high-frequency noise amplification (page 121), and (c) its effectiveness depends on the shape of the peaks. For example, Gaussian peaks can be reduced in width by nearly half (script) and for Lorentzian peaks by an even greater factor, as shown in the figure (script). Intermediate shapes, such as the Gaussian-Lorentzian blend, are in between, as expected (page 126). However, with high degrees of sharpening, the *noise amplification will be extreme* and a carefully calibrated amount of smoothing must be applied to the result, usually by means of a multi-pass sliding average (page 55) or a Fourier filter (page 121). Even then, the technique will be most effective if the signal-to-noise ratio is good, as is often the case in various forms of chromatography (page 363).

Computer software for deconvolution

Matlab and Octave

Matlab and Octave have a built-in function for Fourier deconvolution: deconv. An example of its application is shown in the figure on the next page: the vector *yc* (line 6) represents a **noisy** rectangular pulse (*y*) convoluted with a transfer function *c* before being measured. In line 7, *c* is deconvoluted from *yc*, to recover the original *y*. This requires that the transfer function *c* be known. The rectangular signal pulse is recovered in the lower right (*ydc*), complete with the noise that was present in the original signal. The Fourier deconvolution reverses not only the signal-distorting effect of the convolution by the exponential function, but also its low-pass noise-filtering effect. As explained above, there is a significant amplification of any noise that is added *after* the convolution by the transfer function (line 5). This script can be used to demonstrate that there is a big difference between noise added *before* the convolution (line 3), which is recovered unmodified by the Fourier deconvolution along with the signal, and noise added *after* the convolution (line 6), which is amplified compared to that in the original signal. Download this script. Note that the "sum(c)" term in line 7 is included simply to scale the amplitude of the result (specifically the area under the curve) to match the original y.

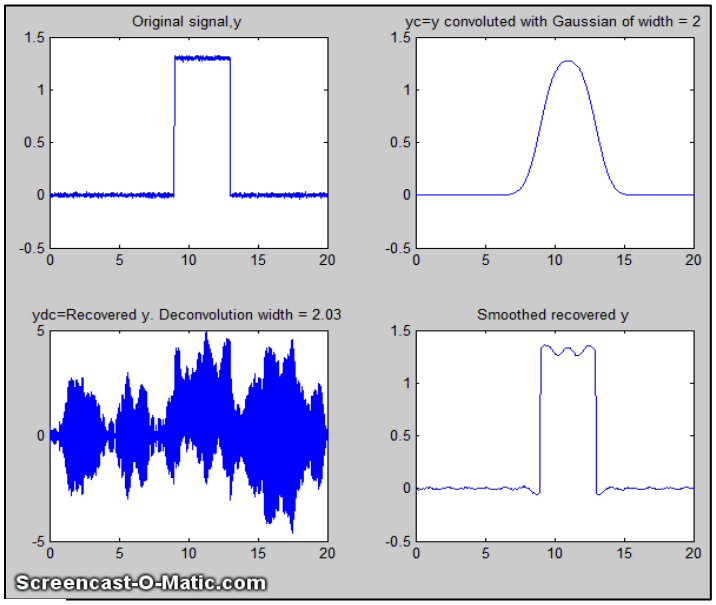


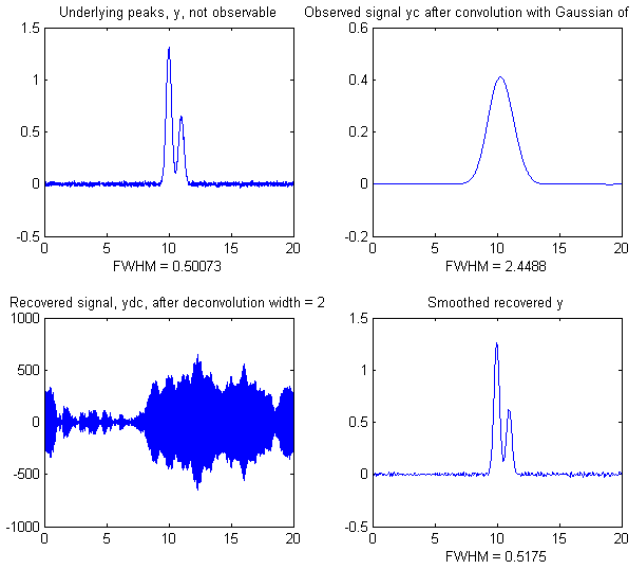

```
x=0:.01:20;y=zeros(size(x));   % 2000 point signal with 200-point
y(900:1100)=1;                 % rectangle in center, y
y=y+.01.*randn(size(y));       % Noise added before the convolution
c=exp(-(1:length(y))./30);     % exponential convolution function, c
yc=conv(y,c,'full')./sum(c);   % Create exponential trailing function, yc
% yc=yc+.01.*randn(size(yc)); % Noise added after the convolution
ydc=deconv(yc,c).*sum(c);      %  Recover y by deconvoluting c from yc
% Plot all the steps
subplot(2,2,1); plot(x,y); title('original y'); subplot(2,2,2);
plot(x,c);title('c'); subplot(2,2,3); plot(x,yc(1:2001)); title('yc');
subplot(2,2,4); plot(x,ydc);title('recovered y')
```

Alternatively, you could perform the Fourier deconvolution yourself *without* using the built-in Matlab/ Octave "deconv" function by dividing the Fourier transforms of *yc* and *c* using the built-in Matlab/ Octave "fft.m" function and inverse transform the result with the built-in Matlab/Octave "ifft.m" function. Note that *c* must be zero-filled to match the size of *yc*. The results are essentially the same (except for the numerical floating-point precision of the computer, which is usually negligible), and it is *faster* than using the deconv function:

```
ydc=ifft(fft(yc)./fft([c zeros(1,2000)])).*sum(c);
```

If you are reading this online, click here for a simple explicit example of Fourier convolution and deconvolution for a small 9-element vector, with the vectors printed out at each stage. The script DeconvDemo3.m is like the previous example, except that it demonstrates *Gaussian* Fourier convolution and deconvolution of the same slightly noisy rectangular pulse, utilizing the fft/ifft formulation just described. The animated screen graphic on the right (If you are reading this online, click link for animation) demonstrates the effect of changing the deconvolution width. The raw deconvoluted signal in this example (bottom left quadrant) is extremely noisy, but that noise is mostly "blue" (high frequency) noise that you can easily reduce by a little smoothing (page 40). As you can see in both animated examples here, deconvolution works *best* when the deconvolution width exactly matches the width of the convolution to which the observed signal has been subjected. The further off you are, the worse the wiggles and other signal artifacts will be. In practice, you must try several different deconvolution widths to find the one that results in the *smallest wiggles*, which of course becomes harder to see if the signal is very noisy. Note that in this example the deconvolution width must be within 1% of the convolution width. In general, the wider the physical convolution width relative to the signal, the more accurately the deconvolution width must be matched to the physical convolution width.

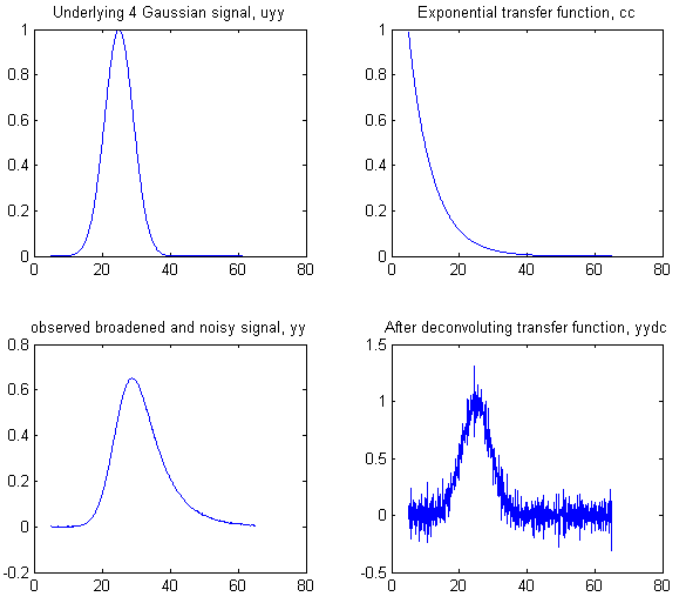




DeconvDemo5.m (left) shows an example with *two* closely-spaced underlying peaks of equal width that are *completely unresolved* in the observed signal but are recovered with their 2:1 height ratio intact in the deconvoluted and smoothed result. This is an example of Gaussian "self-deconvolution". DeconvDemo6.m is the same except that the underlying peaks are Lorentzian. (Note that all these scripts require functions than can be downloaded from http://tinyurl.com/cey8rwh). In all the above simulations, the deconvolution method works as well as it does because the signal-to-noise ratio of the "observed signal" (upper right quadrant) is quite good. The noise is not even visible on the scale presented here. In the absence of any knowledge of the width of the deconvolution function, finding the correct deconvolution width depends upon experimentally minimizing the wiggles that appear when the deconvolution width is incorrect, and a poor signal-to-noise ratio will make this much more difficult. Of course, smoothing can reduce noise, especially high-frequency (blue) noise, but smoothing also slightly increases the width of peaks, which works *counter* to the point of

deconvolution, so it must not be overused. The image on the left shows the widths of the peaks (as full width at half maximum). The widths of the deconvoluted peaks (lower right quadrant) are only slightly larger than in the (unobserved) underlying peaks (upper left quadrant) either because of imperfect deconvolution or the broadening effects of the smoothing needed to reduce the high-frequency noise. As a rough but practical rule of thumb, if there is any *visible* noise in the observed signal, it is likely that the results of self-deconvolution, of the type shown in DeconvDemo5.m, will be too noisy to be useful.

In the example shown on the right. (Download this script), the underlying signal (*uyy*) is a *Gaussian*, but in the observed signal (*yy*) the peak is *broadened exponentially* resulting in a *shifted, shorter, and wider peak.* Assuming that the exponential broadening time constant ('*tc*') is known, or can be guessed or measured (page 83), the Fourier deconvolution of *cc* from *yy* successfully removes the broadening (*yydc*), and restores the original height, position, and width of the underlying Gaussian, but at the expense of considerable noise increase. The noise is caused by the fact that a little constant white noise has been added *after* the broadening convolution (*cc*), to make the simulation more realistic. However, the noise remaining in the deconvoluted signal is "blue" (high-frequency weighted, see page 27) and so is easily reduced by smoothing (page 44) and has less effect on least-square fits than does white noise. (For a greater challenge, try more noise in line 6 or a bad guess of the time constant ('*tc*') in line 7). To plot the recovered signal overlaid with the underlying signal: **`plot(xx,uyy,xx,yydc)`**. To plot the observed signal overlaid with the underlying signal: **`plot(xx,uyy,xx,yy)`**. To curve fit the recovered signal to a Gaussian to determine peak parameters:



`[FitResults,FitError]=peakfit([xx;yydc],26,42,1,1,0,10)`, which yields excellent values for the original peak positions, heights, and widths. You can demonstrate to yourself that with *ten times* the previous noise level (Noise=.01 in line 6), the values of peak parameters determined by curve fitting (page 199) are still quite good, and even with *100x more noise* (Noise=.1 in line 6) the peak parameters are *more accurate than you might expect* for that amount of noise (because that noise is blue). Remember, there is no need to smooth the results of the Fourier deconvolution before curve fitting, as seen previously on page 50

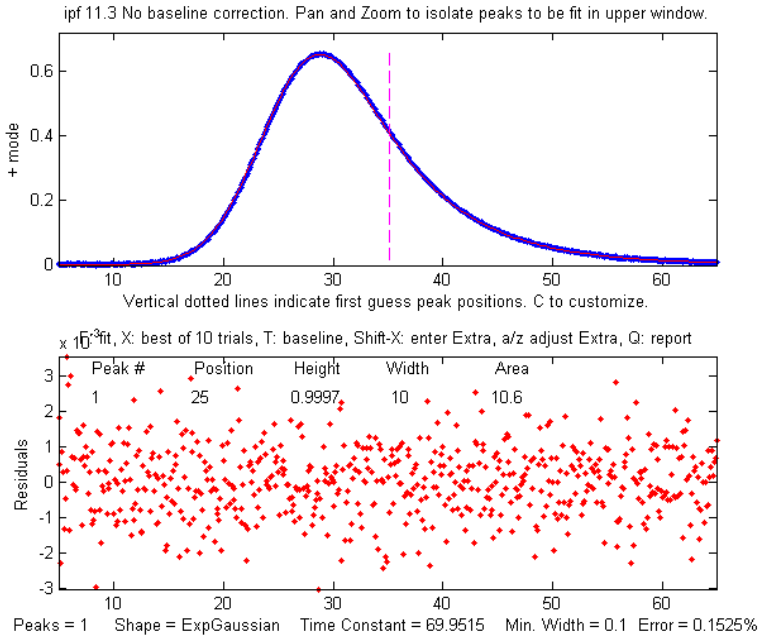
An alternative to the above deconvolution approach is to use iterative curve fitting (page 199) to fit the observed signal directly with an exponentially broadened Gaussian model (shape number 5):

```
>> [FitResults,FitError] = peakfit([xx;yy], 26, 50, 1, 5, 70, 10)
```

Both methods give good values of the peak parameters, but the Fourier deconvolution method is faster because fitting the deconvoluted signal with a simple Gaussian model is faster than iteratively curve

fitting the observed signal with the more complicated exponentially broadened Gaussian model.

If the exponential factor "tc" is not known, it can be determined by iterative curve fitting using ipf.m (page 416), manually adjusting the exponential factor ('extra') interactively with the A and Z keys to get the best fit:




```
>>ipf([xx;yy]);
```

which in this case gives a best fit when the exponential factor "tc" is adjusted to about 69.9 (close the correct value of 70 in this simulation).

Alternatively, you can use peakfit.m with the *unconstrained variable* exponentially broadened Gaussian (shape 31), which will automatically find the best value of "tc", but in that case the best results will be obtained if you give it a rough first guess ("start") as the eighth input argument, with values within a factor of two or so of the correct values:

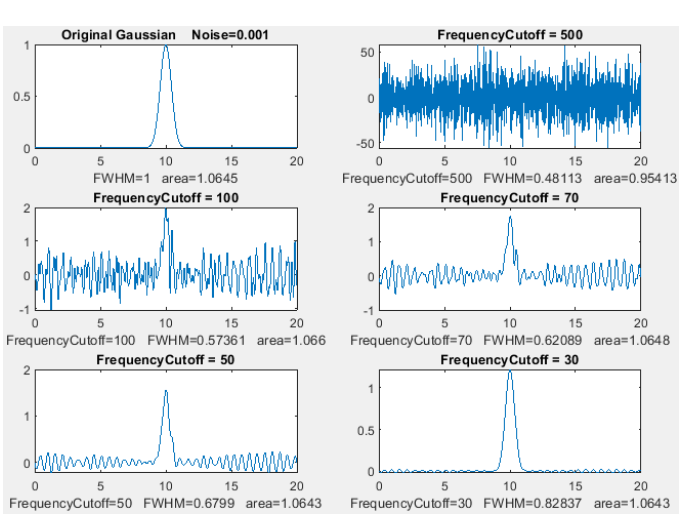
```
>>[FitResults,FitError]=peakfit([xx;yy],0,0,1,31,70,10, [20 10 50])

            Peak#    Position    Height      Width      Area       tc
              1      25.006      0.99828     10.013    10.599    69.83
GoodnessOfFit =
      0.15575       0.99998
```

The value of the exponential factor determined by this method is 69.8, again close to 70. However, if the signal is very noisy, there will be quite a bit of uncertainty in the value of the exponential factor so determined - for example, the value will vary a bit if slightly different regions of the signal are selected for measurement (e.g., by panning or zooming in ipf.m or by changing the center and window arguments in peakfit.m). See page 307 for another example with four overlapping Gaussians.

Noise reduction in deconvoluted signals

The most common way to control the high-frequency noise amplification resulting from deconvolution, as described above, is low-pass filtering, either by some form of sliding average filter or by a Fourier filter. The basic limitation of such filters, unfortunately, is that they are limited in their ability to handle situations where near-zero values in the fft of the deconvolution function in the denominator result in astronomically large values in the deconvoluted signal. To appreciate this problem, consider the figures below, created by the Matlab script DenomAdditionDemo.m, which shows the application of self-deconvolution to sharpen a single isolated Gaussian peak by deconvolving a narrower zero-centered Gaussian function (0.8 times the width of the original peak), using only a Fourier filter (page 129) to reduce noise and ringing. (Double-click to view enlarged figures)



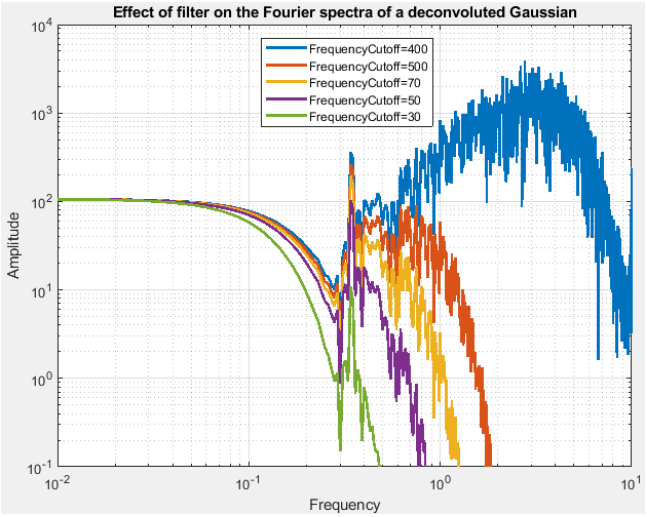


Figure window 1 (left) shows the original peak (top left) and the remaining subplots show the deconvoluted peaks after smoothing by a Fourier filer with 5 different frequency cutoffs to reduce the ringing. The corresponding frequency spectra of those five deconvoluted signals are shown in figure window 2 (right), which shows two distinct frequency regions:

(a) The left-hand (low frequency) side is a smooth curved region. This is the frequency region dominated by the peaks that have been sharpened by deconvolution. The narrower the peak, the more gradually the curve drops off at higher frequencies.

(b) The right-hand (high frequency) end of the spectrum is dominated by the noise and ringing in the deconvoluted signal. You want to reduce this region as much as possible without reducing the lower frequencies too much (which would broaden the peaks).

Observing these spectra can be a useful guide to adjusting the deconvolution parameters. The ringing in the signal peaks in Figure 1 window corresponds to the *pronounced spike near the middle of the spectrum*. As the cutoff frequency is decreased, the high-frequency components are reduced, as expected, but *that spike still remains* even at the lowest cutoff (green line), at which point the width of the deconvoluted peak has been broadened by the filter, opposite to the original intent of sharpening.

Excess noise reduction by denominator addition (regularization)

This problem can be solved by simply adding a small positive non-zero constant or distribution function to the denominator in the deconvolution calculation, which prevents excessively small numbers in the denominator. The quantity added is small, typically 1% to 5% of the amplitude of the denominator. In Matlab, the plain deconvolution operation is coded like so:

```
ydc=ifft(fft(y)./(fft(df))).*sum(df)
```

Here is the code for the case where the demoninator addition is a constant:

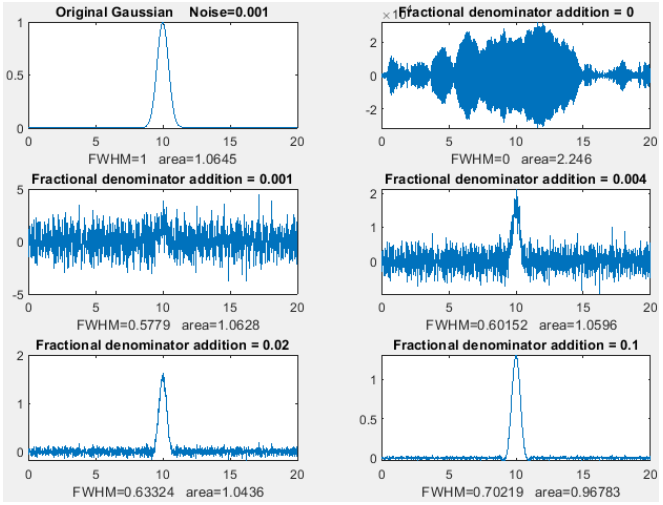
```
ydc=ifft(fft(y)./(fft(df)+FDA.*max(fft(df)))).*sum(df)
```

where *y* is the original signal, *df* is the deconvolution function and *FDA* is the fractional denominator addition. The addition is scaled to the maximum value of the fft of the deconvolution function *df* so that the quantity added will adjust to the varying amplidudes of different experimental signals.

An alternative is to add a constant *only to those members of the denominator below a pecified threshold,* e.g. using the "no lower than" function nlt(a,b):

```
ydcDA=ifft(fft(y)./(nlt(fftc,DA.*0.01.*max(fftc)))).*sum(df);
```

(The denominator addition method is included in the Matlab Live Script example on page 368).



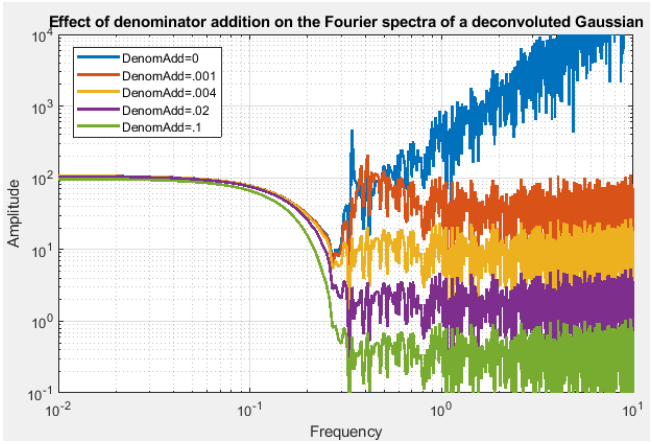


Figure windows 3 and 4 (above) show the effect of varying the fractional denominator addition *without* frequency filtering. The subplot on the top left shows the original peak and the remaining five subplots show the results of adding increasing amounts to the denominator, from zero to 0.1. The frequency spectra of those five deconvoluted signals are shown in figure window 4 (right), showing that the effect of denominator addition is to reduce the overall amplitude of the noisy right-hand half of the frequency spectrum, *without* changing the frequency distribution. With zero addition (blue), the large spike near the center is seen as before, but even the smallest addition (red) eliminates it. With the maximum 10% addition (green), the noise is greatly reduced but the deconvoluted peak is sharper (FWHM=0.7) than the width of the peak with only the filter applied (FWHM=0.82) and has the same SNR. Note: if you run the script DenomAdditionDemo.m, again, a different noise sample will be generated each time.

The two frequency spectra in the above figures show that low-pass filtering and denominator addition are *orthogonal* operations in the frequency domain. Filtering operates "horizontally" along the x (frequency) axis, whereas denominator addition operates "vertically" along the y (amplitude) axis and reduces the amplitude of the all the frequencies that are the most highly amplified by deconvolution. Both methods reduce noise, but they work in different ways and thus *can be used together*.

A good way to explore the interplay between the values of the many variables is to use the Matlab Live Script DenomAdditionDemo.mlx (graphic), which has *sliders* and a *drop down menu* for adjusting the parameters interactively. See page 368. (See page 370 for other interactive tools).

Reference 98 explores this method in greater detail, including adding distributions rather than a constant and showing several examples of applications to experimental signals.

Deconvolution for peak area measurements

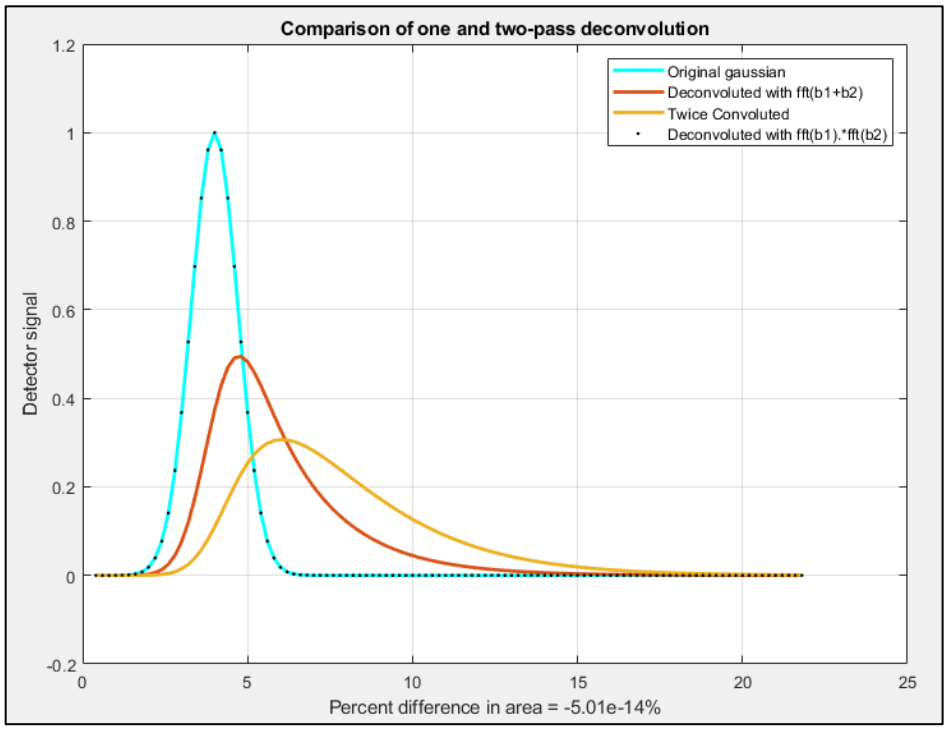
Measuring the areas under peaks is a common requirement in quantitative analysis (page 145), but it

works only if there is sufficient separation between peaks. Because deconvolution sharpens peaks but does not change the area under them, it can be used to improve the measurement of the areas of overlapping peaks. The Matlab Live Script DenomAdditionDemo.mlx (graphic) uses Fourier self-deconvolution to sharpen peaks to improve the accuracy of the perpendicular drop method for a pair of overlapping peaks (check the box to the right of "PeakAreaMeasurements" in line 3. The Live Script format allows interactive controls that allow you to explore the settings for signal generation and peak area measurement). See page 126.

In the Matlab script GLSDPerpDropDemo16.m, the areas of a group of three partially overlapping peaks is measured by the perpendicular drop method, before and after peak sharpening by Fourier self-deconvolution. The measurements are repeated with random peak heights, to test how the peak overlap interferes with precise area measurement. After sixteen trials with randomized peak heights, the true peak areas are plotted against the measured areas, and the R^2 values for each case are compared before and after deconvolution. The results are summarized on this PDF file. Conclusion: in every case, from the "easiest" to the most challenging, the areas of the peaks sharpened by deconvolution are the most accurate.

Multiple sequential deconvolution

In cases where the original signal has been subject to two or more sequential distorting convolutions(page 113), the reversal of those convolutions requires multiple sequential deconvolutions and *cannot be undone accurately by a single larger deconvolution*. As a simple example of that situation, the Matlab/ Octave script

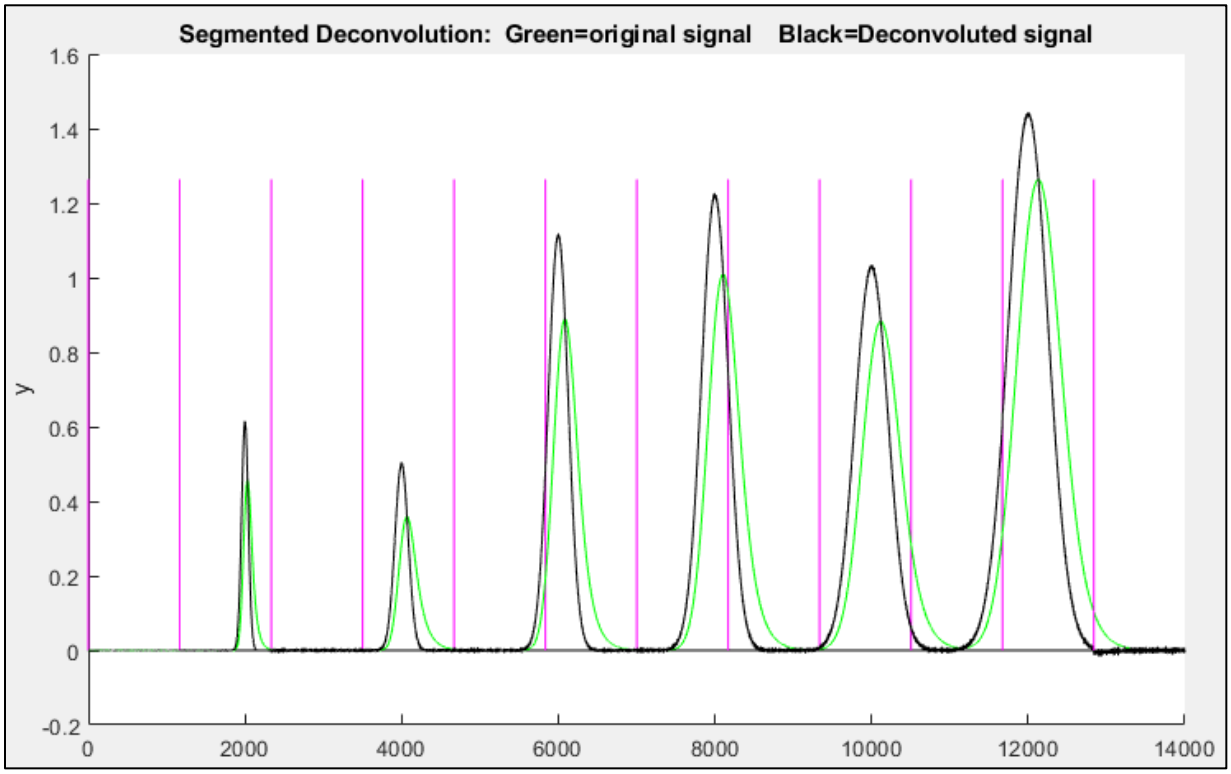


DeconvoluteTwiceBroadenedPeak.m, demonstrates the attempted deconvolution of two exponential broadenings, represented by the vectors of b1 and b2, that have been applied to an originally Gaussian peak. In the resulting graphic on the left, the light blue curve is the unknown original underlying signal, the yellow curve is the observed signal after the original has been twice exponentially broadened, and the red curve is an attempt to deconvolute a single wider exponential function with a larger time

constant, using the de-tailing method described on page 83. That attempt is obviously unsuccessful. In fact, no single simple deconvolution can remove the effects of two or more convolutions. The black dotted line is the result of performing a deconvolution with the product `fft(b1)*fft(b2)`, which is the Fourier transform of the *convolution* of b1 and b2. That attempt is successful: the black dots overlay the original Gaussian, in blue, exactly.

Segmented deconvolution

If the peak widths or tailing vary substantially across the signal, you can use a *segmented* deconvolution, which allows the deconvolution vector to adapt to the local conditions in different signal regions. SegExpDeconv(x,y,tc) divides x,y into several equal-length segments defined by the length of the vector "tc", then each segment is deconvoluted with an exponential decay of the form exp(-x./t) where "t" is the corresponding element of the vector "tc". Any number and sequence of t values can be used. SegExpDeconvPlot.m is the same except that it plots the original and deconvoluted signals and *shows the divisions between the segments by vertical magenta lines* to make it easier to adjust the number and values of the segments. This is demonstrated by the script SegExpDeconvPlotExample.m, shown in the figure below). The inevitable noise increase can be moderated by denominator addition (page 122) or by segmented smoothing (page 333).



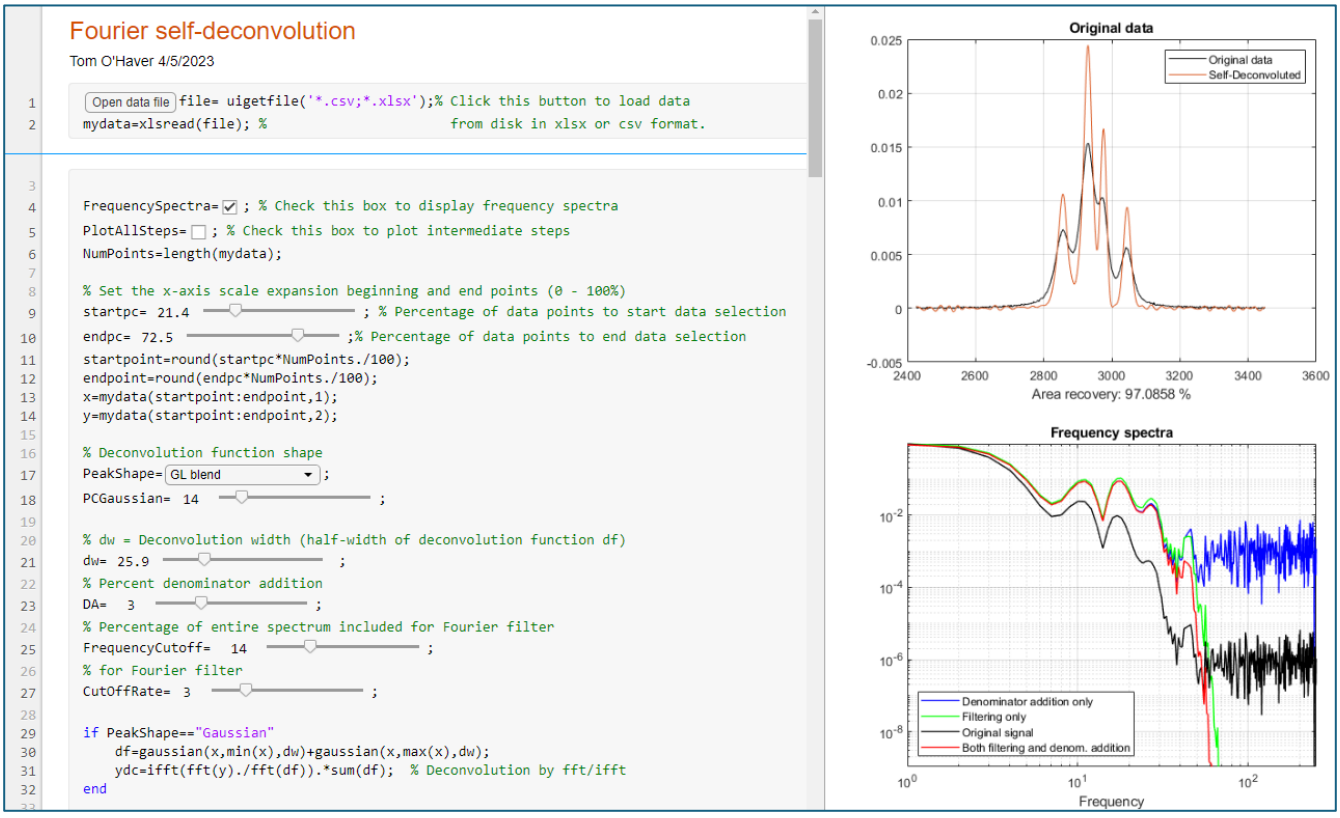
SegGaussDeconv.m and SegGaussDeconvPlot.m are the same except that they perform symmetrical (zero-centered) Gaussian deconvolution. SegDoubleExpDeconv.m and SegDoubleExpDeconvPlot.m perform symmetrical (zero-centered) exponential deconvolution. If the peak widths increase regularly across the signal, you can calculate a reasonable initial value for the vector "tc" by giving only the number of segments ("NumSegments"), the first value, "startt", and the last value, "endt":

```
tstep=(endt-startt)/NumSegments;
tc=startt:tstep:endt;
```

Live script Self-deconvolution Tool

The Live Script DeconvoluteData.mlx can perform Fourier self-deconvolution on your own data stored in disk. Clicking the "Open data file" button in line 1 opens a file browser, allowing you to navigate to your data file (in .csv or .xlsx format. The script assumes that your x,y data are in the first two columns. You can change that in lines 13 and 14). In the case shown here, the data file is 'HepteneTestData.csv', a portion of the IR spectrum of heptane, shown as the 'file' variable in the Matlab workspace. (To view the figures to the right as shown below, right-click on the right-hand panel and select "Disable synchronous scrolling").

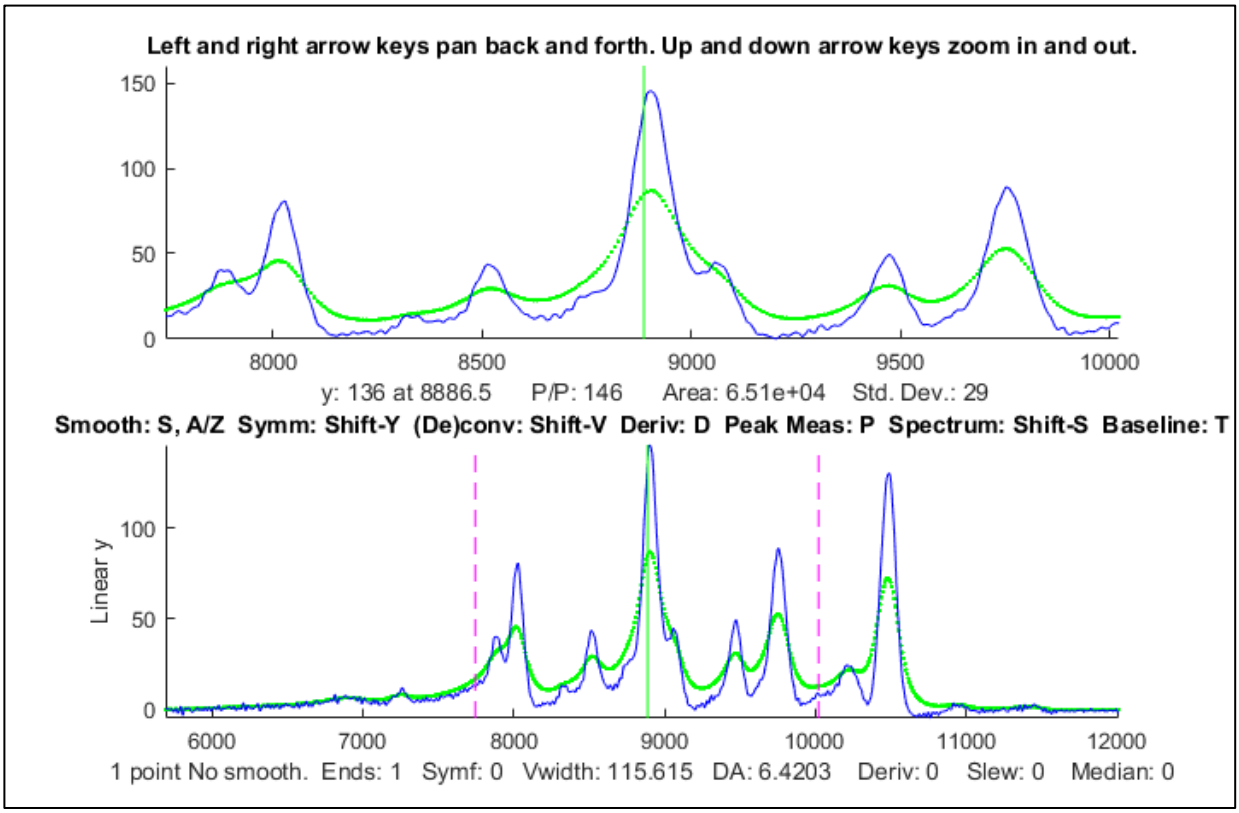
The **startpc** and **endpc** sliders in lines 9 and 10 allow you to select which portion of the data range to process, from 0% to 100% of the total range of the data file. The **PeakShape** drop-down menu in line 17 selects the convolution function shape (in this case, a Gaussian-Lorentzian blend) and the **PCGaussian** slider in the next line allows selection of the percent Gaussian of that shape. The **dw** slider in line 21 controls the deconvolution half-width, the **DA** slider in line 23 controls the percent denominator addition. Smoothing, by Fourier filtering, is controlled by the **FrequencyCutoff** and **CutOffRate** in lines 25 and 27. All variables are accessible in the Matlab workspace. The final signal is 'syDA'. Note: you can double-click any of the sliders to change their ranges if the initial range is insufficient. Click the **FrequencySpectra** check box in line 4 to view the frequency spectra. Click the **PlotAllSteps** check box in line 5 to view all the steps leading up to the final result.



See page 370 for other interactive tools.

Interactive deconvolution with iSignal

In my **iSignal** version 8.3 and later (page 376), you can press **Shift-V** to display the menu of Fourier convolution and deconvolution operations that allow you to convolute or to deconvolute a Gaussian, Lorentzian or exponential function. It will ask you for the initial width or time constant of the deconvolution function (in X units), then you can use the **3** and **4** keys to decrease or increase the width by 10% (or **Shift-3** and **Shift-4** to adjust by 1%). This version of iSignal includes an additional way to reduce ringing and noise in the deconvoluted signal, by adding a constant to the denominator (reference 85) and adjusting it with the 5 and 6 keys to decrease or increase the constant by 10% (or Shift-5 and Shift-6 to adjust by 1%).

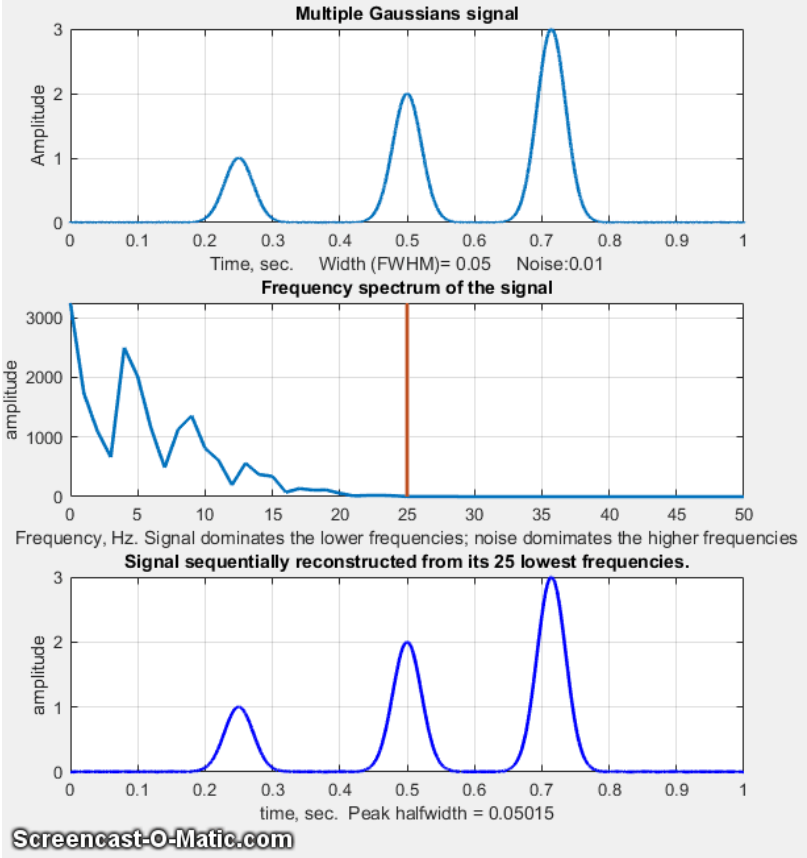


In this example, the original signal is shown as the dotted green line and the result of deconvoluting it with a *Lorentzian* deconvolution function is shown as the blue line. The deconvolution width was adjusted as large as possible without causing significant negative dips between the peaks, which for many types of experimental data, would be non-physical. (Recall that the mathematics of the deconvolution operation is structured so that the *area under the peaks remains unchanged*, even though the widths are reduced, and the heights are increased). The zoomed-in close-up in the upper panel shows that several peaks with shoulders are resolved into distinct peaks, allowing their peak positions to be measured more accurately. Fortunately, the amplitude of those revealed peaks is greater than the small amount of noise remaining in the signal (thanks to the good signal-to-noise ratio of the original signal.

Fourier Filter

A Fourier filter is a type of filtering function that is based on direct manipulation of the frequency components of a signal. It works by taking the Fourier transform of the signal, then attenuating or amplifying specific frequencies or ranges of frequencies, and finally inverse transforming the result.

In many scientific measurements, such as spectroscopy and chromatography, the signals are relatively smooth shapes that can be represented by a surprisingly small number of Fourier components. For example, the figure below (script) shows in the top panel a simulated signal with three smooth peaks, with peak heights of 1, 2, and 3, where the x-axis is time in seconds. The middle panel shows the first 50 frequencies of its Fourier spectrum, where the x-axis is frequency in Hz. The amplitude of the Fourier components is strongest at low frequencies and drops to near zero at 25 Hz.

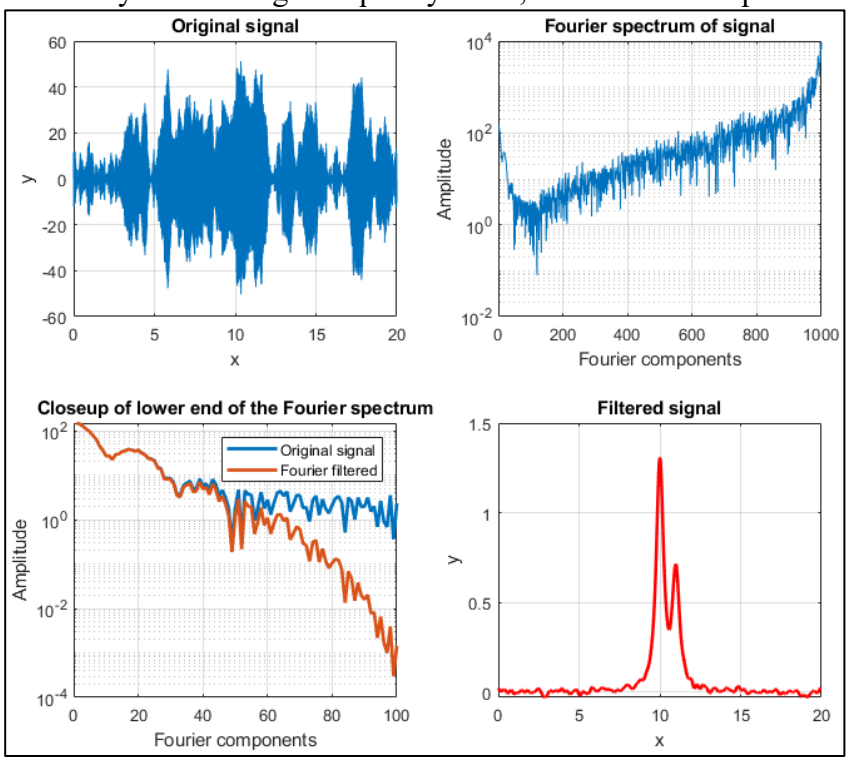


The bottom panel shows the signal reconstructed by adding up the first "n" Fourier components, where n=1, 2, 3…. A GIF *animation* of this process (visible in the Microsoft Word 365 version or click to view in any web browser) shows the results of including the frequencies between 1 through 25 progressively. The reconstructed signal starts as a big featureless blob with only a few frequencies included. The peaks emerge and become narrower as more frequencies are added, and the baseline between the peaks becomes flatter, until the result is visually indistinguishable from the original signal when 26 frequencies are included. But notice what the reconstructed signal looks like even when it gets only to 16 frequencies. By that point, the amplitude of the frequencies has already dropped very low and there is relatively little amplitude in the remaining frequencies, so the three peaks are rendered well. But the baseline has a small but distinct ripple, caused by the abrupt cut-off of the frequencies beyond that point. That can be avoided by including more frequencies or by using a filter with an *adjustable filter shape* that allows the cut-off rate to be controlled.

Optimization of the Fourier filter for the signal-to-noise (SNR) ratio of peak signals faces the same compromise as conventional smoothing functions: the optimum SNR is achieved when the peak height is less than the noiseless maximum. For example, the script GaussianSNRFrequencyReconstruction.m shows that for a Gaussian peak, the optimum SNR is reached when the peak height is about half the true value, but the peak area is the same. (White noise has equal amplitude at all frequencies, (page 27)

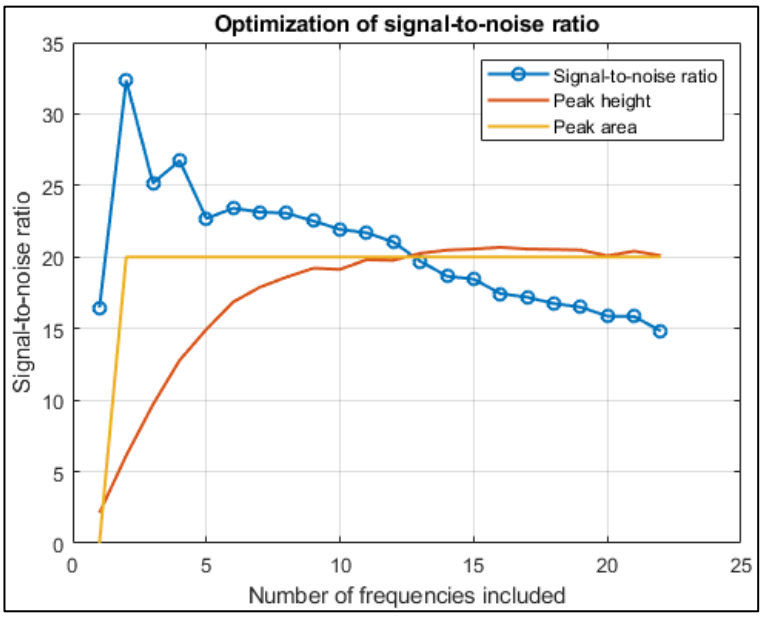
whereas most of the peak signal is concentrated in the first few frequencies).

A more dramatic example is shown the figure on the left (script). In this case, the signal (top left) seems to be only random high-frequency noise, and its Fourier spectrum (top right, shown with a log y scale) shows that high-frequency components dominate the spectrum over most of its frequency range. The bottom left panel shows the Fourier spectrum expanded in the X and Y directions to show the low-frequency region more clearly. There, the series of relatively smooth bumps, with peaks at the 1st, 20th, and 40th frequencies, are most likely the actual signal. Working on the hypothesis that the components above the 40th harmonic are increasingly dominated by noise, using a Fourier filter function (FouFilter.m) that can gradually reduce the higher harmonics and reconstruct the signal from the modified Fourier transform (red line). The result (bottom right) shows that the signal contains two partly overlapping Lorentzian peaks that were totally obscured by high-frequency noise in the original signal.



Computer software for Fourier Filtering

MATLAB. The simplest possible code for a Fourier simply cuts out all frequencies above a certain limit. To do this correctly, care must be taken to use both the sine *and* cosine (or equivalently the frequency *and* phase or the real *and* imaginary) components of the Fourier transform. The operation must account for the mirror-image structure of the Matlab's Fourier transform: the lowest frequencies are at the extremes of the fft and the highest frequencies are in the *center* portion. So, to pass the lowest *n* frequencies, you must pass the first *n* points *and* the last *n* points and *zero out the others*.

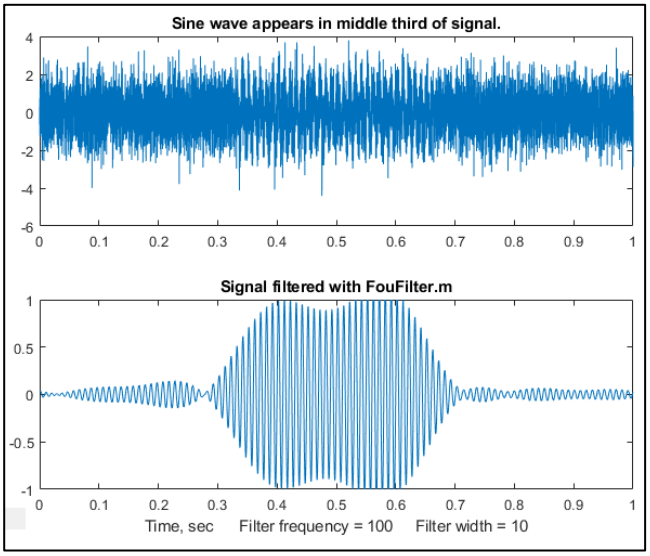


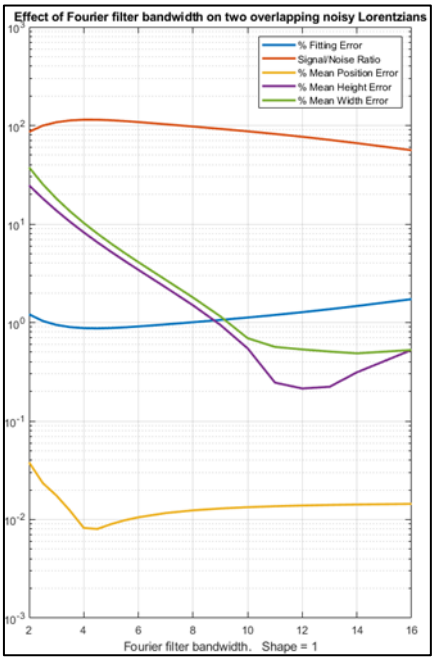
```
ffty=fft(y); % ffty is the fft of y
lfft=length(ffty); % Length of the FFT
ffty(n:lfft-n)=0; % Frequencies between n and lfft-n in the fft are set to zero.
fy=real(ifft(ffty)); % Real part of the inverse fft
```

The function form of this simple Fourier low pass filter is flp.m. This is the minimal essence of a Fourier filter, but it is not really a practical filter, however, because its abrupt cutoff usually results in ringing on the baseline, as shown above.

General-purpose Fourier filter function. To make the Fourier filter more generally useful, you must add code not only low-pass filters, but also for high-pass, band-pass, and band-reject filter modes, plus a provision for more gentle and variable cut-off rates.

The custom Matlab/Octave function FouFilter.m is a more flexible Fourier filter that can serve as a low pass, high pass, bandpass, or band-reject (notch) filter *with variable cut-off rate*. This function has as the form **`[ry,fy,ffilter,ffy] = FouFilter (y,samplingtime,centerfrequency, frequencywidth,shape,mode),`** where y is the time-series signal vector, 'samplingtime' is the total duration of sampled signal in seconds, milliseconds, or microseconds. 'centerfrequency' and 'frequencywidth' are the center frequency and width of the filter in Hz, KHz, or MHz, respectively. 'Shape' determines the sharpness of the cut-off. If shape = 1, the filter is Gaussian. As shape increases the filter shape becomes more and more rectangular. Set mode = 0 for band-pass filter, mode = 1 for band-reject (notch) filter. FouFilter returns the filtered signal in 'ry'. It can handle signals of virtually any length, limited only by your computer's memory. Here are two example scripts that call FouFilter.m: TestFouFilter.m demonstrates a Fourier bandpass filter applied to a noisy 100 Hz sine wave that appears in the middle third of the signal record, shown in the figure above. You can see that this filter is effective in extracting the signal from the noise, but that the response time is slow. The script TestFouFilter2.m demonstrates a Fourier bandpass filter applied to a noisy 100 Hz sine wave signal with the filter center frequency swept from 50 to 150 Hz. Both require the FouFilter.m function in the Matlab/Octave path.



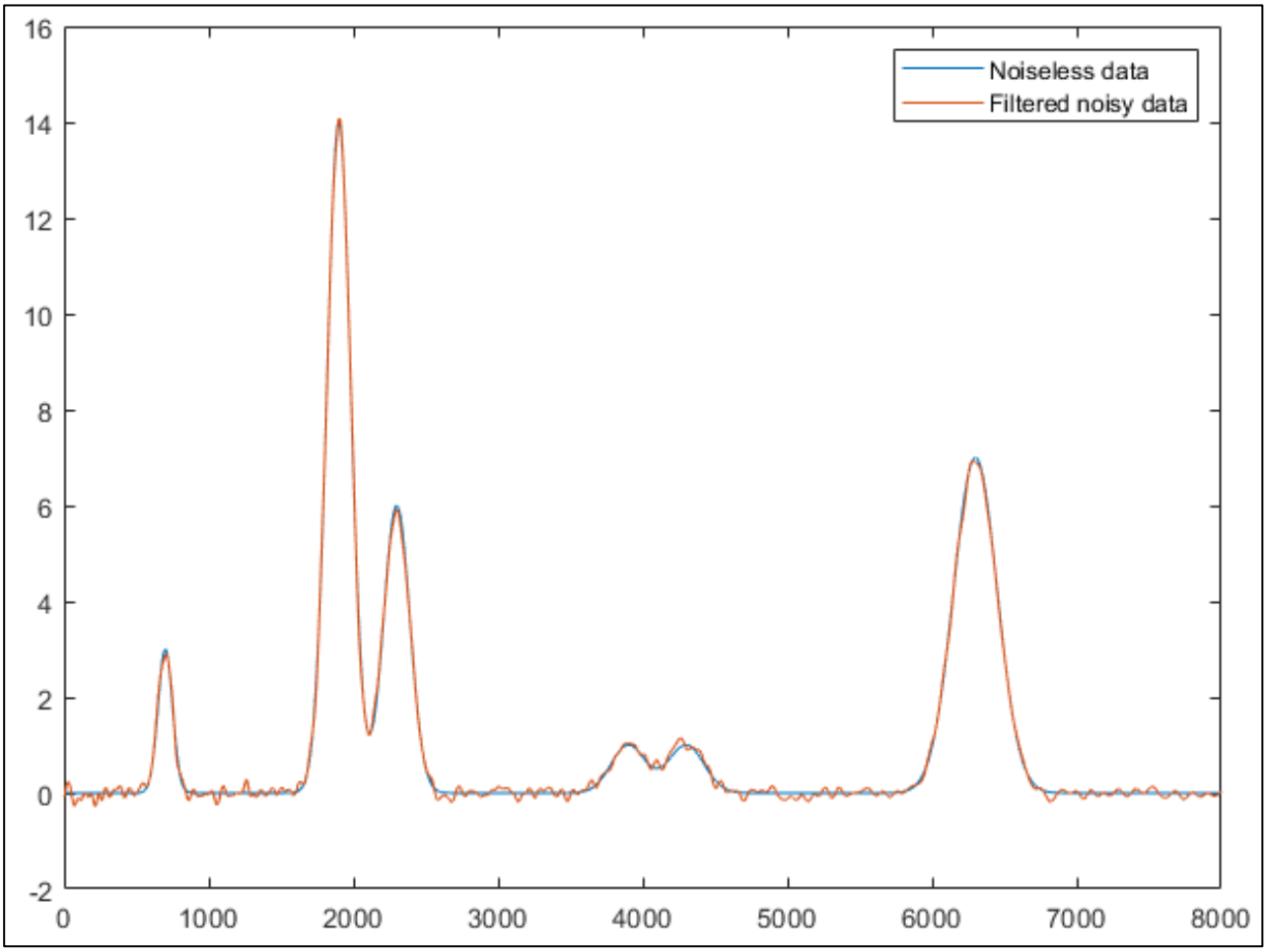


The figure on the right (Matlab script) demonstrates the effect of the bandwidth of a Fourier low-pass filter applied to a typical peak signal with white noise. The graph shows a semi-log plot of the signal-to-noise ratio (red) and the percent errors in the peak parameters (height, width, and position) as a function of filter width. The

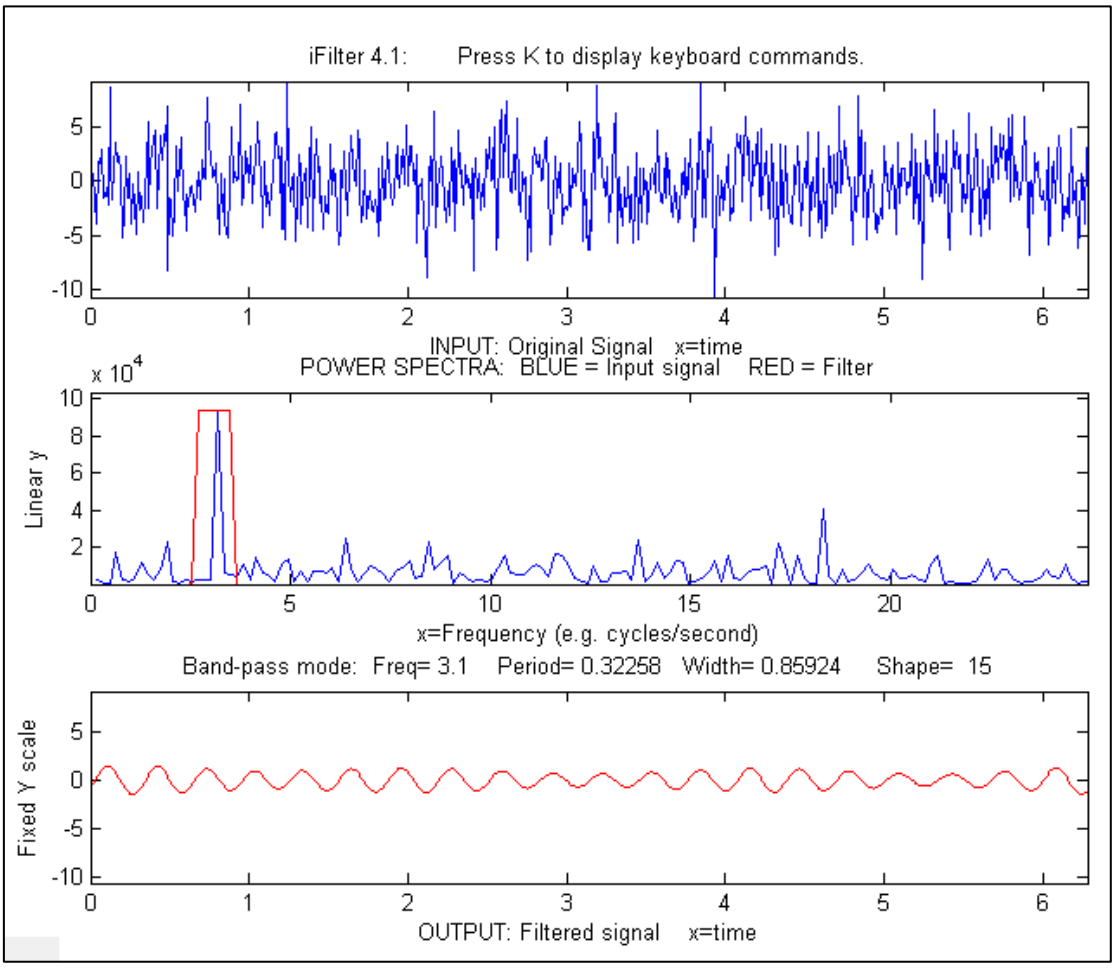
signal to noise ratio is also shown. As usual, the results are poor if the bandwidth is either too low or too high, but in this case the signal-to-noise ratio is best at a relatively low bandwidth whereas most of the peak height and width measurements are most accurate at much higher values. A slightly different result is obtained with shape=2, for which the cut-off rate is faster. As usual, compromise is a must.

The interactive Live Script data smoothing tool includes Fourier filtering among several other data smoothing methods. (Page 59. The download link is: DataSmoothing.mlx).

Segmented Fourier filter. SegmentedFouFilter.m is a *segmented* version of FouFilter.m, which applies different center frequencies and widths to different segments of the signal. The syntax is the same as FouFilter.m except that the two input arguments *centerFrequency* and *filterWidth* must be vectors with the values of *centerFrequency* and *filterWidth* for each segment. The function divides the signal into several equal-length segments determined by the length of *centerFrequency* and *filterWidth*, which must be equal in length. For help and examples, type "help SegmentedFouFilter". The figure below shows Example 2, which demonstrates a Fourier low-pass filter with decreasing bandwidth as the peak widths become wider from left to right. If you look closely, you can see that the random noise in the filtered signal, which was constant in the original raw signal, decreases from left to right as the filter width increases.



Further Matlab-based applications using an *interactive* Fourier filter, iFilter.m, shown above, allows you to adjust the filter parameters with keystrokes while observing the effect on your signal dynamically. This is described on page 391. There is also a version for Octave users, ifilteroctave.m which uses different keys for the filter center and width adjustments.



A demonstration of a *real-time* Fourier filter is discussed on page 347.

A Live Script version of this interactive Fourier filter is described on page 396.

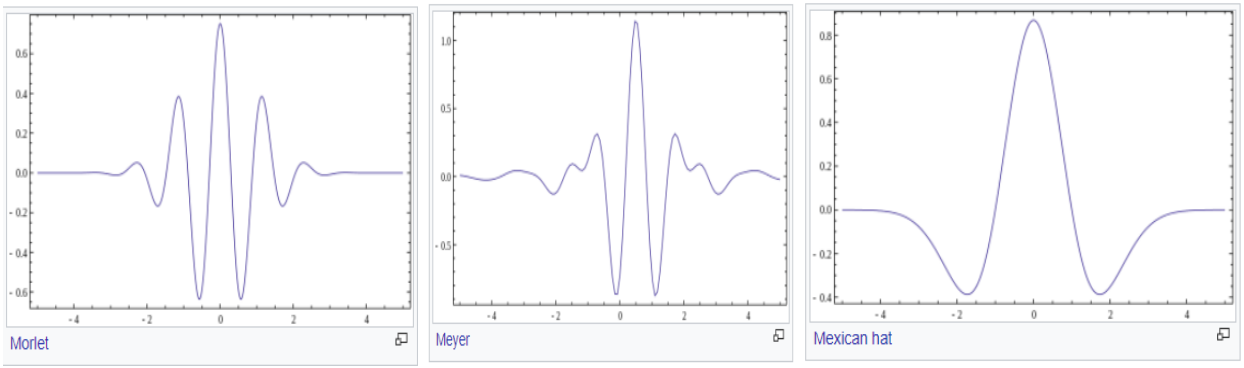
Wavelets and wavelet denoising

Wavelets are literally "little waves", small oscillating waveforms that begin from zero, swell to a maximum, and then quickly decay to zero again. They can be contrasted to sine or cosine waves, which go on "forever", repeating out to positive and negative infinity. In the previous sections you have seen how useful it is to use the Fourier Transform of a signal, which expresses a signal as the sum of sine and cosine waves, allowing such useful operations as convolution, deconvolution, and Fourier filtering. But there is a downside to the Fourier Transform. It covers the *entire* signal duration, giving only the *average* frequency content. Although you saw on page 104 that it is possible to use segmented or time-resolved variations of the Fourier transform to overcome this difficulty, a more sophisticated way to solve this limitation of Fourier analysis is to use wavelets as a basis set for representing signals rather than sine and cosine waves. Like sine waves, wavelets can be stretched or compressed along their "x" or time axis to cover different frequencies. But unlike sine waves, wavelets can be *translated along the time axis* of a signal to probe the time variations, because wavelets are of short duration compared with the signals they are used with.

Wavelets were introduced by mathematicians and mathematical physicists in the early years of the 20th century and the subsequent development has been highly mathematical. Many of the treatments of wavelets in the literature are aimed at the formal mathematical aspects, which have been "worked out in excruciating detail" (according to reference 82). The value system of mathematics – rigorous proofs, exhaustive exploration, assumption of mathematical background, and the need for compact notation - makes it difficult for the non-specialists. Because of this, there are many "easy" introductions to the subject (references 79 - 82) that promise to soften the blow of mathematical abstraction. For that reason, I will not repeat all those mathematical details here. Rather, I will attempt to show what you can accomplish using wavelets *without* understanding all the underlying mathematics. I am particularly interested in situations when wavelets work better than the best available conventional techniques, but I also call out a few situations where the conventional techniques remain superior.

A wavelet transform (WT) is a decomposition of a signal into a set of "basis functions" consisting of contractions, expansions, and translations of a wavelet function (reference 85). It can be computed by repeated convolution of the signal (page 109) with the chosen wavelet as the wavelet is translated across the time dimension (to probe the time variation) and as it is stretched or compressed (to probe different frequencies). Because two dimensions are being probed, the result is naturally a 3D surface (time-frequency-amplitude) that can be conveniently displayed as a time-frequency contour plot with different colors representing the amplitudes at that time and frequency. Of course, such calculations will require more complex algorithms and greater execution times, often taking about 5 to 20 times longer than conventional methods. That might have been a problem in the early days of computers, but with modern fast processors and great memory capacity, it is less likely to be of concern now.

Wavelets are used for the visualization, analysis, compression, and denoising of complex data. There are dozens of different wavelet shapes, which by itself is a big difference from sinewave-based Fourier analysis. The Wikipedia article on wavelets mentions three of them, which are shown below, from left to right: the Meyer, the Morlet and the "Mexican hat". Wavelets are conventionally constructed so that the area under the curve is zero and the integral of their squares is 1.0.

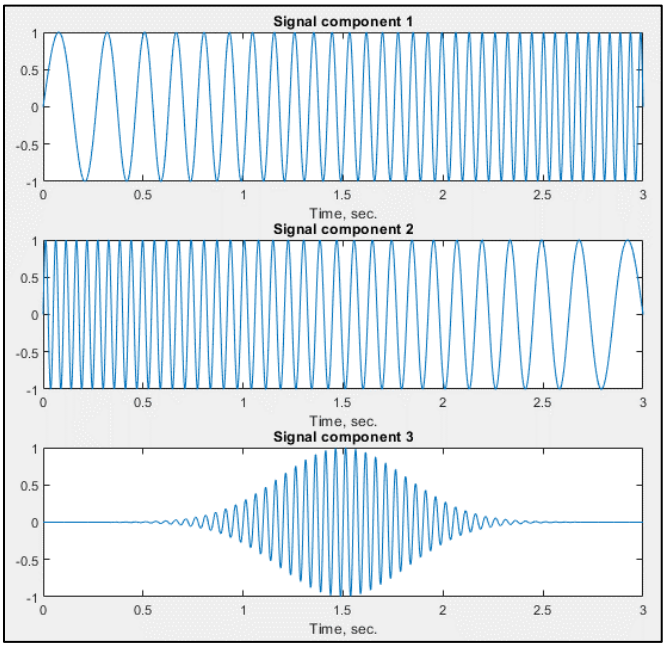


In Matlab, the easiest way to access these tools is to use the *Wavelet Toolbox*, if that is included in your school or company campus Matlab site license or if you purchase it separately. This toolbox includes a graphical user interface (GUI) for a Wavelet Analyzer, Signal Multiresolution Analyzer, and a Wavelet Signal Denoiser, as well as an extensive collection of command-line wavelet functions. Documentation is available at https://www.mathworks.com/products/wavelet.html.

However, it is not necessary to have the Wavelet Toolbox, because wavelet code has been published on the Internet in a variety of languages. For example, several papers (reference 84, 90) include or reference Matlab or Python code that implement wavelets using only inbuilt or readily imported functions. In this chapter you will see all these software approaches to describe the properties and applications of wavelets to scientific measurement.

Visualization and analysis

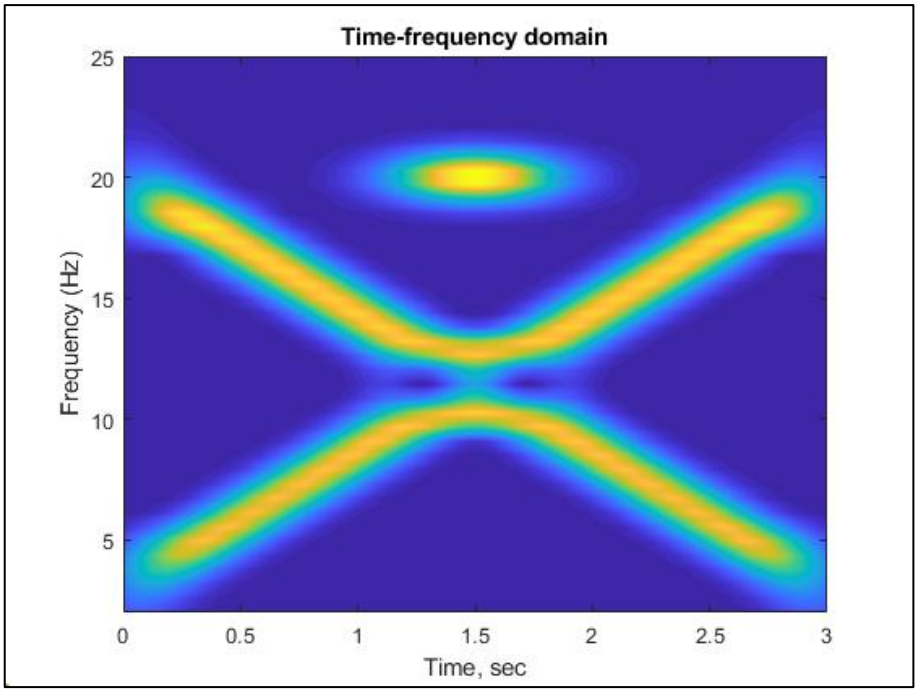
Wavelets are quite effective at visualizing complicated signals and helping observers make sense of them. A good example is given in reference 84, which describes a 3-second-long signal sampled at 1000 Hz consisting of three overlapping components that are initially unknown to the experimenter. These components are shown in the figure below: (1) a swept sine wave (called a 'chirp') going from 5 Hz to 20 Hz, (2) another simultaneous 'chirp' going in reverse from 20 Hz to 5 Hz, and finally (3) a Gaussian-modulated 20 Hz sine wave that peaks in the center of the signal.



When there three are added are up, the resulting waveform, shown in the upper panel of the figure below (displayed in iSignal.m) is a complicated jumble that offers no hint of its underlying structure. The conventional Fourier transform spectrum, shown in the lower panel, shows only that the frequency components of the signal fall in the range between about 3 Hz and 25 Hz. In fact, the Fourier spectrum is misleading. It suggests that there might be *two* components, one at a higher frequency range than the other, with a small gap in between near 12 Hz. But there are in fact *three* components, two of them covering a wide frequency range and the third one fixed at about 12 Hz. The Fourier spectrum is no help there.



In contrast, the wavelet-based time-frequency-amplitude contour shown below, which was computed using the Morlet wavelet by the Matlab code in reference 86, helps to unravel the complexities, showing all three components clearly. In this display, *yellow* corresponds to the greatest amplitudes and

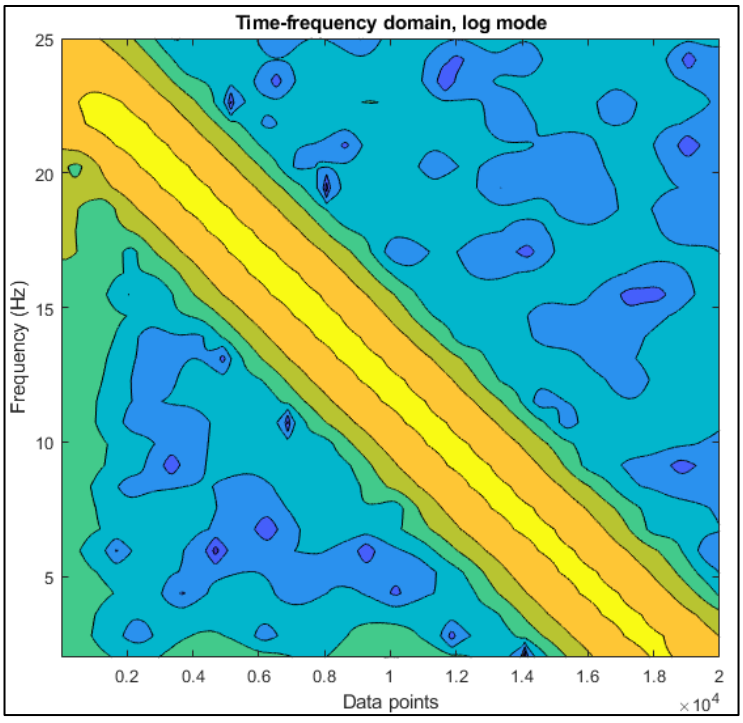


blue to the lowest. The Gaussian-modulated 20 Hz sine wave is clearly visible at the top center. (Incidentally, there is an ambiguity concerning the two swept sine waves at the point where they cross in frequency in the middle of the signal and cancel out momentarily. Do they keep going in the same direction, forming an "X", or do they both reverse direction, forming a "V" and its reflection? The two behaviors would result in the same final signal. The simplest assumption would be the former).

Another example is closer to a typical scientific application: digging a signal out of an excess of noise and interference. This one is based on the "buried peaks" signal that I used before on page 105. The signal (top panel in the iSignal screen on the left) has a pair of hidden Gaussian peaks that are totally buried in a much stronger interfering swept-frequency sine wave and random white noise. The Fourier spectrum, observed here in the bottom panel, again offers no obvious hint of the underlying peaks.



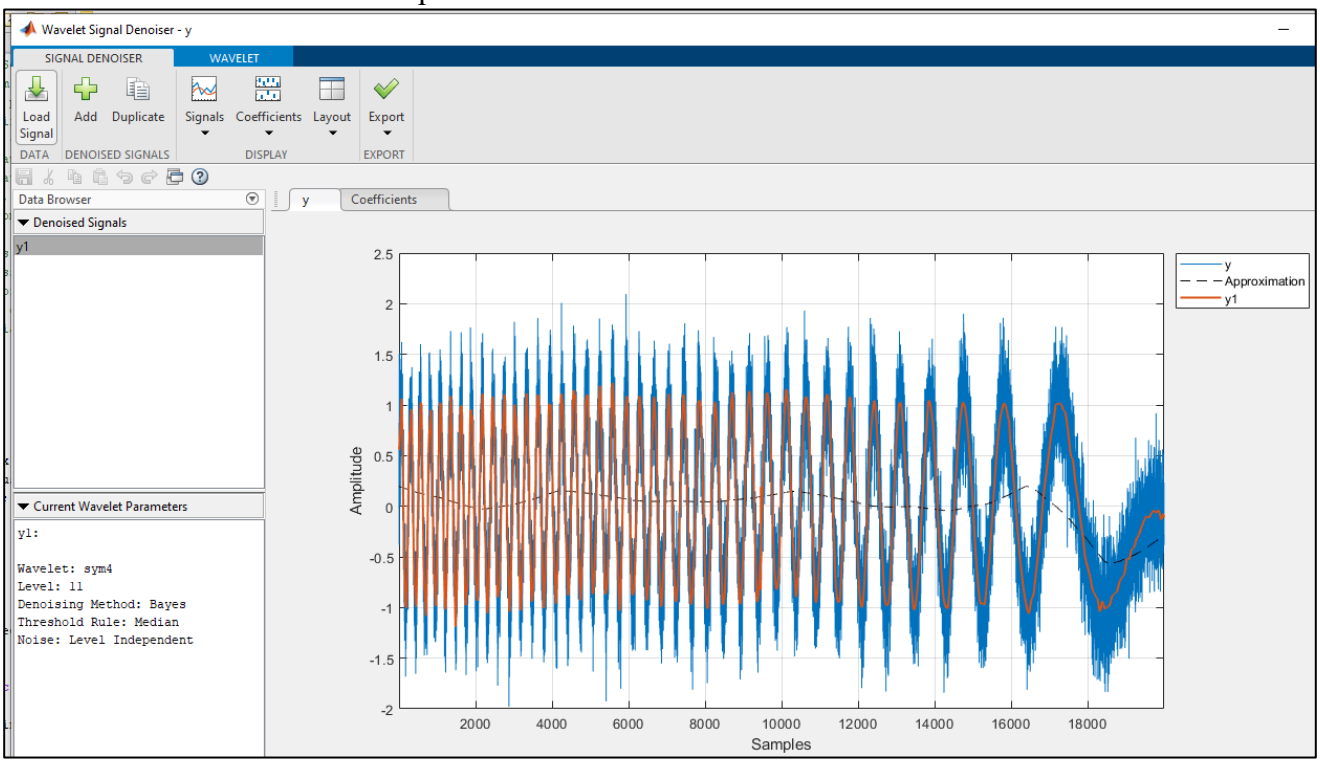
Here I applied the Morlet wavelet to this signal to create the time-frequency-amplitude matrix shown on the left (script and Morlet wavelet function). Don't be distracted by the big yellow diagonal stripe. That corresponds to the swept sine wave. Look instead at the two weaker green humps at the bottom left, at the low-frequency end, near data points 5000 and 10000. Those are the two Gaussian peaks. On the basis of that observation, you would be justified to perform smoothing or curve-fitting in that specific region, as we did before on page 105. (You can compare this graphic to the segmented Fourier spectrum display for this signal shown on that page, which is cruder but displays similar information. The wavelet is clearly a finer-grained tool).



Wavelet denoising

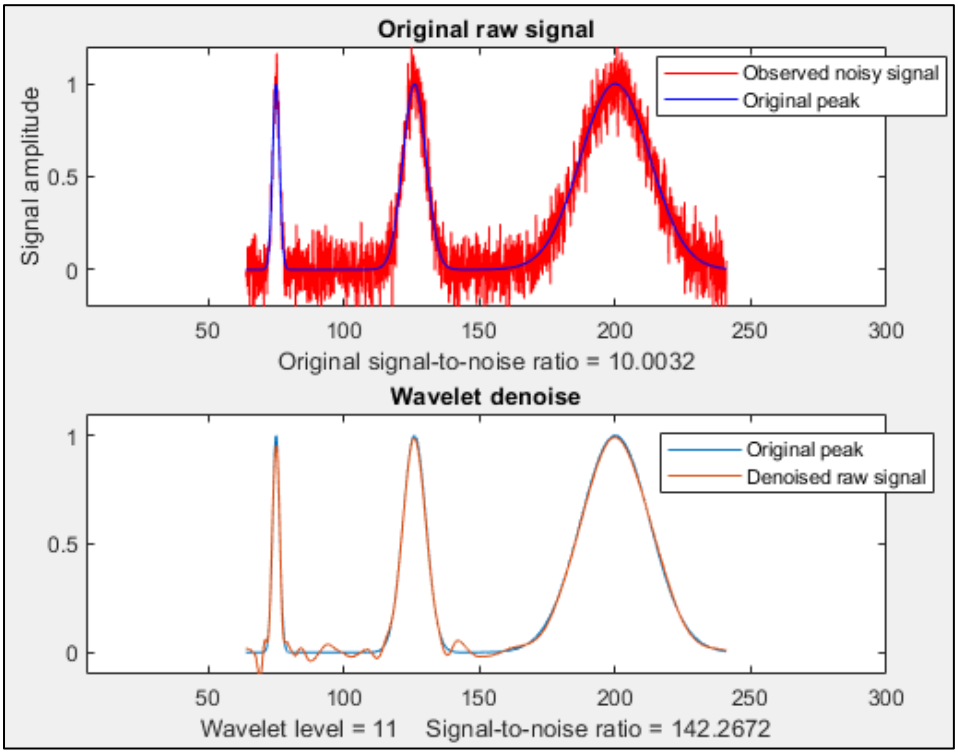
In the context of wavelets, the term "denoising" means reducing the noise as much as possible without distorting the signal. Denoising makes use of the time-frequency-amplitude matrix created by the wavelet transform. It assumes that the undesired noise will be separated from the desired signal by their frequency ranges. Most commonly in scientific measurements, the desired signal components are located at relatively *low* frequencies and the noise is mostly at *higher* frequencies. The process is controlled both by the selection of wavelet type and by a positive integer number called the wavelet "level". The higher the level, the lower is the frequency divider between signal and noise. (To that extent, the wavelet level is qualitatively like the smooth width of a smoothing operation).

Again, Matlab's Wavelet Toolbox provides some useful tools. First, there is the GUI app called the "Wavelet Signal Denoiser". The selection of the wavelet type and level are all selectable manually in that app. I used it to analyze the "buried peaks" signal described on the previous page, using the "sym4" wavelet at a relatively high level of 11, because lower levels allow too much of the interfering swept sine wave to come through and higher levels would damp out the Gaussian peaks too much. (The number in the wavelet name refers to the number of so-called "vanishing moments". More vanishing moments means that it can represent more complex functions). The "Approximation" result (the dotted line in the graph below) is the *low-frequency* information in the data, and you can clearly see that this is a "denoised" version of the original signal (shown in blue). The two bumps at sample numbers 5000 and 10000 are the two Gaussian peaks.

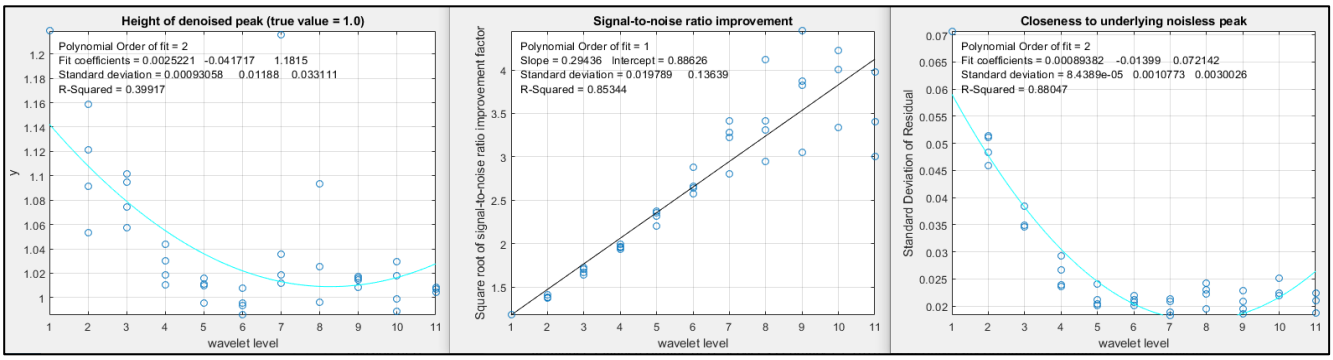


So, both the sym4 wavelet in the Wavelet Signal Denoiser pictured above and the Morlet wavelet's time-frequency-amplitude matrix shown on the previous page give evidence of the hidden Gaussian peaks, but they display them in different ways.

In addition to the GUI app, there is also a command-line denoising function called "wdenoise" (syntax: **`wdenoise(noisydata,level,…)`**. The selection of the wavelet type and level are set by including optional input arguments to this function. The advantage of a *function*, compared to a GUI app, is that it is possible to write scripts that quickly and automatically compare many different wavelet settings, or that compare the results to several conventional noise reduction methods, or that automate the batch processing of large numbers of stored data sets (see page 345). For example, the question of the optimal selection of wavelet level is answered by the script OptimizationOfWaveletLevel3peaks.m, which creates a signal consisting of three noisy unit-height Gaussian (or Lorentzian) peaks with different peak widths, with added white noise, as in the following figure.



The script uses the wdenoise.m function to denoise the signal with the "coiflet" wavelet from levels 1 to 11, measuring three quantities for each level: (a) the height of the peaks, (b) the signal-to-noise ratio improvement, and (c) the closeness to the noiseless underlying signal, as shown in the three plots below.



You can see from these plots that a level of about 7 is optimum in this case. Above 7, the signal-to-noise ratio (center graph) continues to improve, but the results are unreliable and tend to scatter around too much. (Changing to Lorentzian peaks - line 28 of the script - yields similar results).

The script WaveletsComparison.m compares five different wavelet types on the same signal: BlockJS, bior5.5, coif2, sym8, and db4, all at level 12 (graphic). The results are similar but the sym8 has a slight edge. For most smooth peak shapes with additive white noise, the different wavelet types perform similarly. For signals with *high*-frequency weighted noise, the bior5.5 wavelet works better than the others (script; graphic). For square pulses, the Haar wavelet is clearly superior.

Another script, SmoothVsWavelets2Gaussians.m, compares five different non-wavelet smoothing techniques and two different wavelets, all using the same simulated signal consisting of two Gaussian peaks with a *50-fold difference* in peak width, with additive white noise. For each method, the percent

errors in the peak height, width, and area are measured, as well as the difference between the underlying noiseless signal and the denoised (or smoothed) noisy signal. This illustrates a significant advantage that wavelet denoising has over smoothing: *it adapts much better to differences in peak width*. A summary of typical results is shown in this table. (Peak 1 is the narrower peak).

Typical results		Percent errors of peak1|peak2		
Method	Residuals	Height	Width	Area
Original	9.88%	6.29%|25.8%	6.31%|-23.24%	-2.49%|0.86%
Gaussian	2.53%	-3.34%|-3.79%	5.72%|4.35%	-2.6%|0.82%
Segmented	2.04%	-24.48%|3.09%	37.8%|0.21%	-7.07%|0.85%
Savitsky-Golay	2.93%	-2.08%|6.45%	8.66%|-3.04%	-1.9%|0.86%
RC filter	3.29%	-6.53%|9.58%	16.06%|-5.45%	-11.76%|0.86%
Hamming filter	2.91%	-5.12%|8.7%	8.19%|-5.31%	-2.17%|0.86%
coif2 wavelet	1.02%	1.78%|1.18%	-5.59%|0.22%	-6.54%|0.75%
db2 wavelet	1.17%	11.47%|3.82%	3.36%|2.38%	-5.34%|0.81%

The "Residuals" are the percent differences between the underlying noiseless signal and the signal with random noise after denoising. It accounts for both the residual noise in the signal and distortion of the signal shape. As you can see, *the coif2 wavelet comes out ahead by most measures*. This illustrates the most significant practical advantages of wavelet denoising: (1) it gives results that are at least as good as, and often better than, conventional smoothing methods. (2) it is easier to use because it automatically adapts to different peak widths. And (3) it is easier to optimize because in most cases only the level setting makes much difference in the practical results.

However, there are a few situations where conventional methods are still better. For example, in calculating the second derivatives of noisy peaks of variable width, a segmented Gaussian-weighted smooth (page 333) gives a signal-to-noise ratio better than that of a wavelet denoise (script; graphic), especially if the signal-to-noise ratio is poor (graphic), presumably because the frequency spectrum of the noise is so strongly high-frequency weighted. Also, wavelet denoising does not work at all if the amplitude of the noise is *proportional* to the signal amplitude, rather than *constant* (script; graphic). Sometimes, if the original signal-to-ratio is very poor, wavelet denoising produces narrow spike artifacts in the denoised signals, even when soft thresholding is used. These are special cases, however. There are many more situations where the wavelet denoise is really the method of choice, assuming that the slower speed of wavelet methods is not an issue.

The signal-to-noise ratio improvement performance of wavelet denoising is compared to traditional smoothing methods, as a function of smooth ratio, in the previous chapter on smoothing, page 60. The wavelet method performs better but is 10 times slower than even the Savitzky-Golay smooth.

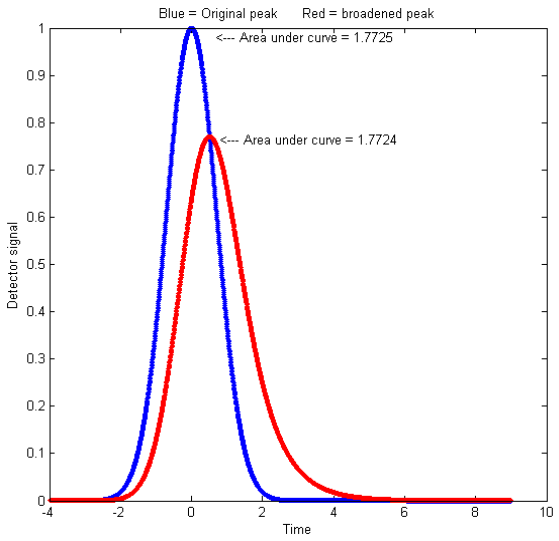
Wavelet denoising is one of the several types of noise reduction smoothing available in the Live Script data smoothing tool described on page 59. This is a very convenient way to compare wavelet denoising to other forms of noise reduction, such as the moving average, Savitsky-Golay, and Fourier filtering.

For Python programmers, there is a wavelet package called PyWavelets that has over 100 built-in wavelet filters, support for custom wavelets, and is compatible with the Matlab wavelet toolbox. See also "A gentle introduction to wavelet for data analysis" for a graphics-intensive treatment that is based on Python.

Integration and peak area measurement

Symbolic integration of functions and calculation of definite integrals are topics that are introduced in elementary Calculus courses. The numerical integration of digitized signals is applied in analytical signal processing as a method for measuring the areas under the curves of peak-type signals.

For example, peak area measurements are very important in chromatography, a class of chemical measurement techniques in which a mixture of components is made to flow through a chemically prepared tube or layer that allows some of the components in the mixture to travel faster than others, followed by a device called a *detector* that measures and records the components after separation.



Ideally, the components are sufficiently separated so that each one forms a distinct *peak* in the detector signal. The magnitudes of the peaks are calibrated to the concentration of that component by measuring the peaks obtained from "standard solutions" of known concentration. In chromatography it is common to measure the *area* under the detector peaks rather than the *height* of the peaks because peak areas are less sensitive to the influence of random noise and to peak broadening (dispersion) mechanisms that cause the molecules of a specific substance to be diluted and spread out rather than being concentrated on one "plug" of material as it travels down the tube or layer. These dispersion effects, which arise from many sources, cause chromatographic peaks to become shorter, broader, and in some cases more unsymmetrical, but they have *little effect on the total area under the peak*, if the total number of molecules remains the same. If the detector response is linear with respect to the concentration of the material, the peak *area* remains proportional to the total quantity of substance passing into the detector, even though the peak *height* is smaller. A graphical example is shown on the left (Matlab/Octave code), which plots detector signal vs time, where the **blue curve** represents the original signal and the **red curve** shows the effect of broadening by dispersion effects. The peak height is lower, and the width is greater, but the *area* under the curve is almost the same. If the extent of broadening changes between the time that the *standards* are run and the time that the unknown *samples* are run, then *peak area measurements will be more accurate and reliable* than peak height measurements. (Peak height will be proportional to the quantity of material only if the peak width and shape are constant). Here is another example with greater broadening: (script and graphic).

Peak area measurements are occasionally used also in *spectroscopy*, for example in flow injection methods and in graphite furnace atomic absorption (reference 87). In that application, calibration curves based on area measurements are more linear than peak height measurements because most of the area of a peak is measured when the transient absorbance is less than maximum and where Beer's Law is more strictly obeyed.

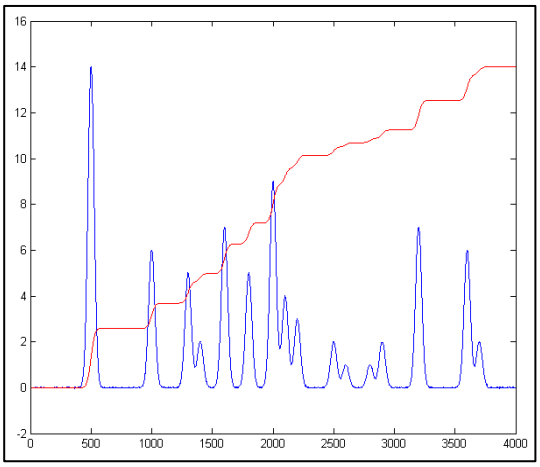
Conversely, peak height measurements are *simpler to make* and are *less prone to interference* by neighboring, overlapping peaks. And a further disadvantage of peak area measurement is that the peak

start and stop points must be determined, which may be difficult especially if the multiple peaks overlap. In principle, curve fitting (page 199) can measure the areas of peaks even then they overlap, but that requires that the shapes of the peaks be known at least approximately (however, see PeakShapeAnalyticalCurve.m described on page 337).

Chromatographic peaks are often described as a Gaussian function or as a convolution of a Gaussian with an exponential function. A detailed quantitative comparison of peak height and peak area measurement is given in on page 314: Why measure peak area rather than peak height? (In spectroscopy, there are other broadening mechanisms, such as Doppler broadening caused by thermal motion, which results in a Gaussian broadening function).

Before computers, researchers used a variety of clever but archaic methods to compute peak areas:

(**a**) Plot the signal on a gridded paper chart, cut out the peak with scissors, then weigh the cutout piece on a micro-balance compared to a square section of a known area.
(**b**) Count the grid squares under a curve recorded on gridded graph paper.
(c) Use a mechanical ball-and-disk integrator.
(**d**) Use a straight-edge to construct a triangle with its sides tangent to the sides of the peak, and calculate the geometrical area of that triangle.
(**e**) Calculate the cumulative sum of the signal magnitude and measure the heights of the resulting steps, as in the figure below. (This is a method that was formerly used in proton NMR spectroscopy, where the area under each peak or group of peaks is proportional to the number of equivalent hydrogen atoms responsible for that peak).



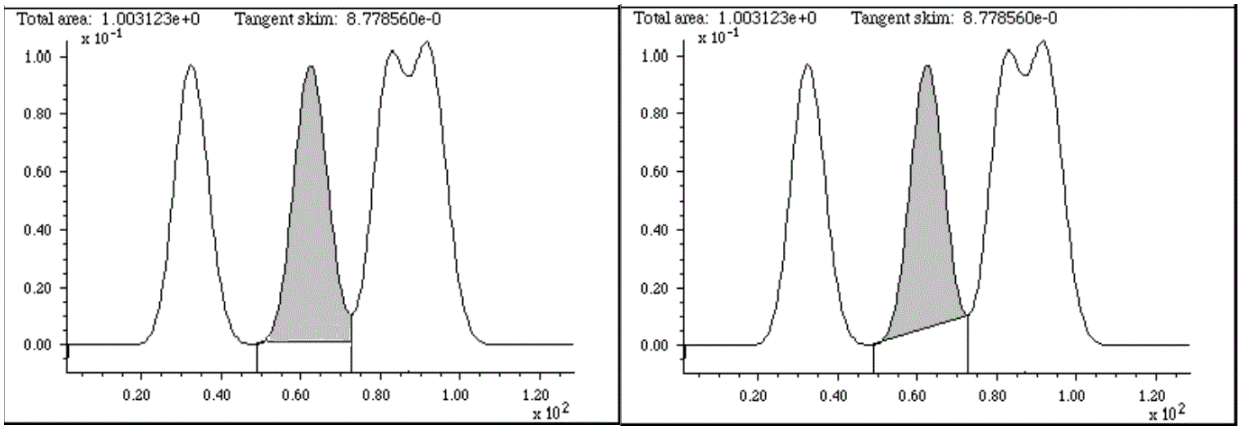
Now that computing power is built into or connected to almost every measuring instrument, more accurate and convenient digital methods can be employed. No matter how it is measured, the *units* of peak area are the *product* of the x and y units. Thus, in a chromato-gram where the x is time in minutes and y is volts, the area is in volts-minute. In absorption spectra where the x is nm (nanometers) and y is absorbance, the area is absorbance-nm. Because of this, the numerical magnitude of peak area will always be different from that of peak height. If you are performing a quantitative analysis of unknown samples by means of a calibration curve, you must use the same method of measurement for both the standards and the samples, *even if the measurements are inaccurate*, as long as the *error is the same* for all standards and samples (which is why an approximate method like triangle construction works better than expected).

The best method for calculating the area under a peak depends on whether the peak is isolated or overlapped with other peaks or superimposed on a non-zero baseline or not. For an isolated peak, Yuri Kalambet (reference 72) has shown that the trapezoidal rule area, such as calculated by Matlab's "trapz.m" function, is an efficient estimate of the full peak area with extraordinary low error, even if there are only a few data points across the width of the peak, whereas Simpson's rule is less efficient in full area integration. For a Gaussian peak, the trapezoidal rule requires only 0.62 points per standard deviation (2.5 points within the 4*sigma basewidth) to achieve an integration error of only 0.1%. A

digital simulation supports this result. For asymmetrical peaks, however, more data points are required.

Dealing with overlapping peaks

The classical way to handle the overlapping peak problem is to draw two vertical lines from the left and right bounds of the peak down to the x-axis and then to measure the total area bounded by the signal curve, the x-axis (y=0 line), and the two vertical lines, shown the shaded area in the figure on the left.

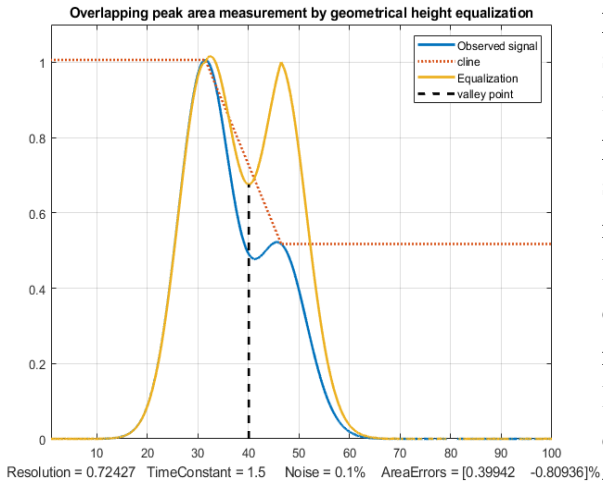


*Peak area measurement for overlapping Gaussian peaks, using the **perpendicular drop method** (left, shaded area) and **tangent skim method** (right, shaded area).*

This is often called the ***perpendicular drop*** method. It would be tedious to do by hand but is an easy task for a computer. The left and right bounds of each peak are either taken as (a) the valleys (minima) between the peaks or (b) as the point half-way between the adjacent peak centers. The half-way point method has the advantage that the SNR at a signal *maximum* is generally better than at a *minimum*, so it's likely that maxima can be more precisely located than minima. The basic assumption of the perpendicular drop method is that the area missed by cutting off the feet of one peak is made up for by including the feet of the adjacent peak. This works well enough if the peaks are Gaussian, symmetrical, not too overlapped, and not too different in height. In addition, the baseline must be zero. Any extraneous background signal must be subtracted before measurement. Using this method, it is possible to estimate the area of the second peak in the example above to an accuracy of about 0.3%. The last two peaks, however, give errors greater than 4%, and that is only because the two peaks in this example have the same height and width. More generally, the error is much more if two peaks overlap this much. As a rough rule, the valley between the peaks must be quite low, perhaps a quarter or a fifth of the lower adjacent peak height, for this method to be acceptable. Other geometrical methods exist that can reduce such errors in many cases. Sharpening peaks by area-conserving methods such as self-deconvolution (page 114) or by derivative-based methods (page 79) reduces the degree of overlap and can greatly reduce the peak area measurement errors made by the perpendicular drop method, for example as in the Excel/OpenOffice Calc PeakSharpeningAreaMeasurementDemo.xlsm (screen image). Moreover, *asymmetrical* peaks that are the result of exponential broadening can be symmetrized by the weighted addition of its first derivative, making the perpendicular drop areas more accurate (page 145). In both cases, it may be necessary to set the strength of sharpening higher than previously recommended, if it that is the only way to form a valley between peaks whose areas you want to measure. In the case where a single peak is superimposed on a straight or broadly curved baseline, you might use the ***tangent skim method***, which measures the area between the curve and a

linear baseline drawn across the bottom of the peak (e.g., the *shaded area* in the figure on the right, above). In general, the hardest part of the problem and the greatest source of uncertainty is determining the shape of the baseline under the peaks and determining when each peak begins and ends. Once those are determined, you subtract the baseline from each point between the start and endpoints, add them up, and multiply by the x-axis interval. Incidentally, smoothing a noisy signal does not change the areas under the peaks, but it may make the peak start and stop points easier to determine. The downside of smoothing is that increases peak width and the overlap between adjacent peaks. Numerical methods of peak sharpening, for example, derivative sharpening and Fourier deconvolution, can help with the problem of peak overlap, and both of these techniques have the useful property that they do not change the total area under the peaks. See 3peaks.pdf for a series of examples (Matlab script GLSDPerpDropDemo16.m). Also see the appendix on page 373 for another example.

Other methods. Although the perpendicular drop method remains the standard, here are two other possible geometrical methods that can work better in some cases. The "equalization" method, illustrated in the figure on the left, uses another method of locating the perpendicular drop point. A set of three straight-line segments is constructed that touches the estimated maxima of the two peaks, shown by the dotted red line labeled "cline" in the figure on the left. The quotient of the original signal, in blue, divided by this line, results in a temporarily normalized signal (the yellow line) that has more nearly equal peak heights. The effect of this treatment is to *deepen the valley* between the peaks, so that it remains distinct for lower values of the second peak height. *This is used only for the purpose of determining the separation point between the peaks*, shown as a vertical black line, and then is discarded. The perpendicular drop areas are then calculated on the *original* observed signal (blue line). Note that this new separation point is not quite the same as the valley of the original signal, nor is it exactly the half-way point between the two peak positions. This operation would be difficult to do by hand, but software can do it easily, given only an initial estimate for the two peak positions based on the observed signal.



The "reflection/subtraction" method, shown on the right, is simpler, but it requires that the larger peak be perfectly symmetrical. An estimate of the isolated larger peak is constructed by reflecting its left-hand half and using it to replace the right-hand half, resulting in the red dotted line in the figure. Then that estimated peak is simply subtracted from the entire signal to reveal the isolated second peak (dotted yellow line). The two areas are then separately calculated by the "trapz" function. This process is also easily automated, given only the peak position of the first peak. It works perfectly only if the larger peak is symmetrical and if the peak separation is sufficient so that the left-hand tail of the smaller peak does not significantly increase the height of the first peak.

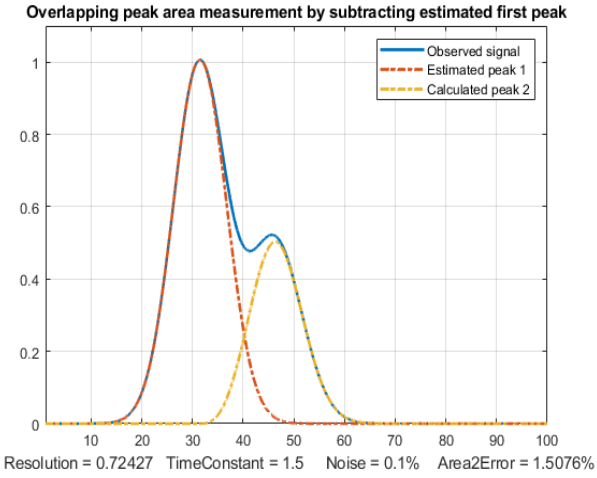
If the *shape* of peaks is known, a good way to measure the areas of overlapping peaks is to use *least-squares curve fitting*, as discussed starting on page 173. If the peak positions, widths, and amplitudes are unknown, and only the fundamental peak shapes are known, then the iterative least-squares method can be employed. In some cases, even the background can be accounted for by curve fitting.



For gas chromatography and mass spectrometry specifically, **Philip Wenig's OpenChrom** is an open-source data system that can import binary and textual chromatographic data files directly. It includes methods to detect baselines and to measure peak areas in a chromatogram. Extensive documentation is available. It is available for Windows, Linux, Solaris, and Mac OS X. A screenshot is shown on the right (click to enlarge). The author has regularly updated the program and its documentation. Another freely-available open-source program for mass spectroscopy is "Skyline" from MacCoss Lab Software, which is specifically aimed at reaction monitoring. Tutorials and videos are available. There is also commercial software, such as Ampersand's Chrom&Spec software and Shimadzu's LabSolutions, which perform sophisticated factor analysis, peak deconvolution, etc.

Peak area measurement in spreadsheets.

EffectOfDx.xlsx (screen image) demonstrates that the simple equation sum(y)*dx accurately measures the peak area of an isolated Gaussian peak if there are at least 4 or 5 points visibly above the baseline and as long as you include the points out to plus and minus at least 2 or 3 standard deviations of the Gaussian. It also shows that an exponentially broadened Gaussian needs to include more points on the tailing (right-hand, in this case) side to achieve the best accuracy. EffectOfNoiseAndBaseline.xlsx (screen image) demonstrates the effect of random noise and non-zero baseline, showing that the area is more sensitive to a



non-zero baseline than the same amount of random noise. CumulativeSum.xls (screen image) illustrates the integration of a peak-type signal by normalized cumulative sum. You can paste your own data into columns A and B. CumulativeSumExample.xls is an example with data.

The **Excel** and **Calc** spreadsheets PeakDetectionAndMeasurement and CurveFitter can measure the areas under partly overlapping Gaussian peaks in time-series data, using the findpeaks algorithm and iterative non-linear curve fitting techniques (page 199), respectively. But neither is as versatile as using a dedicated chromatography program such as OpenChrom.

Using sharpening for overlapping peak area measurements.

I have created a set of spreadsheet templates for perpendicular drop area measurements of overlapping peaks using even derivative sharpening. There is an empty template you can Copy/Paste your data into (PeakSharpeningAreaMeasurementTemplate.xlsm), an example version with some typical sample data and settings already entered (PeakSharpeningArea-MeasurementExample.xlsm), and a "demo" that creates and measures *simulated data with known areas* (PeakSharpeningAreaMeasurementDemo.xlsm) so you can see how sharpening effects area measurement accuracy. There are very brief instructions in row 2 of each of these. In addition, there are *mouse-over pop-up notes* on many of the cells (indicated by a red marker in the upper right corner of the cell). All three have clickable ActiveX buttons for convenient interactive adjustment of the K2 and K4 factors by 1% or by 10% for each click. Of course, the problem is knowing what values of the 2nd and 4th derivative weighting factors (K1 and K2) to use. Those values depend on the peak separation, peak width, and the relative peak height of the two peaks, and you must determine them experimentally based on your preferred trade-off between extent of sharpening and extent of baseline upset. A good place to start for Gaussian peaks is (sigma^2)/30 for the 2nd derivative factor and (sigma^4)/200 for the 4th derivative factor, where sigma is the standard deviation of the Gaussian, then adjust to give the narrowest peaks without significant negative dips. Do not assume that increasing the Ks until baseline resolution is achieved will always give the best area accuracy. The optimum values depend on the ratio of peak heights: at 1:1, with equal widths and shapes, the perpendicular drop method (page 142) works perfectly with no sharpening, but if there is inequality in shapes, heights, or widths, increased K values give lower errors, but overdoing the sharpening can sacrifice accuracy. The two-screen images screen1 and screen2, which use the *same* K values, show that it is possible to find K values that give excellent accuracy for peak 2 over a range of relative peak heights, even when the smaller peak is quite small. *Without* sharpening, accurate perpendicular drop area measurements are impossible because there is no valley between the peaks.

The template PeakSymmetrizationTemplate.xlsm (graphic) performs the symmetrization of exponentially broadened peaks by the weighted addition of the first derivative. See page 83. PeakSymmetrizationExample.xlsm is an example application with sample data already typed in. The procedure here is first to adjust k1 to get the most symmetrical peak shapes (judged by equal but opposite slopes on the leading and trailing edges), then enter the start time, valley time, and end time from the graph for the pair of peaks you want to measure into cells B4, B5, and B6, and finally (optionally) adjust the second derivative sharpening factor k2. The perpendicular drop areas of those two peaks are reported in the table in columns F and G. These spreadsheets have Active-X clickable buttons to adjust the first derivative weighting factor (k1) in cell J4 and the second derivative sharpening factor k2 (cell J5). There is also a demo version that allows you to determine the accuracy

of perpendicular drop peak areas under different conditions by internally generating overlapping peaks of known peak areas, with specified asymmetry (B6), relative peak height (B3), width (B4), and noise (B5): PeakSymmetrizationDemo.xlsm (graphic).

Peak area measurement using Matlab and Octave

Matlab and Octave have built-in commands for the sum of elements ("sum", and the cumulative sum "cumsum") and the trapezoidal numerical integration ("trapz"). For example, consider these three Matlab commands.

```
>> x=-5:.1:5;
>> y=exp(-(x).^2);
>> trapz(x,y)
```

These lines accurately compute the numerical value of the area under the curve of x,y, in this case an isolated Gaussian, whose area can be shown to be the square root of pi, which is equal to 1.7725:

$$\int_{-\infty}^{\infty} e^{-x^2} dx = \left[\int_{-\infty}^{\infty} e^{-x^2} dx \int_{-\infty}^{\infty} e^{-y^2} dy\right]^{1/2} = \left[\int_0^{2\pi} \int_0^{\infty} e^{-r^2} r \, dr \, d\theta\right]^{1/2} = \left[\pi \int_0^{\infty} e^{-u} du\right]^{1/2} = \sqrt{\pi}$$

If the interval between x values, dx, is *constant*, then the area is simply `yi=sum(y).*dx`. Alternatively, the signal can be integrated using **`yi=cumsum(y).*dx`**, then the area of the peak will be equal to the height of the resulting step, **`max(yi)-min(yi)=1.7725`**.

The area of a peak is proportional to the product of its height and its width, but the proportionality constant depends on the peak shape. A pure Gaussian peak with a peak height *h* and full-width at half-maximum *w* has a total area of 1.064467**h***w*. A pure Lorentzian peak has a total area of (pi/2)**h***w*. A Gaussian-Lorentzian blend with *p* percent Gaussian character has an area of ((100- *p*)/100) * ((pi/2) **w***h*) + (*p* /100)*(1.064467**w***h*). The graphic LorentzianVsGaussian.png compares Gaussian and Lorentzian peaks of the same height and width. *The Lorentzian has more area in the outer wings*, you must measure over a very wide range on both sides of the peak. To get an area within 1%, you need to expand that to 64 times the FWHM! (See LorentzianAreaProblem.m, graphic). Some real signals in practice have difficult peak shapes (Lorentzian or Gaussian/Lorentzian blends) that are too close together to allow the theoretical areas to be measured directly by integration. Possible solution include peak sharpening (page 145), iterative curve fitting (page 153), or even measuring the height and width and then calculating the area analytically using the above analytical expressions.

The peaks in real signals have some other complications: (a) The shapes of the peaks might not be known; (b) they may be superimposed on a variable baseline; and (c) they may be overlapped with other peaks; (d) there is always some random noise. You can use signal simulation to test the influence of those complications. For example, the Matlab/Octave script AreasOfIsolatedPeaks.m creates a simulated multi-peak signal and then tests the peak area measurement accuracy of that signal with a specified integration window width and baseline correction range. (In that case, noise is the culprit).

Computer code for the perpendicular drop method. The following Matlab/Octave code measures the areas of two overlapping symmetrical peaks in the data vectors x,y by the perpendicular drop method. Variables m1 and m2 are the estimated x-axis positions of the two peaks, which are typically determined by some peak finding algorithm based on the first derivative. The "val2ind" function returns the index number of the value in a vector that matches the specified value. The third line finds

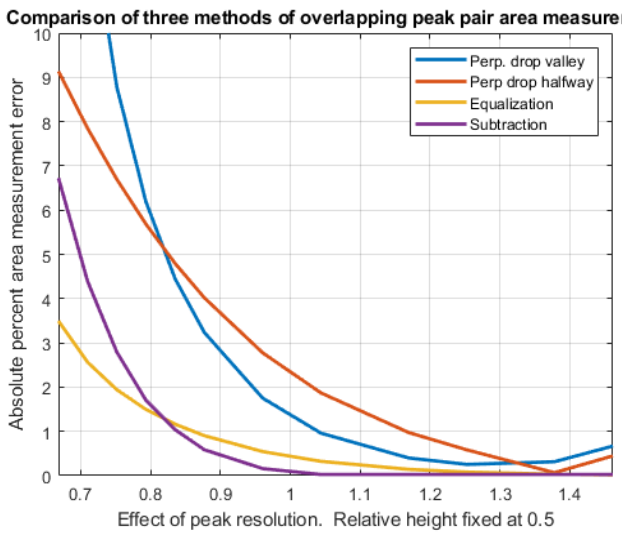
the half-way point between the two peaks. The last two lines use Matlab's "trapz" function to measure the areas before and after the valley point.

```
index1=val2ind(x,m1);
index2=val2ind(x,m2);
valleyindex=val2ind(x,(m1+m2)/2),
PDMeasArea1=trapz(x(1:valleyindex),y(1:valleyindex));
PDMeasArea2=trapz(x(valleyindex:length(x)),y(valleyindex:length(x));
```

Alternatively, you could replace "valleyindex" with **`valleyy=min(y(index1:index2));`** **`valleyindex=val2ind(y,valleyy);`** which uses the *minimum* between the peaks rather than the half-way point. But the *half-way point method has the advantage* that the SNR at a peak maximum is usually better than at a peak minimum, and moreover it works even when there is not a discernible minimum between the peaks. My function PerpDropAreas.m uses the half-way point method to measure the areas of any number of overlapping peaks, given a list of their peak maxima positions. These methods work best if the peak *widths* are equal or nearly so. Alternative methods include EqualPerpDrop.m, which performs area measurements by the "equalization" method, and EqualPerpDropTest.m, which demonstrates the use of the function applied to the measurement of two simulated overlapping EMG (exponentially modified Gaussian) peaks. Matlab/Octave code for all these methods is contained in the script "OverlapAreaComparison.m". For the case of Gaussian peak with a resolution of 0.7 and a height ratio of 1 to 0.5, the relative percent errors of the peak areas are:

```
                                    Peak 1   Peak 2
Perpendicular drop, valley point:   -6.44%  12.89%
Perpendicular drop, half-way point:  3.91%  -7.83%
Equalization method:                 1.27%  -2.54%
Subtraction method:                 -2.12%   4.25%
```

You can change the parameters in lines 5 through 10 to test with other peak separations and relative peak heights. The equalization method is often, but not always, the most accurate method. (Note: the script requires my val2ind.m, halfwidth.m, ExpBroaden.m, and plotit.m functions to be in the path).



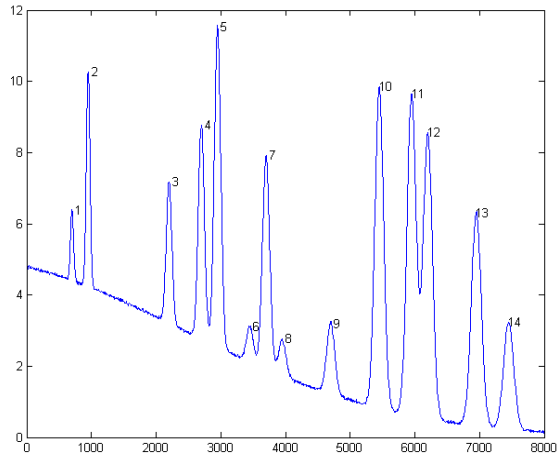
A more thorough investigation of these methods demonstrates the effect of changing the peak resolution, shown on the left (script, graphic) and of changing the height of the smaller peak, shown on the right below (script, graphic). These scripts include the effect of *random noise* in the signal, because noise can influence the location of peak maxima and the separation point between the peaks, whether they are determined manually or by a computer algorithm (as it is here). The random noise is set by the variable "noise", which is the fractional random white noise added to the signal. In addition, these scripts include the effect of *asymmetry* of the peak shapes, which can cause errors in area measurement by all these methods. After all, the very reason for measuring peak area rather than peak heights is to reduce the effect of

uncontrolled variations in peak broadening. The asymmetry is set by the variable "TimeConstant", which is the time constant of the exponential convolution applied to the signal that reduces the height and stretches out the right-hand half. Both of those are zero in the above figures for simplicity and to show the best possible accuracy. For example, with a resolution of 1.0, a tau of 2, and noise set to 0.01 (1%), the valley perpendicular drop and the equalization method outperform the other methods (graphic). Things are much easier and more forgiving in *quantitative* analysis using a calibration curve, because in that case absolute area accuracy is not really necessary. Rather, it is really the *reproducibility* of the areas that is key. *Systematic* errors in the area measurement simply change the *slope* of the calibration curve, and if the conditions are the same between calibration and analysis (always a requirement in any case), the error will cancel out exactly. For example, if you run the above scripts with very asymmetrical peaks (TimeConstant=3), poor resolution (resolution =0.68), and visible amounts of random noise=5%, the *systematic* area measurement errors are quite large (5%-15%), but nevertheless good linear calibration curves are produced by both the halfway point perpendicular drop and the equalization method, over the range of relative peak heights of 0.1 to 0.99, with correlation coefficients of 0.999. The calibration curve compensates for the systematic error and the area measurement integrates multiple data points over the peak.

All of these methods can produce significant errors if the peaks are highly overlapped or asymmetrical. However, asymmetry that is the result of *exponential broadening* can be symmetrized *before* computing the areas using the first derivative addition method, which sharpens the peaks and removes the asymmetry *without changing the peak areas*. Other methods of peak sharpening, especially self-deconvolution (page 114) could also be used when the peak to be measured is too weak or too poorly resolved to allow easy measurement. Ultimately, in the most difficult cases, you may have to consider the use of iterative curve fitting, though it is admittedly more complex mathematically and is subject to its own limitations.

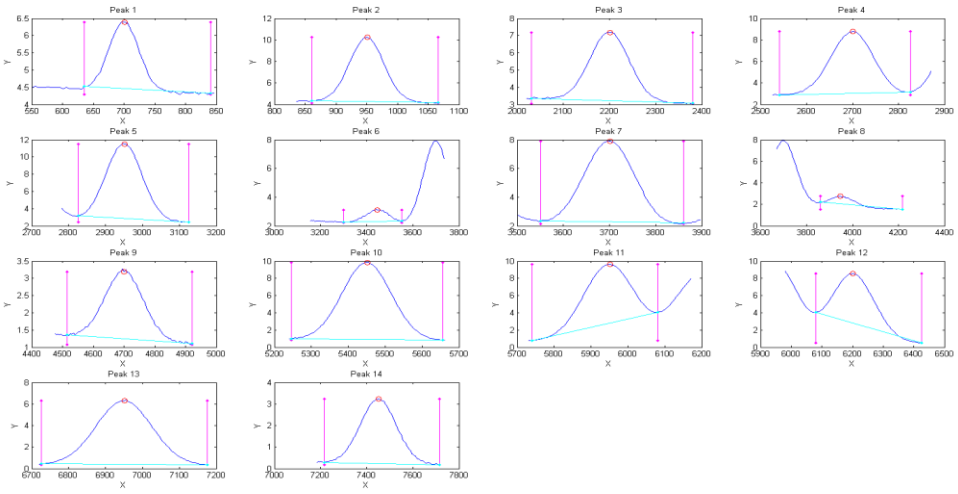
Automatic multiple peak detection

Measurepeaks.m (The syntax is **`M=measurepeaks (x,y,SlopeThreshold,AmpThreshold, SmoothWidth,FitWidth,plots)`**) is a function that *quickly and automatically* detects peaks in a signal, using the derivative zero-crossing method described previously, and measures their areas using the perpendicular drop and tangent skim methods. It shares the first 6 input arguments with findpeaksG. It returns a table containing the peak number, peak position, absolute peak height, peak-valley difference, perpendicular drop area, *and* the tangent skim area of each peak it detects. If the last input argument ('plots') is set to 1, it plots the signal with numbered peaks (shown on the left) and also *plots the* individual peaks (in blue) with the maximum (red circles), valley points (magenta), and tangent lines (cyan) marked as shown on the right. The peak heights and x-positions are indicated by the red circles, the perpendicular drop area is the total area measured



between the two magenta vertical lines down to zero, and the tangent skim area is the area between the cyan baseline and the blue peak (which compensates for a linear local baseline). Type "help measurepeaks" and try the seven examples there, or run HeightAndArea.m to test the accuracy of peak height and area measurement with signals that have multiple peaks with noise, background, and some peak overlap. Generally, the values for absolute peak height and perpendicular drop area are best for peaks that have no background, even if they are slightly overlapped, whereas the values for peak-valley difference and for tangential skim area are better for isolated peaks on a straight or slightly curved background. Note: this function uses smoothing (specified by the SmoothWidth input argument) only for peak *detection*. It performs measurements on the *raw unsmoothed* y data. If the raw data are noisy, the location of the valleys may be uncertain, in which case it may be beneficial to smooth the y data yourself before calling measurepeaks.m, using any smooth function of your choice. (Smoothing does not change the peak area of an isolated peak).

[M,A]=autopeaks.m is basically a combination of autofindpeaks.m and measurepeaks.m. It has a similar syntax to measurepeaks.m, except that the peak detection parameters (SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, and smoothtype) can be omitted and the function will calculate trial values in the manner of autofindpeaks.m. Using the simple syntax [M,A]=autopeaks(x, y) works well in some cases, but if not try [M,A]=autopeaks(x, y, *n*), using different values of *n* (roughly the number of peaks that would fit into the signal record) until it detects the peaks that you want to measure. Just like measurepeaks.m, it returns a table M containing the peak number, peak position, absolute peak height, peak-valley difference, perpendicular drop area, and tangent skim area of each peak it detects, but is also can optionally return a vector A containing the peak detection parameters that it calculates (for use by other peak detection and fitting functions). For the most precise control over peak detection, you can specify all the peak detection parameters by typing M=autopeaks (x, y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup). The function autopeaksplot.m is the same but it also plots the signal and the individual peaks in the manner of measurepeaks.m (shown above). The script testautopeaks.m runs all the examples in the autopeaks help file, with a 1-second pause between each one, printing out results in the command window and additionally plotting and numbering the peaks (Figure window 1) and each individual peak (Figure window 2). It requires gaussian.m and fastsmooth.m in the Matlab path. Autopeaks.m and autopeaksplot.m returns a matrix, **M**, which lists each peak detected in the rows and has the following peak measurements in the columns:

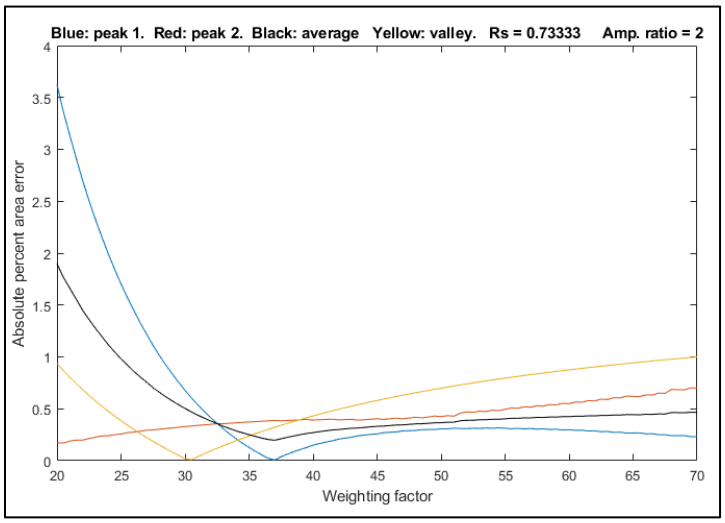


Peak	Position	PeakMax	Peak-valley	Perp drop	Tan skim
1	6.0000	1.3112	1.2987	1.7541	1.7121
2	. . . etc.				

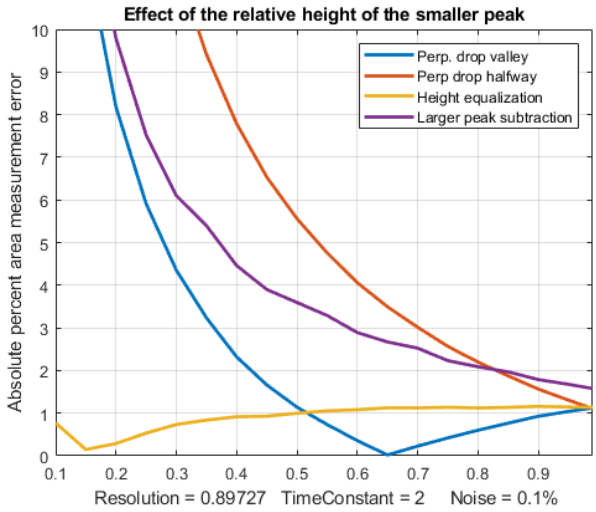
For determining the effect of smoothing, peak sharpening, deconvolution, or other signal enhancement methods on the areas of overlapping peaks measured by the perpendicular drop method, the Matlab/ Octave function ComparePDAreas.m uses autopeaks.m to measure the peak areas of original and processed signals, "orig" and "processed", and displays a scatter plot of original vs processed areas for each peak and returns the peak tables, P1 and P2 respectively, and the slope, intercept, and R^2 values, which should ideally be 1,0, and 1, if the processing has had no effect at all on peak area.

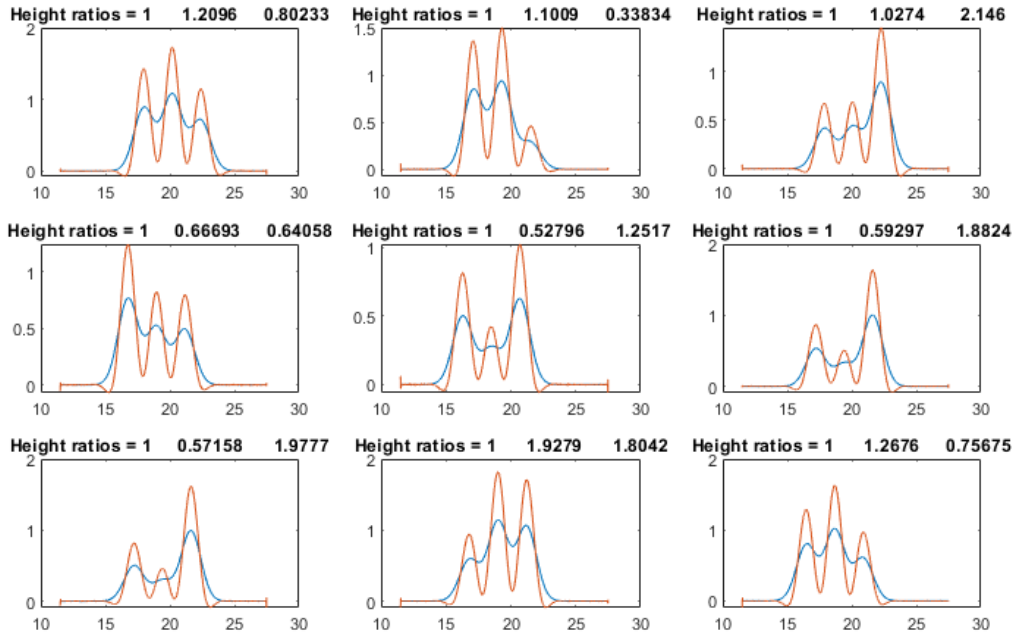
The related functions wmeasurepeaks.m and testwmeasurepeaks.m utilize *wavelet denoising* (page 136) rather than smoothing, but that makes little difference, because the peak parameter measurements are based on least-squares fitting to the *raw data*, not the *smoothed* data, so the usual wavelet denoising advantage of avoiding smoothing distortion does not apply here.

The Matlab/Octave automatic peak-finding function findpeaksG.m computes peak area assuming that the peak shape is Gaussian (or Lorentzian, for the variant findpeaksL.m). The related function findpeaksT.m uses the *triangle construction method* to compute the peak parameters. Even for well-separated Gaussian peaks, the area measure-ments by the triangle construction method are not very accurate. The results are about 3% below the correct values. (However, this method does perform better than findpeaksG.m when the peaks are noticeably asymmetric. See triangulationdemo for some examples). In contrast, measurepeaks.m makes no assumptions about the shape of the peak.



Peak sharpening (page 78) can often help in the measurement of the areas of overlapping peaks, by creating (or deepening) the valleys between peaks that are needed by the perpendicular drop method. SharpenedOverlapDemo.m is a script that automatically determines the optimum degree of even-derivative sharpening that minimizes the errors of measuring peak areas of two overlapping Gaussians by the perpendicular drop method using the autopeaks.m function. It does this by applying different degrees of sharpening and plotting the area errors (percent difference between the true and





measured errors) vs the sharpening weighting factor, as shown on the right. It also shows the height of the valley between the peaks (yellow line). This demonstrates that:

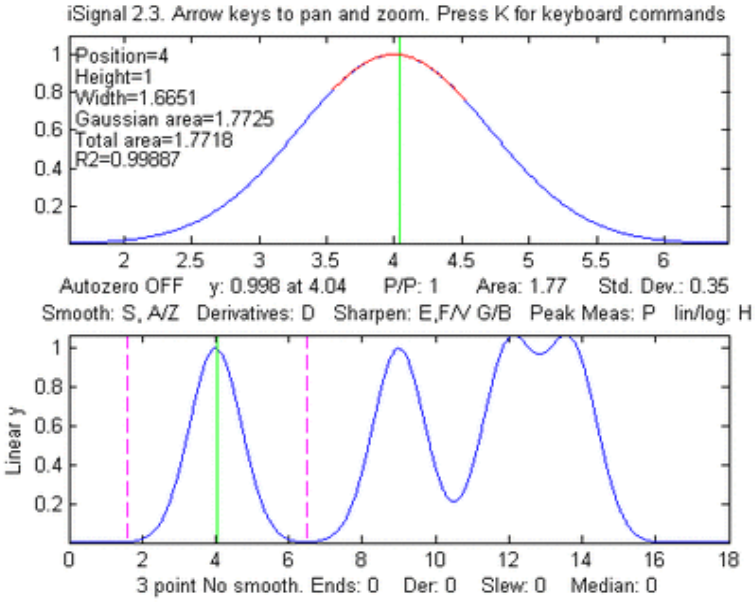
(1) the optimum sharpening factor depends upon the width and separation of the two peaks and on their height ratio,
(2) the degree of sharpening is not overly critical, often exhibiting a broad optimum region,
(3) the optimum for the two peaks is not necessarily the same, and
(4) the optimum for area measurement usually does not occur at the point where the valley is zero. (To run this script you must have gaussian.m, derivxy.m, autopeaks.m, val2ind.m, and halfwidth.m in the Matlab search path. Download these from https://terpconnect.umd.edu/~toh/spectrum/).

The script SharpenedOverlapCalibrationCurve.m simulates the construction and use of calibration curves of three overlapping Gaussian peaks (the blue lines in the signal plots on the next page). Even-derivative sharpening (the red line in the signal plots) is used to improve the resolution of the peaks to allow perpendicular drop area measurement. A straight line is fit to the calibration curve and the R^2 is calculated, to demonstrate (1) the linearity of the response, and (2) in independence of the overlapping adjacent peaks. You can change the following parameters:

1. The resolution, Rs, by changing the peak width *w* in line 15. Initially w=2, which makes Rs=0.55.
2. The peak height ratios, by changing the minimum and maximum peaks in lines 21 and 22. (Default is 0.2 and 1.0, a 1:5 ratio range). Naturally, if peak 2 is *too* small there won't be a valley between peaks.
3. The number of standards in the calibration curves, in line 24. Larger numbers give better results.
4. The number of simulated samples, in line 25. Larger numbers give more reliable average errors.

SymmetrizedOverlapCalibrationCurve.m does the same thing for symmetrization of overlapping exponentially modified Gaussian peaks by first-derivative addition. The critical variable is "factor" in line 27, which for best results should match or slightly exceed "tau", the exponential time constant in line 19. You must have gaussian.m, derivxy.m, autopeaks.m, val2ind.m, halfwidth.m, fastsmooth.m, and plotit.m in the Matlab search path.

iSignal (page 376) is a downloadable interactive Matlab function that performs various signal processing functions described in this tutorial, including measurement of peak area using Simpson's Rule and the perpendicular drop method. Click to view or **right-click > Save link as...** here, or you can download the ZIP file with sample data for testing. It is shown below applying the perpendicular drop method to a series of four peaks of equal area. Look at the bottom panel to see how the measurement intervals, marked by the vertical dotted magenta lines, are positioned at the valley *minimum* on either side of each of the four peaks. You can see this animation on the page if you view it in Microsoft Word 365, otherwise click this link.



The following lines of Matlab/ Octave code create four computer-synthesized Gaussian peaks that *all have the same height* (1.000), *width* (1.665), and *area* (1.772) but with *different degrees of peak overlap*, as in the figure on the right.

```
x=[0:.01:18];
y=exp(-(x-4).^2) + exp(-(x-9).^2) + exp(-(x-12).^2) + exp(-(x-13.7).^2);
isignal(x,y);
```

To use ***iSignal*** to measure the areas of each of these peaks by the perpendicular drop method, use the pan and zoom keys to position the two outer cursor lines (dotted magenta lines) in the valley on either side of the peak. The total of each peak area will be displayed below the upper window.

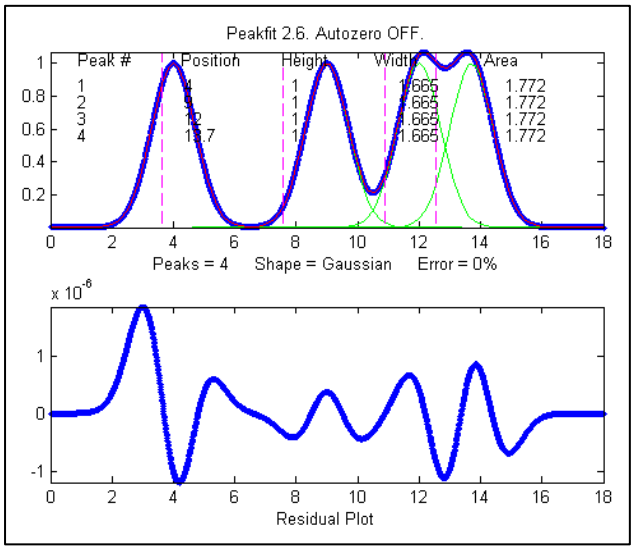
Peak #	Position	Height	Width	Area
1	4.00	1.00	1.661	1.7725
2	9.001	1.0003	1.6673	1.77
3	12.16	1.068	2.3	1.78
4	13.55	1.0685	2.21	1.79

The area results are reasonably accurate in this example only because the perpendicular drop method roughly compensates for partial overlaps between peaks, but only if the peaks are symmetrical, about equal in height, and have zero background.

iSignal includes an additional command (**J** key) that calls the autopeaksplot function, which automatically detects the peaks in the signal and measures their peak position, absolute peak height, peak-valley difference, perpendicular drop area, and tangent skim area. It asks you to type in the peak density (roughly the number of peaks that would fit into the signal record). The greater this number, the more sensitive it is to narrow peaks. It displays the measured peaks just as does the measurepeaks function described above. (To return to iSignal, press any cursor arrow key).

Area measurement by iterative curve fitting

In general, the most flexible peak area measurements for overlapping peaks, assuming that the basic *shape* of the peaks is known or can be guessed, are made with iterative least-squares peak fitting, for example using **peakfit.m**, shown below (for Matlab and Octave). This function can fit any number of overlapping peaks with model shapes selected from a list of different types. It uses the "trapz" function to calculate the area of each of the component model peaks. For example, using the **peakfit** function on the same data set as above, the results are much more accurate:




```
>>peakfit([x;y],9,18,4,1,0,10,0,0,0)
          Peak #     Position       Height        Width         Area
            1            4            1       1.6651       1.7725
            2            9            1       1.6651       1.7725
            3           12            1       1.6651       1.7725
            4         13.7            1       1.6651       1.7725
```

The interactive function **iPeak** (page 253), can also be used to estimate peak areas. It has the advantage that it can detect and measure *all the peaks in a signal in one operation.* The default area measurement method in iPeak is Gaussian estimation, by assuming that the peaks are Gaussian and fitting the top part of the peak. For example (using the same x and y vectors defined on the previous page):

```
>> ipeak([x,y],10)
          Peak #     Position       Height        Width         Area
            1            4            1       1.6651       1.7727
            2       9.0005       1.0001       1.6674       1.7754
            3        12.16       1.0684       2.2546       2.5644
            4        13.54       1.0684       2.2521       2.5615
```

Peaks 1 and 2 are measured accurately by ***iPeak***, but the peak widths and areas for peaks 3 and 4 are not accurate because of the peak overlap. Fortunately, **iPeak** has a built-in "peakfit" function (activated by the **N** key) that uses these estimates of peak position and width as its first guess, resulting in good accuracy for all four peaks.

```
Fitting Error 0.0002165%
          Peak#      Position       Height        Width         Area
            1            4            1       1.6651       1.7724
            2            9            1       1.6651       1.7725
            3           12            1       1.6651       1.7725
            4         13.7      0.99999       1.6651       1.7724
```

So, in this artificially ideal situation, the results are in perfect agreement with expectations.

Correction for background/baseline

The presence of a baseline or background signal, on which the peaks are superimposed, will greatly influence the measured peak area if not corrected or compensated. ***iSignal***, ***iPeak***, **measurepeaks**,

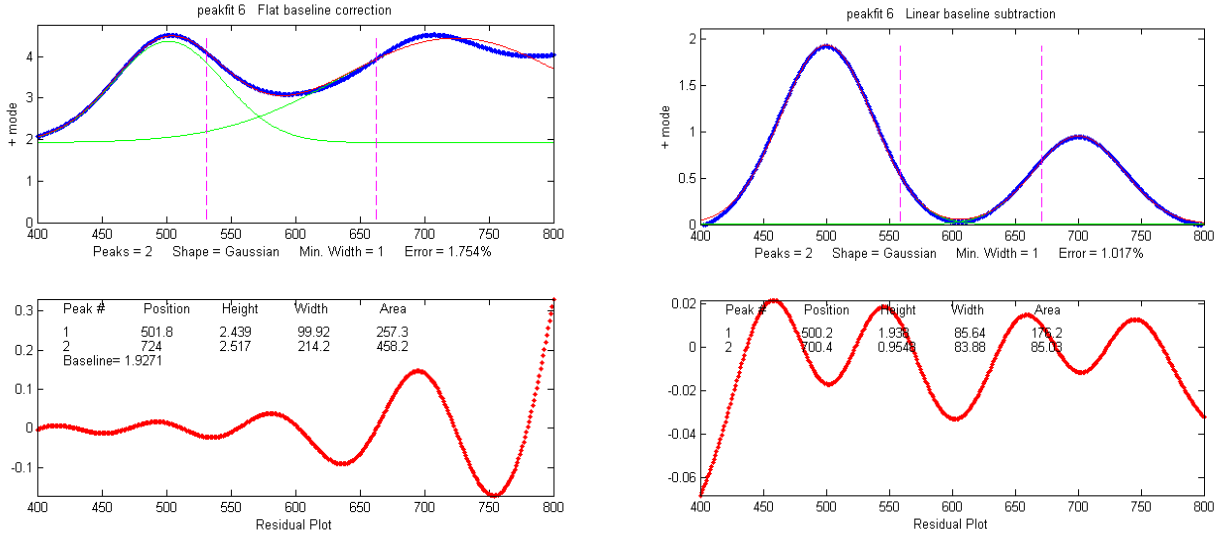
and ***peakfit*** all have several different baseline correction modes, for flat, tilted linear, and curved baselines, and *iSignal* and *iPeak* additionally have a multipoint piece-wise linear baseline subtraction function allows the manually estimated background to be subtracted from the entire signal. If the baseline is caused by the edges of a strong overlapping adjacent peak, then it is possible to include that peak in the curve-fitting operation, as seen in Example 22 on page 408.

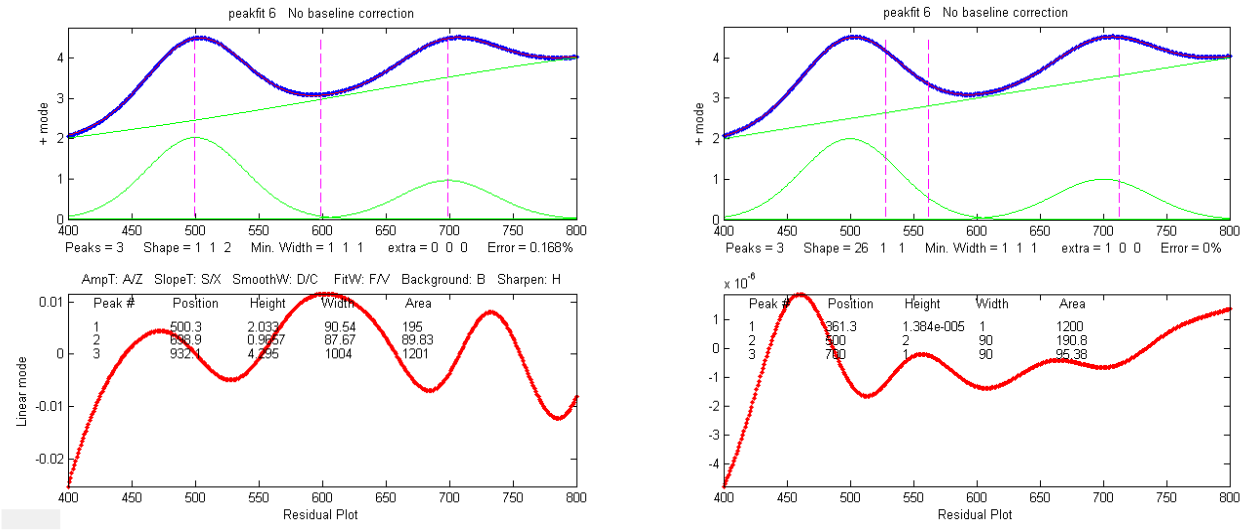
The script AreasOfIsolatedPeaks2.m demonstrates the use of peakfit.m for a simulated experimental signal consisting of several isolated peaks on a straight tilted baseline. In this example, the positions of the peaks are assumed to be reproducible enough that a pre-determined set of measurement segments can be used to fit each peak separately and determine its exact position, height, width, and area.

The following line of Matlab/Octave creates a simulated signal that consists of two noiseless, slightly overlapping Gaussian peaks with theoretical peak heights of 2.00 and 1.00 and areas of 191.63 and 95.81 units, respectively. The baseline is tilted and linear, and slightly greater in magnitude than the peak heights themselves, but the most serious problem is that the *signal never returns to the baseline* long enough to make it easy to distinguish the signal from the baseline.

```
>> x=400:1:800;y=2.*gaussian(x,500,90)+1.*gaussian(x,700,90)+2.*(x./400);
```

A straightforward application of iSignal, using baseline mode 1 and the perpendicular drop method, seriously underestimates both peak areas (168.6 and 81.78), because baseline mode 1 only works when the signal returns completely to the local baseline at the edges of the fitted range, which is not the case here.





An automated tangent skim measurement by measurepeaks is not accurate in this case because the peaks do not go all the way down to the baseline at the edges of the signal and because of the slight overlap:

```
>> measurepeaks(x,y,.0001,.8,2,5,1)
    Position  PeakMax  Peak-valley  Perp drop  Tan skim
1   503.67    4.5091     1.895       672.29     171.44
2   707.44    4.5184    0.8857       761.65     76.685
```

An attempt to use curve fitting with **peakfit.m** in the flat baseline correction mode 3 (`peakfit([x;y],0,0,2,1,0,1,0,3)`, above, top left-most figure) fails because the actual baseline is tilted, not flat. The linear baseline mode, `peakfit([x;y],0,0,2,1,0,1,0,1)`, top right figure) is not much better in this case (page 219). A more accurate approach is to set the baseline mode to *zero* and to include a *third* peak in the model to fit the baseline, for example with either a Lorentzian model - `peakfit([x;y],0,0,3,[1 1 2])`, bottom left figure - or with a "slope" model - shape 26 in peakfit version 6, bottom right figure. The latter method gives both the lowest fitting error (less than 0.01%) and the most accurate peak areas (less than ½% error in peak area):

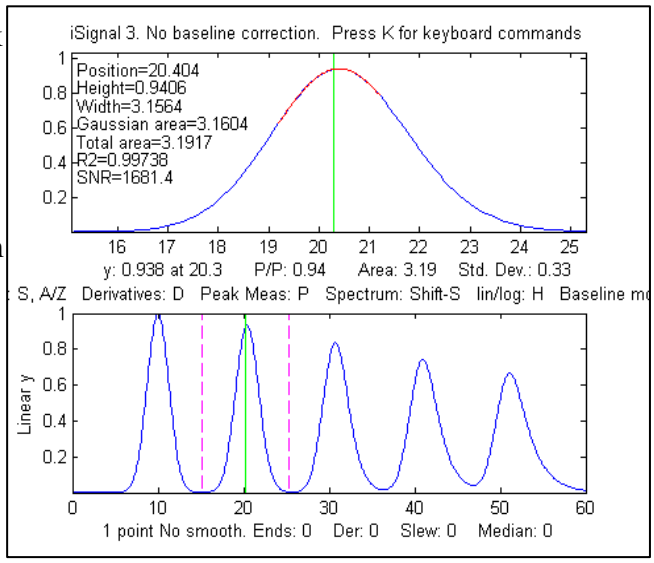
```
>> [FitResults,FitError]=peakfit([x;y],0,0,3,[1 1 26])
        Peak#     Position       Height        Width       Area
           1          500       2.0001       90.005      190.77
           2          700      0.99999       89.998      95.373
           3       5740.2  8.7115e-007            1      1200.1

FitError =0.0085798
```

Note that in this last case the number of peaks is 3 and the shape argument is a vector [1 1 26] specifying two Gaussian components plus the "linear slope" shape 26. If the baseline seems to be non-linear, you might prefer to model it using a quadratic (shape 46; see example 38 on page 411). If the baseline seems to be different on either side of the peak, try modeling the baseline with an S-shape (sigmoid), either an up-sigmoid, shape 10 (click for graphic), `peakfit([x;y],0,0,2,[1 10],[0 0])`, or a down-sigmoid, shape 23 (click for graphic), `peakfit([x;y],0,0,2,[1 23],[0 0])`, in these examples leaving the peak modeled as a Gaussian.

Asymmetrical peaks and peak broadening: perpendicular drop vs curve fitting

AsymmetricalAreaTest.m is a Matlab/Octave script that compares the accuracy of peak area measurement methods for a single noisy asymmetrical peak measured by different methods: (A) Gaussian estimation, (B) triangulation, (C) perpendicular drop method, and curve fitting by (D) exponentially broadened Gaussian, and (E) two overlapping Gaussians. AsymmetricalAreaTest2.m is similar except that it compares the precision (standard deviation) of the areas. For a single peak with a baseline of zero, the perpendicular drop and curve fitting methods work equally well, both considerably better than Gaussian estimation or triangulation. The advantage of the curve fitting methods is that they can deal more accurately with peaks that overlap or that are superimposed on a baseline.



Here's a Matlab/Octave experiment that creates a signal containing five Gaussian peaks with the *same* initial peak *height* (1.0) and *width* (3.0) but which are subsequently broadened by *increasing degrees of exponential broadening*, similar to the broadening of peaks commonly encountered in chromatography:

```
>> x=5:.1:65;
>> y=modelpeaks2(x, [1 5 5 5 5], [1 1 1 1 1], [10 20 30 40 50], [3 3 3 3
3], [0 -5 -10 -15 -20]);
>> isignal(x,y);
```

The theoretical area under these Gaussians is *all the same*: 1.0645*Height*Width = 1*3*1.0645 = **3.1938**. A perfect area-measuring algorithm would return this number for all five peaks.

As the broadening is increased from left to right, the peak height *decreases* (by about 35%) and peak width *increases* (by about 32%). Because the area under the peak is proportional to the *product* of the peak height and the peak width, these two changes *approximately cancel each other out* and the result is that the peak area is nearly independent of peak broadening (see the summary of results in 5ExponentialBroadenedGaussianFit.xlsx). The perpendicular drop method (page 146), PerpDropAreas (x, y, min(x), max(x), positions), where "positions" are the x-axis positions of the original peaks, gives the areas [3.1933 3.1926 3.1738 3.1006 3.3045], an average errors of 1.4%, which is not quite perfect.

The Matlab/Octave peak-finding function **findpeaksG.m**, finds all five peaks and measures their areas assuming a Gaussian shape. This works well for the unbroadened peak 1 (script), but it underestimates the areas as the broadening increases in peaks 2-5:

Peak	Position	Height	Width	Area
1	10.0000	1.0000	3.0000	3.1938
2	20.4112	0.9393	3.1819	3.1819
3	30.7471	0.8359	3.4910	3.1066
4	40.9924	0.7426	3.7786	2.9872
5	51.1759	0.6657	4.0791	2.8910

The triangle construction method (using **findpeaksT.m**) underestimates even the area of the unbroadened peak 1 and is less accurate for the broadened peaks (script; graphic):

```
Peak      Position    Height     Width     Area
   1        10.0000    1.1615    2.6607    3.0905
   2        20.3889    1.0958    2.8108    3.0802
   3        30.6655    0.9676    3.1223    3.0210
   4        40.8463    0.8530    3.4438    2.9376
   5        50.9784    0.7563    3.8072    2.8795
```

The automated function measurepeaks.m gives better results using the perpendicular drop method (5th column of the table).

```
>> M=measurepeaks(x,y,0.0011074,0.10041,3,3,1)

  Peak  Position  PeakMax   Peak-val.  Perp drop  Tan skim
   1     10          1        .99047     3.1871     3.1123
   2     20.4        .94018   .92897     3.1839     3.0905
   3     30.709      .83756   .81805     3.1597     2.9794
   4     40.93       .74379   .70762     3.1188     2.7634
   5     51.095      .66748   .61043     3.0835     2.5151
```

Using **iSignal** (page 376) and the manual peak-by-peak perpendicular drop method yields areas of 3.193, 3.194, 3.187, 3.178, and 3.231, a mean of 3.1966 (close to the theoretical value of 3.1938) and standard deviation of only 0.02 (0.63%). Alternatively, integrating the signal, `cumsum(y).*dx)`, where dx is the difference between adjacent x-axis values (0.1 in this case), and then measuring the heights of the resulting steps, gives similar results: 3.19, 3.19, 3.18, 3.17, 3.23. By either method, the peak areas are not quite equal as they should be.

But you can obtain a more accurate automated measurement of all five peaks, using **peakfit.m** with multiple shapes, one Gaussian and four exponentially modified Gaussians (shape 5) with different exponential factors (extra vector):

```
>> [FitResults,FittingError]=peakfit([x;y],30,54,5,[1 5 5 5 5],
[0  -5 -10 -15 -20],10, 0, 0)

          Peak#   Position       Height       Width        Area
            1        9.9933     0.98051      3.1181       3.2541
            2       20.002      1.0316       2.8348       3.1128
            3       29.985      0.95265      3.233        3.2784
            4       40.022      0.9495       3.2186       3.2531
            5       49.979      0.83202      3.8244       3.2974
FittingError = 2.184%
```

The fitting error is not much better than the simple Gaussian fit. Better results can be had using preliminary position and width results obtained from the findpeaks function or by curve fitting with a simple Gaussian fit and using those results as the "start" vector (eight input argument):

```
>> [FitResults,FittingError]=peakfit([x;y],30,54,5, [1 5 5 5 5], [0  -5 -10
-15 -20], 10, [10 3.5 20 3.5 31 3.5 41 3.5 51 3.5], 0)
```

```
          Peak#    Position       Height       Width          Area
             1        9.9999     0.99995      3.0005       3.1936
             2       20          0.99998      3.001        3.1944
             3       30.001      1.0002       3.0006       3.1948
             4       40          0.99982      2.9996       3.1924
             5       49.999      1.0001       3.003        3.1243
FittingError = 0.02%
```

Even more accurate results for area are obtained using peakfit with one Gaussian and four *equal-width* exponentially modified Gaussians (shape 8):

```
>> [FitResults,FittingError]=peakfit([x;y],30,54,5, [1 8 8 8 8], [0  -5 -10
-15 -20],10, [10 3.5 20 3.5 31 3.5 41 3.5 51 3.5],0)
          Peak#  Position   Height         Width          Area
             1    10        1.0001         2.9995       3.1929
             2    20        0.99998        3.0005       3.1939
             3    30        0.99987        3.0008       3.1939
             4    40        0.99987        2.9997       3.1926
             5    50        1.0006         2.9978       3.1207
FittingError = 0.008%
```

The latter approach works because, although the *broadened* peaks clearly have different widths (as shown in the simple Gaussian fit), the underlying pre-broadening peaks all have the *same* width. In general, if you expect that the peaks should have equal widths or fixed widths, then it is better to use a constrained model that fits that knowledge. You'll get better estimates of the measured unknown properties, even though the fitting error will be higher than for an unconstrained model.

The *disadvantages* of the exponentially broadened model are:

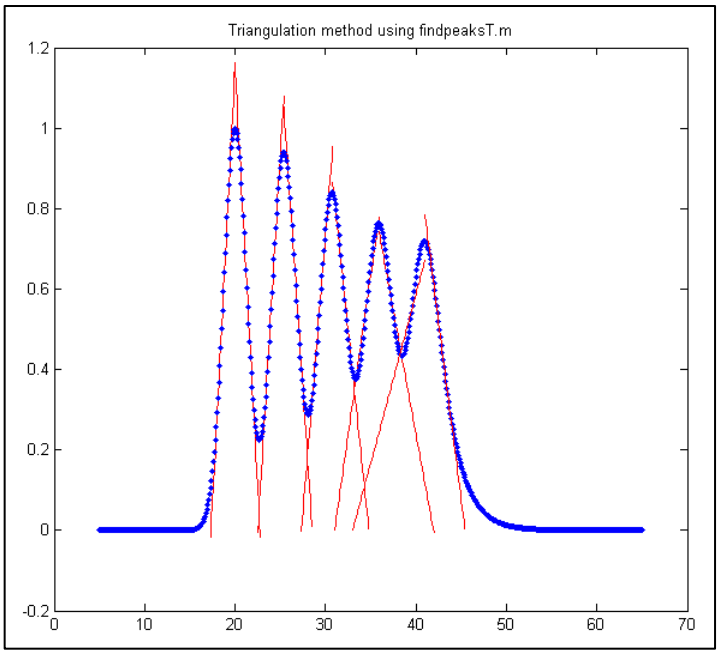
(a) it may not be a perfect match to the actual physical broadening process;
(b) it is slower than a simple Gaussian fit, and
(c) it sometimes needs help, in the form of a start vector or equal-widths constraints, as seen above, to get the best results.

Alternatively, if the objective is only to measure the peak *areas*, and not the peak positions and widths, then it is not even necessary to model the physical peak-broadening of each peak. You can simply aim for a good fit using two (or more) closely spaced simple Gaussians for each peak and simply *add up the areas* of the best-fit model. For example, the 5th peak in the above example (which is the most asymmetrical) can be fit very well with two overlapping Gaussians, resulting in a total area of 1.9983 + 1.1948 = 3.1931, very close to the theoretical area of 3.1938. Even more overlapping Gaussians can be used if the peak shape is more complex. This is called the "sum rule" in integral calculus: the integral of a sum of two functions is equal to the sum of their integrals. As a demonstration, the script SumOfAreas.m shows that even drastically non-Gaussian peaks can be fitted with multiple Gaussian components and that the total area of the components approaches the area under the non-Gaussian peak as the number of components increases (graphic). When using this technique, it is best to set the number of trials (*NumTrials*, the 7th input argument of the peakfit.m function) to 10 or more. Additionally, if the peak of interest is on a baseline, you must add up the areas of only those peak that contribute to fitting the peak itself and *not* those that are fitting the baseline.

An alternative to curve fitting with an exponentially broadened model is to use symmetrize.m or iSignal.m on each peak to convert them into symmetrical peaks and then to fit them with an appropriate symmetrical model (in this case a Gaussian). See page 83.

By making the peaks **closer together**, you can create a tougher and more realistic challenge.

```
>> y=modelpeaks2(x,[1 5 5 5 5],[1 1 1 1 1],[20 25 30 35 40],[3 3 3 3 3],[0
-5 -10 -15 -20]);
```



In this case, the triangle construction method gives areas of 3.1294, 3.202 3.3958, 4.1563, and 4.4039, seriously over-estimating the areas of the last two peaks, and measurepeaks.m using the perpendicular drop method (page 146) gives areas of 3.233, 3.2108, 3.0884, 3.0647 3.3602, compared to the theoretical value of 3.1938, better but not perfect. The integration-step height method is almost useless because the steps are no longer clearly distinct.

The peakfit function does better, again using the approximate result of **findpeaksG.m** as the 'start' value (8th input argument) for peakfit.

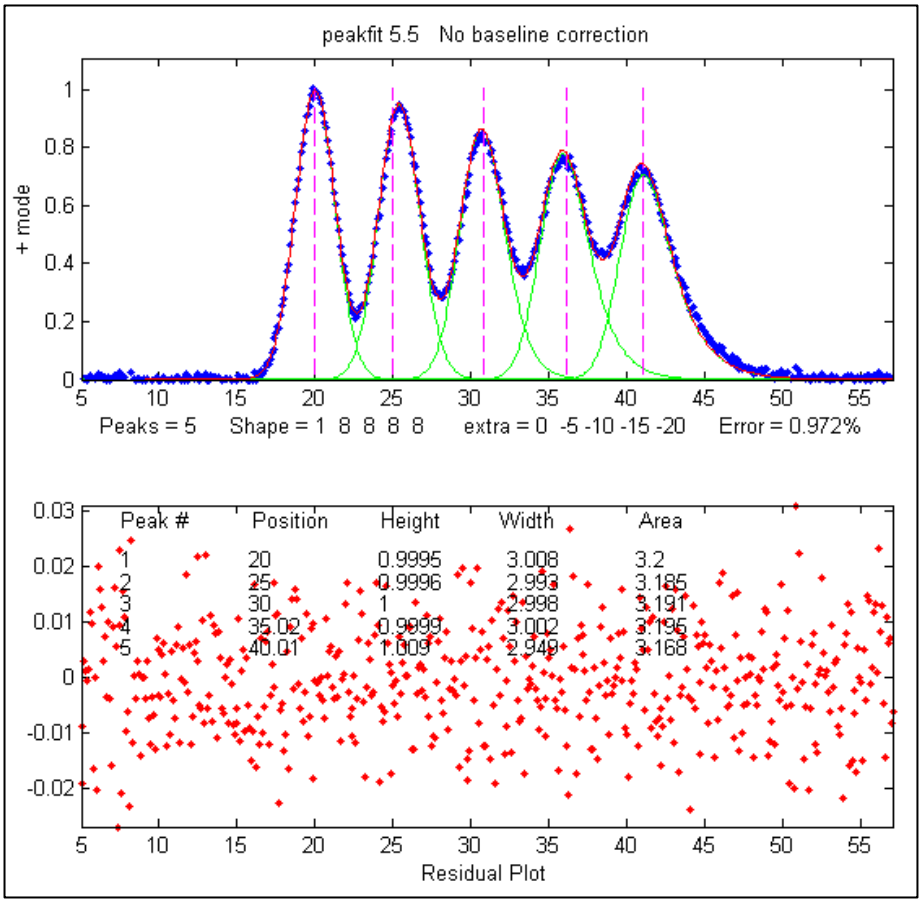
```
>>[FitResults,FittingError]=peakfit([x;y],30,54,5,[1 8 8 8 8],[0  -5 -10 -
15 -20],10, [20 3.5 25 3.5 31 3.5 36 3.5 41 3.5],0)
```

Peak#	Position	Height	Width	Area
1	20	0.99999	3.0002	3.1935
2	25	0.99988	3.0014	3.1945...
3	30	1.0004	2.9971	3.1918
4	35	0.9992	3.0043	3.1955
5	40.001	1.0001	2.9981	3.1915

```
FittingError = 0.01%
```

Next, make an even tougher challenge with different peak heights (1, 2, 3, 4 and 5, respectively) and a

bit of *added random noise*. The theoretical areas (Height*Width*1.0645) are 3.1938, 6.3876, 9.5814,



12.775, and 15.969.

```
>> y=modelpeaks2(x,[1 5 5 5 5],[1 2 3 4 5], [20 25 30 35 40], [3 3 3 3 3],
[0 -5 -10 -15 -20])+.01*randn(size(x));

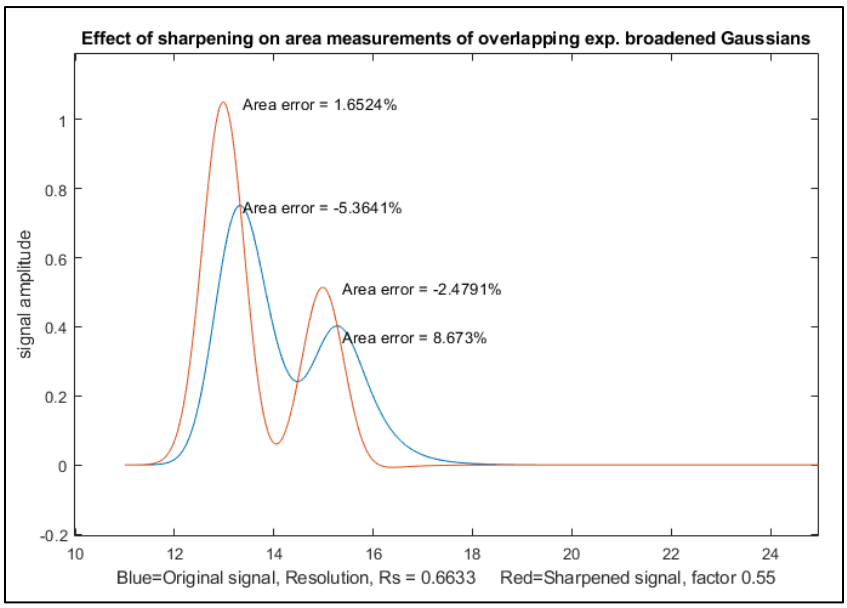
>> [FitResults,FittingError]=peakfit([x;y],30,54,5, [1 8 8 8 8], [0  -5 -10
-15 -20] ,20, [20 3.5 25 3.5 31 3.5 36 3.5 41 3.5],0)
```

Peak#	Position	Height	Width	Area
1	19.999	1.0015	2.9978	3.1958
2	25.001	1.9942	3.0165	6.4034
3	30	3.0056	2.9851	9.5507
4	34.997	3.9918	3.0076	12.78
5	40.001	4.9965	3.0021	15.966

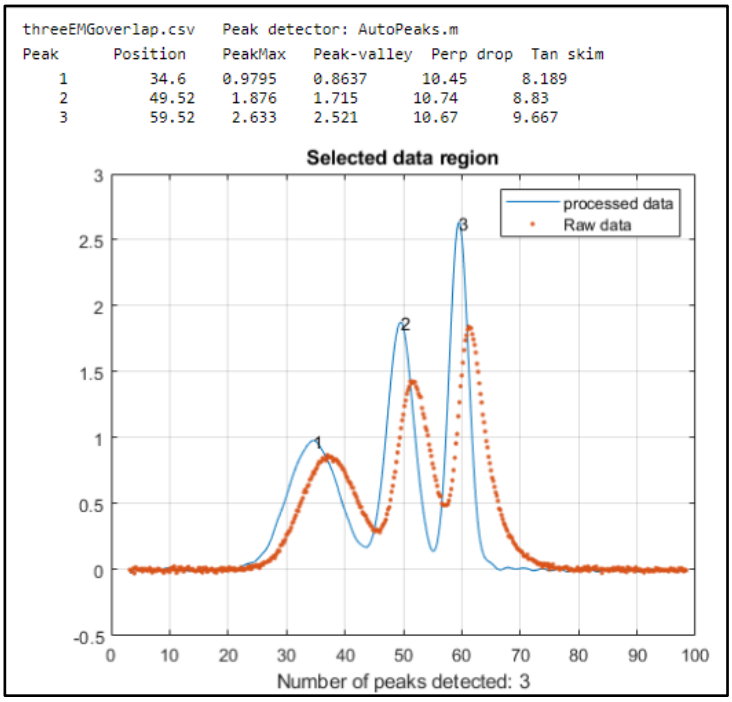
```
FittingError = 0.2755
```

The measured areas in this case (last column) are very close to the theoretical values, whereas all the other methods give substantially poorer accuracy. The more overlap between peaks, and the more unequal are the peak heights, the poorer the accuracy of the perpendicular drop and triangle construction methods. If the peaks are so overlapped that distinct maxima are not visible, both methods fail completely, whereas curve fitting can often retrieve a reasonable result, but *only if you can provide approximate first-guess value*.

Although curve fitting is generally the most powerful method for dealing with the combined effects of overlapping asymmetrical peaks superimposed on an irrelevant background, the simpler and computationally faster technique of first derivative sharpening (page 83) can be useful as a method to reduce or eliminate the effects of exponential broadening, resulting in a simpler shape that is easier and faster to fit. As is the case with curve fitting, it is most effective if there is an isolated peak with the same exponential broadening because that peak can be used to determine more easily the best value of the first derivative weighting factor.



SymmetizedOverlapDemo.m, illustrated on the left, demonstrates the optimization of the first derivative symmetrization for the measurement of the areas of two overlapping exponentially broadened Gaussians. It plots and compares the original (blue) and sharpened peaks (red), then tries first-derivative weighting factors from +10% to -10% of the correct tau value in line 14 and plots absolute peak area errors vs factor values. You can change the resolution by changing either the peak positions in lines 17 and 18 or the peak width in line 13. Change the height in line 16. You must have derivxy.m, autopeaks.m, and halfwidth.m in the Matlab search path. This method also easily deals with double exponential broadening, page 91, which is not so easily handled by curve fitting alone.



Peak area measurements by multiple methods is a part of the Live Script Peak detection tool **PeakDetection.mlx** described on page 253 and illustrated on the left. This interactive tool allows optional first-derivative symmetrization of skewed peaks, as well as symmetrical sharpening by Fourier self-deconvolution, to enhance the resolution of overlapping peaks and improve the accuracy of peak are measurement.

Curve fitting A: Linear Least-squares

The objective of curve fitting is to find the parameters of a mathematical model that describes a set of (often noisy) data in a way that minimizes the difference between the model and the data. The most common approach is the "linear least-squares" method, also called "polynomial least-squares", a well-known mathematical procedure for finding the coefficients of polynomial equations that are a "best fit" to a set of X,Y data. A polynomial equation expresses the dependent variable Y as a weighted sum of a series of single-valued functions of the independent variable X, most commonly as a straight line ($Y = \mathbf{a} + \mathbf{b}X$, where **a** is the *intercept* and **b** is the *slope*), or a quadratic ($Y = \mathbf{a} + \mathbf{b}X + \mathbf{c}X^2$), or a cubic ($Y = \mathbf{a} + \mathbf{b}X + \mathbf{c}X^2 + \mathbf{d}X^3$), and so on to higher-order polynomials. Those coefficients (**a**, **b**, **c**, etc.) can be used to predict values of Y for each X. In all these cases, Y is a *linear function* of the parameters **a**, **b**, **c**, and/or **d**. ***This*** *is the reason we call it a "linear" least-squares fit,* ***not*** *because the plot of X vs Y is linear.* Only for the *first*-order polynomial $Y = \mathbf{a} + \mathbf{b}X$ is the plot of X vs Y linear. And if the model *cannot* be described by a weighted sum of single-valued functions, then a different, more computationally laborious, "non-linear" least-squares method may be used, introduced on page 199.

“Best fit” simply means that the differences between the actual measured Y values and the Y values predicted by the model equation are *minimized*. It does *not* mean a “perfect" fit. In most cases, a least-squares best fit *does not go through all the points* in the data set. Above all, a least-squares fit *must conform to the selected model* - for example, a straight line or a quadratic parabola - and there will almost always be some data points that do not fall exactly on the best-fit line, either because of random error in the data or because the model is not capable of describing the data exactly.

Another thing: it is not correct to say "fit data to ..." a straight line or to some other model. It is the other way around: you are fitting a *model* to the *data*. The *data* are not being modified in any way. It is the *model* that is being adjusted to fit the data. (Actually, in some special cases it can be useful to transform the data before curve fitting; see page 172).

Least-squares best fits can be calculated by some hand-held calculators, spreadsheets, and dedicated computer programs (see Math Details below). Although it is possible to estimate the best-fit straight line by visual estimation and a straightedge, the least-square method is more objective and easier to automate. (If you were to give a plot of X, Y data to five different people and ask them to estimate the best-fit line visually, you would get five slightly different answers, but if you gave the data set to five different computer programs, you would get the exact same answer every time).

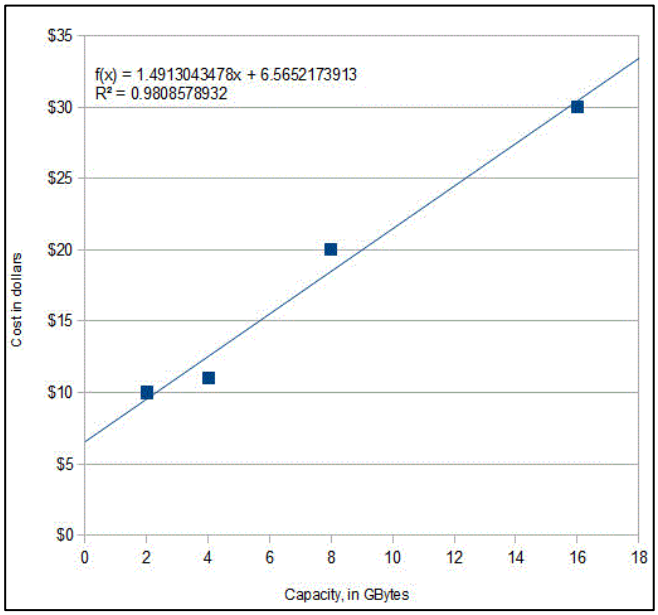
Examples of polynomial fits

Here is a very simple example: the historical prices of different sizes of SD memory cards advertised in the February 19, 2012, issue of the New York Times. (Yes, I know, the prices are *much* lower now, but these really were the prices in a big-box store back in 2012).

Memory Capacity (GBytes)	Price in US dollars
2	**$9.99**
4	**$10.99**
8	**$19.99**
16	**$29.99**

What is the relationship between memory capacity and cost? Naturally, you expect that the larger-

capacity cards should cost more than the smaller-capacity ones, and if you plot cost vs capacity (graph on the next page), you can see a rough straight-line relationship. A least-squares algorithm can compute the values of "**a**" (intercept) and "**b**" (slope) of the straight line that is a "best fit" to the data points. Using a linear least-squares calculation, where **X = capacity** and **Y = cost**, the straight-line mathematical equation that most simply describes these data (rounding to the nearest penny) is:



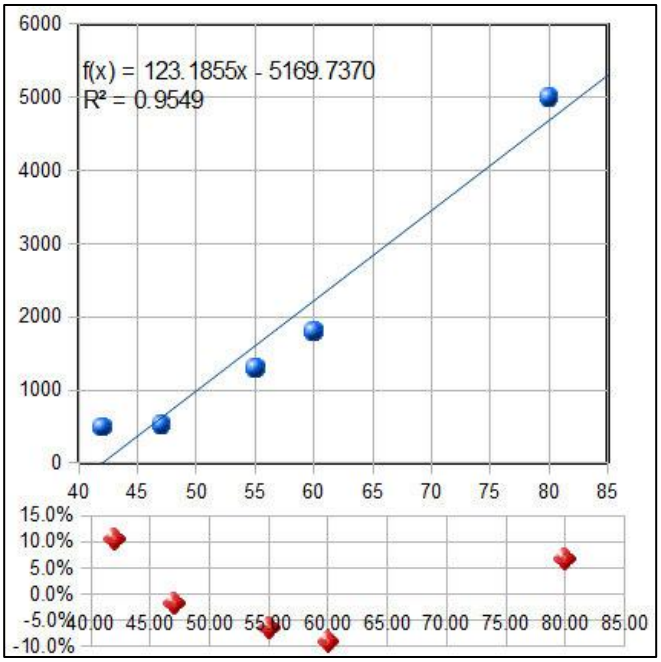
Cost = $6.56 + Capacity * $1.49

So, $1.49 is the *slope* and $6.56 is the *intercept*. (The equation is plotted as the solid line that passes among the data points in the figure). Basically, this is saying that the cost of a memory card consists of a fixed cost of $6.56 plus $1.49 for each GBytes of capacity. How can you interpret this? The $6.56 represents the costs that are the same regardless of the memory capacity: a reasonable guess is that it includes things like packaging (the different cards are the same physical size and are packaged the same way), shipping, marketing, advertising, and retail shop shelf space. The $1.49 (1.49 dollars/Gbyte) represents the increasing retail price of the larger chips inside the larger capacity cards, mainly because they *have more value for the consumer* but also probably cost more to make because they use more silicon, are more complex, or maybe have a higher chip-testing rejection rate in the production line. So, in this case, the slope and intercept have actual physical and economic meanings.

What can you do with this information? First, you can see how closely the actual prices conform to this equation: approximately *but not perfectly*. The line of the equation passes *among* the data points but does not go exactly *through* each one. That's because actual retail prices are also influenced by several factors that are unpredictable and random: local competition, supply, demand, and even rounding of prices to the nearest "neat" number. All those factors constitute the "noise" in these data. The least-squares procedure also calculates R^2, called the *coefficient of determination* or the *correlation coefficient*, which is an indicator of the "goodness of fit". R^2 is exactly 1.0000 when the fit is perfect, less than that when the fit is imperfect. The closer to 1.0000 the better. An R^2 value of 0.99 usually means a "good" fit and 0.999 is a "very good" fit. (The R^2 value is calculated as shown on page 177).

The second way you can use these data is to predict the likely prices of other card capacities, if they were available, by putting in the memory capacity into the equation and evaluating the cost. For example, a 12 Gbyte card would be expected to cost $24.44 according to this model. And a 32 Gbyte card would be predicted to cost $54.29, but *that would be predicting beyond the range of the available data* - it is called "extrapolation"- *and it is very risky* because you do not really know what other factors may influence the data beyond the last data point. (You could also solve the equation for capacity as a function of cost and use it to predict how much capacity could be expected to be bought for a given amount of money if such a product were available).

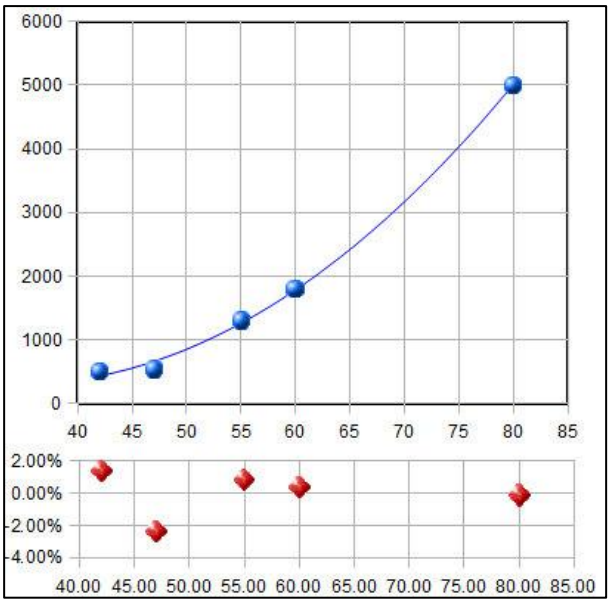
As I said, these were the prices back in 2012. Why not do a little "homework"? Look up the *current*



prices and see how they compare. Did you get a lower slope, lower intercept, or both?

Here's another related example: the historical prices of standard high definition (not UHD) flat-screen LCD TVs as a function of screen size, as advertised on the Web in the spring of 2012. The prices of five selected models, *similar except for screen size*, are plotted against the screen size in inches in the figure on the left and are fit to a first order (straight-line) model. As for the previous example, the fit is not perfect. The equation of the best-fit model is shown at the top of the graph, along with the R^2 value (0.9549) indicating that the fit is not very good. And you can see from the best-fit line that a 40-inch set would be predicted to have a *negative cost*, which is impossible.

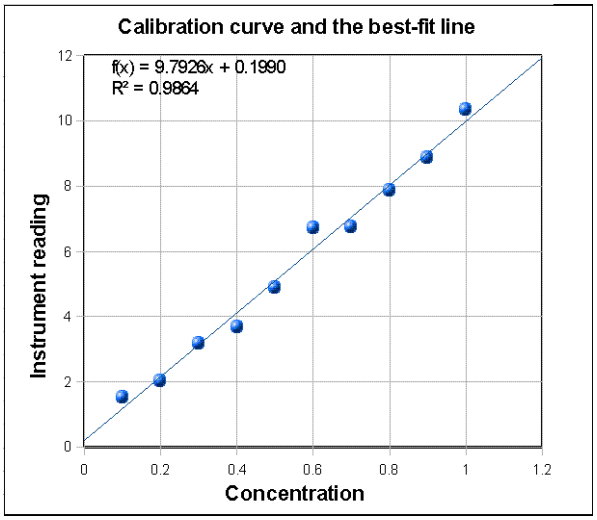
The goodness of fit is shown even more clearly in the little graph at the bottom of the figure, with the red dots. This shows the "residuals", the differences between each data point and the least-squares fit at that point. You can see that the deviations from zero are large (±10%), but more importantly, *they are not completely random*. They form a *clearly visible U-shaped curve*. This is a tip-off that the straight-line model you have used here may not be ideal and that you might get a better fit with another model. (Or it might be just *chance*: the first and last points might be higher than expected because those were unusually expensive TVs for those sizes. How would you really know unless your data collection were very careful?)



Least-squares calculations can fit not only straight-line data, but *any set of data that can be described by a polynomial*, for example a second order (quadratic) equation ($Y = \mathbf{a} + \mathbf{b}X + \mathbf{c}X^2$). Applying a second order fit to these data, you get the graph on the right. Now the R^2 value is higher, 0.9985, indicating that the fit is better (but again not really perfect), and the residuals (the red dots at the bottom) are smaller and more random. This should not really be a surprise, because of the nature of these data. The size of a TV screen is always quoted as the *length of the diagonal*, from one corner of the screen to its opposite corner, but the quantity of material, the difficulty of manufacture, the weight, and the power supply requirements of the screen should all scale with the *screen area*. Area is proportional to the *square* of the linear measure, so the inclusion of an X^2 term in the model is quite reasonable in this case. Applying a quadratic fit, which has an X^2 term, the 40-inch set would be predicted to cost under $500, which is more sensible than the

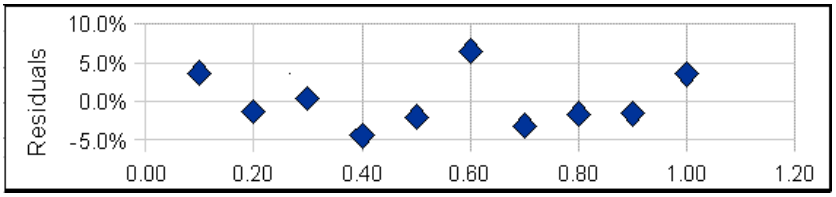
linear fit. (The actual interpretation of the meaning of the best-fit coefficients **a**, **b**, and **c** is, however, impossible unless you know much more about the manufacture and marketing of TVs). The least-squares procedure allows us to model the data with a more-or-less simple polynomial equation. The point here is that a quadratic model is justified not just because it fits the data better, but in this case, it is justified because it is *expected in principle* based on the relationship between length and area. (Incidentally, as you might expect, prices have dropped considerably since 2012. In 2024, a Vizio 65" 4K set was available at Sam's Club for under $380, far cheaper than the prediction from this fit).

In general, fitting *any* set of data with a higher order polynomial, like a quadratic, cubic or higher, will reduce the fitting error and make the R^2 values closer to 1.000. That is because a higher-order model has more variable coefficients that the program can adjust to fit the data. For example, you *could* fit the SD card price data to a *quadratic* (graphic), but *there is no reason to do so* and the fit would only be slightly better. The danger is that you could be "fitting the noise", that is, adjusting to the random noise in *that data* set, whereas *another* measurement with different random noise might give markedly different results. In fact, if you use a polynomial order that is *one less than the number of data points*, the fit will be perfect and R^2=1.000. For example, the SD card data have only 4 data points, and if you fit those data to a 3rd order (cubic) polynomial, you'll get a mathematically *perfect fit* (graphic), but one that makes no sense in the real world (the price turns back down above x=14 Gbytes). It is meaningless and misleading - *any* 4-point data would have fit a cubic model perfectly, *even pure random noise*! The only justification for using a higher-order polynomial is if you have reason to believe, or have observed, that there is a *consistent* non-linearity in the data set, as in the TV price example above.



The graph here shows a third example, taken from the field of analytical chemistry: it is a straight-line calibration data set where X = concentration and Y = the reading of some instrument that is supposed to be linearly proportional to the concentration X (in other words, Y = **a** + **b**X). If you are reading this online, you can click to download that data set. The blue dots are the data points. They do not all fall in a perfect straight line because of random noise and measurement error in the instrument readings and possibly also volumetric errors in the concentrations of the standards (which are usually prepared in the laboratory by diluting a stock solution). For this set of data, the measured slope **b** is 9.7926 and the intercept **c** is 0.199. In analytical chemistry, the slope of the calibration curve is often called the "sensitivity". The intercept indicates the instrument reading that would be expected if the concentration were zero. Ordinarily, instruments are adjusted ("zeroed") by the operator to give a reading of zero for a concentration of zero, but random noise and instrument drift can cause the intercept to be non-zero for any calibration set. In this case, the data are in fact computer-generated, and the "true" value of the slope was exactly 10 and of the intercept was exactly zero before noise was added, and the noise was added by a zero-centered normally-distributed random-number generator. The presence of the noise caused this measurement of the slope to be off by about 2%. (Had there been a larger number of points in this data set, the calculated values of slope and intercept would almost certainly have been better. On

average, the accuracy of measurements of slope and intercept improve with the *square root of the number of points* in the data set). With this many data points, it is *mathematically* possible to use an even higher polynomial degree, up to one less than the number of data points, but it is not *physically* reasonable in most cases. For example, you could fit a 9th-degree polynomial perfectly to these data, but the result is pretty wild (graphic link). No analytical instrument has a calibration curve that behaves like that!



A plot of the residuals for the calibration data (right) raises a question. Except for the 6th data point (at a concentration of 0.6), the other points seem to form a rough U-shaped curve, indicating that a quadratic equation might be a better model for those points than a straight line. Can you reject the 6th point as being an "outlier", perhaps caused by a mistake in preparing that solution standard or in reading the instrument for that point? Discarding that point would improve the quality of fit (R^2=0.992 instead of 0.986) especially if a quadratic fit were used (R^2=0.998). The only way to know for sure is to repeat that standard solution preparation and calibration and see if that U shape persists in the residuals. Many instruments do give a very linear calibration response, while others may show a slightly non-linear response under some circumstances (for example). But in fact, the calibration data used for *this* example were computer-generated to be *perfectly linear,* with normally-distributed random numbers added to simulate noise. So that 6th point is *not an outlier* and the underlying data are not really curved, but *you would not know that in a real application*. It would have been a mistake to discard that 6th point and use a quadratic fit in this case. Moral: do not throw out data points just because they seem a little off, unless you have a good reason, and do not use higher-order polynomial fits just to get better fits if the instrument is known to give linear response under those circumstances. Even perfectly normally-distributed random errors can occasionally give individual deviations that are quite far from the average and might tempt you into thinking that they are outliers. Do not be fooled. (*Full disclosure*: I obtained the above example by "cherry-picking" from among dozens of randomly generated linear data sets with added random noise, in order to find one that, although actually random, *seemed* to have an outlier).

Solving the calibration equation for concentration. Once the calibration curve is established, it can be used to determine the concentrations of unknown samples that are measured on the same instrument, for example by *solving the equation for concentration as a function of instrument reading*. The result for the linear case is that the concentration of the sample Cx is given by Cx = (Sx - *intercept*)/*slope*, where Sx is the signal given by the sample solution, and "*slope*" and "*intercept*" are the results of the least-squares fit. If a quadratic fit is used, then you must use the more complex "quadratic equation" to solve for concentration (see QuadraticEquation.m), but the problem of solving the calibration equation for concentration becomes forbiddingly complex for higher-order polynomial fits. (Note: The concentration and the instrument readings can be recorded in any convenient units if the same units are used for calibration and for the measurement of unknowns).

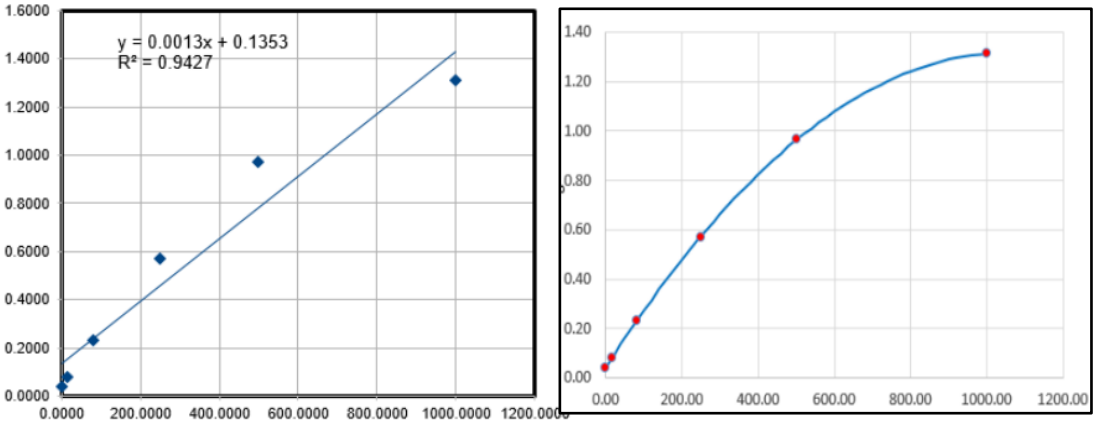
Reliability of curve fitting results

How reliable are the slope, intercept and other polynomial coefficients obtained from least-squares calculations on experimental data? The single most important factor is the appropriateness of the chosen model. It is critical that the model (e.g., linear, quadratic, gaussian, etc.) be a good match to the actual underlying shape of the data. You can do that either by choosing a model based on the known and expected behavior of that system (like using a linear calibration model for an instrument that is known to give linear response under those conditions) or by choosing a model that always gives randomly scattered residuals that do not exhibit a regular shape. But even with a perfect model, the least-squares procedure applied to repetitive sets of measurements will not give the same results every time because of random error (noise) in the data. If you were to repeat the entire set of measurements many times and do least-squares calculations on each data set, the standard deviations of the coefficients would vary directly with the standard deviation of the noise and inversely with the square root of the number of data points in each fit, all else being equal. The problem, obviously, is that it is not always possible to repeat the entire set of measurements many times. You may have only *one* set of measurements, and each experiment may be very expensive to repeat. So, it would be great if you had a short-cut method that would let us *predict* the standard deviations of the coefficients from a *single measurement* of the signal, without repeating the measurements.

Here I will describe three general ways to predict the standard deviations of the polynomial coefficients: algebraic propagation of errors, Monte Carlo simulation, and the bootstrap sampling method.

Algebraic Propagation of errors

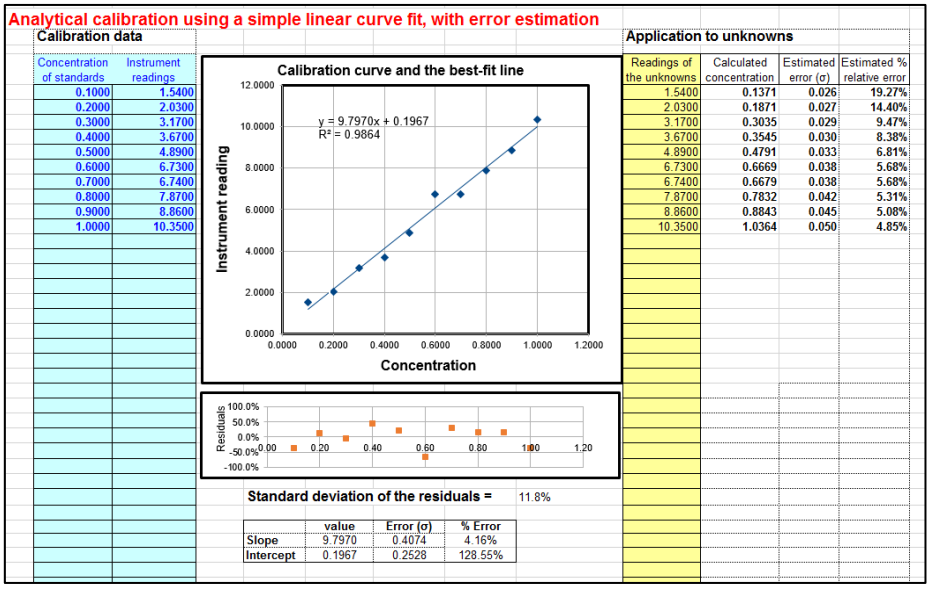
The classical way is based on the rules for mathematical error propagation. The propagation of errors of the entire curve-fitting method can be described in closed-form algebra by breaking down the method into a series of simple differences, sums, products, and ratios, and applying the rules for error propagation to each step. The results of this procedure for a first order (straight line) least-squares fit are shown in the last three lines of the set of equations in Math Details, on page 177. Essentially, these equations make use of the deviations from the least-squares line (the "residuals") to estimate the standard deviations of the slope and intercept, based on the assumption that the noise in that single data set is *random* and is representative of the noise that would be obtained upon repeated measurements. *Because these predictions are based only on a single data set, they are good only insofar as that data set is typical of others that might be obtained in repeated measurements.* If your random errors happen to be *small* when you acquire your data set, you will get a deceptively *good-looking* fit, but then your



estimates of the standard deviation of the slope and intercept will be too *low*, on average. If your random errors happen to be *large* in that data set, you will get a deceptively *bad-looking* fit, but then your estimates of the standard deviation will be too *high*, on average. This problem becomes worse when the number of data points is small. This is not to say that it is not worth the trouble to calculate the predicted standard deviations of slope and intercept, but keep in mind that these predictions are accurate only if the number of data points is large (and only if the noise is random and normally-distributed). Beware: if the deviations from linearity in your data set are *systematic* and not *random* - for example, if try to fit a straight line to a smooth curved data set (previous page, left), then the estimates the standard deviations of the slope and intercept by these last two equations *will be too high*, because they assume the deviations are caused by random noise that varies from measurement to measurement, whereas in fact a smooth curved data set without random noise (previous page, right) will give the *same* slope and intercept from measurement to measurement.

In the application to analytical calibration, the concentration of the sample Cx is given by Cx = (Sx - *intercept*)/*slope*, where Sx is the signal given by the sample solution. The uncertainty of all three terms contributes to the uncertainty of Cx. The standard deviation of Cx can be estimated from the standard deviations of the slope, intercept, and Sx using the rules for mathematical error propagation. But the problem is that, in analytical chemistry, the labor and cost of preparing and running large numbers of standards solution often limits the number of standards to a rather small set, by statistical standards, so these estimates of standard deviation are often poor.

A spreadsheet that performs these error-propagation calculations for your own first-order (linear) analytical calibration data can be downloaded from http://terpconnect.umd.edu/~toh/models/CalibrationLinear.xls. For example, the linear calibration example just given in the previous section, where the "true" value of the slope was 10 and the intercept was zero, this spreadsheet (whose screenshot shown on the right) predicts that the slope is 9.8 with a standard deviation 0.407 (4.2%) and that the intercept is 0.197 with a standard deviation 0.25 (128%), both well within two standard deviations of the true values. This spreadsheet also performs the propagation of error calculations for the calculated concentrations of each unknown in the last two columns on the right. In the example in this figure, the instrument readings of the standards are taken as the unknowns, showing that the predicted percent concentration errors range from about 5% to 19% of the true values of those standards. (Note that the standard deviation of the concentration is greater at high concentrations than the standard deviation of the slope, and considerably greater at low concentrations because of the

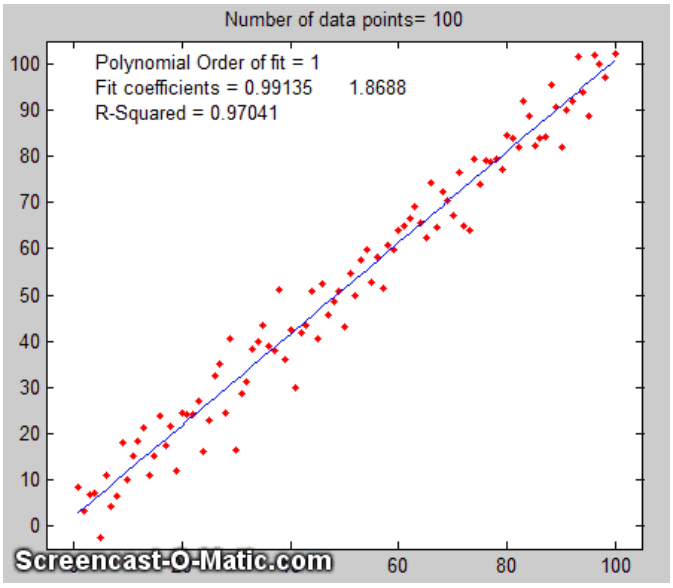


greater influence of the uncertainty in the intercept). For further discussion and some examples, see "The Calibration Curve Method with Linear Curve Fit". My Matlab/Octave plotit.m function uses the algebraic method to compute the standard deviations of least-squares coefficients for any polynomial order.

Monte Carlo simulation

The second way of estimating the standard deviations of the least-squares coefficients is to perform a random-number simulation (a type of Monte Carlo simulation). This requires that you know (by previous measurements) the average standard deviation of the random noise in the data. Using a computer, you construct a model of your data over the normal range of X and Y values (e.g. Y = **intercept** + **slope***X + **noise**, where **noise** is the vector of noise in the data), compute the slope and intercept of each simulated noisy data set, then repeat that process many times (usually a few thousand) with different sets of random noise, and finally compute the standard deviation of all the resulting slopes and intercepts. This is ordinarily done with normally-distributed random noise (e.g., the RAND or RANDN functions that many programming languages have). These random number generators produce "white" noise, but other noise colors can be derived. If the model is good and the noise in the data is well-characterized in terms of frequency distribution and signal amplitude dependence, the results will be a very good estimate of the expected standard deviations of the least-squares coefficients. (If the noise is not constant, but rather varies with the X or Y values, or if the noise is not white or is not normally-distributed, then that behavior must be included in the simulation).

An animated example is shown on the right (which you can see if you view it in Microsoft Word 365, otherwise click this link), for the case of a 100-point straight-line data set with slope=1, intercept=0, and standard deviation of the added noise equal to 5% of the maximum value of y. For each repeated set of simulated data, the fit coefficients (least-squares measured slope and intercept) are slightly different because of the noise.



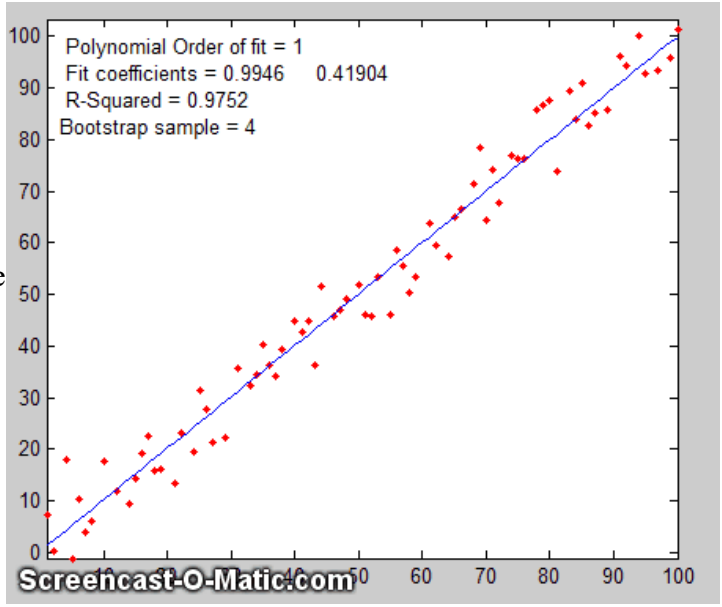
Obviously, this method involves programming a computer to compute the model and is not as convenient as evaluating a simple algebraic expression. But there are two important advantages to this method: (1) is has great generality.it can be applied to curve fitting methods that are too complicated for the classical closed-form algebraic propagation-of-error calculations, even iterative non-linear methods; and (2) its predictions are based on the average noise in the data, not the noise in just a single data set. For that reason, it gives more reliable estimations, particularly when the number of data points in each data set is small. Nevertheless, you cannot always apply this method because you do not always know the average standard deviation of the random noise in the data. You can do this type of computation easily in Matlab/Octave and in spreadsheets (page 177).

You can download a Matlab/Octave script that compares the Monte Carlo simulation to the algebraic method above from http://terpconnect.umd.edu/~toh/spectrum/LinearFiMC.m. By running this script with different sizes of data sets ("NumPoints" in line 10), you can see that the standard deviation predicted by the algebraic method fluctuates a lot from run to run when NumPoints is small (e.g., 10), but the Monte Carlo predictions are much steadier. When NumPoints is large (e.g., 1000), both methods agree very well.

The Bootstrap method

The third method is the "bootstrap" method, a procedure that involves choosing random sub-samples with replacement from a single data set and analyzing each sample the same way (e.g. by a least-squares fit). Every sample is returned to the data set after sampling, so that (a) a particular data point from the original data set could appear multiple times in each sample, and (b) the number of elements in each bootstrap sub-sample equals the number of elements in the original data set. This is best explained by a simple example. Consider a data set with 10 x,y pairs assigned the letters *a* through *j*. The original data set is represented as [*a b c d e f g h i j*], and some typical bootstrap sub-samples might be [*a b b d e f f h i i*] or [*a a c c e f g g i j*]. Each bootstrap sample contains 10 data points, but with about a third of the data pairs skipped, a third duplicated, and a third unchanged. (This is equivalent to weighting a third of the data pairs by a factor of 2, a third by 0, and leaving a third unweighted). You would use a computer to generate hundreds of bootstrap samples like that and to apply the calculation procedure under investigation (in this case a linear least-squares) to each set.

If there were *no* noise in the data set, and if the model were properly chosen, then all the points in the original data set and in each the bootstrap sub-sample would fall *exactly on the model line*, with the result that the least-squares results would be the *same for every subsample*.



However, if there *is* noise in the data, each bootstrap sub-sample would give a *slightly different result* (e.g., the least-squares polynomial coefficients), because each sub-sample has a different subset of the random noise. This is illustrated by the animation on the right (which you can view if download the Microsoft Word 365 version, otherwise click this link), for the same 100-point straight-line data set used above. You can see that *the variation in the best-fit coefficients between sub-samples is very close to the Monte Carlo simulation above*. The greater the amount of random noise in the data set, the greater the range of results from sample to sample in the bootstrap set would be. This enables you to estimate the uncertainty of the quantity you are estimating, just as in the Monte-Carlo method above. The difference is that the Monte-Carlo method assumes that the noise is known, random, and can be accurately simulated by a random number generator on a computer, whereas the bootstrap method uses the *actual noise in the data set* at hand, like the algebraic

method, except that it does not need an algebraic solution of error propagation. The bootstrap method thus shares its generality with the Monte Carlo approach but is limited by the assumption that the noise in that (possibly small) single data set is representative of the noise that would be obtained upon repeated measurements. The bootstrap method cannot, however, correctly estimate the parameter errors resulting from poor model selection. The method is examined in detail in its extensive literature. This type of bootstrap computation is easily done in Matlab/Octave and can even be done (with somewhat greater difficulty) in spreadsheets.

Comparison of error prediction methods.

The Matlab/Octave script TestLinearFit.m compares *all three* of these methods (Monte Carlo simulation, the algebraic method, and the bootstrap method) for a 100-point first-order linear least-squares fit. Each method is repeated on different data sets with the same average slope, intercept, and random noise, then the standard deviation (SD) of the slopes (`SDslope`) and intercepts (`SDint`) were compiled and are tabulated below.

```
NumPoints = 100  SD of the Noise = 9.236 x-range = 30
             Simulation     Algebraic equation   Bootstrap method
            SDslope SDint     SDslope SDint       SDslope SDint
Mean SD:    0.1140  4.1158    0.1133  4.4821      0.1096  4.0203
```

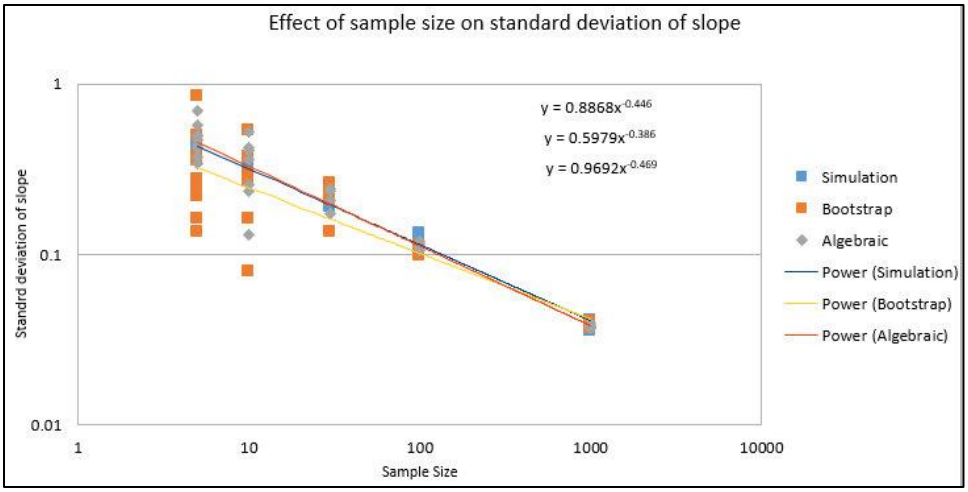
(You can download this script from http://terpconnect.umd.edu/~toh/spectrum/TestLinearFit.m). On average, the mean standard deviations ("Mean SD") of the three methods agree very well, but the algebraic and bootstrap methods fluctuate more than the Monte Carlo simulation each time this script is run, because they are based on the noise in one *single* 100-point data set, whereas the Monte Carlo simulation reports the average of many data sets. Naturally, the algebraic method is simpler and faster to compute than the other methods. However, an algebraic propagation of error solution is not always possible to obtain, whereas the Monte Carlo and bootstrap methods do not depend on an algebraic solution and can be applied readily to more complicated curve-fitting situations, such as non-linear iterative least-squares, as will be seen later.

Effect of the number of data points on least-squares fit precision

The spreadsheets EffectOfSampleSize.ods or EffectOfSampleSize.xlsx, which collect the results of many runs of TestLinearFit.m with different numbers of data points ("NumPoints"), demonstrates that the standard deviation of the slope and the intercept *decrease* if the number of data points is *increased*. On average, the *standard deviations are inversely proportional to the square root of the number of data points,* which is consistent with the observation that the slope of a log-log plot is roughly 1/2.

These plots really dramatize the problem of small sample sizes, but this must be balanced against the cost of obtaining more data points. For example, in analytical chemistry calibration, a larger number of calibration points could be obtained either by preparing and measuring more standard solutions or by reading each of a smaller number of standards repeatedly. The former approach accounts for both the

volumetric errors in preparing solutions and the random noise in the instrument readings, but the labor and cost of preparing and running large numbers of standard solutions, and safely disposing of them afterward, is limiting. The latter approach is less expensive but is less reliable because it accounts only for the random noise in the instrument readings. Overall, it is better to refine the laboratory techniques and instrument settings to minimize error than to attempt to compensate by taking lots of readings.

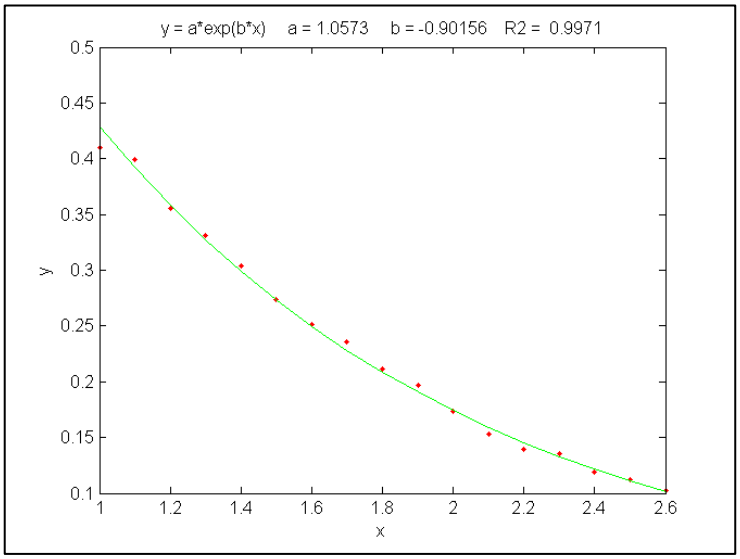


To smooth or not to smooth? *It is usually best that a noisy signal not be* smoothed *before the least-squares calculations*, because doing so will *not* improve the reliability of the least-squares results, but it will cause both the algebraic propagation-of-errors and the bootstrap calculations to *seriously underestimate* the standard deviation of the least-squares results. You can demonstrate using the script TestLinearFit.m by setting SmoothWidth in line 10 to something higher than 1, which will smooth the data before the least-squares calculations. This has no significant effect on the *actual* standard deviation as calculated by the Monte Carlo method, but it does significantly reduce the *predicted* standard deviation calculated by the algebraic propagation-of-errors and (especially) the bootstrap method. For similar reasons, if the noise is pink rather than white, the bootstrap error estimates will also be too low. Conversely, if the noise is blue, as occurs in processed signals that have been subjected to some sort of differentiation process or that have been deconvoluted from some blurring process, then the errors predicted by the algebraic propagation-of-errors and the bootstrap methods will be *high*. (You can prove this to yourself by running TestLinearFit.m with pink and blue noise modes selected in lines 23 and 24). Bottom line: error prediction works best for *white* noise.

Transforming non-linear relationships

In some cases, a fundamentally non-linear relationship can be transformed into a form that is amenable to polynomial curve fitting by means of a coordinate transformation (e.g., taking the log or the reciprocal of the data), and then the least-squares method can be applied to the resulting linear equation. For example, the signal in the figure below is from a simulation of exponential decay that has the mathematical form Y = **a** exp(**b**X), where X=time, Y=signal intensity, **a** is the Y-value at X=0 and **b** is the decay constant. This is a fundamentally non-linear problem because Y is a non-linear function of the parameter **b**. However, by taking the natural log of both sides of the equation, you obtain ln(Y)=ln(**a**) + **b**X. In this equation, Y is a *linear* function of both parameters ln(**a**) and **b**, so it can be fit by the least-squares method to estimate ln(**a**) and **b**, from which you get **a** by computing exp(ln(**a**)). In this example, the "true" values of the coefficients are **a** =1 and **b** = -0.9, but random noise has been added to each data point, with a standard deviation equal to 10% of the value of that data point, to simulate a typical experimental measurement in the laboratory. An estimate of the values of ln(**a**) and **b**, given only the noisy data points, can be determined by the least-squares curve fitting of

ln(Y) vs X.



An exponential least-squares fit (solid line) applied to a noisy data set (points) to estimate the decay constant.

The best-fit equation, shown by the green solid line in the figure, is Y =0.959 exp(- 0.905 X), that is, **a** = 0.959 and **b** = -0.905, which are reasonably close to the expected values of 1 and -0.9, respectively. Thus, even in the presence of substantial random noise (10% relative standard deviation), it is possible to get reasonable estimates of the parameters of the underlying equation (to within about 4%). The most important requirement is that the model must be good, that is, that the equation selected for the model accurately describes the underlying behavior of the system (except for noise). Often that is the most difficult aspect because the underlying models are not always known with certainty. In Matlab and Octave, is fit can be performed in a single line of code: **`polyfit(x,log(y),1)`**, which returns `[b log(a)]`. (In Matlab and Octave, "log" is the natural log, "log10" is the base-10 log).

Another example of the linearization of an exponential relationship is explored on page 326: Signal and Noise in the Stock Market.

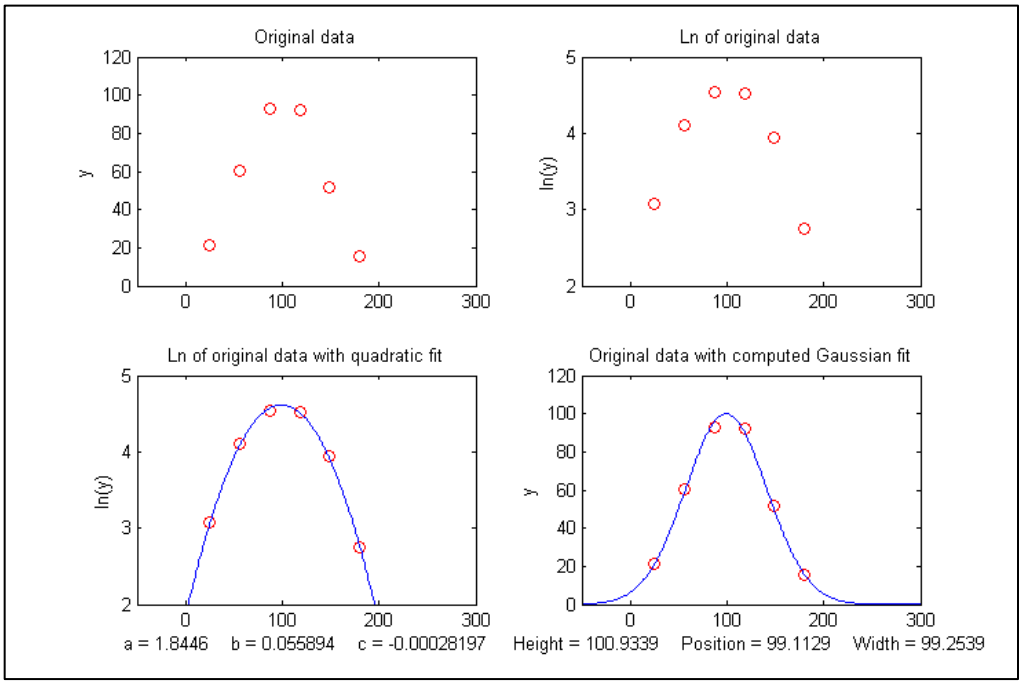
Other examples of non-linear relationships that can be linearized by coordinate transformation include the logarithmic (Y = **a** ln(**b**X)) and power (Y=**a**$X^{\mathbf{b}}$) relationships. Methods of this type used to be very common back in the days before computers, when fitting anything but a straight line was difficult. It is still used today to extend the range of functional relationships that can be handled by common linear least-squares routines available in spreadsheets and hand-held calculators. (My Matlab/Octave function trydatatrans.m tries eight different simple data transformations on any given x,y data set and fits the transformed data to a straight line or polynomial). Only a few non-linear relationships can be handled by simple data transformation, however. To fit *any* arbitrary custom function, you may have to resort to the *iterative* curve fitting method, which will be treated in Curve Fitting C.

Simple fitting of Gaussian and Lorentzian peaks by data transformation

An interesting example of the use of transformation to convert a non-linear relationship into a form that is amenable to polynomial curve fitting is the use of the natural log (ln) transformation to convert a positive Gaussian peak, which has the fundamental functional form exp(-x^2), into a parabola of the form -x^2, which can be fit with a second-order polynomial (quadratic) function (y = **a** + **b**x + **c**x^2). The

equation for a Gaussian peak is y = **h***exp(-((x-**p**)./(1/(2*sqrt(ln(2)))***w**)) ^2)), where **h** is the peak height, **p** is the x-axis location of the peak maximum, **w** is the full width of the peak at half-maximum. The natural log of *y* can be shown to be log(**h**)-(4 **p**^2 log(2))/**w**^2+(8 **p** *x* log(2))/**w**^2-(4 *x*^2 log(2))/**w**^2, which is a quadratic form in the independent variable x because it is the sum of x^2, x, and constant terms. Expressing each of the peak parameters **h**, **p**, and **w** in terms of the three quadratic coefficients, a little algebra (courtesy of Wolfram Alpha) will show that all three parameters of the peak (height, maximum position, and width) can be calculated from the three quadratic coefficients **a**, **b**, and **c**. The peak height is given by exp(**a**-**c***(**b**/(2***c**))^2), the peak position by -**b**/(2***c**), and the peak half-width by 2.35482/(sqrt(2)*sqrt(-**c**)). This is called "Caruana's Algorithm", named after Rich Caruana, an American computer scientist and researcher. The area under the Gaussian peak can also be calculated and is approximately 1.064467***h****w**.

One advantage of this type of Gaussian curve fitting, as opposed to simple visual estimation, is illustrated in the figure below. The signal is a synthesized Gaussian peak with a true peak height of exactly 100 units, a true peak position of 100 units, and a true half-width of 100 units, but it is *sparsely*

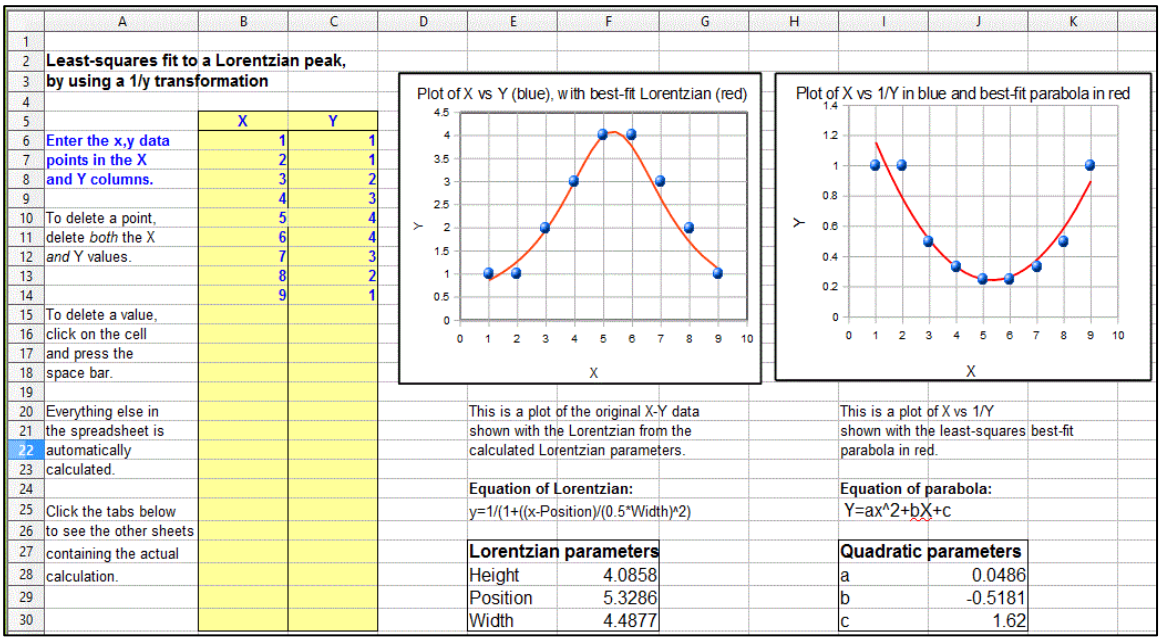


sampled only every 31 units on the x-axis. The resulting data set, shown by the red points in the upper left, *has only 6 data points on the peak itself*. If you were to take the maximum of those 6 points (the 3rd point from the left, with x=87, y=95) as the peak maximum, you would get a poor approximation to the true values of peak position (100) and height (100). If you were to take the distance between the 2nd the 5th data points as the peak width, you would get only 3*31=93, compared to the true value of 100. If you were to attempt to calculate the *area* under the peak from those measurements, you would get 1.064467*95*93=9404.6, much lower than the theoretical width of 1.064467*height*width=10644.67. These are all very poor estimates. *However*, taking the *natural log* of the data (upper right) produces a *parabola* that can be fitted with a quadratic least-squares fit (shown by the blue line in the lower left). From the three coefficients of the quadratic fit, you can calculate much more accurate values of the Gaussian peak parameters, shown at the bottom of the figure: height=100.93; position=99.11; width=99.25; area= 10663. The plot in the lower right shows the resulting Gaussian fit (in blue)

displayed with the original data (red points). The accuracy of those peak parameters (about 1% in this example) is limited only by the noise in the data: *much more accurate, at little computational cost.*

The figure above was created in Matlab (or Octave), using this script. (The Matlab/Octave function gaussfit.m performs the calculation for an x,y data set. You can also download a spreadsheet that does the same calculation. It is available in OpenOffice Calc (Download link, Screenshot) and Excel formats). The method is simple and very fast, but for this method to work properly, the data set *must not contain any zeros or negative points*. If the signal-to-noise ratio is very poor, it may be useful to skip those points or to pre-smooth the data slightly to reduce this problem. Moreover, the original Gaussian peak signal must be a single isolated peak with a zero baseline, that is, must tend to zero far from the peak center. In practice, this means that any non-zero baseline must be subtracted from the data set before applying this method. (A more general but computationally slower approach to fitting Gaussian peaks, which works for data sets with zeros and negative numbers and also for data with multiple overlapping peaks, is the non-linear iterative curve fitting method, which will be treated later, on page 199).

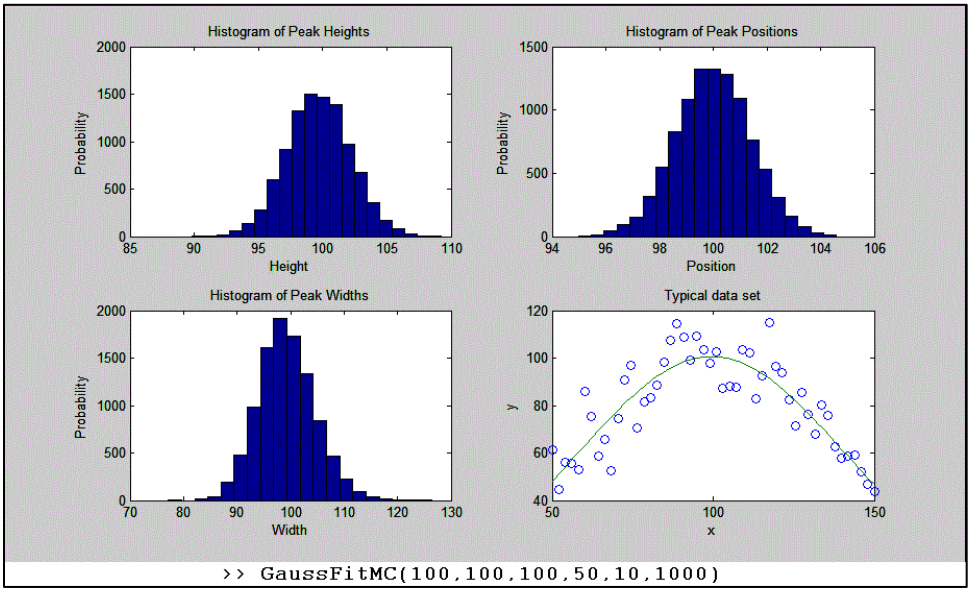
A similar method can be derived for a Lorentzian peak, which has the fundamental form y=**h**/(1+((x-**p**)/(0.5***w**))^2), by fitting a quadratic to the reciprocal of y. As for the Gaussian peak, all three parameters of the peak (height **h**, maximum position **p**, and width **w**) can be calculated from the three quadratic coefficients **a**, **b**, and **c** of the quadratic fit: **h**=4***a**/((4***a****c**)-**b**^2), **p**= -**b**/(2***a**), and **w**= sqrt(((4***a****c**)-**b**^2)/**a**)/sqrt(**a**). Just as for the Gaussian case, the data set must not contain any zero or negative y values. The Matlab/Octave function lorentzfit.m performs the calculation for an x,y data set, and the Calc and Excel spreadsheets LorentzianLeastSquares.ods and LorentzianLeastSquares.xls perform the same calculation (illustrated below).



(By the way, a quick way to test either of the above methods is to use this *minimal peak data set*: x=5, 20, 35 and y=5, 10, 5, which has a height, position, and width equal to 10, 20, and 30, respectively, for a single isolated symmetrical peak of any shape, assuming only a baseline of zero). Try it.

To apply the above methods to signals containing *two or more* Gaussian or Lorentzian peaks, it is necessary to locate all the peak maxima first, so that the proper groups of points centered on each peak can be processed with the algorithms just discussed. That is discussed on page 234.

However, there is a downside to using coordinate transformation methods to convert non-linear relationships into simple polynomial form, and that is that the noise is also affected by the transformation, with the result that the propagation of error from the original data to the final results is often difficult to predict. For example, in the method just described for measuring the peak height, position, and width of Gaussian or Lorentzian peaks, the results depend not only on the amplitude of noise in the signal, but also on how many points across the peak are taken for fitting. As you take more



points far from the peak center, where the y-values approach zero, the natural log of those points approaches negative infinity as y approaches zero. The result is that the noise of those low-magnitude points is unduly magnified and has a disproportional effect on the curve fitting. This runs counter the usual expectation that the quality of the parameters derived from curve fitting improves with the square root of the number of data points (page 221). A reasonable compromise in this case is to take *only the points in the top half of the peak*, with Y-values down to one-half of the peak maximum. If you do that, the error propagation (predicted by a Monte Carlo simulation with constant normally-distributed random noise) shows that the relative standard deviations of the measured peak parameters are directly proportional to the noise in the data and inversely proportional to the square root of the number of data points (as expected), but that the proportionality constants differ:

(a) the relative standard deviation of the peak height = 1.73**noise*/sqrt(*N*),

(b) the relative standard deviation of the peak position = *noise*/sqrt(*N*),

(c) the relative standard deviation of the peak width = 3.62**noise*/sqrt(*N*),

where *noise* is the standard deviation of the noise in the data and *N* is the number of data points taken for the least-squares fit. You can see from these results that the measurement of peak *position* is most precise, followed by the peak *height*, with the peak *width* being the least precise. If one were to include points far from the peak maximum, where the signal-to-noise ratio is very low, the results would be poorer than predicted. These predictions depend on knowledge of the noise in the signal. If only a

single sample of that noise is available for measurement, there is no guarantee that sample is a representative sample, especially if the total number of points in the measured signal is small. The standard deviation of small samples is notoriously variable. Moreover, these predictions are based on a simulation with *constant normally-distributed white* noise. Had the actual noise varied with signal level or with x-axis value, or if the probability distribution had been something other than normal, those predictions would not necessarily have been accurate. In such cases, the bootstrap method (page 170) has the advantage that it samples the actual noise in the signal.

You can download the Matlab/Octave code for this Monte Carlo simulation from http://terpconnect.umd.edu/~toh/spectrum/GaussFitMC.m, view screen capture. A similar simulation (http://terpconnect.umd.edu/~toh/spectrum/GaussFitMC2.m, view screen capture) compares this method to fitting the entire Gaussian peak with the iterative method in Curve Fitting 3, finding that the precision of the results is only slightly better with the (slower) iterative method.

Note 1: If you are reading this online, you can right-click on any of the m-file links above and select "**Save Link As...**" to download them to your computer for use within Matlab/Octave.

Note 2: In the curve fitting techniques described here and in the next two chapters, there is no requirement that the x-axis interval between data points be uniform, as is the assumption in many of the other signal processing techniques previously covered. Curve fitting algorithms typically accept a set of arbitrarily spaced x-axis values and a corresponding set of y-axis values.

Math and software details for linear least squares

The least-squares best fit for an x,y data set can be computed using only basic arithmetic. Here are the relevant equations for computing the slope and intercept of the first order best-fit equation, y = intercept + slope*x, as well as the predicted standard deviation of the slope and intercept, and the coefficient of determination, R^2, which is an indicator of the "goodness of fit". (R^2 is 1.0000 if the fit is perfect and less than that if the fit is imperfect).

n = number of x,y data points
sumx = Σx
sumy = Σy
sumxy = Σx*y
sumx2 = Σx*x
meanx = sumx / n
meany = sumy / n
slope = (n*sumxy - sumx*sumy) / (n*sumx2 - sumx*sumx)
intercept = meany-(slope*meanx)
ssy = Σ(y-meany)^2
ssr = Σ(y-intercept-slope*x)^2
R^2 = 1-(ssr/ssy)
Standard deviation of the slope = SQRT(ssr/(n-2))*SQRT(n/(n*sumx2 - sumx*sumx))
Standard deviation of the intercept = SQRT(ssr/(n-2))*SQRT(sumx2/(n*sumx2 - sumx*sumx))

(In these equations, Σ represents summation. For example, Σx means the sum of all the x values, and Σx*y means the sum of all the x*y products, etc.)

The last two lines predict the standard deviation of the slope and the intercept, based only on that data sample, assuming that the deviations from the line are random and normally-distributed. These are estimates of the variability of slopes and intercepts you are likely to get if you repeated the data measurements over and over multiple times under the same conditions, assuming that the deviations from the straight line are due to *random variability* and not a systematic error caused by non-linearity. If the deviations are random, they will be slightly different from time to time, causing the slope and intercept to vary from measurement to measurement, with a standard deviation predicted by these last two equations. However, if the deviations are caused by systematic non-linearity, they will be the same from measurement to measurement, in which case the prediction of these last two equations will not be relevant, and you might be better off using a polynomial fit such as a quadratic or cubic.

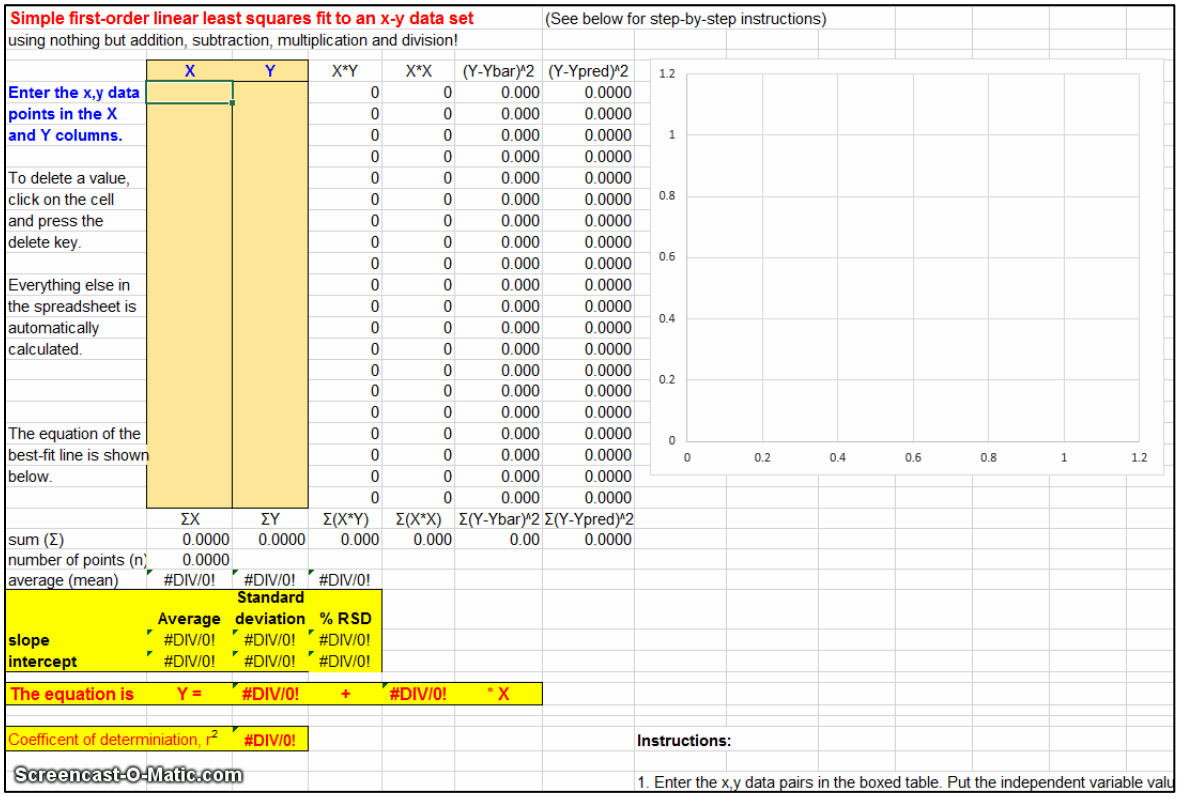
The reliability of these standard deviation estimates depends on the assumption of random deviations and on the number of data points in the curve fit. They improve with the square root of the number of points. A slightly more complex set of equations can be written to fit a second-order (quadratic or parabolic) equations to a set of data. Instead of a slope and intercept, three coefficients are calculated, **a**, **b**, and **c**, representing the coefficients of the quadratic equation $\mathbf{a}x^2+\mathbf{b}x+\mathbf{c}$.

These calculations could be performed step-by-step by hand, with the aid of a calculator or a spreadsheet, with a program written in any programming language, such as a Matlab or Octave script.

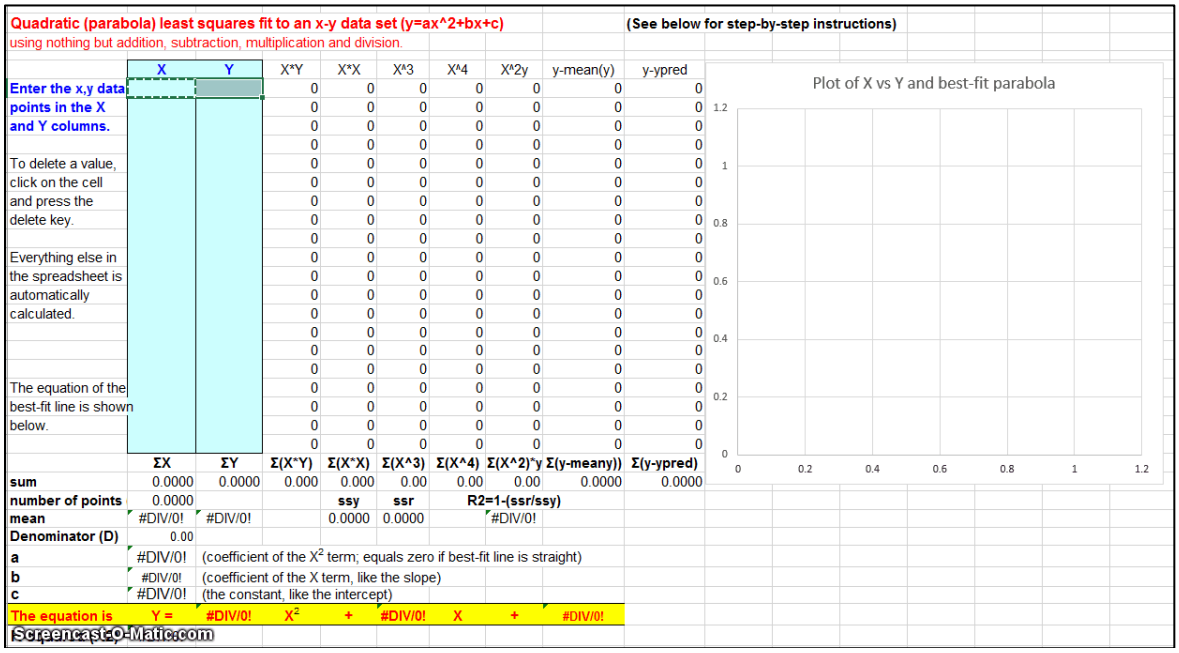
Web sites: Wolfram Alpha includes some capabilities for least-squares regression analysis, including linear, polynomial, exponential, and logarithmic fits. Statpages.org can perform a huge range of statistical calculations and tests, and there are several Web sites that specialize in plotting and data visualization that have curve-fitting capabilities, including most notably Plotly and MyCurveFit. Web sites such as these can be very handy when working from a smartphone, tablet, or a computer that does not have suitable computational software. If you are reading this online, you can **Ctrl-Click** on these links to open them in your browser.

Spreadsheets for linear least squares

Spreadsheets can perform the math described above easily. The spreadsheets pictured below (LeastSquares.xls and LeastSquares.odt for linear fits and QuadraticLeastSquares.xls and QuadraticLeastSquares.ods for quadratic fits), utilize the expressions given above to compute and plot linear and quadratic (parabolic) least-squares fit, respectively. (Viewed in Word 365 or on the web, these animations show the result of entering a small set of data point by point). The advantage of spreadsheets is that they are highly customizable for your application and can be deployed on mobile devices such as tablets or smartphones. For straight-line fits, you can use the convenient built-in functions *slope* and *intercept*.



Animation of spreadsheet template data entry for first order (linear) least-squares fit (https://terpconnect.umd.edu/~toh/spectrum/LeastSquares.GIF)



Animation of spreadsheet template data entry for second order (quadratic) least-squares fit. (https://terpconnect.umd.edu/~toh/spectrum/QuadraticLeastSquares.GIF)

The LINEST function. Modern spreadsheets also have *built-in* facilities for computing polynomial least-squares curve fits of *any* order. For example, the LINEST function in both Excel and OpenOffice Calc can be used to compute polynomial and other curvilinear least-squares fits. In addition to the best-

fit polynomial coefficients, the LINEST function also calculates at the same time the standard error values, the determination coefficient (R^2), the standard error value for the *y* estimate, the F statistic, the number of degrees of freedom, the regression sum of squares, and the residual sum of squares. A significant inconvenience of LINEST, compared to working out the math using the series of mathematical expressions described above, is that it is more difficult to adjust to a variable number of data points and to remove suspect data points or to change the order of the polynomial. LINEST is an *array function*, which means that when you enter the formula in one cell, multiple cells will be used for the output of the function. *You cannot edit a LINEST function just like any other spreadsheet function.* To specify that LINEST is an array function, do the following. Highlight the entire formula, including the "=" sign. On a Macintosh, hold down the "apple" key and press "**Enter**." On a PC hold down the "Ctrl" and "Shift" keys and press "**Enter**." Excel adds "{ }" brackets around the formula, to show that it is an array. Note that you cannot type in the "{ }" characters yourself. If you do Excel will treat the cell contents as characters and not a formula. *Highlighting the full formula and typing the "apple" key or "Ctrl"," Shift" and "return" is the only way to enter an array formula.* This instruction sheet from Colby College may help: http://www.colby.edu/chemistry/PChem/notes/linest.pdf.

Practical Note: If you are working with a template that uses the LINEST function, and you wish to change the number of data points, the easiest way to do that is to select the rows or columns containing the data, right-click on the row or column *heading* (1,2,3 or, A, B, C, etc.) and use the **Insert** or **Delete** in the right-click menu. If you do it that way, the LINEST function referring to those rows or columns will be adjusted *automatically*. That is easier than trying to edit the LINEST function directly. (If you are inserting rows or columns, you must drag-copy the formulas from the older rows or columns into the newly inserted empty ones). See CalibrationCubic5Points.xls for an example.

Application to analytical calibration and measurement

There are specific versions of these spreadsheets that apply curve fitting to calibration curves (plots of signal measurements vs standards of known concentration) and which also calculate the concentrations of unknown sample (download complete set as CalibrationSpreadsheets.zip). Of course, these spreadsheets can be used for just about any measurement calibration application. Just change the labels of the columns and axes to suit your application.

Another set of spreadsheets is used to perform Monte Carlo simulations of the calibration and measurement process using several widely-used analytical calibration methods, including first-order (straight line) and second-order (curved line) least-squares fits. Typical systematic and random errors in both signal and in volumetric measurements are included, for the purpose of demonstrating how non-linearity, interferences, and random errors combine to influence the result (the so-called "propagation of errors").

For fitting *peaks*, GaussianLeastSquares.odt, is an OpenOffice spreadsheet that fits a quadratic function to the natural log of y(x) and computes the height, position, and width of the Gaussian that is the best fit to y(x). There is also an Excel version (GaussianLeastSquares.xls). LorentzianLeastSquares.ods and LorentzianLeastSquares.xls fits a quadratic function to the reciprocal of y(x) and computes the height, position, and width of the Lorentzian that is a best fit to y(x). Note that for either of these fits, the data may not contain zeros or negative points, and the baseline (the value that y approaches far from the peak center) must be zero. See Fitting Peaks, above.

Matlab and Octave

Matlab and **Octave** have simple built-in functions for least-squares curve fitting: polyfit and polyval. For example, if you have a set of x,y data points in the vectors "x" and "y", then the coefficients for the least-squares fit are given by **`coef=polyfit(x,y,n)`**, where "n" is the order of the polynomial fit: n = 1 for a straight-line fit, 2 for a quadratic (parabola) fit, etc. The polynomial coefficients 'coef" are given in decreasing powers of x. For a straight-line fit (n=1), `coef(1)` is the slope ("b") and `coef(2)` is the intercept ("a"). For a quadratic fit (n=2), `coef(1)` is the x^2 term ("c"), `coef(2)` is the x term ("b") and `coef(3)` is the constant term ("a").

The fit equation can be evaluated using the function polyval, for example, **`fity=polyval(coef,x).`** This works for any order of polynomial fit ("n"). You can plot the data and the fitted equation together using the `plot` function: **`plot(x,y,'ob',x,polyval(coef,x),'-r'),`** which plots the data as blue circles and the fitted equation as a red line. You can plot the residuals by writing **`plot(x,y-polyval(coef,x))`**.

When the number of data points is small, you might notice that the fitted curve is displayed as a series of straight-line segments, which can look ugly. You can get a smoother plot of the fitted equation, evaluated at more finely divided values of x, by defining **`xx=linspace(min(x),max(x));`** and then using xx rather than x to evaluate and plot the fit: **`plot(x,y,'ob',xx,polyval(coef,xx),'-r')`**.

`[coef,S] = polyfit(x,y,n)` returns the polynomial coefficients coef and a structure 'S' used to obtain error estimates.

```
>> [coef,S]=polyfit(x,y,1)
coef =
    1.4913    6.5552
S =
        R: [2x2 double]
       df: 2
    normr: 2.2341
>> S.R
ans =
  -18.4391   -1.6270
         0   -1.1632
```

The vector of standard deviations of the coefficients can be computed from S by the expression **`sqrt(diag(inv(S.R)*inv(S.R')).*S.normr.^2./S.df)'`**, in the same order as the coefficients.

Matrix Method

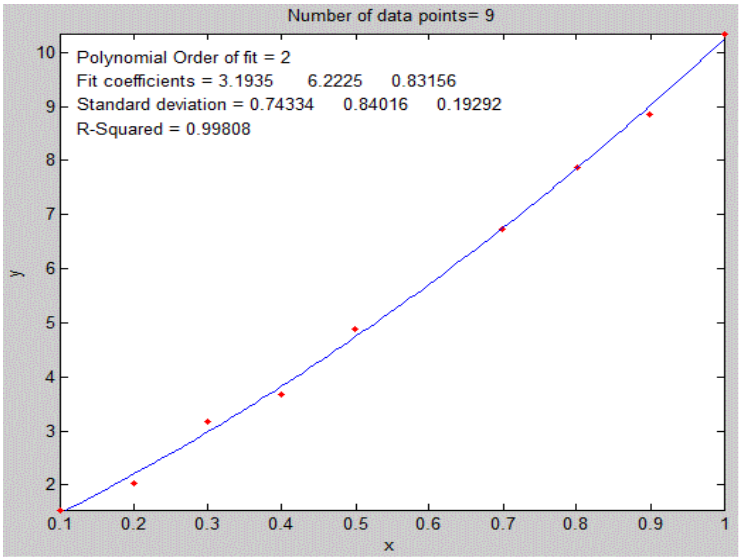
Alternatively, if you don't like using the Matlab/Octave built-in polyfit function, you may perform the polynomial least-squares calculations for the row vectors x,y by using the matrix method with the Matlab "/" symbol, meaning "right matrix divide". The coefficients of a first-order fit are given by **`y/[x;ones(size(y))]`** and a second-order (quadratic) fit by **`y/[x.^2;x;ones(size(y))]`** . For higher-order polynomials, just add another row to the denominator matrix, for example, a third-order fit is given by **`y/[x.^3;x.^2;x;ones(size(y))]`** and so on. The coefficients are returned in the same order as polyfit, in decreasing powers of x (e.g., for a first-order fit, *slope* first (the x^1 term) then *intercept* (the x^0 term). Using the example of the first-order fit to the SD memory card prices:

```
>>x=[2 4 8 16];
>> y=[9.99 10.99 19.99 29.99];
>> polyfit(x,y,1)
ans =
    1.4913    6.5552
>> y/[x;ones(size(y))]
ans =
    1.4913    6.5552
```

This shows that the *slope and intercept results for the polyfit function and for the matrix method are the same*. (The slope and intercept results may be the same, but the polyfit function has the advantage that it also can compute the *error estimates* with little extra effort, as described above, page 167).

The plotit.m function

The graph below was generated by my Matlab/Octave function plotit.m, in the form plotit(data) or plotit(data,polyorder). This function uses *all the techniques mentioned in the previous paragraph*. It accepts 'data' in the form of a single vector, or a pair of vectors "x" and "y", or a 2xn or nx2 matrix with x in the first row or column and y in the second and plots the data points as red dots. If the optional input argument "polyorder" is provided, plotit fits a polynomial of order "polyorder" to the data and plots the fit as a blue line and displays the fit coefficients and the goodness-of-fit measure R^2 in the upper left corner of the graph.



On the next page is a Matlab/Octave example of the use of plotit.m to perform the coordinate transformation described, on page 172, to fit an exponential relationship, showing both the original exponential data and the transformed data with a linear fit in the figure(2) and figure(1) windows, respectively. If you are reading this online, click to download.

```
x=1:.1:2.6;
a=1;
b=-.9;
y=a.*exp(b.*x);
y=y+y.*.1.*rand(size(x));
figure(1)
[coeff,R2]=plotit(x,log(y),1);
ylabel('ln(y)');
title('Plot of x vs the natural log (ln) of y')
aa=exp(coeff(2));
bb=coeff(1);
yy= aa.*exp(bb.*x);
figure(2)
plot(x,y,'r.',x,yy,'g')
xlabel('x');
```

```
ylabel('y');
title(['y = a*exp(b*x)      a = ' num2str(aa)   '      b = '
num2str(bb)   '    R2 =   ' num2str(R2) ] ) ;
```

In version 5 or 6 the syntax of plotit can be **[coef, RSquared, StdDevs] =plotit(x,y,n)**. It returns the best-fit coefficients **'coeff'**, in decreasing powers of x, the standard deviations of those coefficients **'StdDevs'** in the same order, and the R-squared value. To compute the *relative* standard deviations, just type **StdDevs./coef**. For example, the following script computes a straight line with five data points and a slope of 10, an intercept of zero, and noise equal to 1.0. It then uses plotit.m to plot and fit the data to a first-order linear model (straight line) and compute the estimated standard deviation of the slope and intercept, if you run this repeatedly, you will observe that the measured slope and intercept are usually within two standard deviations of 10 and zero respectively. Try it with different values of "Noise".

```
NumPoints=5;
slope=10;
Noise=1;
x=round(10.*rand(size(1:NumPoints)));
y=slope*x+Noise.*randn(size(x));
[coef,RSquared,StdDevs]=plotit(x,y,1)
```

Comparing two data sets. Plotit can also be used to compare to two different dependent variable vectors (e.g., y1 and y2) *if they share the same independent variables x*, for example to determine the similarity of two different spectra measured over the same wavelengths as was done on page 14:

```
[coeff,R2]=plotit(y1,y2,1);
```

R^2 is a measure of similarity. The closer R^2 is to 1.000, the more similar they are. If y1 and y2 are two measurements of the *same* signal with different random noise, the plot will show a random scatter of points along a straight line with a slope, coeff(1), of 1.00. If the y1 and y2 are the same signal with different amplitudes, the slope of the line will equal their average ratio. If the data points are curved and loop around, the difference between the two y vectors is greater than the random noise.

The syntax can be optionally **plotit(x,y,n,datastyle,fitstyle)**, where `datastyle` and `fitstyle` are optional strings specifying the line and symbol style and color, in standard Matlab convention. The strings, in single quotes, are made from one element from any or all the following 3 columns:

```
b   blue        .   point              -    solid
g   green       o   circle             :    dotted
r   red         x   x-mark             -.   dashdot
c   cyan        +   plus               --   dashed
m   magenta     *   star            (none)  no line
y   yellow      s   square
k   black       d   diamond
w   white       v   triangle (down)
                ^   triangle (up)
                <   triangle (left)
                >   triangle (right)
                p   pentagram
                h   hexagram
```

For example, `plotit(x,y,3,'or','-g')` plots the data as red circles and the fit as a green solid line (the default is red dots and a blue line, respectively).

You can use plotit.m in Matlab to linearize and plot other nonlinear relationships, such as:

y = a exp(bx) : `[coeff,R2]=plotit(x,log(y),1); f=exp(coeff(2)); b=coeff(1);`
y = a ln(bx) : `[coeff,R2]=plotit(log(x),y,1); a=coeff(1); b=log(coeff(2));`
$y=ax^b$: `[coeff,R2]=plotit(log(x),log(y),1); a=exp(coeff(2)); b=coeff(1);`
$y=start(1+rate)^x$: `[coeff,R2]=plotit(x,log(y),1); start=exp(coeff(2)); rate=exp(coeff(1))-1;`

This last one is the expression for *compound interest,* covered on page 326: Signal and Noise in the Stock Market.

Do not forget that in Matlab/Octave, "log" means *natural log*. The *base-10* log is denoted by "log10".

Estimating the coefficient errors. The plotit function also has a built-in bootstrap routine that computes coefficient error estimates by the bootstrap method (Standard deviation STD and relative standard deviation RSD) and returns the results in the matrix "BootResults" (of size 5 x polyorder+1). You can change the number of bootstrap samples in line 101. The calculation is triggered by including a 4th *output* argument, e.g.

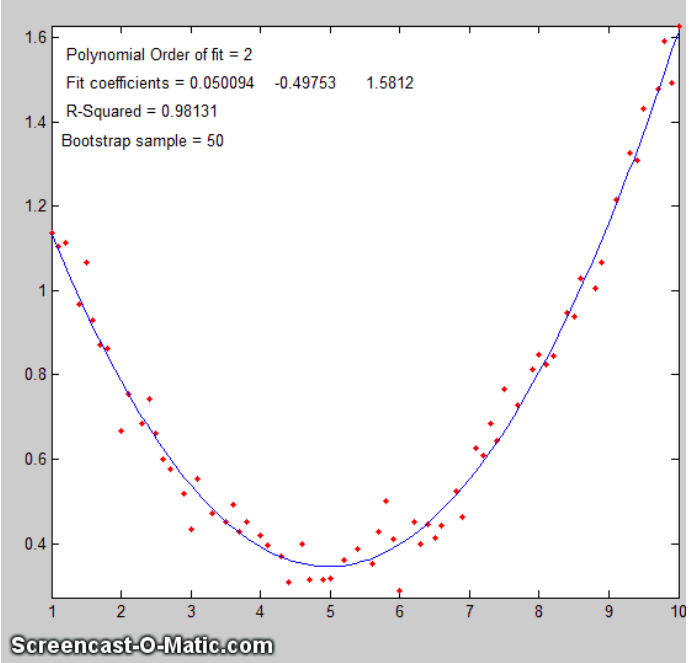
```
[coef,RSquared,StdDevs, BootResults] = plotit(x,y,polyorder).
```

This works for any polynomial order. For example:

```
>> x=0:100;
>> y=100+(x*100)+100.*randn(size(x));
>> [FitResults, GOF, baseline, coeff, residual, xi, yi, BootResults] =
plotit(x,y,1);
```

The above statements compute a straight line with an intercept and slope of 100, plus random noise with a standard deviation of 100, then fits a straight line to the data and prints out a table of bootstrap error estimates, with the slope in the first column and the intercept in the second column:

```
Bootstrap Results
Mean:          100.359       88.01638
STD:           0.204564       15.4803
STD (IQR):     0.291484       20.5882
% RSD:         0.203832       17.5879
% RSD (IQR):  0.290441       23.3914
```

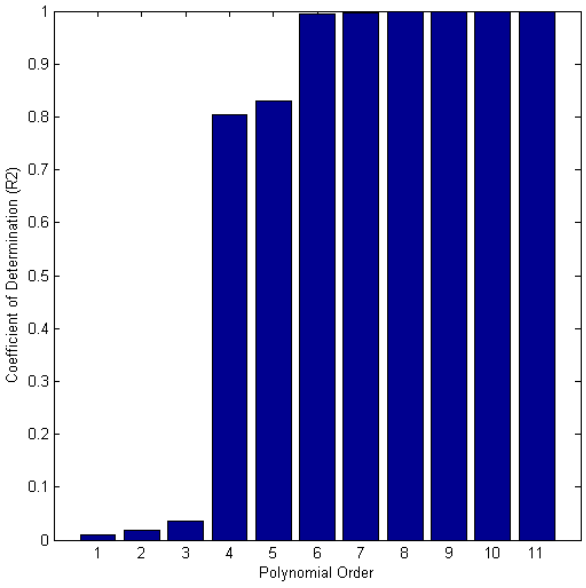


The variation plotfita *animates* the bootstrap process for instructional purposes, as shown in the animation on the right for a quadratic fit. You must include the output arguments, for example:

```
[coef, RSquared, BootResults] = plotfita([1 2 3 4 5 6],[1 3 4 3 2 1],2);
```

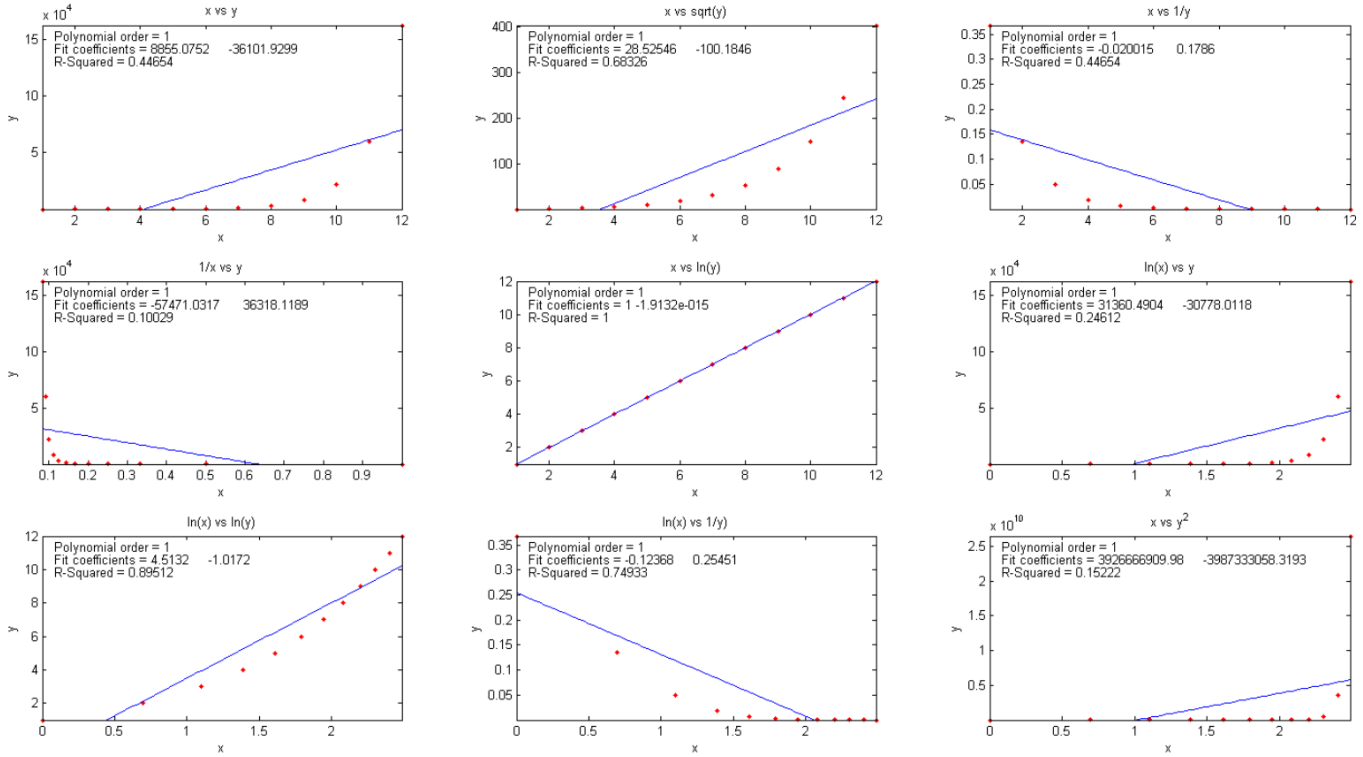
The variation logplotfit plots and fits log(x) vs log(y), for data that follows a power-law relationship or that covers a very wide numerical range.

Comparing polynomial orders. My function trypoly(x,y) fits the data in x,y with a series of polynomials of degree 1 through length(x)-1 and returns the coefficients of determination (R2) of each fit as a vector, showing that, for *any* data, the coefficient of determination R^2 approaches 1 as the polynomial order approaches length(x)-1. The variant trypolyplot(x,y) creates a bar graph such as shown on the left.



Comparing data transformations. The function trydatatrans(x, y, polyorder) tries 8 different simple data transformations on the data x,y, fits the transformed data to a polynomial of order 'polyorder', displays results graphically in 3 x 3 array of small plots and returns the R^2 values in a vector. In the example below, for polyorder=1, it is

the 5th one that is best, namely x vs ln(y). An example is shown on the next page.



Fitting Single Gaussian and Lorentzian peaks

A simple user-defined Matlab/Octave function that fits a single Gaussian function to an x,y signal is gaussfit.m, which implements the x vs ln(y) quadratic fitting method described above. It takes the form **`[Height,Position,Width]=gaussfit(x,y).`**

For example,

```
>> x=50:150;
>> y=100.*gaussian(x,100,100)+10.*randn(size(x));
>> [Height,Position,Width]=gaussfit(x,y)
```

returns [Height,Position,Width] clustered around 100,100,100. A similar function for Lorentzian peaks is lorentzfit.m, which takes the form

`[Height,Position,Width]=lorentzfit(x,y).`

An expanded variant of the gaussfit.m function is bootgaussfit.m, which does the same thing but also optionally plots the data and the fit and computes estimates of the random error in the height, width, and position of the fitted Gaussian function by the bootstrap sampling method. For example:

```
>> x=50:150;
>> y=100.*gaussian(x,100,100)+10.*randn(size(x));
>> [Height, Position, Width, BootResults]=bootgaussfit(x,y,1);
```

This does the same thing as the previous example but also displays error estimates in a table (next page) and returns the 3x5 matrix `BootResults`. Type "help bootgaussfit" for help.

```
                Height      Position      Width
Bootstrap Mean: 100.84      101.325       98.341
Bootstrap STD:  1.3458      0.63091       2.0686
Bootstrap IQR:  1.7692      0.86874       2.9735
Percent RSD:    1.3346      0.62266       2.1035
Percent IQR:    1.7543      0.85737       3.0237
```

It is important that the noisy signal not be smoothed if the bootstrap error predictions are to be accurate. Smoothing causes the bootstrap method to seriously underestimate the precision of the results.

The gaussfit.m and lorentzfit.m functions are simple and easy, but they do not work well with very noisy peaks or for multiple overlapping peaks. As a demonstration, OverlappingPeaks.m is a script that shows how to use gaussfit.m to measure two overlapping partially Gaussian peaks. It requires careful selection of the optimum data regions around the top of each peak. Try changing the relative position and height of the second peak or adding noise (line 3) and see how it affects the accuracy. This function needs the gaussian.m, gaussfit.m, and peakfit.m functions in the Matlab search path. The script also performs measurements by the iterative method (page 199) using peakfit.m, which is more accurate but takes about times longer to compute.

My Matlab-only functions iSignal.m (page 376) and ipf.m (page 416), whose principal functions are fitting *peaks*, also have a function for fitting *polynomials* of any order (**Shift-o**).

Recent versions of Matlab have a convenient tool for interactive manually-controlled (rather than programmed) polynomial curve fitting in the Figure window. If you are reading this online, click for a video example: (external link to YouTube).

The *Matlab Statistics Toolbox* includes two types of bootstrap functions, "bootstrp" and "jackknife". To open the reference page in Matlab's help browser, type "doc bootstrp" or "doc jackknife".

Curve fitting B: Multicomponent Spectroscopy

The spectroscopic analysis of mixtures, when the spectrum of the mixture is the simple sum of the spectra of known components that may overlap but are not identical, can be performed using special calibration methods based on a type of linear least-squares called *multiple linear regression*. This method is widely used in modern spectrometers such as diode-array, Fourier transform, and digitally controlled scanning spectrometers (because perfect wavelength reproducibility is a key requirement). To understand the required math, it is helpful to do a little basic matrix algebra (a.k.a., linear algebra), which is just a shorthand notation for dealing with signals expressed as equations with one term for each point. Because that area of math may not be a part of everyone's math background, I will do some elementary matrix math derivations in this section.

Definitions:
n = number of distinct chemical components in the mixture
s = number of samples
s1, s2 = sample 1, sample 2, etc.
c = molar concentration
c1, c2 = component 1, component 2, etc.
w = number of wavelengths at which signal is measured
w1, w2 = wavelength 1, wavelength 2, etc.
Ɛ = analytical sensitivity (slope of a plot of A vs c)
A = analytical signal
$\mathbf{M}^T$ = matrix transpose of matrix **M** (rows and columns switched).
$\mathbf{M}^{-1}$ = matrix inverse of matrix **M**.

Assumptions:
The analytical signal, A (such as absorbance in absorption spectroscopy, fluorescence intensity in fluorescence spectroscopy, and reflectance in reflectance spectroscopy) is directly proportional to concentration, c. The proportionality constant, which is the slope of a plot of A vs c, is Ɛ.

$$A = \varepsilon c$$

The total signal is the sum of the signals for each component in a mixture:

$A_{total} = A_{c1} + A_{c2} + \ldots$ for all *n* components.

Classical Least-squares (CLS) multivariate calibration

This method is commonly applied to the quantitative spectroscopic analysis of a mixture of components, provided that (a) you can measure the spectra of the individual components and (b) the total signal of the mixture is simply the sum of the signals for each component in the mixture. Measurement of the spectra of known concentrations of the separate components allows their analytical sensitivity Ɛ at each wavelength to be determined. Then it follows that:

$A_{w1}=\varepsilon_{c1,w1}\, c_{c1} + \varepsilon_{c2,w1}\, c_{c2} + \varepsilon_{c3,w1}\, \mathbf{c}_{c3} + \ldots$ for all *n* components.
$A_{w2}=\varepsilon_{c1,w2}\, \mathbf{c}_{c1} + \varepsilon_{c2,w2}\, \mathbf{c}_{c2} + \varepsilon_{c3,w2}\, \mathbf{c}_{c3} + \ldots$
and so on for all wavelengths - w3, w4, etc. It is impractical to write out all these individual terms,

especially because there may be *hundreds* of wavelengths in modern array-detector spectrometers. Moreover, despite the mass of raw data, these are just nothing more than linear equations. The calculations required here are rather simple and certainly very easy for a computer to do. So, *you really need a correspondingly simple notation* that is more compact. To do this, it is conventional to use **bold-face letters** to represent a *vector* (like a column or row of numbers in a spreadsheet) or a *matrix* (like a *block* of numbers in a spreadsheet). For example, **A** could represent the list of absorbances at each wavelength in an absorption spectrum. So, this big set of linear equations above can be written:

$$\mathbf{A} = \boldsymbol{\varepsilon}\mathbf{C}$$

where **A** is the *w*-length vector of measured signals (i.e., the signal spectrum) of the mixture, $\boldsymbol{\varepsilon}$ is the $n \times w$ rectangular matrix of the known $\boldsymbol{\varepsilon}$-values for each of the *n* components at each of the *w* wavelengths, and **C** is the *n*-length vector of concentrations of all the components. $\boldsymbol{\varepsilon}\mathbf{C}$ means that $\boldsymbol{\varepsilon}$ "pre-multiplies" **C**. *Each column of matrix* $\boldsymbol{\varepsilon}$ *is multiplied point-by-point* by the vector **C.**

If you have a sample solution containing unknown amounts of those n components, you measure its spectrum **A** and seek to calculate the concentration vector of concentrations **C**. To solve the above matrix equation for **C**, the number of wavelengths *w* must be equal to or greater than the number of components *n*. If $w = n$, then you have a system of *n* equations in *n* unknowns which can be solved by pre-multiplying both sides of the equation by $\boldsymbol{\varepsilon}^{-1}$, the matrix inverse of $\boldsymbol{\varepsilon}$, and using the property that any matrix times its inverse is unity:

$$\mathbf{C} = \boldsymbol{\varepsilon}^{-1}\mathbf{A}$$

Because real experimental spectra are subject to random noise (e.g., photon noise and detector noise), the solution will be more precise if the signals at a larger number of wavelengths are used, i.e., if $w > n$. This is easily done with no increase in labor by using a modern array-detector spectrophotometer. But then the equation cannot be solved by simple matrix inversion, because the $\boldsymbol{\varepsilon}$ matrix is a $w \times n$ matrix and *a matrix inverse exists only for square matrices*. However, a solution can be obtained in this case by pre-multiplying both sides of the equation by the expression $(\boldsymbol{\varepsilon}^{T}\boldsymbol{\varepsilon})^{-1}\boldsymbol{\varepsilon}^{T}$, assuming that $\boldsymbol{\varepsilon}$ is not zero:

$$(\boldsymbol{\varepsilon}^{T}\boldsymbol{\varepsilon})^{-1}\boldsymbol{\varepsilon}^{T}\mathbf{A} = (\boldsymbol{\varepsilon}^{T}\boldsymbol{\varepsilon})^{-1}\boldsymbol{\varepsilon}^{T}\boldsymbol{\varepsilon}\mathbf{C} = (\boldsymbol{\varepsilon}^{T}\boldsymbol{\varepsilon})^{-1}(\boldsymbol{\varepsilon}^{T}\boldsymbol{\varepsilon})\mathbf{C}$$

where $\boldsymbol{\varepsilon}^{T}$ is the *transpose* of $\boldsymbol{\varepsilon}$ (rows and columns switched). But the quantity $(\boldsymbol{\varepsilon}^{T}\boldsymbol{\varepsilon})^{-1}(\boldsymbol{\varepsilon}^{T}\boldsymbol{\varepsilon})$ is a matrix multiplied by its inverse and is therefore unity. Thus, you can simplify the result to:

$$\mathbf{C} = (\boldsymbol{\varepsilon}^{T}\boldsymbol{\varepsilon})^{-1}\boldsymbol{\varepsilon}^{T}\mathbf{A}$$

This is often called the "normal equation". In this expression, $\boldsymbol{\varepsilon}^{T}\boldsymbol{\varepsilon}$ is a square matrix of order *n*, the number of components. In most practical spectroscopic applications, the number of chemical components, *n*, is relatively small, perhaps only 2 to 5. Vector **A** is of length *w*, the number of wavelengths. That can be quite large, perhaps several hundred in a diode-array spectrometer. The more wavelengths are used, the more effectively the random noise will be averaged out (although it will not help to use wavelengths in spectral regions where none of the components produce analytical signals). The determination of the optimum wavelength region must usually be determined empirically. All components that contribute to the spectrum must be accounted for and included in the $\boldsymbol{\varepsilon}$ matrix. This computational method is called "Classical Least Squares" or simply "CLS", so-called because it is a rather old method introduced long before computers made it practical.

Three extensions of the CLS method are commonly made:

(a) If you have *s* multiple unknown samples to measure, you can compute them all at once using the same notation as above, by combining their spectra into a $w \times s$ matrix **A**, which will result in an $n \times s$ matrix **C**. (This capability is used in the Appendix: "Spectroscopy and chromatography combined: time-resolved Classical Least-squares" on page 362).

(b) To account for the baseline shift caused by drift, background, and light scattering, a column of 1s is added to the **Ɛ** matrix. This has the effect of introducing into the solution an additional component with a flat spectrum. This is referred to as “background correction”.

(c) To account for the fact that the precision of measurement may vary with wavelength, it is common to perform a *weighted* least-squares solution that de-emphasizes wavelength regions where precision is poor:

$$\mathbf{C} = (\mathbf{Ɛ}^T \mathbf{Ɛ}^{-1}\mathbf{Ɛ})^{-1} \mathbf{Ɛ}^T \mathbf{V}^{-1} \mathbf{A}$$

where **V** is a $w \times w$ diagonal matrix of variances at each wavelength. In absorption spectroscopy, where the precision of measurement is poor in spectral regions where the absorbance is very high (and the light level and signal-to-noise ratio therefore low), it is common to use the transmittance T or its square T^2 as weighting factors.

The CLS method is in principle applicable to any number of overlapping components. Its accuracy is limited by how accurately the spectra of the individual components are known, the amount of noise in the signal, the extent of overlap of the spectra, and the linearity of the analytical curves of each component (the extent to which the signal amplitudes are proportional to concentration). In practice, the method does not work well with old-style instruments with mechanical wavelength control, because of insufficient wavelength reproducibility. This is because many of the data points end up being on the *sides* of spectral bands, where *even small failures in the reproducibility of wavelength settings between measurements would result in large intensity changes*. However, it is especially well suited to diode-array and Fourier transform instruments, which have extremely good wavelength reproducibility. The method also depends on the linearity of analytical signal with respect to concentration. In absorption spectrophotometry specifically, there are well-known instrumental deviations from analytical curve linearity that set a limit to the performance to this method, but that problem can be avoided by applying iterative curve fitting (page 199) and Fourier convolution (page 109) to the *transmission* spectra, an idea that will be developed later, on page 276.

Inverse Least-squares (ILS) calibration

ILS is a method that can be used to measure the concentration of an analyte in samples in which the spectrum of the analyte in the sample is *not* known beforehand. Whereas the classical least-squares (CLS) method models the signal at each wavelength as the sum of the concentrations of the analyte times the analytical sensitivity, the inverse least-squares (ILS) methods use the inverse approach and models the analyte concentration c in each sample as the sum of the signals A at each wavelength times calibration coefficients m that express how the concentration of that component is related to the signal at each wavelength:

$$c_{s1} = m_{w1}A_{s1,w1} + m_{w2}A_{s1,w2} + m_{w3}A_{s1,w3} + \ldots \text{ for all } w \text{ wavelengths.}$$
$$c_{s2} = m_{w1}A_{s2,w1} + m_{w2}A_{s2,w2} + m_{w3}A_{s2,w3} + \ldots$$

and so on for all s samples. In matrix form

$$\mathbf{C} = \mathbf{AM}$$

where **C** is the s-length vector of concentrations of the analyte in the s samples, **A** is the $w \times s$ matrix of measured signals at the w wavelengths in the s samples, and **M** is the w-length vector of calibration coefficients.

Now, suppose that you have a s-length set of "standard samples that are typical of the type of sample that you wish to be able to measure and which contain a range of analyte concentrations that span the range of concentrations expected to be found in other samples of that type. This will serve as the *calibration set.* You measure the spectrum of each of the samples in this calibration set and put these data into a $w \times s$ matrix of measured signals **A**. You then measure the analyte concentrations in each of the samples *by some reliable and independent analytical method* and put those data into a vector of concentrations **C.** Together these data allow you to calculate the calibration vector **M** by solving the above equation. If the number of samples in the calibration set is greater than the number of wavelengths, the least-squares solution is:

$$\mathbf{M} = (\mathbf{A}^{\mathrm{T}}\mathbf{A})^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{C}$$

(Note that $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ is a square matrix of size w, the number of wavelengths, which must be less than s). This calibration vector can be used to compute the analyte concentrations of other samples which are similar to those in the calibration set, from the measured spectra of the samples:

$$\mathbf{C} = \mathbf{AM}$$

Clearly, this will work well only if the analytical samples are similar in composition to the calibration set. The advantage of this method is that the spectrum of an unknown sample can be measured much more quickly and cheaply than the more laborious standard reference methods that are used to measure the calibration set, but if the unknowns are similar enough to the calibration set, the concentrations calculated by the above equation will be accurate enough for many purposes.

Computer software for multiwavelength spectroscopy

Spreadsheets

Modern spreadsheets have basic matrix manipulation capabilities that can be used for multi-component calibration, for example Excel and OpenOffice Calc. The spreadsheets RegressionDemo.xls and RegressionDemo.ods (for Excel and Calc, respectively) demonstrate the classical least-squares procedure for a simulated spectrum of a 5-component mixture measured at 100 wavelengths. A screenshot is shown below. The matrix expression described above, which solves for the concentration of the components on the unknown mixture

$$\mathbf{C} = (\boldsymbol{\varepsilon}^{\mathrm{T}}\boldsymbol{\varepsilon})^{-1}\boldsymbol{\varepsilon}^{\mathrm{T}}\mathbf{A}$$

is performed in these spreadsheets by the TRANSPOSE (matrix transpose), MMULT (matrix multiplication), and MINVERSE (matrix inverse) array functions, laid out step-by-step in rows 123 to 158 of this spreadsheet. Alternatively, all these array operations may be combined into one big cell equation, although admittedly this is harder to read than separating the steps as I have done.

C = MMULT (MMULT (MINVERSE (MMULT (TRANSPOSE (**E**);**E**)); TRANSPOSE (**E**));**A**)

where **C** is the vector of the 5 concentrations of all the components in the mixture, **E** is the 5 × 100 rectangular matrix of the known sensitivities (e.g., absorptivities) for each of the 5 components at each of the 100 wavelengths, and **A** is the vector of measured signals at each of the 100 wavelengths (i.e., the signal spectrum) of the unknown mixture. (Note: spreadsheet array functions like this must be entered by typing **Ctrl-Shift-Enter**, not just **Enter** as usual. See "Guidelines and examples of array formulas".

Multiwavelength Spectrophotometric Analysis by Classical Least Squares

Simulated mixture with 5 components (Gaussian bands)

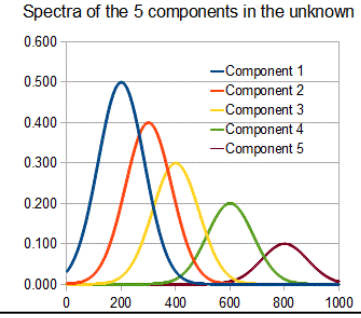
	Component 1	Component 2	Component 3	Component 4	Component 5
Amplitude	0.500	0.400	0.300	0.200	0.100
Position	200	300	400	600	800
Width	200	200	200	200	200

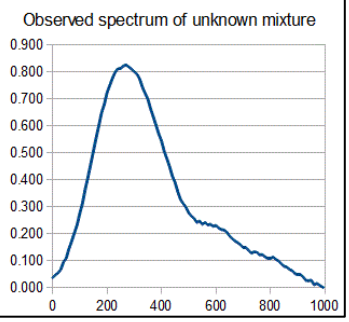
Noise	0.01

Measurements based on noisy mixture spectrum

	Component 1	Component 2	Component 3	Component 4	Component 5
	0.4998109295	0.4012296537	0.2973088641	0.2019537902	0.0973494797
% error	-0.04%	0.31%	-0.90%	0.98%	-2.65%

See rows 123 to 158 for the step-by-step calculation procedure.





.

OpenOffice Calc spreadsheet demonstrating the CLS procedure for the measurement of a 5-component unknown mixture at 100 wavelengths.

Alternatively, you can skip over all the details above and use the built-in **LINEST** function, in both Excel or OpenOffice Calc, which performs this type of calculation in a single function statement. This is illustrated in RegressionTemplate.xls, in cell Q23. (A slight modification of the function syntax, shown in cell Q32, performs a *baseline-corrected* calculation). A significant advantage of the LINEST function is that it can automatically compute the standard errors of the coefficients and the R^2 value in the same operation. Other methods would require some extra work. LINEST is also an array function that must also be entered by typing **Ctrl-Shift-Enter**, not just **Enter**. Note that this is the *same* LINEST function that was previously used for polynomial least-squares (page 161). The difference is that in polynomial least-squares, the multiple columns of x values are *computed*, for example by taking the powers (squares, cubes, etc.) of the x's, whereas, in the multicomponent CLS method, the multiple columns of x values are *experimental* spectra of the different standard solutions. The *math* is the same, but the *origin of the x data* is very different.

A template for performing a 5-component analysis on your own data, with step-by-step instructions, is available as RegressionTemplate.xls and RegressionTemplate.ods (Graphic on the next page) from the demo above). Paste your own data into columns B - G. You must adjust the formulas if your number of data points or of components is different from this example. The easiest way to add more wavelengths is to select an entire row anywhere between row 40 and the end, right-click on the row number on the left and select **Insert**. That will insert a new blank row and will automatically adjust all the cell formulas (including the LINEST function) and the graph to include the new row. Repeat that as many

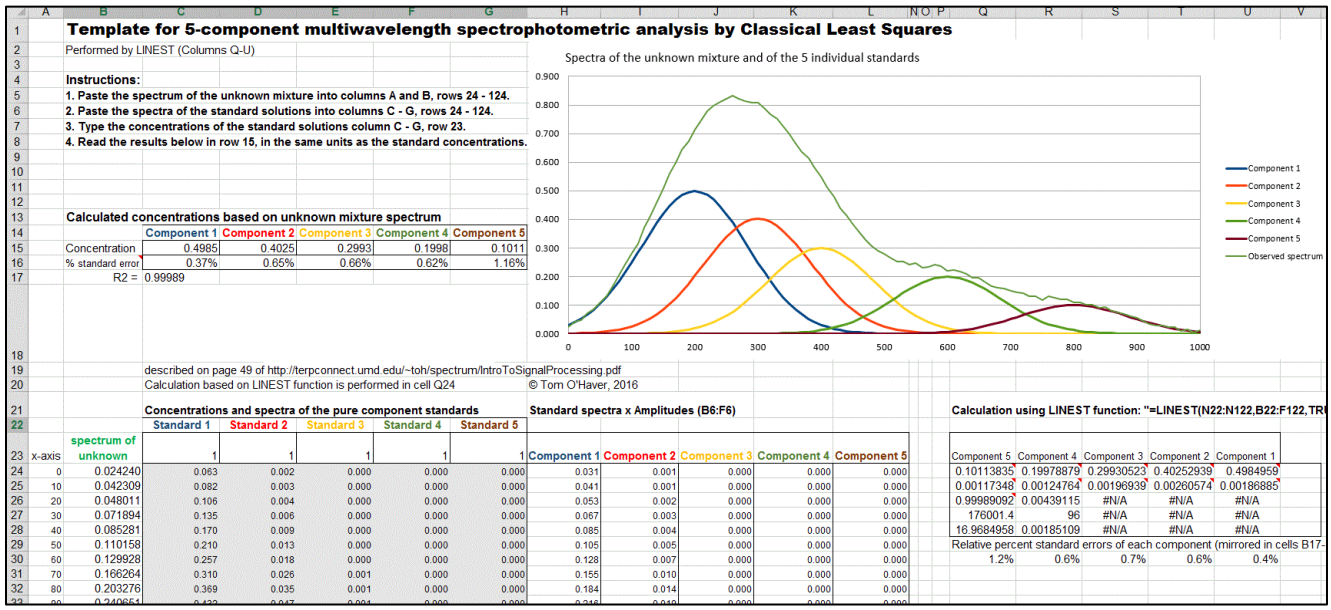
times as needed. Finally, select the entire row just before the insertion (that is, the last non-blank row) and drag-copy in down to fill in all the new blank rows. Changing the number of components is more difficult: it involves inserting or deleting columns between C and G and between H and L and adjusting the formulas in rows 15 and 16 and also in Q29-U29.

Spreadsheets of this type, though easy to use once constructed, must be carefully modified for different applications that have different numbers of components or different numbers of wavelengths, which is inconvenient and can be error prone. However, it is possible to construct these spreadsheets in such a way that they *automatically* adjust to any number of components or wavelengths. This is done by using two new functions:

(a) the COUNT function in cells B18 and F18, which counts the number of wavelengths in column A and the number of components in row Q22-U22, respectively, and

(b) the INDIRECT function (see page 353) in cell Q23 and in rows 12 and 13, which allows the address of a cell or range of cells to be *calculated within the spreadsheet* (based on the number of wavelengths and components just counted) rather than using a fixed address range.

This technique is used in RegressionTemplate2.xls and in two examples showing the *same template* with data entered for different numbers of wavelengths and for mixtures of 5 components at 100 wavelengths (RegressionTemplate2Example.xls) and for 2 components at 59 wavelengths (RegressionTemplate3Example.xls). If you inspect the LINEST functions in cell Q23, you will see that it is the same in both of those two example templates, even though the number of wavelengths and the number of components is different. You will still have to adjust the graph, however, to cover the desired x-axis range. See page 353.



Excel template applied to the measurement of a 5-component unknown mixture at 100 wavelengths.

Matlab/Octave and Python

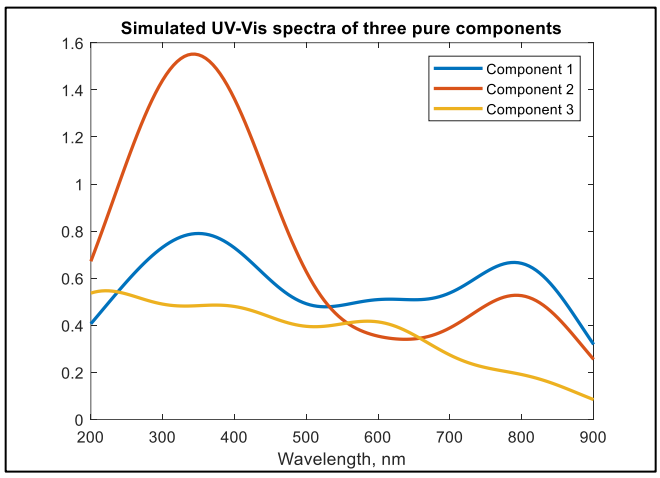
Matlab/Octave and Python are really the natural languages for multicomponent analysis because they handle all types of matrix math so easily, compactly, and quickly, and they readily adapt to any number

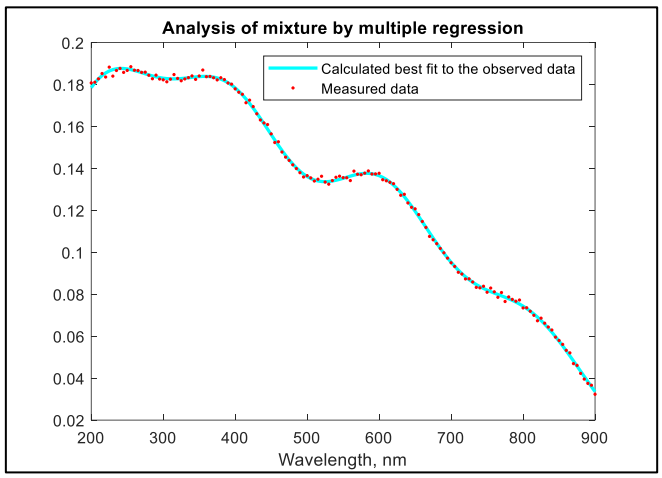
of wavelengths or number of components without any special tricks. In Matlab/Octave, the notation is very compact: the transpose of matrix A is A', the inverse of A is inv(A), and matrix multiplication is designated by an asterisk (*). Thus, the solution to the Classical Least-Squares method above is written in Matlab/Octave notation as

```
C = inv(E'*E)*E'*A
```

where E is the rectangular matrix of sensitivities at each wavelength for each component and A is the observed spectrum of the mixture. Note that the Matlab/Octave notation is not only shorter than the spreadsheet notation, but also closer to the traditional mathematical notation. Even more compactly, you can write C = A/E, using the Matlab forward slash or "right divide" operator, which yields the same results but is more accurate with respect to the numerical precision of the computer (which in most applications is negligible compared to the noise in the signal; see page 340).

The script RegressionDemo3.m (for Matlab or Octave) demonstrates the classical least-squares





procedure for a simulated absorption spectrum of a 3-component mixture at 141 wavelengths in the 200 to 900 nm range, in which each of the 3 components has a distinct but overlapping spectrum. In the figure on the left, the colored bands represent the three components in the mixture, whose spectra have been measured separately but whose concentrations in the mixture are unknown. In the figure on the right, the red dots represent the observed spectrum of the mixture at about a 10-fold lower concentration than on the left (hence the visible noise). The blue line represents the "best fit" to the observed spectrum calculated by the program. Most of this script is just signal generation and plotting. The actual least-squares regression is performed in line 48:

```
MeasuredAmp = ObservedSpectrum'/spectra;
```

Here, the matrix "spectra" contains the spectra of each of the components in each of its rows. "ObservedSprectum" is the measured spectrum of the unknown mixture, including random noise. In this example, the concentrations of the three components are measured to an accuracy of about 1% relative, limited by the noise in the observed spectrum. In practice, the accuracy is more likely to be limited by chemical and physical factors, such as temperature, pH, volumetric errors, etc.

The CLS technique is fast enough (especially in Matlab) that it can be applied in real time to 2D chromatography with array detectors, where a complete spectrum is acquired multiple times per second

over the entire chromatogram. See “Spectroscopy and chromatography combined: time-resolved Classical Least Squares” on page 363.

Extensions:

(a) The extension to **multiple unknown samples**, each with its own "ObservedSpectrum" is straightforward in Matlab/Octave. If you have "s" samples, just assemble their observed spectra onto a *matrix* with "s" rows and "w" columns ("w" is the number of wavelengths), then use the same formulation as before:

```
MeasuredAmp = ObservedSpectrum*A'*inv(A*A')
```

The resulting "MeasuredAmp" will be an "s" × "n" *matrix* rather than an n-length vector ("n" is the number of measured components). This is a great example of the convenience of the vector/matrix nature of this language. (RegressionDemoMultipleSamples.m demonstrates this).

(b) The extension to **"background correction"** is easily accomplished in Matlab/Octave by adding a column of 1s to the **A** matrix containing the absorption spectrum of each of the components:

```
background=ones(size(ObservedSpectrum));
A=[background A1 A2 A3];
```

where A1, A2, A3... are the absorption spectra vectors of the individual components.

(c) Performing a **Transmission-weighted regression** is also readily performed:

```
MeasuredAmp=([T T] .* A)\(ObservedSpectrum .* T);
```

where T is the transmission spectrum vector. Here, the matrix division backslash "\" is used as a short-cut to the classical least-squares matrix solution (c.f. http://www.mathworks.com/help/techdoc/ref/mldivide.html).

The cls.m function: Ordinarily, the calibration matrix **M** is assembled from the experimentally measured signals (e.g. spectra) of the individual components of the mixture, but it is also possible to fit a computer-generated model of basic peak shapes (e.g. Gaussians, Lorentzians, etc.) to a signal to determine if that signal can be represented as the weighted sum of overlapping basic peak shapes. The function cls.m computes such a model consisting of the sum of any number of peaks of *known shape, width, and position*, but of *unknown height*, and fit it to noisy x,y data sets. The syntax is

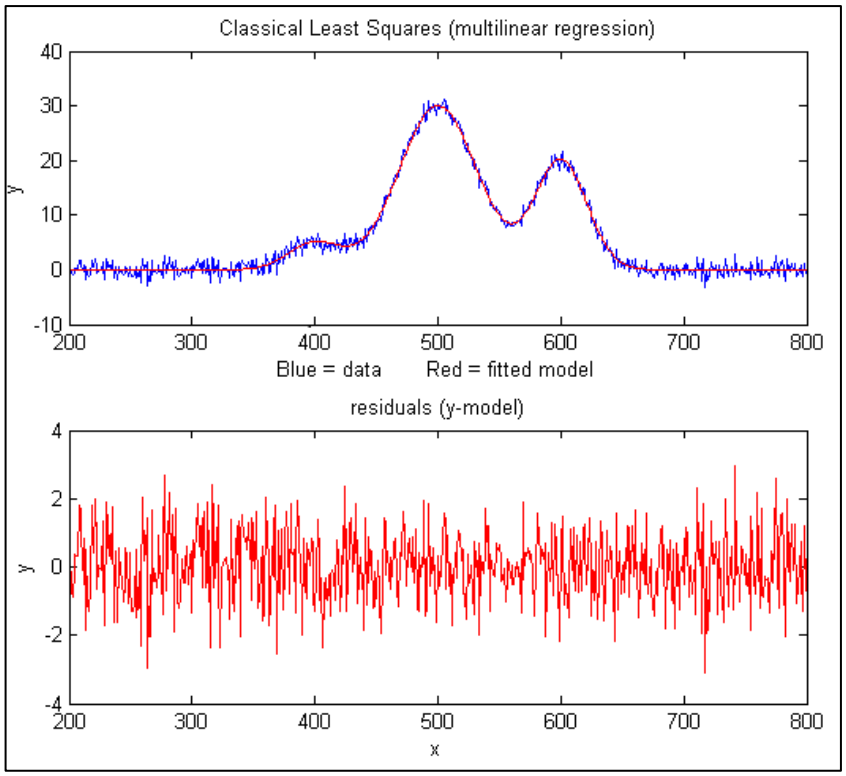
```
heights=cls(x, y, NumPeaks, PeakShape, Positions, Widths, extra)
```

where x and y are the vectors of measured data (e.g. x might be wavelength and y might be the absorbance at each wavelength), 'NumPeaks' is the number of peaks, 'PeakShape' is the peak shape number (1=Gaussian, 2=Lorentzian, 3=logistic, 4=Pearson, 5=exponentially broadened Gaussian; 6=equal-width Gaussians; 7=Equal-width Lorentzians; 8=exponentially broadened equal-width Gaussian, 9=exponential pulse, 10=sigmoid, 11=Fixed-width Gaussian, 12=Fixed-width Lorentzian; 13=Gaussian/Lorentzian blend; 14=BiGaussian, 15=BiLorentzian), 'Positions' is the vector of peak positions on the x axis (one entry per peak), 'Widths' is the vector of peak widths in x units (one entry

per peak), and 'extra' is the additional shape parameter required by the exponentially broadened, Pearson, Gaussian/Lorentzian blend, BiGaussian and BiLorentzian shapes. Cls.m returns a vector of measured peak heights for each peak.

The cls2.m function is similar to cls.m, except that it also measures the baseline (assumed to be flat), using the extension to "background correction" described above, and returns a vector containing the background B and measured peak heights H for each peak 1,2,3, e.g., [B H1 H2 H3...].

The demonstration script **clsdemo.m** (on the right) creates some noisy model data, fits it with cls.m, computes the accuracy of the measured heights, then repeats the calculation *by iterative non-linear least-squares peak fitting* ("INLS", covered on page 199) using my Matlab/ Octave function peakfit.m, making use of the known peak positions and widths only as *starting guesses* ("start"). You can see that CLS is 30 times faster and slightly more accurate, especially if the peaks are highly overlapped. (This script requires the functions cls.m, modelpeaks.m, and peakfit.m in the Matlab/Octave search path). A typical result is:



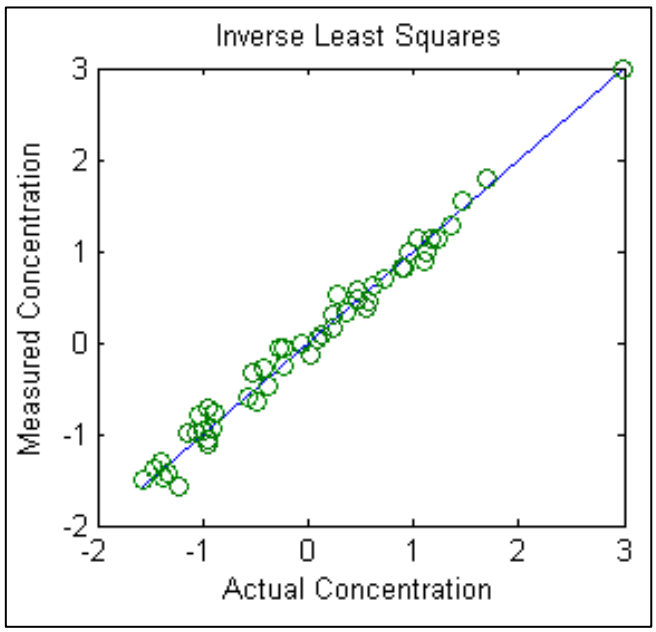

```
Figure window(1): Classical Least-squares (multilinear regression)
Elapsed time is 0.012 seconds.
Average peak height accuracy = 0.9145%

Figure window(2): Iterative non-linear peak fitting with peakfit.m
Elapsed time is 0.39 seconds.
Average peak height accuracy = 1.6215%
```

On the other hand, INLS can be better than CLS if there are *unsuspected shifts* in peak position and/or peak width between calibration and measurement (for example caused by drifting spectrometer calibration or by changing temperature, pressure, or solution variables), because INLS can track and compensate for changes in peak position and width. You can test this by changing the variable "PeakShift" (line 16) to a non-zero value in clsdemo.m.

Weighted linear regression: My Matlab/Octave function "tfit.m" simulates the measurement of the absorption spectrum of a mixture of three components by *weighted and unweighted* linear regression, demonstrates the effect of the amount of noise in the signal, the extent of overlap of the spectra, and the linearity of the analytical curves of each component. This function also compares the results to a more advanced method described later (line 66) that applies curve fitting to the *transmission* spectra rather than to the *absorbance* spectra (page 276), which improves the linearity and accuracy of the method.

The **Inverse Least-squares (ILS)** technique is demonstrated in Matlab by this script and the graph in the right. The mathematics, described above on page 190, is similar to the Classical Least-squares method and can be done by any of the Matlab/ Octave, Python, or spreadsheet methods described in this section. This example is based on a real data set derived from the near-infrared (NIR) reflectance spectroscopy of agricultural wheat paste samples analyzed for protein content. There are 50 calibration samples measured at 6 infrared wavelengths. These calibration samples had already been analyzed for protein content by a reliable (but laborious and time-consuming) reference method. The purpose of this calibration is to establish whether the results of the quicker and easier near-infrared reflectance method correlate to the protein contents determined by the more costly reference method. These results indicate that it does, at least for this set of 50 wheat samples, and therefore it is likely that near-infrared spectroscopy should do a pretty good job of estimating the protein content of similar unknown samples. *The key is that the unknown samples must be similar to the calibration samples* (except of course for the protein content). This is a very common situation in industrial and agricultural *quality control*, where many samples of products or crops of a similar predictable type must be tested quickly and cheaply, sometimes in the field using simple portable instruments. It may take a good bit of time and effort to calibrate the instrument initially, but once that is done, it can be applied quickly and cheaply to the type of sample for which it was calibrated. Near-infrared (NIR) reflectance spectroscopy, in conjunction with inverse least-squares or related more sophisticated mathematical methods, is very widely used in industry, agriculture, food science, and environmental applications.



Incidentally, the above method, specifically near infrared methods for agricultural samples, was pioneered in the 1950s and 1960s by Dr. Karl Norris at the Beltsville Agricultural Research Center in Maryland, just up the road from my home institution in College Park. (Several of my graduate students went to work there after finishing their degrees). The early history of near infrared methods is reviewed by Dr. Norris himself in https://journals.sagepub.com/doi/10.1255/jnirs.941.

Note: If you are reading this online, you can right-click on any of the m-file links above and select **Save Link As...** to download them to your computer for use within Matlab.

Classical Least Squares in Python

The equivalent expression in Python for the "Normal Equation" is

```
C = inv(E.T.dot(E)).dot(E.T).dot(A)
```

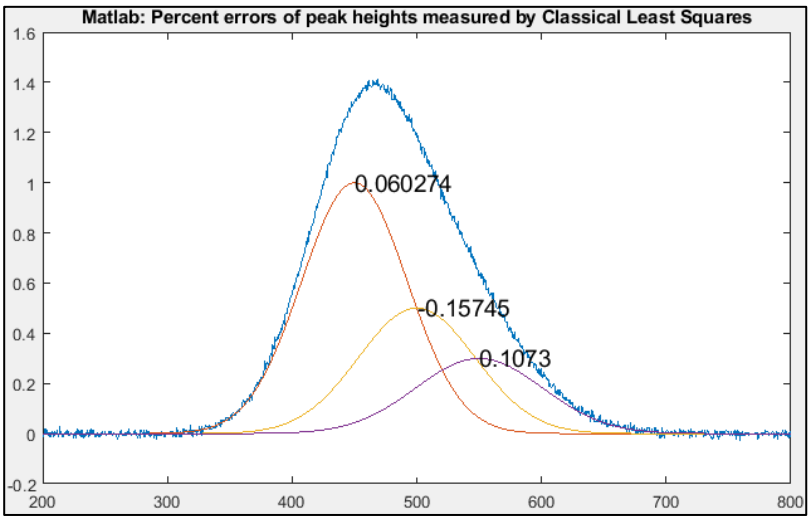
where "inv()" means the matrix inverse, the expression "E.T" means the *transpose* of matrix E, and ".dot" means the *dot product*. This is arguably more explicit than the more compact Matlab version:

```
C = inv(E'*E)*E'*A;
```

Just remember that before doing this in Python you must do the following imports:

```
numpy as np
from numpy.linalg import inv
```

The pair of matched scripts, NormalEquationDemo.py and NormalEquationDemo.m compare these aspects in Python and in Matlab. Both scripts use *the same sequence of operations and the same variable names* to create and plot a noisy simulated signal vector ***A*** that is the sum of three strongly overlapping peaks of known shape (G1, G2, and G3) but unknown amplitude, and then measure the amplitudes ***C*** in the noisy signal by Classical Least Squares. The percent error between the true and measured peak heights are printed next to each peak. The code similaries are striking, the results are the same, and the execution times are similar for the Matlab and Python version. The actual computation of concentrations C by CLS takes only one line in both languages: `C = inv(E.T.dot(E)).dot(E.T).dot(A)` in Python and `C = inv(E'*E)*E'*A;` in Matlab.



Curve fitting C: Non-linear Iterative Curve Fitting

Least-squares curve fitting, described in "Curve Fitting A" on page 161, is simple and fast, but it is limited to situations where the dependent variable can be modeled as a polynomial with linear coefficients.

The most general way of fitting any model to a set of data is the iterative method, which is a "trial and error" procedure in which the parameters of the model are adjusted in a systematic fashion until the equation fits the data as close as required. This sounds like a brute-force approach, and it is. In fact, in the days before computers, this method was not widely used. But its great generality, coupled with advances in computer speed and algorithm efficiency, means that iterative methods are more widely used now than ever before. There are extensive applications of this technique in the areas of biomedical research and pharmacology, environmental Science, physics and material Science, engineering and drug development, neuroscience and neuroimaging, and agronomy and biomedicine. But here I will focus on the application to peak fitting: that is, determining whether it is possible to model a complex signal shape as the sum of several simple components whose height, position, width, and shape are initially unknown.

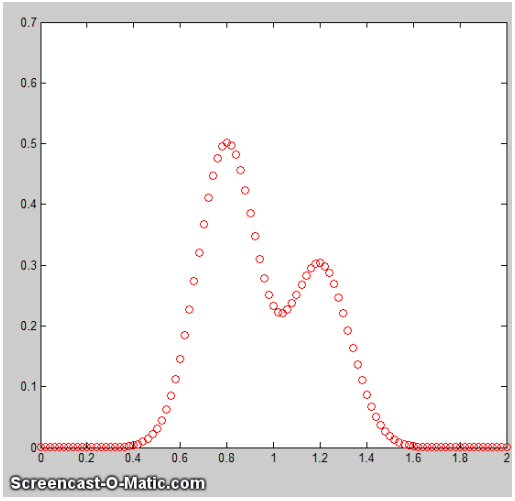
An iterative method proceeds in the following general way:

(1) Select a model for the data, for example, in the example below, the sum of two Gaussian peaks.

(2) Make first guesses of all the adjustable non-linear parameters (i.e. for application to peak fitting, these would be the position and width of the peaks that are presumed to be in the data).

(3) A computer program computes the model and compares it to the data set, calculating a fitting error.

(4) The program systematically changes the parameters and loops back around to the previous step and repeats until the required fitting accuracy is achieved.



One popular technique for doing this is the Nelder-Mead Modified Simplex. This is a way of organizing and optimizing the changes in parameters (step 4, above) to shorten the time required to fit the function to the required degree of accuracy. It might sound complicated, but with contemporary personal computers, the entire process *typically takes only a fraction of a second* to a few seconds, depending on the complexity of the model and the number of independently adjustable parameters in the model. The animation here (Matlab script) shows the working of the iterative process for a 2-peak unconstrained Gaussian fit to a small set of x,y data. This model has *four nonlinear variables* (the positions and width of the two Gaussians, which are determined by iteration) and *two linear variables* (the peak heights of the two Gaussians, which are determined directly by regression for each trial iteration). To allow the process to be observed in action, this animation is *slowed down artificially* by (a) plotting step-by-step, (b) making a bad initial guess, and (c) adding a "`pause()`" statement inside the iteration loop. Without

all that slowing down, *the entire process normally takes only about 0.05 seconds on a standard PC or laptop*, depending mainly on the number of nonlinear variables that must be iterated. This even works if the peaks are so overlapped that they bend into a single peak, as shown by the script Demofitgauss2AnimatedBlended.m, and the corresponding animation, but it takes more iterations and it is more sensitive to any random noise in the data (added in line 13).

Note: the script that created this animation, Demofitgauss2animated.m, requires the gaussian.m, fitgauss2animated.m, and fminsearch.m functions to be in the Matlab/Octave path. As an instructional aid, a modified version of this script, DemofitgaussanimatedError.m, plots the fitting error vs iteration number, showing that a poor initial guess (start value) can be detrimental, requiring many more iterations (and more time) to find a good fit, which is not very efficient from the point of view of a human observer like yourself who can visually estimate the peak parameters much more intelligently. A good guess requires fewer iterations. One way to get very good start values for peak-type signals is to precede the curve fitting with a fast, single-pass peak detector algorithm to determine the number of peaks and their approximate positions and widths (as will be covered in a later section), but that will only work when the peaks to be fit have distinct maxima and are not blended into a single peak. As you might expect, a 3-peak fit, with *six* non-linear parameters to optimize, will require more iterations.

The main difficulty of the iterative methods is that they sometimes fail to converge at an overall optimum solution in difficult cases, especially if given a bad initial starting guess. To understand how that could happen, think of a physical analogy: you deposit a blind robot on a hilly landscape and program it to walk to the highest peak, given only a long stick to probe the immediate surroundings to test if the ground is higher or lower on each side. If it is higher, it walks there and probes around that area. It stops when all the areas around it are lower than where it is. This works well if there is a single smooth hill. But what if there are multiple small hills, or if the ground is rocky and uneven? The robot is very likely to stop at one of the smaller hills, not seeing that there might be an even higher hill nearby. To walk the *entire* landscape in a grid to probe for *all* the hills would take a lot of time.

The standard approach to handle this is to restart the process with random variations of the first guesses, repeat several times, and take the one with the lowest fitting error. That entire process is automated in the peak fitting functions described on page 396. If that is not enough, you can make our own starting guesses. With all this, it is no surprise that iterative curve fitting takes longer than linear regression - with typical personal computers, an iterative fit might take fractions of a second where a regression would take fractions of a millisecond. Still, that's fast enough for many purposes.

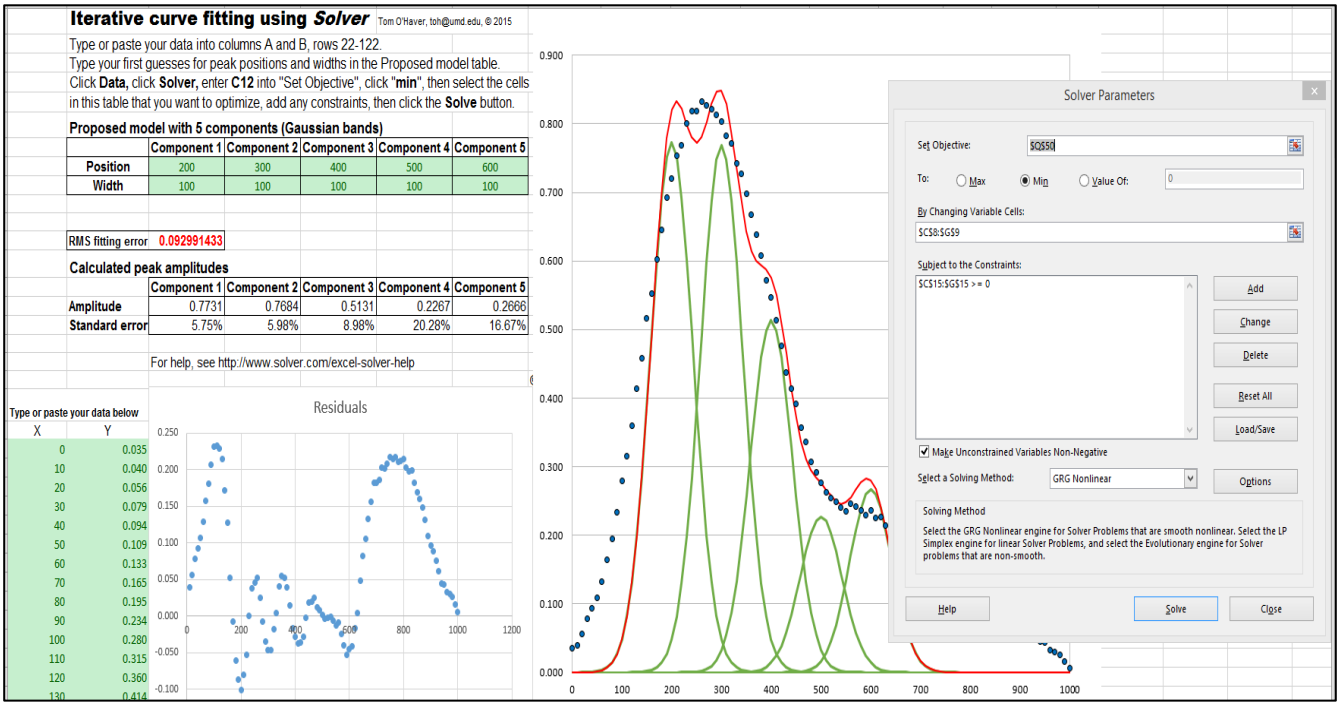
The precision of the model parameters measured by iterative fitting (page 210), like classical least-squares fitting, depends strongly on a good model, accurate compensation for the background/baseline, the signal-to-noise ratio, and the number of data points across the feature that you wish to measure. It is not practical to predict the standard deviations of the measured model parameters using the algebraic approach, but both the Monte Carlo simulation and bootstrap methods (page 170) are applicable.
Note: the term "spectral deconvolution" or "band deconvolution" is often used to refer to this technique, but in this book, "deconvolution" specifically means *Fourier* deconvolution, an independent concept that is treated elsewhere. In Fourier deconvolution, the underlying peak shape is *unknown,* but the broadening function is assumed to be *known*. In iterative least-squares curve fitting, it is just the

reverse: the peak shape must be known but the width of the broadening process, which determines the width and shape of the peaks in the recorded data, is unknown. Thus, the term “spectral deconvolution” is ambiguous: it might mean the Fourier deconvolution of a response function from a spectrum, or it might mean the decomposing of a spectrum into its separate additive peak components. These are different processes. Do not get them confused. See page 306.

Spreadsheets and stand-alone programs

Both *Excel* and *OpenOffice Calc* have a "Solver" capability that will change specified cells in an attempt to produce a specified goal. This can be used in peak fitting to minimize the fitting error between a set of data and a proposed calculated model, such as a set of overlapping Gaussian bands. The latest version includes three different solving methods. This Excel spreadsheet example (screenshot) demonstrates how this is used to fit four Gaussian components to a sample set of x,y data that has already been entered into columns A and B, rows 22 to 101 (you could type or paste in your own data there).

After entering the data, you do a visual estimate of how many Gaussian peaks it might take to represent the data, and their locations and widths, and type those estimated values into the 'Proposed model' table. The spreadsheet calculates the best-fit values for the peak *heights* by multilinear regression (page 188) in the 'Calculated amplitudes' table and plots the data and the fit. It also plots the "residuals", which are the point-by-point *differences* between the data and the model. Ideally, the residuals would be zero, or at least small. (Adjust the x-axis scale of these graphs to fit your data).



The next step is to use *Solver* function to "fine-tune" the position and width of each component to minimize the % fitting error (in red) and to make the residuals plot as random as possible: click **Data** in the top menu bar, click **Solver** (upper right) to open the Solver box, into which you type "C12" into "Set Objective", click "min", select the cells in the "Proposed Model" that you want to optimize, add any desired constraints in the "Subject to the Constraints" box, and click the **Solve** button. Solver

automatically optimizes the position, width, and amplitude of all the components and best fit is displayed. (You can see that the Solver has changed the selected entries in the proposed model table, reduced the fitting error (cell C12, in red), and made the residuals smaller and more random). If the fit fails, change the starting values, click **Solver**, and click the **Solve** button. You can automate the above process and reduce it to a single function-key press by using *macros*, as described on page 315.

So, how many Gaussian components does it take to fit the data? One way to tell is to look at the plot of the residuals (which shows the point-by-point difference between the data and the fitted model), and add components until the residuals are *random, not wavy*, but this works only if the data are *not smoothed* before fitting. Here's an example - a set of real data that are fitted with an increasing sequence of two Gaussians, three Gaussians, four Gaussians, and five Gaussians. As you look at this sequence of screenshots, you will see the percent fitting error decrease, the R^2 value becomes closer to 1.000, and the residuals become smaller and more random. (Note that in the 5-component fit, the first and last components are not *peaks* within the 250-600 x-range of the data, but rather they account for the *background*). There is no need to try a 6-component fit because the residuals are already random at 5 components and more components than that would just "fit the noise" and would likely be unstable and give a very different result with another sample of that signal with different noise.

If you use a spreadsheet for this type of curve fitting, you need to build a custom spreadsheet for each problem, with the right number of rows for the data and with the desired number of components. For example, my CurveFitter.xlsx template is only for a 100-point signal and a 5-component Gaussian model. It is easy to extend to a larger number of data points by inserting rows between 22 and 100, columns A through N, and drag-copying the formulas down into the new cells (e.g. CurveFitter2.xlsx is extended to 256 points). To handle other numbers of components or model shapes you would have to insert or delete columns between C and G and between Q and U and edit the formulas, as has been done in this set of templates for 2 Gaussians, 3 Gaussians, 4 Gaussians, 5 Gaussians, and 6 Gaussians.

If your peaks are superimposed on a baseline, you can *include a model for the baseline* as one of the components. For instance, if you wish to fit 2 Gaussian peaks on a linear tilted slope baseline, select a *3-component* spreadsheet template and change one of the Gaussian components to the equation for a straight line ($y=mx+b$, where m is the slope and b is the intercept). A template for that particular case is CurveFitter2GaussianBaseline.xlsx (graphic). Do not click "Make Unconstrained Variables Non-Negative" in this case, because the baseline model may well need negative variables, as it does in this particular example. If you want to use another peak shape or another baseline shape, you'd have to modify the equation in row 22 of the corresponding columns C through G and drag-copy the modified cell down to the last row, as was done to change the Gaussian peak shape into a Lorentzian shape in CurveFitter6Lorentzian.xlsx. Or you could make columns C through G contain equations for *different* peak or baseline shapes. For exponentially broadened Gaussian peak shapes, you can use CurveFitter2ExpGaussianTemplate.xlsx for two overlapping peaks (screen graphic). In this case, each peak has *four* parameters: height, position, width, and lambda (which determines the asymmetry - the extent of exponential broadening).

Using a spreadsheet has one big advantage: it is easy to add *constraints* to the variables determined by iteration, for example, to constrain them to be greater than zero or to fall between two limits, or to be

equal, etc. Doing so will force the solutions to adhere to known expectations and avoid nonphysical solutions. This is especially important for complex shapes such as the exponentially broadened Gaussian just discussed in the previous paragraph. You can do this by adding those constraints using the "Subject to the Constraints:" box in the center of the "Solver Parameters" box (see the graphic on the previous page). For details, see https://www.solver.com/excel-solver-add-change-or-delete-constraint?

The point of all this is that you can do - in fact, you *must* do - a lot of custom editing to get a spreadsheet template that fits your data. In contrast, my Matlab/Octave peakfit.m function (page 397) *automatically* adapts to any number of data points and is easily set to over 40 different model peak shapes (graphic on page 424) and any number of peaks simply by changing the input arguments. Using my *Interactive Peak Fitter* function ipf.m in Matlab (page 416), you can *press a single keystroke* to instantly change the peak shape, the number of peaks, the baseline mode (page 219), or to re-calculate the fit with a different start value or with a bootstrap subset of the data (to estimate the peak parameters errors). That is far quicker and easier than the spreadsheet. But conversely, a *real advantage of spreadsheets* in this application is that it is relatively easy to add your own *custom shape functions and constraints*, even complicated ones, using standard spreadsheet cell formula construction. And if you are hiring help, it is probably easier to find an experienced spreadsheet programmer than a Matlab programmer. So, if you are not sure which to use, my advice is to try both methods and decide for yourself.

For Python programmers, there are the scipy.optimize.minimize and LMFit packages, an extension of the Levenberg-Marquardt method. See page 451 for a Matlab/Python comparison of iterative fitting.

There are a number of downloadable non-linear iterative curve fitting add-ons and macros for Excel and OpenOffice. For example, Dr. Roger Nix of Queen Mary University of London has developed a very nice Excel/VBA spreadsheet for curve fitting X-ray photoelectron spectroscopy (XPS) data, but it could be used to fit other types of spectroscopic data. A 4-page instruction sheet is also provided.

There are also many examples of stand-alone programs, both freeware and commercial, including PeakFit, Data Master 2003, MyCurveFit, Curve Expert, Origin, ndcurvemaster, and the R language.

Matlab and Octave

Matlab and **Octave** have a convenient and efficient function called "fminsearch" that uses the Nelder-Mead method. It was originally designed for finding the minimum values of functions, but it can be applied to least-squares curve fitting by creating a so-called anonymous function (a.k.a. "*lambda"* function) that computes the model, compares it to the data, and returns the fitting error. For example, writing `parameters = fminsearch(@(lambda)(fitfunction(lambda, x, y)), start)` performs an iterative fit of the data in the vectors x,y to a model described in a previously-created function called `fitfunction`, using the first guesses in the vector "start". The parameters of the best-fit model are returned in the vector "parameters", in the same order that they appear in "start".

Note: for Octave users, the fminsearch function is contained in the "Optim" add-on package (use the latest version 1.2.2 or later), downloadable from Octave Forge. It is a good idea to install *all* these add-on packages just in case they are needed. Follow the instructions on the Octave Forge web page. For

Matlab users, fminsearch is a built-in function, although there are other optimization routines in the optional Optimization Toolbox, which is not needed for the examples and programs in this document.

A simple example is fitting the blackbody equation to the spectrum of an incandescent body for the purpose of estimating its color temperature. In this case, there is only *one* nonlinear parameter, temperature. The script BlackbodyDataFit.m demonstrates the technique, placing the experimentally measured spectrum in the vectors "wavelength" and "radiance" and then calling fminsearch with the fitting function fitblackbody.m. Incandescent lightbulbs, of the type that used to be common in household lighting before LEDs, are examples of blackbody radiators. (If a blackbody source is not thermally homogeneous, it may be possible to model it as the *sum* of two or more regions of different temperature, as in example 3 of fitshape1.m).

Another application is demonstrated by Matlab's built-in demo fitdemo.m and its corresponding fitting function fitfun.m, which model the sum of two exponential decays. To see this, just type "fitdemo" in the Matlab command window. (Octave does not have this demo function).

Fitting peaks

Many instrumental methods of measurement produce signals in the form of peaks of various shapes. A common requirement is to measure the positions, heights, widths, and/or areas of those peaks, even when they are noisy or overlapped with one another. This cannot be done by linear least-squares methods, because such signals cannot be modeled as polynomials with linear coefficients (the positions and widths of the peaks are not linear functions), so iterative curve fitting techniques are used instead, often using Gaussian, Lorentzian, or some other fundamental simple peak shapes as a model.

The Matlab/Octave demonstration script Demofitgauss.m demonstrates fitting a Gaussian function to a set of data, using the fitting function fitgauss.m. In this case, there are two non-linear parameters: peak position and peak width (the peak height is a linear parameter and is determined by regression in a single step in line 9 of the fitting function fitgauss.m and is returned in the global variable "c"). Compared to the simpler polynomial least-squares methods for measuring peaks (page 173), the iterative method has the advantage of using all the data points across the entire peak, including zero and negative points, and it can be applied to multiple overlapping peaks as demonstrated the script Demofitgauss2.m.

To accommodate the possibility that the baseline may shift, you can add a column of 1s to the A matrix, just as was done in the CLS method (page 188). This has the effect of introducing into the model an additional component that is simply a flat line. Its amplitude is returned along with the peak heights in the global vector “c”. Demofitgaussb.m and fitgauss2b.m illustrate this addition. (Demofitlorentzianb.m and fitlorentzianb.m for Lorentzian peaks).

This peak fitting technique is easily extended to any number of overlapping peaks of the same type using the *same* fitting function fitgauss.m, which easily adapts to any number of peaks, depending on the length of the first-guess "start" vector *lambda* that is passed to the function as input arguments, along with the data vectors *t* and *y*:

```
1 function err = fitgauss(lambda, t,y)
2 %  Fitting functions for a Gaussian band spectrum.
```

```
3 %  T. C. O'Haver. March 2006
4 global c
5 A = zeros(length(t),round(length(lambda)/2));
6 for j = 1:length(lambda)/2,
7     A(:,j) = gaussian(t, lambda(2*j-1),lambda(2*j))';
8 end
9 c = A\y'; % c = abs(A\y') for positive peak heights only
10 z = A*c;
11 err = norm(z-y');
```

If there are *n* peaks in the model, then the length of lambda is 2*n*, one entry for each iterated variable ([position1 width1 position2 width2....etc.]). The "for" loop (lines 5-7) constructs an *n* × length(t) matrix containing the model for each peak separately, using a user-defined peak shape function (in this case gaussian.m), then it computes the *n*-length peak height vector *c* by least-squares regression in line 9, using the Matlab shortcut "\" notation. (To constrain the fit to *positive* values of peak height, replace A\y' with abs(A\y') in line 9). The resulting peak heights are used to compute *z*, the sum of all *n* model peaks, by matrix multiplication in line 10, and then "err", the root-mean-square difference between the model *z* and the actual data *y*, is computed in line 11 by the Matlab 'norm' function and returned to the calling function ('fminsearch'), which repeats the process many times, trying different values of the peak positions and the peak widths until the value of "err" is low enough.

This fitting function above would be called by Matlab's fminsearch function like so:

```
params=fminsearch(@(lambda)(fitgauss(lambda, x,y)),[50 20])
```

where the square brackets contain a vector of first guesses for position and width for each peak ([position1 width1 position2 width2....etc.]). The output argument 'params' returns a 2 × *n* matrix of best-fit values of position and width for each peak, and the peak heights are contained in the *n*-length global variable vector c. Similar fitting functions can be defined for other peak shapes simply by calling the corresponding peak shape function, such as lorentzian.m in line 7. (Note: for this and other scripts like Demofitgauss.m or Demofitgauss2.m to work on your version of Matlab, all the functions that they call must be loaded into Matlab beforehand, in this case fitgauss.m and gaussian.m. Scripts that call sub-functions must have those functions in the Matlab search path. Functions, conversely, can have all their required sub-functions defined within the main function itself and thus can be self-contained, as are the next two examples).

Simplified general-purpose peak-fitting function

The function fitshape2.m (syntax: **`[Positions, Heights, Widths, FittingError] = fitshape2(x, y, start)`**) pulls all of this together into a simplified general-purpose Matlab/Octave *function* for fitting multiple overlapping model shapes to the data contained in the vector variables x and y. The model is the weighted sum of any number of basic peak shapes which are defined mathematically as a function of x, with two variables that the program will independently determine for each peak - positions and widths - in addition to the peak heights (i.e., the weights of the weighted sum). You must provide the first guess starting vector 'start', in the form [position1 width1 position2 width2 ...etc.], which specifies the first-guess position and width of each component (one pair of position and width for each peak in the model). The function returns the parameters of the best-fit model in the vectors **`Positions, Heights, Widths`**, and computes the percent error between the

data and the model in **`FittingError`**. It also plots the data as dots and the fitted model as a line.

The interesting thing about this function is that *the only part that defines the shape of the model is the last line*. In fitshape2.m, that line initially contains the expression for a *Gaussian peak of unit height*, but you could change that to *any other expression or algorithm* that computes a unit-height *g* as a function of *x* with two unknown parameters 'pos' and 'wid' (position and width, respectively, for peak-type shapes, but they could represent anything for other function types, such as the exponential pulse, sigmoidal, etc.) Everything else in the fitshape.m function can remain the same. It is all about the bottom line. This makes fitshape.m a good platform for experimenting with different mathematical expressions as proposed models to fit data. There are also two other variations of this function for models with *one* iterated variable plus peak height (fitshape1.m) and *three* iterated variables plus peak height (fitshape3.m). Each has illustrative examples in the help file (type "help fitshape..."). An alternative version is FitMultipleShapes2, which allows you to specify any of 16 common peak shape functions by number. Examples are in the internal help file . The syntax is:

```
[Positions,Heights,Widths,FittingError]=FitMultipleShapes2(x,y,shape,start,m)
```

Variable shape types

The Voigt profile, Pearson, Breit-Wigner-Fano, Gauss-Lorentz blend, and the exponentially-broadened Gaussian and Lorentzian, are defined not only by the peak position, height, and width, but also by *an additional parameter that fine-tunes the shape of the peak*. If that parameter is *equal and known* for all peaks in a group, it can be passed as an additional input argument to the shape function, as shown in the demo function VoigtFixedAlpha.m for the Voigt profile, which is calculated as a convolution of Gaussian and Lorentzian components with different widths. If the shape parameter (*alpha*, the ratio of the two widths) is allowed to be *different* for each peak in the group and is to be determined by iteration (just as is position and width), then the routine must be modified to accommodate *three, rather than two*, iterated variables, as shown in the demo function VoigtVariableAlpha.m. Although the fitting error is lower with variable alphas, the execution time is longer and the alpha values so determined are not very stable, with respect to noise in the data and the starting guess values, especially for multiple peaks. (These are self-contained functions). Version 9.5 of the general-purpose Matlab/Octave function peakfit.m includes fixed and variable shape types for the Pearson, ExpGaussian, Voigt, and Gaussian/Lorentzian blend, as well as the 3-parameter logistic or Gompertz function (whose three parameters are labeled Bo, Kh, and L, rather than position, width, and shape factor). The script VoigtShapeFittingDemonstration.m uses the peakfit.m version 9.5 to fit a single Voigt profile and to calculate the Gaussian width component, Lorentzian width component, and alpha. It computes the theoretical Voigt profile with added random noise for realism. The script VoigtShapeFittingDemonstration2.m does the same for *two* overlapping Voigt profiles, using both fixed alpha and variable alpha models (shape numbers 20 and 30). (Requires voigt.m, halfwidth.m, and peakfit.m in the Matlab search path). The script GLBlendComparison compares the Voigt to the simpler Gauss/Lorentzian blend, showing that they are nearly identical within 0.3% relative difference.

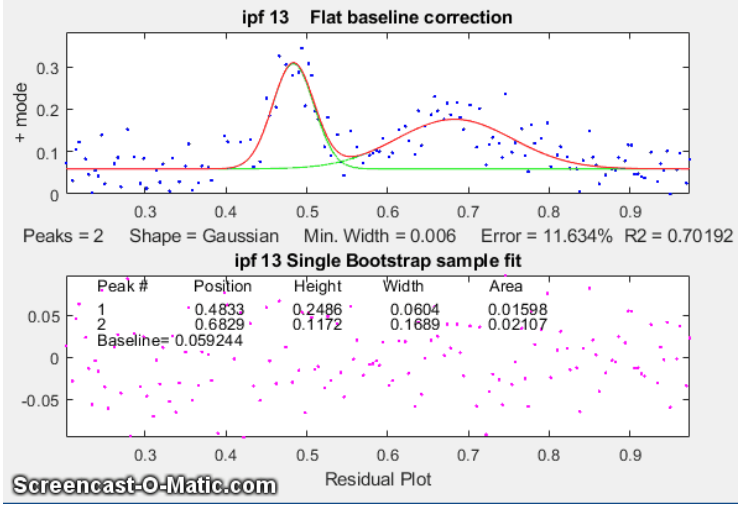
If you do *not* know the shape of your peaks, you can use peakfit.m or ipf.m (page 416) to try different shapes to see if one of the standard shapes included in those programs fits the data. Try to find a peak in your data that is typical, isolated, and that has a good signal-to-noise ratio. For example, the Matlab functions ShapeTestS.m and ShapeTestA.m test the data in its input arguments x,y, assumed to be a

single isolated peak, fits it with *different candidate model peak shapes* using peakfit.m, plots each fit in a separate figure window, and prints out a table of fitting errors in the command window. ShapeTestS.m tries seven different candidate *symmetrical* model peaks, and ShapeTestA.m tries six different candidate *asymmetrical* model peaks. The one with the lowest fitting error (and R^2 closest to 1.000) is presumably the best candidate. Try the examples in the help files for each of these functions. But beware, if there is too much noise in your data, the results can be misleading. For example, even if the actual peak shape is something other than a Gaussian, the multiple Gaussians model is likely to fit slightly better because it has more degrees of freedom and can "fit the noise". The Matlab function peakfit.m has many more built-in shapes to choose from, but it is still a *finite* list and there is always the possibility that the actual underlying peak shape is not available in the software you are using or that it is simply not describable by a mathematical function.

Signals with *peaks of different shape types in one signal* can be fit by the fitting function fitmultiple.m, which takes as input arguments a vector of peak types and a vector of shape variables. The sequence of peak types and shape parameters must be determined beforehand. To see how this is used, see Demofitmultiple.m.

You can create your own fitting functions for any purpose. They are *not* limited to single algebraic expressions but can be arbitrarily complex multiple-step algorithms. For example, in the "Tfit" method for quantitative absorption spectroscopy (page 276), a model of the instrumentally-broadened transmission spectrum is fit to the observed transmission data, using a fitting function that performs Fourier convolution (page 109) of the transmission spectrum model with the known slit function of the spectrometer. The result is an alternative method of calculating absorbance that allows the optimization of signal-to-noise ratio and extends the dynamic range and calibration linearity of absorption spectroscopy far beyond the normal limits.

Peak Fitting Functions for Matlab and Octave

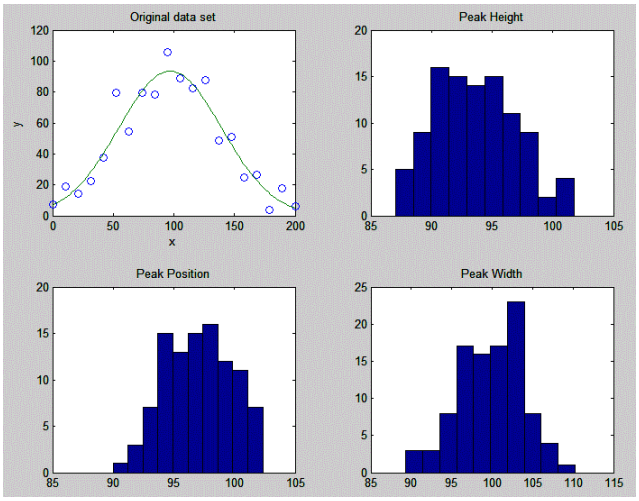


Here I describe more complete peak fitting functions that have additional capabilities: a built-in set of basic peak shapes that you can select, provision for estimating the "start" vector if you do not provide one, provision for handling baselines, ability to estimate peak parameter uncertainties, etc. These functions accept signals of any length, including those with non-integer and non-uniform x-values, and can fit any number of peaks with Gaussian, equal-width Gaussian, fixed-width Gaussian, exponentially-broadened Gaussian, exponentially-broadened equal-width Gaussians, bifurcated Gaussian, Lorentzian, fixed-width Lorentzians, equal-width Lorentzians, exponentially-broadened Lorentzian, bifurcated Lorentzian, logistic distribution, logistic function, triangular, alpha function, Pearson 7, exponential pulse, up sigmoid, down sigmoid,

Gaussian/ Lorentzian blend, Breit-Wigner-Fano, and Voigt profile shapes. (A graphic that shows the basic peak shapes available is on page 424).

There are two different versions, a command-line version called peakfit.m, for Matlab or Octave, and a keypress-operated interactive version called ipf.m or ipfoctave.m (page 416). For adding as an element on your own programs and for automating the fitting of large numbers of signals, peakfit is better. But ipf.m is best for exploring a few signals to determine the best fitting range, peak shapes, number of peaks, baseline correction mode, etc. Both functions allow for simplified operation by providing default values for any unspecified input arguments. For example, the starting values, if not specified in the input arguments, are estimated by the program based on the length and x-axis interval of the data. Compared to the fitshape.m function described above, peakfit.m has a large number of built-in peak shapes, it does not require (although it can be given) the first-guess position and width of each component, and it has features for background correction and other useful features to improve the quality and reliability of fits. All of my functions have been extensively tested on experimental data sent to me by hundreds of researchers in many different fields, helping to expand and improve my software over time).

These functions can optionally estimate the expected standard deviation and interquartile range of the peak parameters using the bootstrap sampling method (page 170). See DemoPeakfitBootstrap for a self-contained demonstration of this function.



The effect of random noise on the uncertainty of the peak parameters determined by iterative least-squares fitting is readily estimated by the bootstrap sampling method (introduced on page 170). A simple demonstration of bootstrap estimation of the variability of an iterative least-squares fit to a single noisy Gaussian peak is given by my Matlab/ Octave function "BootstrapIterativeFit.m", which creates a single x,y data set consisting of a single noisy Gaussian peak, extracts bootstrap samples from that data set, performs an iterative fit to the peak on each of the bootstrap samples, and plots the distributions (histograms) of peak height, position, and width of the bootstrap samples. The syntax is **`BootstrapIterativeFit(TrueHeight, TruePosition, TrueWidth, NumPoints, Noise, NumTrials)`** where **`TrueHeight`** is the true peak height of the Gaussian peak, **`TruePosition`** is the true x-axis value at the peak maximum, **`TrueWidth`** is the true half-width (FWHM) of the peak, NumPoints is the number of points taken for the least-squares fit, Noise is the standard deviation of (normally-distributed) random noise, and NumTrials is the number of bootstrap samples.

A typical example for **`BootstrapIterativeFit( 100,100,100,20,10,100);`** is displayed in the figure on the right, above. The results, displayed in the command window, are:

```
>> BootstrapIterativeFit(100,100,100,20,10,100);
```

```
      Peak Height  Peak Position  Peak Width
mean:   99.27028    100.4002       94.5059
STD:    2.8292      1.3264         2.9939
IQR:    4.0897      1.6822         4.0164
IQR/STD Ratio:    1.3518
```

A similar demonstration function for *two* overlapping Gaussian peaks is available in "BootstrapIterativeFit2.m". Type "help BootstrapIterativeFit2" for more information. In both these simulations, the standard deviation (STD), as well as the interquartile range (IQR) of each of the peak parameters, are computed. This is done because the interquartile range is much less influenced by *outliers*. The distribution of peak parameters measured by iterative fitting is often non-normal, exhibiting a greater fraction of large deviations from the mean than is expected for a normal distribution. This is because the iterative procedure sometimes converges on an abnormal result, especially for multiple peak fits with many variable parameters. (You may be able to see this in the histograms plotted by these simulations, especially for the weaker peak in BootstrapIterativeFit2). In those cases, the standard deviation will be too high because of the outliers, and the IQR/STD ratio will be much less than the value of 1.34896 that is expected for a normal distribution. In that case, a better estimate of the standard deviation of the central portion of the distribution (without the outliers) is IQR/1.34896.

It is important to emphasize that the bootstrap method predicts only the effect of random noise on the peak parameters for a fixed fitting model. It does not consider the possibility of peak parameter inaccuracy caused by using a non-optimum data range, or choosing an imperfect model, or by inaccurate compensation for the background/baseline, all of which are at least partly subjective and thus beyond the range of influences that can easily be treated by random statistics. If the data have relatively little random noise or have been smoothed to reduce the noise, then it is likely that model selection and baseline correction will be the major sources of peak parameter inaccuracy, which are not well predicted by the bootstrap method.

For the quantitative measurement of peaks, it is instructive to compare the iterative least-squares method with simpler, less computationally intensive, methods. For example, the measurement of the peak height of a single peak of uncertain width and position could be done simply by taking the maximum of the signal in that region. If the signal is noisy, a more accurate peak height will be obtained if the signal is smoothed beforehand (page 42). But smoothing can distort the signal and reduce peak heights. Using an iterative peak fitting method, assuming only that the peak shape is known, can give the best possible accuracy *and* precision, without requiring smoothing even under high noise conditions, e.g. when the signal-to-noise ratio is 1, as in the demo script SmoothVsFit.m:

```
True peak height = 1    NumTrials = 100    SmoothWidth = 50

      Method         Maximum y    Max Smoothed y    Peakfit
Average peak height    3.65          0.96625         1.0165
Standard deviation     0.36395       0.10364         0.1157
```

If peak *area* is measured rather than peak *height*, smoothing is unnecessary (unless to locate the peak beginning and end) but peak fitting still yields the best precision. See SmoothVsFitArea.m.

It is also instructive to compare the iterative least-squares method with *classical least-squares curve fitting*, discussed on page 188, which can also fit peaks in a signal. The difference is that in the classical least-squares method, the positions, widths, and shapes of all the individual components are all known beforehand. The *only* unknown quantities are the amplitudes (e.g., peak heights) of the components in the mixture. In non-linear iterative curve fitting, on the other hand, the positions, widths, and heights of the peaks a*re all unknown* beforehand. The *only* thing that is known is the fundamental underlying *shape* of the peaks. The non-linear iterative curve fitting is more difficult to do (for the computer, anyway) and more prone to error, but it is necessary if you need to track shifts in peak position or width or to decompose a complex overlapping peak signal into fundamental components knowing only their shape. The Matlab/Octave script "CLSvsINLS.m" compares the classical least-squares (CLS) method with different variations of the iterative method (INLS) method for measuring the peak heights of three Gaussian peaks in a noisy test signal on a standard Windows PC, demonstrating that the fewer the number of unknown parameters, the faster and more accurate is the peak height calculation.

```
Method  Positions  Widths       Execution time   % Accuracy
  CLS     known     known          0.00133         0.30831
  INLS    unknown   unknown        0.61289         0.6693
  INLS    known     unknown        0.16385         0.67824
  INLS    unknown   known          0.24631         0.33026
  INLS    unknown   known (equal)  0.15883         0.31131
```

Another comparison of multiple measurement techniques is presented in Case Study D (page 298).

Note: If you are reading this book online, you can right-click on any of the m-file links and select **Save Link As...** to download them to your computer, then place them in the Matlab search path for use within Matlab.

Accuracy and precision of peak parameters

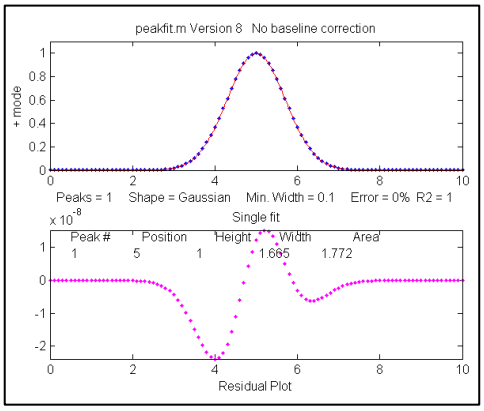
Iterative curve fitting is often used to measure the position, height, and width of peaks in a signal, especially when they overlap significantly. There are four major sources of error in measuring these peak parameters by iterative curve fitting. This section makes use of my peakfit.m function. Instructions are here or type "help peakfit". (Once you have peakfit.m in the Matlab search path, you can copy and paste, or drag and drop, any of the following single-line or multi-line code examples into the Matlab or Octave editor or into the command line and press **Enter** to execute it).

a. Model errors.

Peak shape. If you have the wrong model for your peaks, the results cannot be expected to be accurate. For instance, if your actual peaks are Lorentzian in shape, but you fit them with a Gaussian model or *vice versa*. For example, a single isolated Gaussian peak at x=5, with a height of 1.000 fits a Gaussian model virtually perfectly, using the Matlab user-defined peakfit function (page 397), as shown on the right. (The 5th input argument for the peakfit function specifies the shape of peaks to be used in the fit. Peak shape "1" means Gaussian. See page 398 for a list of shape numbers).

```
>> x=[0:.1:10];y=exp(-(x-5).^2);
>> [FitResults,MeanFitError]=peakfit([x' y'],5,10,1,1)
```

```
Peak#      Position      Height       Width         Area
1             5             1        1.6651        1.7725
MeanFitError =     7.8579e-07       R2=              1
```

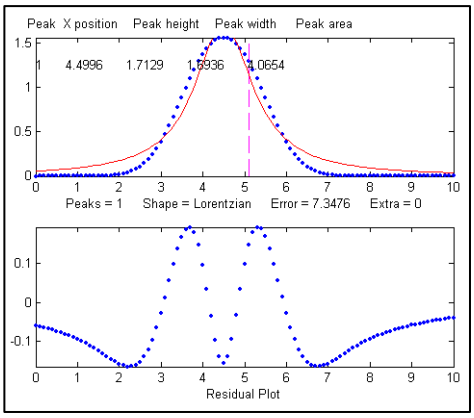


The "FitResults" are, from left to right, peak number, peak position, peak height, peak width, and peak area. The MeanFitError, or just "fitting error", is the square root of the sum of the squares of the differences between the data and the best-fit model, as a percentage of the maximum signal in the fitted region. Recent versions of peakfit return also the R2, the "R-squared" or coefficient of determination, which is exactly 1 for a perfect fit. Note the agreement between the area (1.7725) with the *theoretical* area under the curve of $\exp(-x^2)$, which is the square root of pi. If you are reading this online, click for Wolfram Alpha solution. But this same peak, when fitted with the incorrect model (a *Logistic* model, peak shape number 3), gives a fitting error of 1.4% and height and width errors of 3% and 6%, respectively. However, the peak area error is only 1.7%, because the height and width errors partially cancel out. So, you do not have to have a perfect model to get a decent area measurement.

```
>> [FitResults,MeanFitError]=peakfit([x' y'],5,10,1,3)
Peak#      Position      Height       Width           Area
          Peak#      Position       Height        Width       Area
1        5.0002       0.96652         1.762        1.7419
MeanFitError =1.4095
```

When fit with an even more incorrect *Lorentzian* model (peak shape 2, shown on the right, below), this peak gives a 7% fitting error and height, width, and area errors of 8%, 20%, and 17%, respectively.

```
>> [FitResults,MeanFitError]=peakfit([x' y'],5,10,1,2)
Peak#      Position     Height       Width          Area
1             5         1.0876       1.3139        2.0579
MeanFitError =5.7893
```

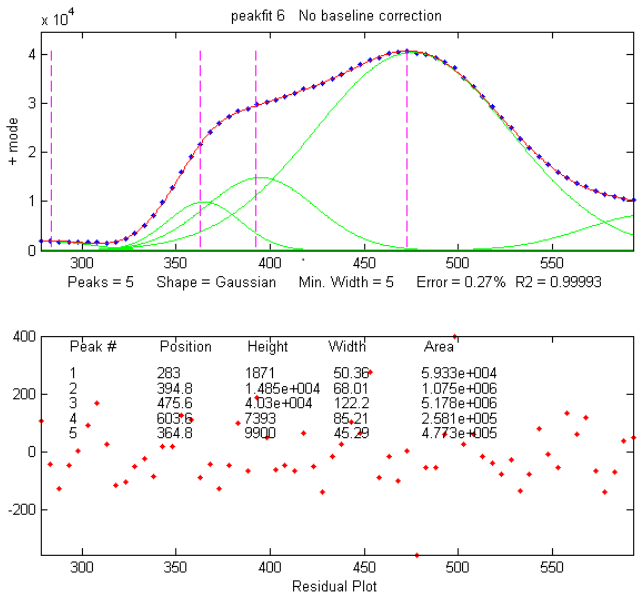


But it is unlikely that your estimate of a model will be that far off. It is much more likely that your actual peaks are some unknown combination of peak shapes, such as Gaussian with a little Lorentzian mixed in or vice versa, or some slightly asymmetrical modification of a standard symmetrical shape. So, if you use an available model that is at least *close* to the actual shape, the parameter errors may not be so bad and may, in fact, be better than other measurement methods.

So clearly the larger the fitting errors, the larger are the parameter errors, but the parameter errors are of course not *equal* to the fitting error (that would be *too* easy). Also, the peak *height* and *width* are the parameters most susceptible to errors. The peak *positions* are measured accurately even if the model is wrong if the peak is symmetrical and is not highly

overlapping with other peaks.

A good fit is not by itself proof that the shape function you have chosen is the correct one. In some cases, the wrong function can give a fit that looks perfect. For example, the graph on the left above shows a fit of a real data set to a 5-peak Gaussian model that exhibits a low percent fitting error residuals that look random - usually an indicator of a good fit. But in fact*, in this case, the model is wrong*. Those data came from an experimental domain where the underlying shape is *fundamentally non-Gaussian* but, in some cases, can look very like a Gaussian. As another example, a data set consisting of peaks with a Voigt profile peak shape can be fitted with a weighted sum of a Gaussian and a Lorentzian almost as well as with an actual Voigt model, even though those models are *not the same mathematically*. The difference in fitting error is so small that it would likely be obscured by the random noise if it were a real experimental signal. The script GLBlendComparison compares the Voigt to the simpler Gauss/Lorentzian blend, showing that they are nearly identical within 0.3% relative difference. The same thing can occur in sigmoidal signal shapes: a pair of simple 2-parameter logistic functions seems to fit this example data pretty well, with a fitting error of less than 1%. You would have no reason to doubt the goodness of fit unless the random noise is low enough so you can see that the residuals are wavy. In fact, a *3-parameter* logistic (Gompertz function) fits much better, and the residuals are random, not wavy, which indicates a better fit. In such cases you cannot depend solely on what *looks* like a good fit to determine whether the model is optimum. Sometimes you need to know more about the peak shape expected in that kind of experiment, especially if the data are noisy. At best, if you do get a good fit with random non-wavy residuals, you can claim only that the data *are consistent with* the proposed model.



In some applications the accuracy of the model is *not* so important. Take the example of *quantitative analysis applications*, where the peak heights or areas measured by curve fitting is used to determine the *concentration of the substance that created the peak* by constructing a calibration curve (page 465) based on laboratory prepared standards solutions of known concentrations. In that case, the necessity of using the exact peak model is lessened, if the shape of the unknown peak is *constant and independent of concentration*. If the wrong model shape is used, the R^2 *for curve fitting* will be poor (much less than 1.000) and the peak heights and areas measured by curve fitting will be inaccurate, but the *error will be the same* for the unknown samples and the known calibration standards, so the error will cancel out. As a result, the R^2 *for the calibration curve* can be very high (0.9999 or better) and the *measured concentrations will be no less accurate than they would have been with a perfect peak shape model*. Even so, it is useful to use as accurate a model peak shape as possible, because the R^2 for curve fitting will work better as a warning indicator if something unexpected goes wrong during the analysis (such as an increase in the

noise or the appearance of an interfering peak from a foreign substance).
See PeakShapeAnalyticalCurve.m for a Matlab/Octave demonstration.

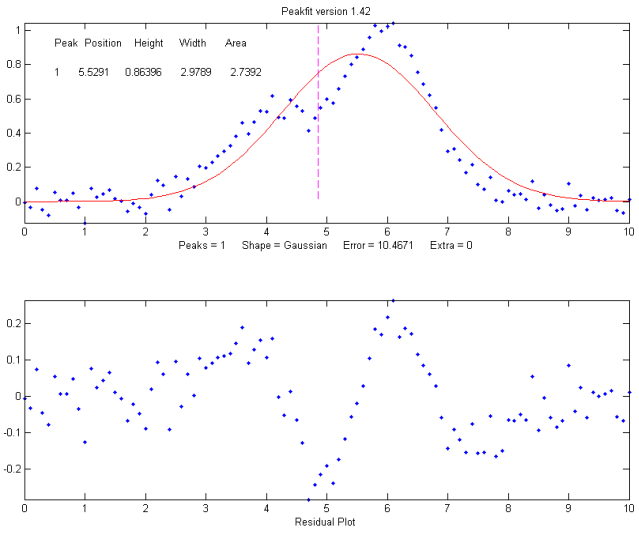
Number of peaks. Another source of model error occurs if you have the wrong *number of peaks* in your model, for example if the signal actually has *two* peaks but you try to fit it with only *one* peak. Let's take an obvious case first. In the example below, the Matlab code generates a simulated signal with of two Gaussian peaks at x=4 and x=6 with peaks heights of 1.000 and 0.5000 respectively and widths of 1.665, plus random noise with a standard deviation 5% of the height of the largest peak (a signal-to-noise ratio of 20):

```
>> x=[0:.1:10];
>> y=exp(-(x-6).^2)+.5*exp(-(x-4).^2)+.05*randn(size(x));
```

In a real experiment, you would not usually know the peak positions, heights, and widths. You would be using curve fitting to *measure* those parameters. Let us assume that, based on previous experience or some preliminary trial fits, you have established that the optimum peak *shape* model is Gaussian, but you do not know for sure how many peaks are in this group. If you start out by fitting this signal with a *single*-peak Gaussian model, you get:

```
>> [FitResults,MeanFitError]=peakfit([x' y'],5,10,1,1)

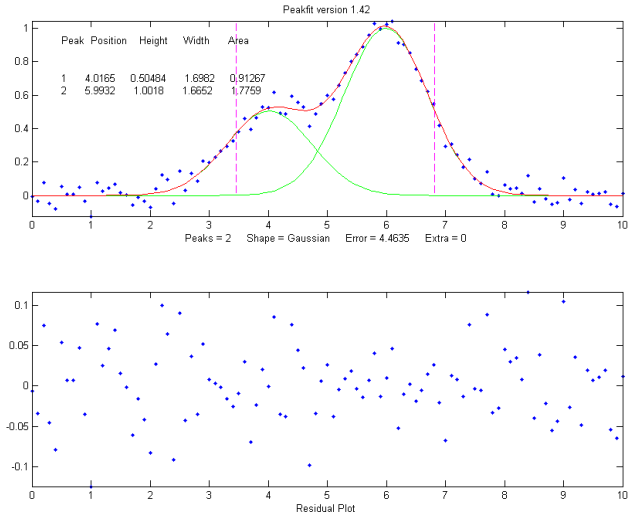
   Peak#    Position    Height      Width       Area
   1        5.5291      0.86396     2.9789     2.7392
MeanFitError = 10.467
```



Obviously, this is not right. The residual plot (bottom panel) shows a "wavy" structure that is clearly visible in the random scatter of points due to the random noise in the signal. This means that the fitting error is not limited by the random noise. It is a clue that the model is not quite complete.

However, a fit with *two* peaks yields much better results (The 4th input argument for the peakfit function specifies the number of peaks to be used in the fit).

```
>> [FitResults,MeanFitError]=peakfit([x' y'],5,10,2,1)
      Peak#     Position   Height     Width     Area
       1       4.0165      0.50484    1.6982    0.91267
       2       5.9932      1.0018     1.6652    1.7759
MeanFitError = 4.4635
```

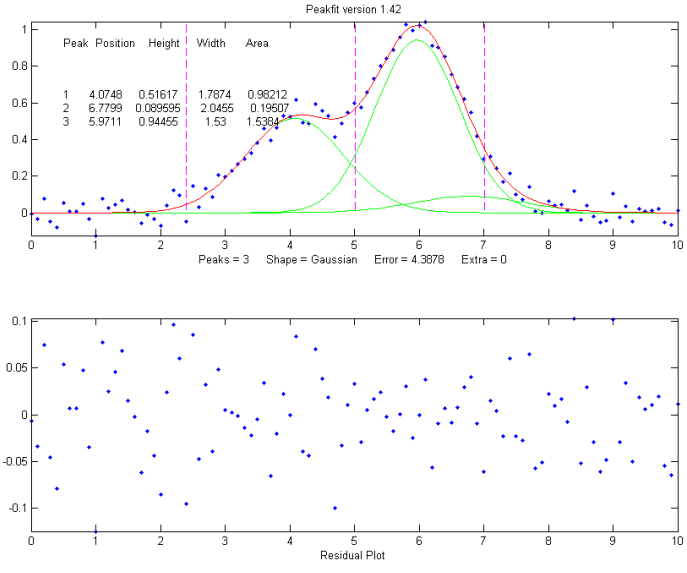


Now the residuals have a random scatter of points, as would be expected if the signal had been accurately fit except for the random noise. Moreover, the fitting error is less than half of the error with only one peak. In fact, the fitting error is just about what you would expect in this case based on the 5% random noise in the signal (estimating the relative standard deviation of the points in the baseline visible at the edges of the signal). Because this is a simulation in which you know beforehand the true values of the peak parameters (peaks at x=4 and x=6 with peaks heights of 1.0 and 0.50 respectively and widths of 1.665), you can calculate the parameter errors (the difference between the real peak positions, heights, and widths and the measured values). Note that they are quite accurate (in this case within about 1% relative on the peak height and 2% on the widths), *which is better than the 5% random noise in this signal because of the averaging effect of fitting to multiple data points in the signal*.

However, if going from one peak to two peaks gave us a better fit, why not go to *three* peaks? If there were no noise in the data, and if the underlying peak shape were perfectly matched by the model, then the fitting error would have already been essentially zero with two model peaks. Adding a third peak to the model would yield a vanishingly small height for that third peak. But in our examples here, as in real data, there is always some random noise, and the result is that the third peak height will not be zero. Changing the number of peaks to three gives these results:

```
>> [FitResults,MeanFitError]=

peakfit([x' y'],5,10,3,1)
```

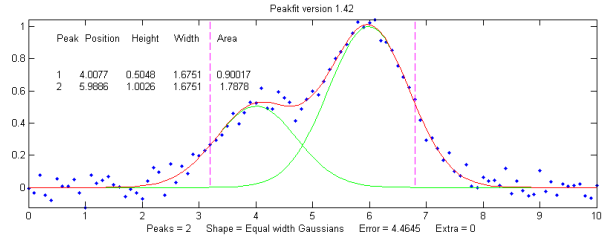


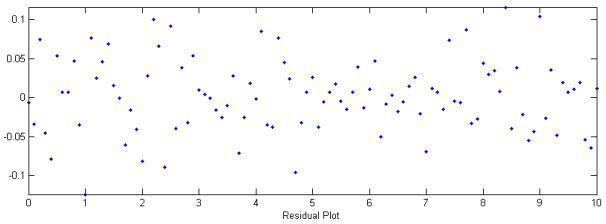

```
Peak#   Position    Height     Width      Area
1     4.0748      0.51617    1.7874     0.98212
2     6.7799      0.089595   2.0455     0.19507
3     5.9711      0.94455    1.53       1.5384
MeanFitError = 4.3878
```

The fitting algorithm has now tried to fit an additional low-amplitude peak (numbered peak 2 in this case) located at x=6.78. The fitting error is lower than for the 2-peak fit, but only slightly lower, and *the residuals are no less visually random* that with a 2-peak fit. So, knowing nothing else, a 3-peak fit might be rejected on that basis alone. In fact, there is a serious downside to fitting more peaks than are present in the signal: *it increases the parameter measurement errors* of the peaks that are present. Again, you can prove this because you know beforehand the true values of the peak parameters: clearly, the peak positions, heights, and widths of the two real peaks than are in the signal (peaks 1 and 3) are significantly less accurate than those of the 2-peak fit.

Moreover, if you repeat that fit with the *same signal* but with a *different* sample of random noise (simulating a repeat measurement of a stable experimental signal in the presence of random noise), the additional third peak in the 3-peak fit will bounce around all over the place (because the third peak is fitting the random *noise*, not an actual peak in the signal).

```
>> x=[0:.1:10];
>> y=exp(-(x-6).^2)+.5*exp(-(x-4).^2)+.05*randn(size(x));
>> [FitResults,MeanFitError]=peakfit([x' y'],5,10,3,1)
        Peak#    Position     Height      Width       Area
        1        4.115       0.44767     1.8768      0.89442
        2        5.3118      0.09340     2.6986      0.26832
        3        6.0681      0.91085     1.5116      1.4657
MeanFitError = 4.4089
```





With this new set of data, two of the peaks (numbers 1 and 3) have roughly the same position, height, and width, but peak number 2 has changed substantially compared to the previous run. Now you have an even more compelling reason to reject the 3-peak model: the 3-peak solution *is not stable*. And because this is a simulation in which you know the right answers, you can verify that the accuracy of the peak heights is substantially poorer (about 10% error) than expected with this level of random noise in the signal (5%). If you were to run a 2-peak fit on the same new data, you get much better measurements of the peak heights.

```
>> [FitResults,MeanFitError]=peakfit([x' y'],5,10,2,1)
         Peak#    Position     Height      Width       Area
          1       4.1601       0.49981     1.9108      1.0167
          2       6.0585       0.97557     1.548       1.6076
MeanFitError = 4.4113
```

If this is repeated several times, it turns out that the peak parameters of the peaks at x=4 and x=6 are, on average, more accurately measured by the 2-peak fit. In practice, the best way to evaluate a proposed fitting model is to fit several repeat measurements of the same signal (if that is practical experimentally) and to compute the standard deviation of the peak parameter values. In real

experimental work, of course, you usually do not *know* the right answers beforehand, so that is why it is important to use methods that work well when you *do* know. Here's an example of a set of real data that was fit with a succession of **2**, **3**, **4** and **5** Gaussian models, until the residuals became random. With each added component, the fitting error becomes smaller and the residuals become more random. But beyond 5 components point, there is little to be gained by adding more peaks to the model. Another way to determine the minimum number of models peaks needed is to plot the fitting error vs the number of model peaks. The point at which the fitting error reaches a minimum, and increases afterward, would be the fit with the "ideal combination of having the best fit without excess or unnecessary terms" in the words of Wikipedia. The Matlab/Octave function testnumpeaks.m (**`R = testnumpeaks(x, y, peakshape, extra, NumTrials, MaxPeaks)`**) applies this idea by fitting the x,y data to a series of models of shape `peakshape` containing 1 to `MaxPeaks` model peaks. The correct number of underlying peaks is either the model with the lowest fitting error, or, if two or more models have about the same fitting error, the model with the *least* number of peaks. The Matlab/Octave demo script NumPeaksTest.m uses this function with noisy computer-generated signals containing a user-selected 3, 4, 5 or 6 underlying peaks. With very noisy data, however, the technique is not always reliable.

Peak width constraints. Finally, there is one more thing that you can do that might improve the peak parameter measurement accuracy, and it concerns the peak widths. In all the above simulations, the basic assumption that *all* the peak parameters were unknown and independent of one another. In some types of measurements, however, the peak widths of each group of adjacent peaks are all expected to be equal, based on first principles or previous experiments. This is a common situation in analytical chemistry, especially in atomic spectroscopy and in chromatography, where the peak widths are determined largely by instrumental factors that are the same for all peaks in a given region.

In the current simulation, the true widths of both peaks are in fact equal to 1.665, but all the results above show that the *measured* peak widths are close but not quite equal, due to random noise in the signal. The unequal peak widths are a consequence of the random noise, not real differences in peak width. But you can introduce an *equal-width* constraint into the fit by using peak shape 6 (Equal-width Gaussians) or peak shape 7 (Equal-width Lorentzian). Using peak shape 6 on the same set of data as the previous example:

```
>> [FitResults,MeanFitError]=peakfit([x' y'],5,10,2,6)
            Peak#  Position   Height    Width    Area
            1     4.0293     0.52818   1.5666   0.8808
            2     5.9965     1.0192    1.5666   1.6997
MeanFitError = 4.5588
```

This "equal width" fit forces all the peaks within one group to have exactly the same width, but that width is determined by the program from the data. The result is a *slightly higher* fitting error (in this case 4.5% rather than 4.4%), but - perhaps surprisingly - the peak parameter measurements are usually *more accurate* and *more reproducible* (Specifically, the relative standard deviations are on average lower for the equal-width fit than for an unconstrained-width fit to the same data, assuming of course that the true underlying peak widths are equal). *This is an exception* to the general expectation that lower fitting errors result in lower peak parameter errors. It is an illustration of the general rule that

the more you know about the nature of your signals, and the closer your chosen model adheres to that knowledge, the better the results. In this case you knew that the peak shape was Gaussian (although you could have verified that choice by trying other candidate peaks shapes). You determined that the number of peaks was 2 by inspecting the residuals and fitting errors for 1, 2, and 3 peak models. And then you introduced the constraint of equal peak widths within each group of peaks, based on prior knowledge of the experiment rather than on inspection of residuals and fitting errors. Here's another example, with real experimental data from a measurement where the *adjacent peak widths are expected to be equal*, showing the result of an unconstrained fit and an equal width fit. The fitting errors are slightly larger for the equal-width fit, but that is to be preferred in this case. *Not every experiment can be expected to yield peaks of equal width, but when it does, it is better to make use of that constraint.*

Fixed-width shapes. Going one step beyond *equal widths* (in peakfit version 7.6 and later), you can also specify a *fixed-width* shapes (shape numbers 11, 12, 34-37), in which the *widths of the peaks are known beforehand*, but are not necessarily equal, and are specified as a *vector* in input argument 10, one element for each peak, rather than being determined from the data as in the equal-width fit above. Introducing this constraint onto the previous example and supplying an accurate width as the 10th input argument: `peakfit([x' y'],0,0,2,11,0,0,0,0,[1.6661.666])`

```
      Peak#      Position      Height       Width        Area
        1         3.9943      0.49537       1.666      0.8785
        2         5.9924      0.98612       1.666      1.7488
MeanFitError = 4.8128
```

Comparing to the previous equal-width fit, the fitting error of 4.8% is larger here (because there are fewer degrees of freedom to minimize the error), but the parameter errors, particularly the peak heights, are *more accurate* because the width information provided in the input argument was more accurate (1.666) than the width determined by the equal-width fit (1.5666). Again, not every experiment yields peaks of known width, but when it does, it is better to make use of that constraint. For example, see Example 35 (page 411) and the Matlab/Octave script WidthTest.m (typical results for a Gaussian/ Lorentzian blend shape shown below, showing that the more constraints, the greater the fitting error but the lower the parameter errors, if the constraints are accurate).

Relative percent error	Fitting error	Position Error	Height Error	Width Error
Unconstrained shape factor and widths: shape 33	0.78%	0.39%	0.80%	1.66%
Fixed shape factor and variable widths: shape 13	0.79%	0.25%	1.3%	0.98%
Fixed shape factor and fixed widths: shape 35	0.8%	0.19%	0.695	0.0%

Multiple linear regression (peakfit version 9 or later). Finally, note that if the peak *positions* are also known, and only the peak *heights* are unknown, you do not even need to use the iterative fitting method at all. You can use the easier and faster *multilinear regression* technique (also called classical least-

squares, page 188), which is implemented by the function cls.m and by version 9 of peakfit.m as shape number 50. Although multilinear regression results in fitting error slightly *greater* (and R^2 lower), the errors in the measured peak heights are often *less*, as in this example from peakfit9demo.m, where the true peak heights of the three overlapping Gaussian peaks are 10, 30, and 20.

```
Multilinear regression results (known position and width):
           Peak    Position     Height       Width    Area
            1         400      9.9073         70     738.22
            2         500      29.995         85     2714
            3         560      19.932         90     1909.5
%fitting error=1.3048   R2= 0.99832   %MeanHeightError=0.427

Unconstrained iterative non-linear least-squares results:
           Peak   Position     Height      Width    Area
            1      399.7      9.7737      70.234    730.7
            2      503.12     32.262      88.217    3029.6
            3      565.08     17.381      86.58     1601.9
%fitting error=1.3008   R2= 0.99833   %MeanHeightError=7.63
```

This demonstrates dramatically how different measurement methods can *look* the same, and give fitting errors almost the same, and yet differ greatly in parameter measurement accuracy. (The similar script peakfit9demoL.m is the same thing using Lorentzian peaks).

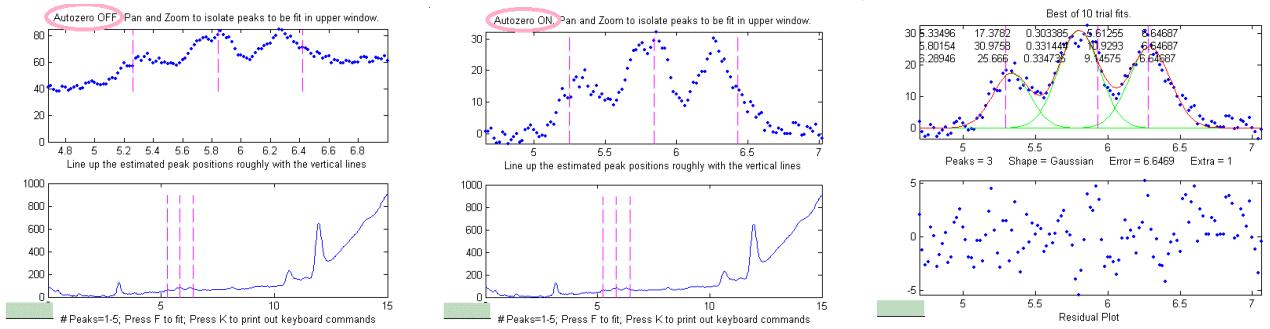
SmallPeak.m is a demonstration script comparing all these techniques applied to the challenging problem of measuring the height of a small peak that is closely overlapped with, and completely obscured by, a much larger peak. It compares unconstrained, equal-width, and fixed-position iterative fits (using peakfit.m) with a classical least-squares fit in which *only* the peak heights are unknown (using cls.m). It helps to spread out the four figure windows, so you can observe the dramatic difference in the stability of the different methods. A final table of relative percent peak height errors shows that *the more the constraints, the better the results* (but only if the constraints are *justified*). The real key is to know which parameters can be relied upon to be constant and which must be allowed to vary.

Here's a screen video (MorePeaksLowerFittingError.mp4) of a real-data experiment using the interactive peak fitter ipf.m (page 416) with a complex experimental signal in which several different fits were performed using models from 4 to 9 variable-width, equal-width, and fixed-width Gaussian peaks. The fitting error gradually *decreases from 11% initially to 1.4%* as more peaks are used, but *is that really justified?* If the objective is simply to get a good fit, for example to estimate the random noise in the signal, then do whatever it takes. But if the objective is to extract some useful information from the model peak parameters, then more specific knowledge about that experiment is needed: how many peaks are really expected. Are the peak widths really expected to be constrained? Note that in this case the residuals (bottom panel) are *not random* and always have a distinct "wavy" character, suggesting that the data *may have been smoothed* before curve fitting (which is not a good idea, as shown by this irreverent post: http://wmbriggs.com/blog/?p=195). Thus, there is a real possibility that some of those 9 peaks are simply "fitting the noise", as will be discussed further on page 291.

b. Background correction

The peaks that are measured in many scientific instruments are sometimes superimposed on a non-specific background or baseline. Ordinarily, the experimental protocol is designed to minimize the background or to compensate for the background, for example by subtracting a "blank" signal from the signal of an actual specimen. But even so, there is often a residual background that cannot be eliminated completely experimentally. The origin and shape of that background depend on the specific measurement method, but often this background is a broad, tilted, or curved shape, and the peaks of interest are comparatively narrow features superimposed on that background. In some cases, the baseline may be another interfering peak that overlaps the peaks of interest. The presence of the background has relatively little effect on the peak *positions*, but it is impossible to measure the peak heights, width, and areas accurately unless something is done to account for the background.

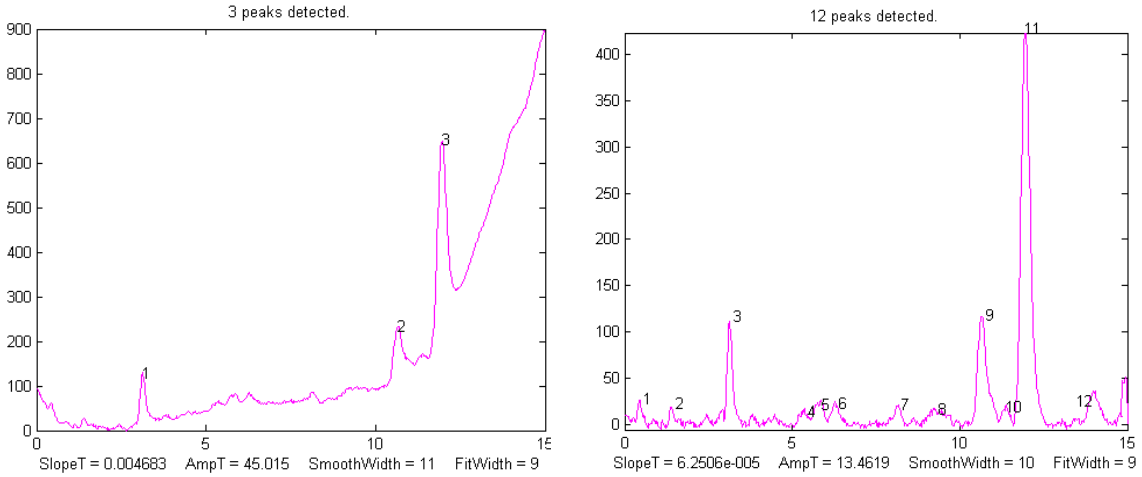
Various methods are described in the literature for estimating and subtracting the background in such cases. The simplest assumption is that the background can be approximated as a simple function in the local group of peaks being fit together, for example as a constant (flat), straight-line (linear) or curved line (quadratic). This is the basis of the "BaselineMode" modes in the ipf.m, iSignal.m, and iPeak.m functions, which are selected by the **T** key to cycle through *OFF*, *linear*, *quadratic*, *flat*, and *mode(y)* modes. In the *flat* mode, a constant baseline is included in the curve fitting calculation. In *linear* mode, a straight-line baseline connecting the two ends of the signal segment in the upper panel will be automatically subtracted *before the iterative curve fitting*. In *quadratic* mode, a parabolic baseline is subtracted. In the last two modes, you must adjust the pan and zoom controls to isolate the group of overlapping peaks to be fit, so that the signal returns to the local background at the left and right ends of the window. In the *mode(y)* mode, the most common value is subtracted from all points.



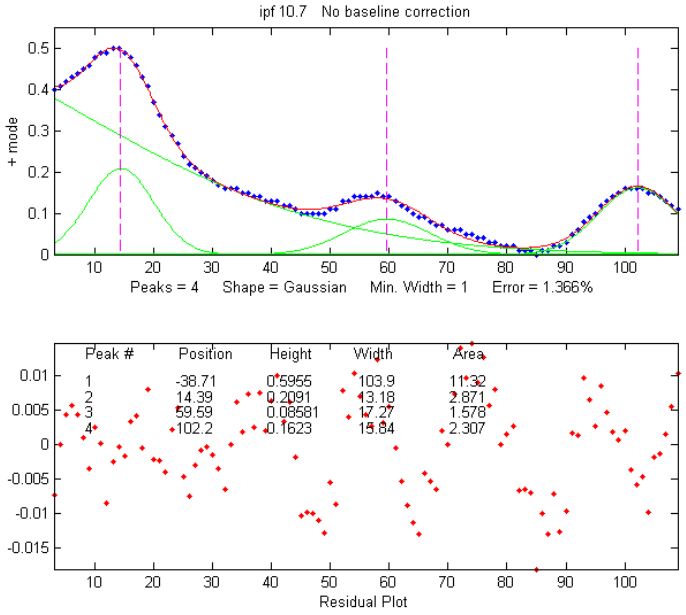
*Example of an experimental chromatographic signal in ipf.m. From left to right, (1) Raw data with peaks on a tilted baseline. The three weak peaks of interest are selected using the pan and zoom controls, adjusted so that the signal returns to the local background at the edges of the segment displayed in the upper window. (2) The linear baseline is subtracted when BaselineMode set to 1 in ipf.m by pressing the **T** key. (3) The selected region is fit with a three-peak Gaussian model, activated by pressing **3**, **G**, **F** (meaning **3** peaks, **G**aussian, **F**it). Press **R** to print out a peak table.*

Alternatively, it may be better to subtract the background from the *entire* signal first, before further operations are performed. As before, the simplest assumption is that the background is piece-wise linear, that is, can be approximated as a series of small straight-line segments. This is the basis of the multiple-point background subtraction mode in ipf.m, iPeak.m, and in iSignal. The user enters the number of points that is thought to be sufficient to define the baseline, then click where the baseline is thought to be along the entire length of the signal in the lower whole-signal display (e.g. on the valleys

between the peaks). After the last point is clicked, the program interpolates between the clicked points and subtracts the piece-wise linear background from the original signal.



From left to right, (1) Raw data with peaks superimposed on the baseline. (2) Background subtracted from the entire signal using the multipoint background subtraction function in iPeak.m *(ipf.m and iSignal.m have the same function).*



Sometimes, even without an actual baseline present, the peaks may overlap enough so that the signal never returns to the baseline, making it seem that there is a baseline to be corrected. This can occur especially with peaks shapes that have gradually sloping sides, such as the Lorentzian, as shown in this example. Curve fitting *without* baseline correction might work in that case.

In some cases, the background may be modeled as a broad peak whose maximum falls *outside* of the range of data acquired, as in the real-data example on the left. It may be possible to fit the off-screen peak simply by including *an extra peak in the model to account for the baseline*. In the example on the left, above, there are three clear peaks visible, superimposed on a tilted baseline. In this case, the signal was fit nicely with four, rather than three, variable-width Gaussians, with an error of only 1.3%. The additional broad Gaussian, with a peak at x = -38.7, serves as the baseline. (Obviously, you should not use the equal-width shapes for this, because the background peak is broader than the other peaks). Another real-data example exhibits four on-screen peaks of very different heights and widths on a broad baseline. Such a signal can be difficult to fit because the starting point for most iterative fits is that all peaks have about the same width. So, in some cases, assigning a custom "start" vector may be necessary. Using the ipf.m (page 417) 'C' and 'W' keys can help.

In another real-data example of an experimental spectrum, the linear baseline subtraction

("BaselineMode") mode described above is used in conjunction with a 5-Gaussian model, with one Gaussian component fitting the broad peak that may be part of the background and the other four fitting the sharper peaks. This fits the data very well (0.5% fitting error), but a fit like this can be difficult to get, because there are so many other solutions with slightly higher fitting errors. It may take several trials. It can help if you specify the *start values* for the iterated variables, rather than using the default choices. All the software programs described here have that capability.

The Matlab/Octave function peakfit.m will accept a peak shape that is a *vector of different shape numbers,* which can be useful for baseline correction. As an example, consider a weak Gaussian peak on a sloped straight-line baseline, using a 2-component fit with one Gaussian component and one variable-slope straight line ('slope', shape 26), specified by using the vector ([1 26]) as the shape argument:

```
x=8:.05:12;y=x+exp(-(x-10).^2);
[FitResults,GOF]= peakfit([x;y],0,0,2,[1 26],[1 1],1,0)

  Peak#    Position    Height     Width      Area
    1          10           1    1.6651    1.7642
    2       4.485     0.22297      0.05    40.045
GOF = 0.0928    0.9999
```

If the baseline seems to be curved rather than straight, you can model the baseline with a *quadratic* (shape 46) rather than a linear slope (peakfit *version 8* and later).

If the baseline seems to be different on either side of the peak, you can try to model the baseline with an *S-shape* (sigmoid), either an up-sigmoid, shape 10 (click for graphic), **`peakfit([x;y],0,0,2,[1 10],[0 0]`**, or a down-sigmoid, shape 23 (click for graphic), `peakfit`**`([x;y],0,0,2,[1 23],[0 0]`**, in these examples leaving the peak modeled as a Gaussian.

If the signal is very weak compared to the baseline, the fit can be helped by adding rough first guesses ('start') using the 'polyfit' function to generate automatic first guesses for the sloping baseline. For example, with *two* overlapping signal peaks and a 3-peak fit with peakshape=[1 1 26].

```
x=4:.05:16;
y=x+exp(-(x-9).^2)+exp(-(x-11).^2)+.02.*randn(size(x));
start=[8 1 10 1 polyfit(x,y,1)];
peakfit([x;y],0,0,3,[1 1 26],[1 1 1],1,start)
```

A similar technique can be employed in a spreadsheet, as demonstrated in CurveFitter2GaussianBaseline.xlsx (graphic).

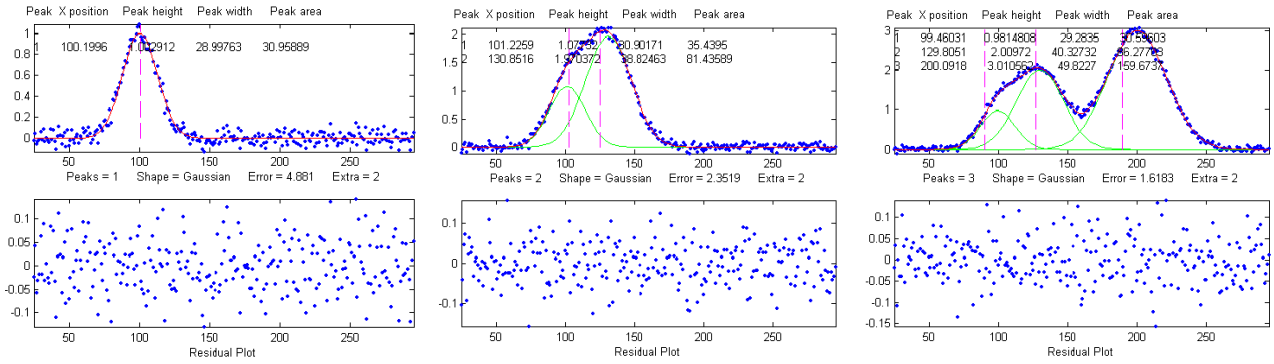
The downside to including the baseline as a variable component is that it increases the number of degrees of freedom, increases the execution time, and increases the possibility of unstable fits. Specifying start values can help.

c. Random noise in the signal.

Any experimental signal has a certain amount of random noise, which means that the individual data points scatter randomly above and below their true values. The assumption is ordinarily made that the scatter is equally above and below the true signal so that the long-term average approaches the true mean value. The noise "averages to zero" as it is often said. The practical problem is that any given

recording of the signal contains only one finite sample of the noise. If another recording of the signal is made, it will contain another independent sample of the noise. These noise samples are not infinitely long and therefore do not represent the true long-term nature of the noise. This presents two problems: (1) an individual sample of the noise will not "average to zero" and thus the parameters of the best-fit model will not necessarily equal the true values, and (2) the magnitude of the noise during one sample might not be typical. The noise might have been randomly greater or smaller than average during that time. This means that the mathematical "propagation of error" methods, which seek to estimate the likely error in the model parameters based on the noise in the signal, will be subject to error (*underestimating* the error if the noise happens to be *lower* than average and *overestimating* the errors if the noise happens to be *larger* than average).

A better way to estimate the parameter errors is to record multiple samples of the signal, fit each of those separately, compute the model parameters from each fit, and calculate the standard error of each parameter. However, if that is not practical, it is possible to simulate such measurements by adding random noise to a model with known parameters, then fitting that simulated noisy signal to determine the parameters, then repeating the procedure repeatedly with different sets of random noise. This is exactly what the script DemoPeakfit.m (which requires the peakfit.m function) does for simulated noisy peak signals such as those illustrated below. It is easy to demonstrate that, as expected, the average fitting error precision and the relative standard deviation of the parameters increases directly with the random noise level in the signal. But the precision and the accuracy of the measured parameters *also* depend on which parameter it is (peak positions are always measured more accurately than their heights, widths, and areas) and on the peak height and extent of peak overlap (the two left-most peaks in this example are not only weaker but also more overlapped than the right-most peak, and therefore exhibit poorer parameter measurements). In this example, the fitting error is 1.6% and the percent relative standard deviation of the parameters ranges from 0.05% for the peak position of the largest peak to 12% for the peak area of the smallest peak.

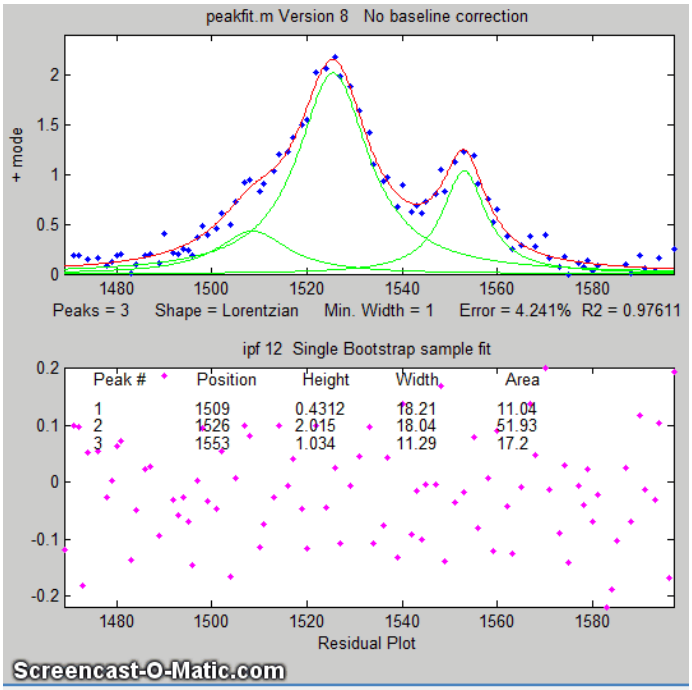


***Overlap matters**: The errors in the values of peak parameters measured by curve fitting depend not only on the characteristics of the peaks in question and the signal-to-noise ratio but also upon other peaks that are overlapping it. From left to right:*

***(a)** a single peak at x=100 with a peak height of 1.0 and width of 30 is fitted with a Gaussian model, yielding a relative fit error of 4.9% and relative standard deviation of peak position, height, and width of 0.2%, 0.95%, and 1.5%, respectively.*

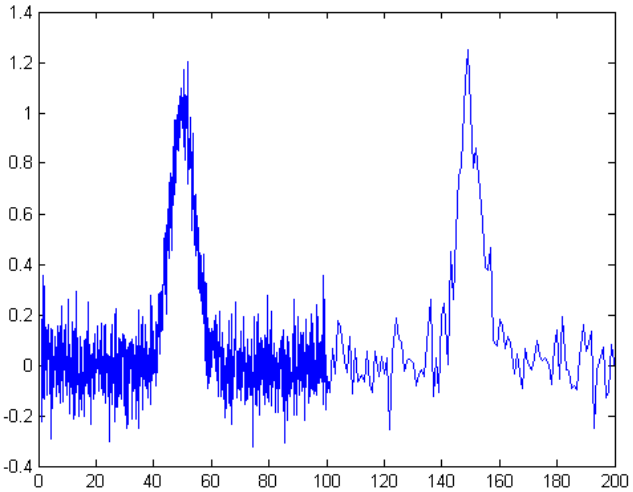
(b) *The same peak, with the same noise level but with another peak overlapping it, actually* ***reduces*** *the relative fit error to 2.4% (simply because the addition of the second peak increases overall signal amplitude). However, it* ***increases*** *the relative standard deviation of peak position, height, and width to 0.84%, 5%, and 4% -* ***a seemingly better fit, but with poorer precision*** *for the first peak.*

(3) *The addition of a third (larger but non-overlapping) peak reduces the fit error further to 1.6%, but the relative standard deviation of peak position, height, and width of the first peak are about the same as with two peaks, because* ***the third peak does not overlap*** *the first one significantly.*



If the average noise in the signal is not known or its probability distribution is uncertain, it is possible to use the bootstrap sampling method (page 170), illustrated by the animation on the left, to estimate the uncertainty of the peak heights, positions, and widths, as illustrated on the left and as described in detail on page 170. The keypress operated interactive function ipf.m (page 416) has this function, which is activated by the '**N**' key. Click on the figure to open a GIF animation shown on the left. Or press '**V**' to compute the expected standard deviations of the all the peak parameters using this method.

One way to reduce the effect of noise is to take more data. If the experiment makes it possible to reduce the x-axis interval between points, or to take multiple readings at each x-axis values, then the resulting increase in the number of data points in each peak should help reduce the effect of noise. As a demonstration, using the script DemoPeakfit.m to create a simulated overlapping peak signal like that shown above left, it is possible to change the interval between x values and thus the total number of data points in the signal. With a noise level of 1% and 75 points in the signal, the fitting error is 0.35 and the average parameter error is 0.8%. With 300 points in the signal and the same noise level, the fitting error is essentially the same, but the average parameter error drops to 0.4%, suggesting that the accuracy of the measured parameters varies inversely with the square root of the number of data points in the peaks.



The figure on the right illustrates the importance of sampling interval and data density. (You can download the data file "udx" in TXT format or in

Matlab MAT format). The signal consists of two Gaussian peaks, one located at x=50 and the second at x=150. Both peaks have a peak height of 1.0 and a peak half-width of 10. Normally-distributed random white noise with a standard deviation of 0.1 has been added to the entire signal. The *x-axis sampling interval*, however, is different for the two peaks. It is 0.1 for the first peak and 1.0 for the second peak. This means that the first peak is characterized by ten times more points than the second peak. When you fit these peaks separately to a Gaussian model (e.g., using peakfit.m or ipf.m), you will find that all the parameters of the first peak are measured more accurately than the second, even though the fitting error is not much different:

First peak:

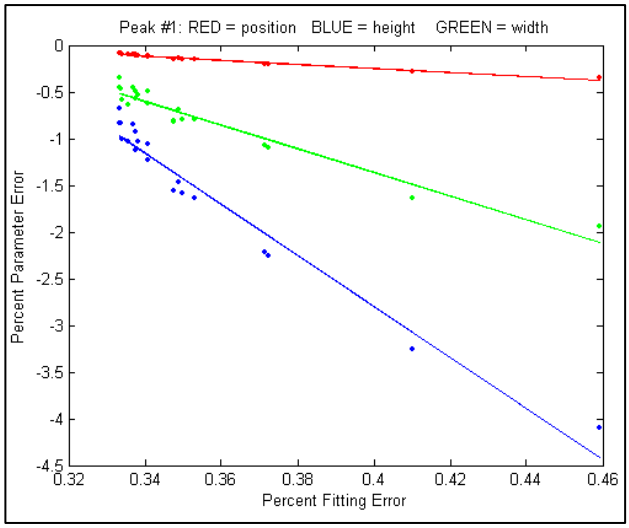
```
Percent Fitting Error=7.6434%
Peak#   Position   Height    Width
1         49.95    1.0049    10.111
```

Second peak:

```
Percent Fitting Error=8.8827%
Peak#   Position    Height    Width
1       149.64     1.0313     9.941
```

Noise color. So far, this discussion has applied to *white* noise. But other noise colors (page 27) have different effects. Low-frequency weighted ("pink") noise has a *greater* effect on the accuracy of peak parameters measured by curve fitting, and, in a nice symmetry, high-frequency "blue" noise has a *smaller* effect on the accuracy of peak parameters that would be expected based on its standard deviation. This is because the information in a smooth peak signal is concentrated at low frequencies. An example of this occurs when you apply curve fitting is to a signal that has been deconvoluted (page 113) to remove a broadening effect. This is the reason smoothing before curve fitting does not help (page 233) because the peak signal information is concentrated in the *low*-frequency range but smoothing reduces mainly the noise in the *high*-frequency range.

Sometimes you may notice that the residuals in a curve-fitting operation are structured into bands or lines rather than being completely random. This can occur if either the independent variable or the dependent variable is *quantized* into discrete steps rather than continuous. It may look strange, but it has little effect on the results if the random noise is larger than the steps. When there is noise in the data (in other words, pretty much always), the exact results will depend on the region selected for the fit - for example, the results will vary slightly with the pan and zoom setting in ipf.m, and the more noise, the greater the effect.



d. Iterative fitting errors

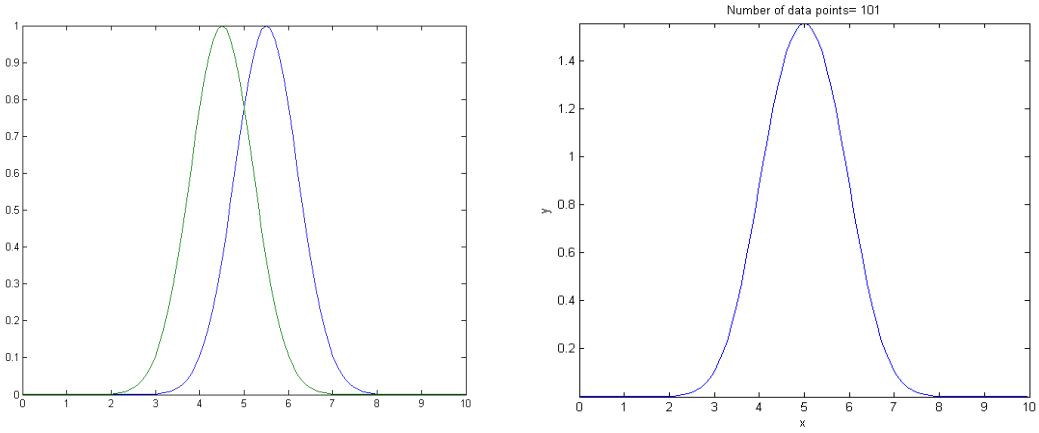
Unlike multiple linear regression, curve fitting, iterative methods may not always converge on the exact same model parameters each time the fit is repeated with slightly different starting values (first guesses). The Interactive Peak Fitter ipf.m (page 416) makes it easy to test this, because it uses slightly different starting values each time the signal is fit (by pressing the **F** key in ipf.m, for example). Even better, by pressing the **X** key, the ipf.m function silently computes 10 fits with different starting values and takes the one with the lowest fitting error. A basic assumption of any curve

fitting operation is that the fitting error (the root-mean-square difference between the model and the data) is minimized. The parameter errors (the difference between the actual parameters and the parameters of the best-fit model) will also be minimized. This is generally a good assumption, as demonstrated by the graph above which shows typical percent parameters errors as a function of fitting error for the left-most peak in one sample of the simulated signal generated by DemoPeakfit.m (shown in the previous section). The variability of the fitting error here is caused by random small variations in the first guesses, rather than by random noise in the signal. In many practical cases, there is enough random noise in the signals that the iterative fitting errors within one sample of the signal are small compared to the random noise errors between samples. Remember that the variability in measured peak parameters from fit to fit of a single sample of the signal is not a good estimate of the precision or accuracy of those parameters, for the simple reason that those results represent only one sample of the signal, noise, and background. The sample-to-sample variations are likely to be much greater than the within-sample variations due to the iterative curve fitting. (In this case, a "sample" is a single recording of signal). To estimate the contribution of random noise to the variability in measured peak parameters when only a single sample if the signal is available, the bootstrap method can be used (page 170).

Selecting the optimum data region of interest. When you perform a peak fitting using ipf.m (page 416), you have control over data region selected by using the pan and zoom controls (or, using the command-line function peakfit.m, by setting the “center” and “window” input arguments). Changing these settings usually changes the resulting fitted peak parameters. If the data were perfect, say, a mathematically perfect peak shape with no random noise, then the pan and zoom settings would make no difference at all. You would get the exact same values for peak parameters at all settings, assuming only that the model you are using matches the actual shape. But of course, in the real world, data are never mathematically perfect and noiseless. The greater the amount of random noise in the data, or the greater the discrepancy between your data and the model you select, the more the measured parameters will vary if you fit different regions using the pan and zoom controls. This is simply an indication of the unavoidable uncertainty in the measured parameters.

A difficult case.

As a dramatic example of the ideas in the previous sections, consider this simulated example signal above. This consists of two Gaussian peaks of equal height = 1.00, shown separately on the left, that overlap closely enough so that their sum, shown on the right, is a single symmetrical peak that looks very much like a single Gaussian.

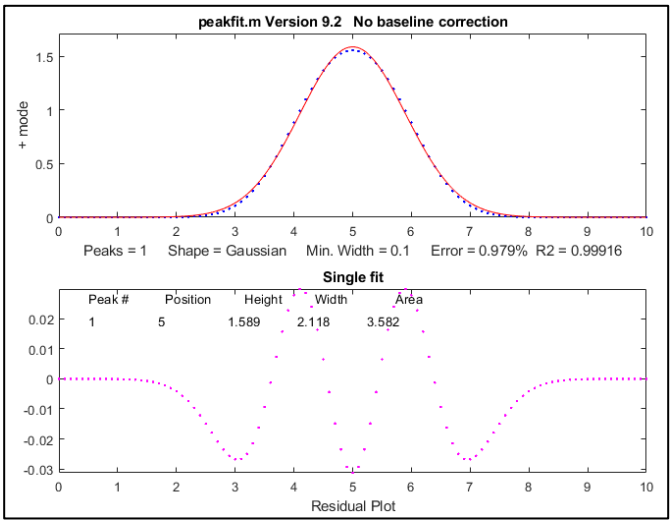



```
>> x=[0:.1:10]';
>> y=exp(-(x-5.5).^2)+exp(-(x-4.5).^2);
```

Attempts to fit this with a *single* Gaussian (shown in the graph below) yield a fit with roughly a fairly low 1% fitting error. But the residual is noticeably wavy and smooth, suggesting that there is little or no random noise in the data but that the *model is not right*.

```
>> peakfit([x y],5,19,1,1)
          Peak#    Position    Height      Width     Area
            1       4.5004     1.001      1.6648    1.773
            2       5.5006     0.99934    1.6641    1.770
```

If there were no noise in the signal, the iterative curve fitting (peakfit.m or ipf.m) routines could easily extract the two equal Gaussian components to an accuracy of 1 part in 1000. But in the presence of



even a little noise (for example, 1% RSD), the results are uneven. One peak is almost always significantly higher than the other:

```
>> y=exp(-(x-5.5).^2)+exp(-(x-4.5).^2)+.01*randn(size(x))
>> peakfit([x y],5,19,2,1)

      Peak#    Position     Height      Width     Area
        1       4.4117     0.83282     1.61      1.43
        2       5.4022     1.1486      1.734     2.12
```

The fit is stable with any one sample of noise, if peakfit.m is run again with slightly different starting values, for example by pressing the **F** key several times in ipf.m (page 416). So, the problem is *not* iterative fitting errors caused by different starting values. The problem is the *noise*: although the signal is completely symmetrical, any sample of the noise is slightly asymmetrical (e.g., the first half of the noise invariably averages either slightly higher or slightly lower than the second half, resulting in an asymmetrical fit result). The surprising thing is that the error in the peak heights is much larger (about 15% relative, on average) than the random noise in the data (1% in this example). So even though *the fit looks good* - the fitting error is low (less than 1%) and the residuals are random and unstructured - *the model parameters can still be very far off.* If you were to simulate another measurement (i.e., generate another independent set of noise), the results would be different but still inaccurate (the first peak has an equal chance of being larger or smaller than the second). Unfortunately, the expected error is not accurately predicted by the bootstrap method (page 170), which seriously underestimates the standard deviation of the peak parameters with repeated measurements of independent signals (because a bootstrap sub-sample of asymmetrical noise is likely to remain asymmetrical). A Monte Carlo simulation (page 169) would give a more reliable estimation of uncertainty in such cases.

Better results can be obtained in cases where the peak widths are expected to be equal, in which case you can use peak shape 6 (equal-width Gaussian) instead of peak shape 1:

```
peakfit([x y],5,19,2,6)
```

It also helps to provide decent first guesses (start) and to set the number of trials (NumTrials) to a number above 1):

```
peakfit([x,y],5,10,2,6,0,10,[4 2 5 2],0,0)
```

The best case will be if the shape, position, and width of the two peaks are known accurately, and if the *only* unknown is their heights. Then the Classical Least-squares (multiple regression) technique can be employed and the results will be much better.

For an even more challenging example like this, where the two closely overlapping peaks are very different in terms of height, see page 322.

So, to sum up, we can make the following observations about the accuracy of model parameters.

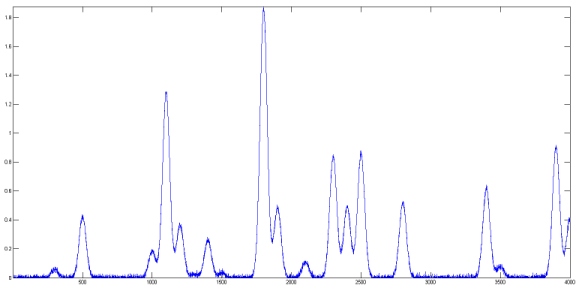
(1) The parameter errors depend on the accuracy of the model chosen and on number of peaks.
(2) The parameter errors are directly proportional to the noise in the data (and worse for low-frequency or pink noise).
(3) All else being equal, parameter errors are proportional to the fitting error, but a model that fits the underlying reality better, e.g., equal or fixed widths or shapes, often gives lower parameter errors even if the fitting error is larger.
(4) The errors are typically least for peak position and worse for peak width and area.
(5) The errors depend on the *data density* (number of independent data points in the width of each peak) and on the *extent of peak overlap* (the parameters of isolated peaks are easier to measure than highly overlapped peaks).
(6) If only a single signal is available, the effect of noise on the standard deviation of the peak

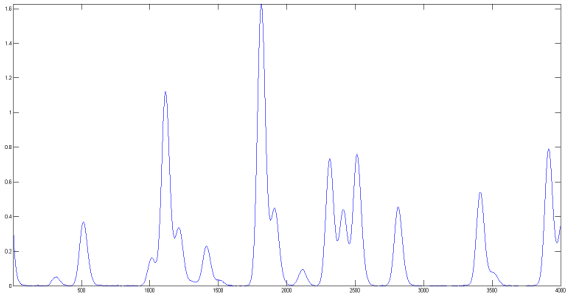
parameters in many cases can be predicted approximately by the bootstrap method, but if the overlap of the peaks is too great, the error of the parameter measurements can be much greater than predicted.

Sometimes curve fitting is complicated if the peaks are *asymmetrical* (wider on one side than the other). AsymmetricalOverlappingPeaks.m illustrates one way to deal with the problem of excessive peak overlap in a multi-step script that uses first-derivative symmetrization as a pre-process performed before iterative least-squares curve fitting to analyze a complex signal consisting of multiple asymmetric overlapping peaks. See page 366 for details. Or you can use an asymmetrical peak model, as described in the next section.

Fitting signals that are subject to exponential broadening.

DataMatrix2 (below left) is a computer-generated test signal consisting of 16 symmetrical Gaussian peaks with random white noise added. The peaks occur in groups of 1, 2, or 3 overlapping peaks, but the peak maxima are located at exactly integer values of x from 300 to 3900 (on the 100's) and the peak widths are always exactly 60 units. The peak heights vary from 0.06 to 1.85. The standard deviation of the noise is 0.01. You can use this signal to test curve-fitting programs and to determine the accuracy of their measurements of peak parameters. Right-click and select "Save" to download this signal, put it in

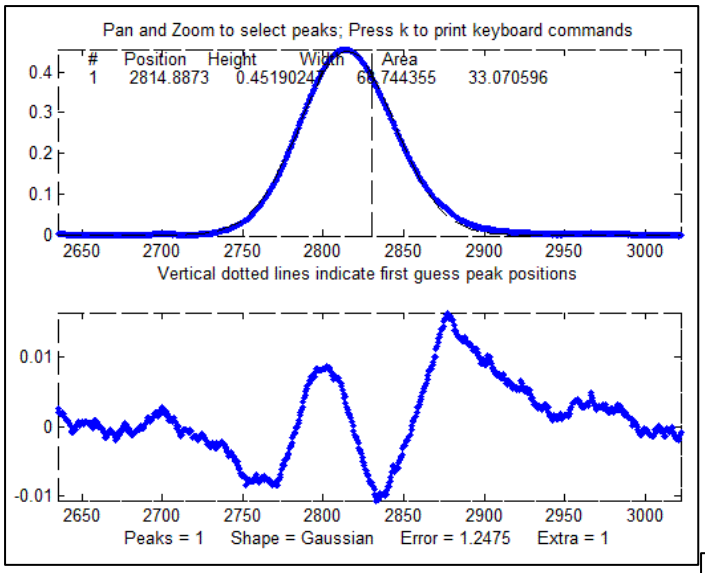




the Matlab search path, then type "load DataMatrix2" at the command prompt to load it into the Matlab workspace. DataMatrix3 (above right) is an exponentially broadened version of DataMatrix2, with a "decay constant", also called "time constant", of 33 points on the x-axis. The result of the exponential broadening is that all the peaks in this signal are asymmetrical, their peak maxima are shifted to longer x values, and their peak heights are smaller, and their peak widths are larger than the corresponding peaks in DataMatrix2. Also, the random noise is damped in this signal compared to the original and is no longer "white", as a consequence of the broadening. This type of effect is common in physical measurements and often arises from some physical or electrical effect in the measurement system that is apart from the fundamental peak characteristics. In such cases, you may wish to compensate for the effect of the broadening, either by deconvolution or by curve fitting, in an attempt to measure *what the peak parameters would have been before the broadening* (and also to measure the broadening itself). This can be done for Gaussian peaks that are exponentially broadened by using the "ExpGaussian" peak shape in peakfit.m and ipf.m (page 416), or the "ExpLorentzian", if the underlying peaks are Lorentzian. Right-click and select "Save" to download this signal, put it in the Matlab search path, then type "load DataMatrix3" to load it into the Matlab workspace. The example illustrated on the right

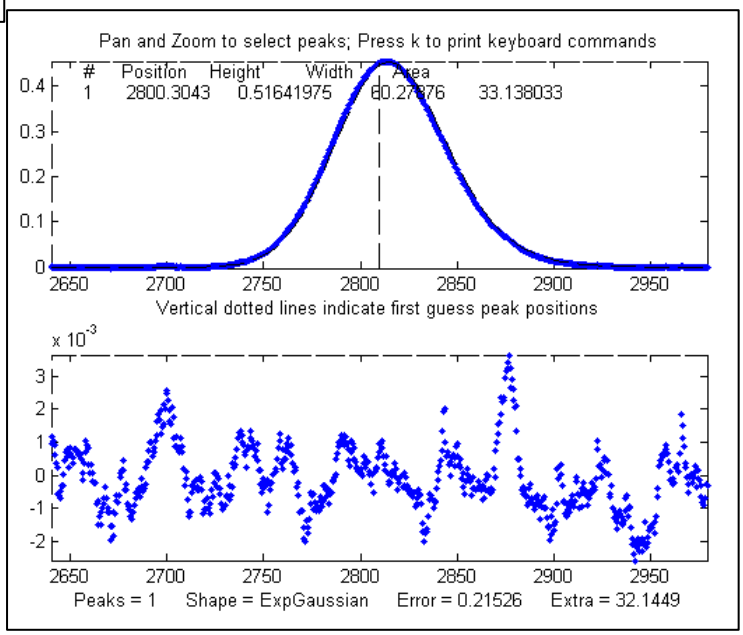
focuses on the single isolated peak whose "true" peak position, height, width, and area in the original unbroadened signal, are 2800, 0.52, 60, and 33.2 respectively. (The relative standard deviation of the noise is 0.01 / 0.52=2%.) In the broadened signal, the peak is visibly asymmetrical, the peak maximum is shifted to larger x values, and it has a shorter height and larger width, as demonstrated by the attempt to fit a normal (symmetrical) Gaussian to the broadened peak. (The peak *area*, on the other hand, is not much affected by the broadening).

```
>> load DataMatrix3
>> ipf(DataMatrix3);
Peak Shape = Gaussian
BaselineMode ON
Number of peaks = 1
Fitted range = 2640 - 2979.5 (339.5)   (2809.75)
Percent Error = 1.2084
Peak#    Position    Height     Width      Area
  1      2814.832   0.451005   68.4412   32.8594
```



The large "wavy" residual in the plot above is a tip-off that the model is not quite right. Moreover, the fitting error (1.2%) is larger than expected for a peak with a half-width of 60 points and a 2% noise RSD (approximately 2/sqrt(60) = 0.25%).

Fitting to an exponentially-broadened Gaussian (pictured on the right) gives a much lower fitting error ("Percent error") and a more nearly random residual plot. But the interesting thing is that it also *recovers the original (pre-broadening) peak position, height, and width to an accuracy of a fraction of 1%*, if that is of interest to you. In performing this fit, the decay constant ("extra") was experimentally determined from the broadened signal by adjusting it with the A and Z keys to give the lowest fitting error. That also results in a reasonably good measurement of the broadening factor (32.6, vs the actual value of 33). Had the original signal been noisier, these measurements would not be so accurate. Note: When using peakshape 5 (fixed decay constant exponentially broadened Gaussian) you must give it a reasonably good value for the decay constant ('extra'), the input argument right after the peakshape number. If the value is too far off, the fit



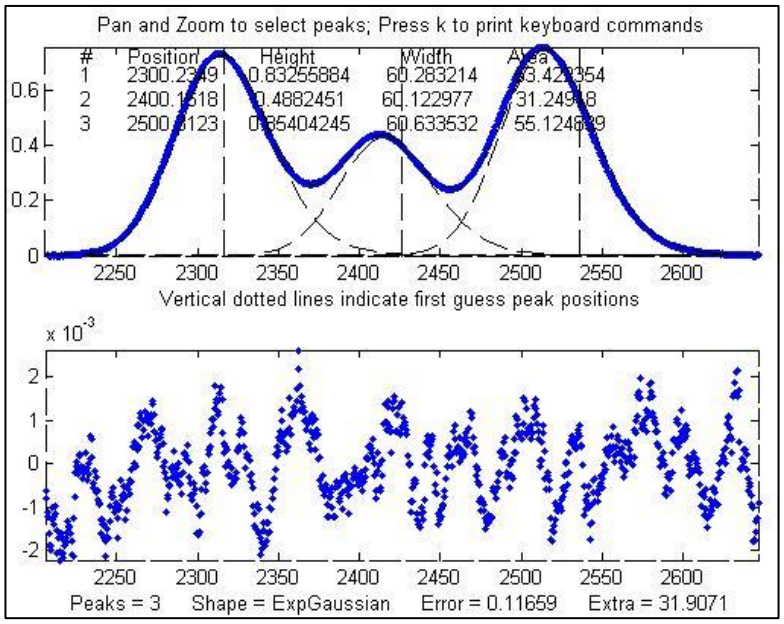
may fail completely, returning all zeros. A little trial and error suffice. Alternatively, you can use the simple *first-derivative addition technique* (page 83) to get a good estimate of the time constant before curve-fitting. Also, peakfit.m version 8.4 has two forms of unconstrained variable decay constant exponentially broadened Gaussian, shape numbers 31 and 39, that will *measure* the decay constant as an iterated variable. Shape 31 (expgaussian.m) creates the shape by performing a Fourier convolution of a specified Gaussian by an exponential decay of specified decay constant, whereas shape 39 (expgaussian2.m) uses a mathematical expression for the final shape so produced. Both result in the *same peak shape* but are parameterized differently. Shape 31 reports the peak height and position as that of the *original* Gaussian before broadening, whereas shape 39 reports the peak height of the broadened *result*. Shape 31 reports the width as the FWHM (full width at half maximum) and shape 39 reports the standard deviation (sigma) of the Gaussian. Shape 31 reports the exponential factor and the *number of data points,* and shape 39 reports the *reciprocal of the decay constant* in time units. (See the script DemoExpgaussian.m for a more detailed numerical example). For multiple-peak fits, both shapes usually require a reasonable first guess ("start") vector for best results, which you can determine automatically from a preliminary fit with a simple Gaussian model (as in this example). If the exponential decay constant of each peak is expected to be different and you need to measure those values, use shapes 31 or 39, but if the decay constant of all the peaks is expected to be the same, use shape 5, and determine the decay constant by fitting an isolated peak. For example:

```
Peak Shape = Exponentially-broadened Gaussian
BaselineMode ON
Number of peaks = 1
Extra = 32.6327
Fitted range = 2640 - 2979.5 (339.5)   (2809.75)
Percent Error = 0.21696

Peak#     Position    Height      Width       Area
  1       2800.130  0.518299   60.08629   33.152429
```

Comparing the two methods, the exponentially broadened Gaussian fit recovers all the underlying peak parameters quite accurately:

	Position	Height	Width	Area
Actual peak parameters	2800	0.52	60	33.2155
Gaussian fit to broadened signal	2814.832	0.45100549	68.441262	32.859436
ExpGaussian fit to broadened signal	2800.1302	0.51829906	60.086295	33.152429



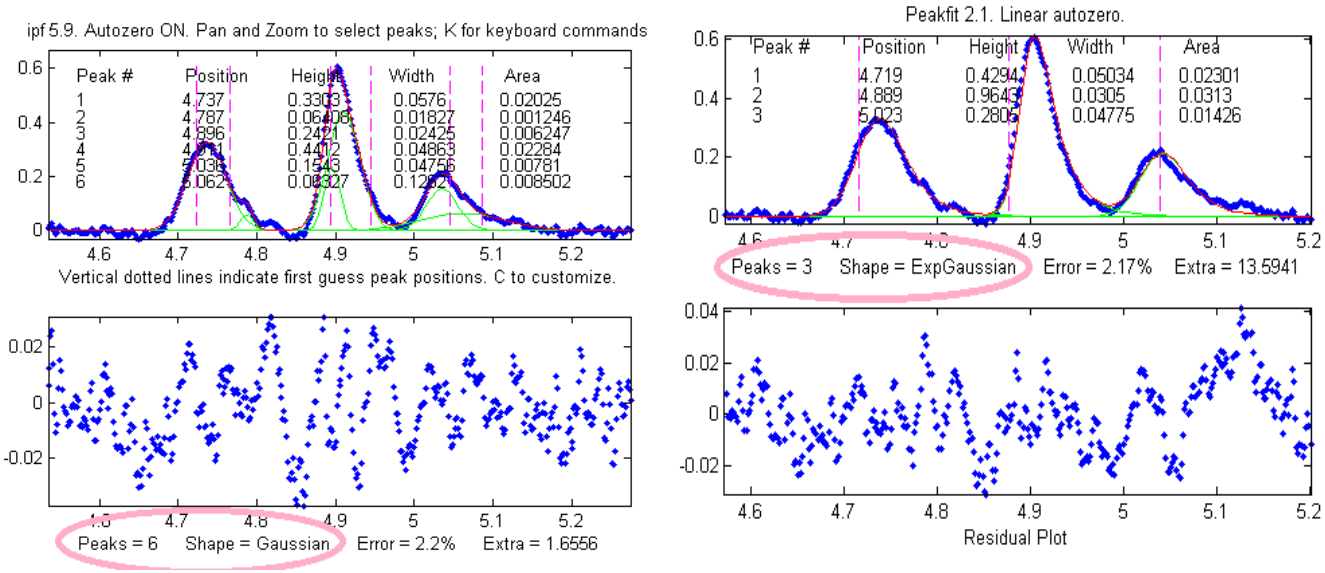
Other peaks in the same signal, under the broadening influence of the same decay constant, can be fitted with similar settings, for example, the set of three overlapping peaks near x=2400. As before, the peak positions are recovered almost exactly and even the width measurements are reasonably accurate (1% or better). If the exponential broadening decay constant is *not* the same for all the peaks in the signal, for example, if it gradually increases for larger x values, then the decay constant setting can be optimized for each group of peaks. The smaller fitting error evident here is just a reflection of the larger peak heights in this group of peaks - the noise is the same everywhere in this signal.

```
Peak Shape = Exponentially-broadened Gaussian
BaselineMode OFF
Number of peaks = 3
Extra = 31.9071
Fitted range = 2206 - 2646.5 (440.5)   (2426.25)
Percent Error = 0.11659

Peak# Position    Height       Width        Area
1     2300.2349   0.83255884   60.283214    53.422354
2     2400.1618   0.4882451    60.122977    31.24918
3     2500.3123   0.85404245   60.633532    55.124839
```

The residual plots in both examples still have some "wavy" character, rather than being completely random and "white". The exponential broadening smooths out any white noise in the original signal that is introduced *before* the exponential effect, acting as a low-pass filter in the time domain and resulting in a low-frequency dominated "pink" noise, which is what remains in the residuals after the broadened peaks have been fit as well as possible. On the other hand, white noise that is introduced *after* the exponential effect would continue to appear white and random on the residuals. In real experimental data, both types of noise may be present in varying amounts.

One final caveat: peak asymmetry such as exponential broadening could possibly be the result of a pair of closely spaced peaks of different peak heights. In fact, a single exponential broadened Gaussian peak can sometimes be fitted with two symmetrical Gaussians to a fitting error at least as low as a single exponential broadened Gaussian fit. This makes it hard to distinguish between these two models based on fitting error alone. However, you can decide that by inspecting the other peaks in the signal:



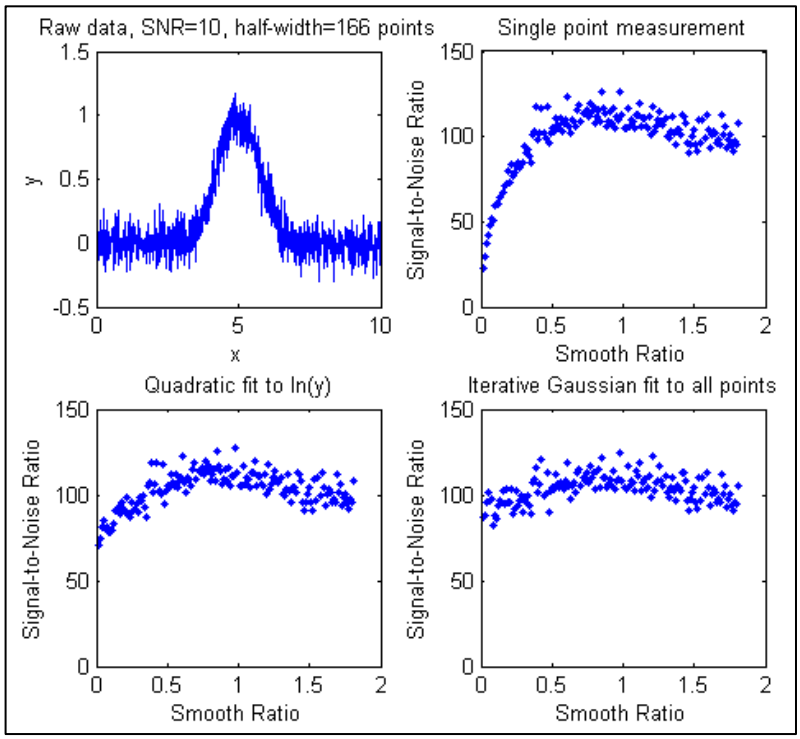
in most experiments, exponential broadening applies to every peak in the signal, and the broadening is either constant or changes gradually over the length of the signal. If only one or a few of the peaks exhibit asymmetry, and the others are symmetrical, it is most likely that the asymmetry is due to closely spaced peaks of different peak heights. If *all* peaks have the same or similar asymmetry, it is more likely to be a broadening factor that applies to the entire signal. The two figures on the previous page provide an example from *real experimental data*. On the left, three asymmetrical peaks are each fitted with two *symmetrical* Gaussians (six peaks total). On the right, those same three peaks are fitted with one *exponentially broadened* Gaussian each (three peaks total). In this case, the three asymmetrical peaks all have the same asymmetry and can be fitted with the same decay constant ("extra"). Moreover, the fitting error is slightly lower for the three-peak exponentially broadened fit. *Both observations argue for the three-peak exponentially broadened fit rather than the six-peak fit.*

Note: if your peaks are trailing off to the left, rather than to the right as in the above examples, simply use a *negative* value for the decay constant. To do that in ipf.m (page 416), press Shift-**X** and type a negative value.

An alternative to this type of curve fitting for exponentially broadened peaks is to use the first-derivative addition technique (page 83) to remove the asymmetry and then fit the resulting peak with a symmetrical model. This is faster in terms of computer execution time, especially for signals with many peaks, but it requires that the exponential time constant is known or estimated experimentally beforehand.

The Effect of Smoothing before least-squares analysis

In general, it is not advisable to smooth a signal before applying least-squares fitting, because doing so



might distort the signal, can make it hard to evaluate the residuals properly, and might bias the results of bootstrap sampling estimations of precision, causing it to underestimate the effect of noise on variations in peak parameters (page 170). SmoothOptimization.m is a Matlab/Octave script that compares the effect of smoothing on the measurements of peak height of a Gaussian peak with a half-width of 166 points, plus white noise with a signal-to-noise ratio (SNR) of 10. The script uses three different methods:

(a) simply taking the single point at the center of the peak as the peak height,
(b) using the "gaussfit" method to fit the top half of the peak (see page 172), and
(c) fitting the entire signal with a Gaussian using the iterative method.

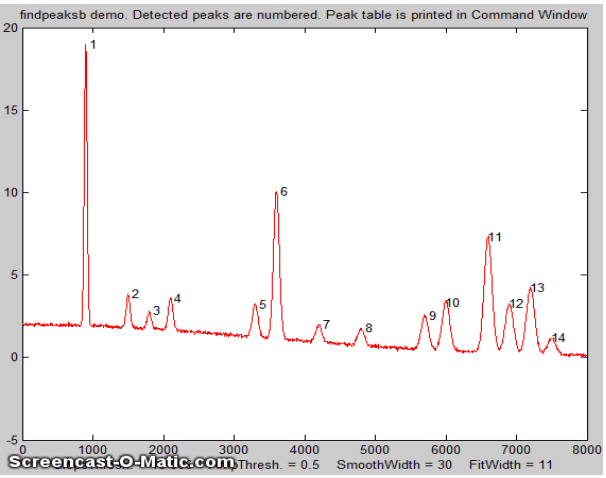
The results of 150 trials with independent white noise samples are shown above: a typical raw signal is shown in the upper left. The other three plots show the effect of the SNR of the measured peak height vs the smooth ratio (the ratio of the smooth width to the half-width of the peak) for those three measurement methods. The results show that the simple single-point measurement is indeed much improved by smoothing, as is expected. However, the optimum SNR (which improves by roughly the square root of the peak width of 166 points) is achieved only when the smooth ratio approaches 1.0, and that much smoothing distorts the peak shape significantly, reducing the peak height by about 40%. The curve-fitting methods are much less affected by smoothing and the iterative method hardly at all. So, the bottom line is that you should *not* smooth data prior to curve-fitting, because it will distort the peak and will not gain any significant SNR advantage. The only situations where it might be advantageous so smooth before fitting are:

(a) when the noise in the signal is high-frequency weighted (i.e. "blue" noise), where low-pass filtering will make the peaks easier to see for the purpose of setting the starting points for an iterative fit, or

(b) if the signal is contaminated with high-amplitude narrow spike artifacts, in which case a median-based pre-filter or another spike killer function, can remove the spikes without much change to the rest of the signal. (Or, in another situation altogether, if you want to fit a curve joining the successive peaks of a modulated wave, called the "envelope", then you can smooth the absolute value of the wave before fitting the envelope).

Peak Finding and Measurement

A requirement in scientific data processing is to detect peaks in a signal and to measure their positions, heights, widths, and/or areas. One way to do this is to make use of the fact that the first derivative of a peak has a downward-going zero-crossing at the peak maximum (page 67). However, the presence of random noise in real experimental signal will cause many false zero-crossing simply due to the noise. To avoid this problem, the technique described here smooths the first derivative of the signal, then looks for downward-going zero-crossings, and then it takes only those zero-crossings whose slope exceeds a certain predetermined minimum (called the "*slope threshold*") at a point where the original signal exceeds a certain minimum (called the "*amplitude threshold*"). By adjusting the smooth width, slope threshold, and amplitude threshold, it is possible to detect only the desired peaks and ignore most peaks that are too small, too wide, or too narrow. Moreover, this technique can be extended to estimate the position, height, and width of each peak by least-squares curve-fitting of a segment of the *original unsmoothed signal* in the vicinity of the zero-crossing. Thus, even if heavy smoothing of the first derivative is necessary to provide reliable discrimination against noise peaks, the peak parameters extracted by curve fitting are *not distorted by the smoothing*, and the effect of random noise in the signal is reduced by curve fitting over multiple data points in the peak. This technique can measure peak positions and heights quite accurately, but the measurements of peak widths and areas are most accurate if the peaks are Gaussian in shape (or Lorentzian, in the variant findpeaksL). For the most accurate measurement of other peak shapes, or of highly overlapped peaks, or of peak superimposed on a baseline, the related functions findpeaksb.m, findpeaksb3.m, findpeaksfit.m utilize non-linear iterative curve fitting with selectable peak shape models and baseline correction modes.



The routine is now available in several different versions that are described below:

(1) a set of command-line functions for Matlab or Octave, each linked to its description: peaksat.m, findpeaksx, findpeaksxw, findpeaksG, findpeksGw, (Wavelet-based denoise, requires the "wdenoise" function found in the Wavelet Toolbox), findvalleys, findpeaksL, measurepeaks, findpeaksGd,

findpeaksb, findpeaksb3, findpeaksplot, findpeaksplotL, peakstats, findpeaksE, findpeaksGSS, findpeaksLSS, findpeaksT, findpeaksfit, autofindpeaks, and autopeaks. These can be used as components in creating your own custom scripts and functions. Do not confuse with the "findpeaks" function in Matlab's Signal Processing Toolbox - that's a completely different algorithm.

(2) an interactive keypress-operated function, called ***iPeak*** (ipeak.m or ipeakoctave), page 253, for adjusting the peak detection criteria interactively to optimize for any particular peak type *iPeak* runs in the Figure window and use a simple set of keystroke commands to reduce screen clutter, minimize overhead, and maximize processing speed.

(3) A set of spreadsheets, available in *Excel* and in *OpenOffice* formats.
(4) *Real-time* peak detection in Matlab is discussed on page 347.

Click here to download the ZIP file "PeakFinder.zip", which includes findpeaksG.m and its variants, ipeak.m, and a sample data file and demo scripts for testing. You can also download *iPeak* and other programs of mine from the Matlab File Exchange.

Simple peak detection

allpeaks.m. `P=allpeaks(x,y)` A super-simple peak detector for x,y, data sets that lists *every y* value that has lower *y* values on both sides. A related version, allpeaksw.m, also estimates the width of the peaks. allvalleys.m lists every *y* value that has *higher y* values on both sides. Type "help allpeaks" or "help allpeaksw" for an example. “help allpeaks” or “help allpeaksw” to see an example of its application.

peaksat.m. (Peaks **A**bove **T**hreshold) Syntax: `P=peaksat(x,y,threshold)` . This function detects every y value that has lower y values on both sides *and* is above the specified threshold. Returns a matrix P with the x and y values of each peak, where *n* is the number of detected peaks. A related version, peaksatw.m, also estimates the width of the peaks. Type "help peaksat" or "help peaksatw" for an example. Type “help peaksat” or “help peaksatw” to see an example of its application.

peaksatG.m. ("Peaks **A**bove **T**hreshold/**G**aussian") `P=peaksatG(x,y,threshold,peakgroup` This function is like peakat.m but in addition it also least-squares fits the top of each detected peak with a Gaussian to estimate their width and area. The number of points at the top of the peak that are fit is determined by the input argument "peakgroup". Returns a 5 by n matrix P with the x and y values of each peak, where n is the number of detected peaks. Type "help peaksatG" for examples.

These simple functions do not have any internal data smoothing. If the data are noisy, smoothing can be applied separately beforehand to prevent the detection of noise peaks (see page 42). The following functions, on the other hand, do apply smoothing before peak detection.

findpeaksx.m is a Matlab/Octave command-line function to *locate and count* the positive peaks in noisy data sets.

`P=findpeaksx(x, y, SlopeThreshold, AmpThreshold, SmoothWidth, PeakGroup, smoothtype)`

It detects peaks by looking for downward zero-crossings in the smoothed first derivative that exceed SlopeThreshold and peak amplitudes that exceed AmpThreshold and returns a list (in matrix **P**) containing the peak number and the measured position and height of each peak (and for the variant findpeaksxw, the full width at half maximum, determined by calling the halfwidth.m function).

It can find and count over 10,000 peaks per second in very large signals. The data are passed to the findpeaksx function in the vectors x and y (x = independent variable, y = dependent variable). The other parameters are user-adjustable:

SlopeThreshold - Slope of the smoothed first derivative that is taken to indicate a peak. This discriminates based on peak width. Larger values of this parameter will neglect the broad features of the signal. A reasonable initial value for Gaussian peaks is 0.7*WidthPoints^-2, where WidthPoints is the *number of data points* in the half-width (FWHM) of the peak.

AmpThreshold - Discriminates based on peak height. Any peaks with height less than this value are ignored.

SmoothWidth - Width of the smooth function that is applied to data before the slope is measured. Larger values of SmoothWidth will neglect small, sharp features. A reasonable value is typically about equal to 1/2 of *the number of data points* in the half-width of the peaks.

PeakGroup - The number of data points around the "top part" of the (unsmoothed) peak that are taken to estimate the peak heights. If the value of PeakGroup is 1 or 2, the maximum y value of the 1 or 2 points at the point of zero-crossing is taken as the peak height value. If PeakGroup **n** is 3 or greater, the *average* of the next **n** points is taken as the peak height value. For spikes or very narrow peaks, keep PeakGroup=1 or 2; for broad or noisy peaks, make PeakGroup larger to reduce the effect of noise.

Smoothtype determines the smoothing algorithm (page 42)
If smoothtype=1, rectangular (sliding-average or boxcar)
If smoothtype=2, triangular (2 passes of sliding-average)
If smoothtype=3, p-spline (3 passes of sliding-average)

Basically, higher values yield a greater reduction in high-frequency noise, at the expense of slower execution. For a comparison of these smoothing types, see page 60.

The demonstration scripts **demofindpeaksx.m** and **demofindpeaksxw.m** finds, numbers, plots, and measures noisy peaks with unknown random positions. (Note that if two peaks overlap too much, the reported width will be the width of the *blended* peak; in that case, it is better to use findpeaksG.m.

Speed demonstration. Here I compare the above peak finding functions, in Matlab, on a typical desktop or laptop PC. Note: Matlab's "tic" and "toc" functions are used to determine elapsed time.

Peaksat.m: `x=[0:.01:500]'; y=x.*sin(x.^2).^2; tic; P=peaksat(x,y,0); toc; NumPeaks=length(P)`
Elapsed time is 0.025 sec on a Dell XPS i7 3.5Ghz desktop, which is *523,000 peaks per second*.

findpeaksx.m: `x=[0:.01:500]'; y=x.*sin(x.^2).^2; tic; P=findpeaksx(x,y,0,0,3,3); toc; NumPeaks=length(P)`
Elapsed time is 0.11 sec on the same computer, which is *110,000 peaks per second*.

Gaussian peak measurement

```
P=findpeaksG(x, y, SlopeThreshold, AmpThreshold, SmoothWidth, FitWidth, smoothtype)
```

```
P=findpeaksL(…)
```

These Matlab/Octave functions locate the positive peaks in a noisy data set, perform a least-squares curve-fit of a Gaussian or Lorentzian function to the top part of the peak, and compute the position, height, and width (FWHM) of each peak from that least-squares fit. (The 6th input argument, FitWidth, is the number of data points around each peak top that is fit). The other arguments are the same as findpeaksx. It returns a list (in matrix P) containing the peak number and the estimated *position, height, width, and area* of each peak. It can find and curve-fit over *1800 peaks per second* in very large signals. (This is useful primarily for signals that have several data points in each peak, not for spikes that have only one or two points, for which findpeaksx is better).

```
>> x=[0:.01:50]; y=(1+cos(x)).^2; P=findpeaksG(x,y,0,-1,5,5); plot(x,y)
P =
     1       6.2832       4       2.3548       10.028
     2       12.566       4       2.3548       10.028
     3        18.85       4       2.3548       10.028…
```

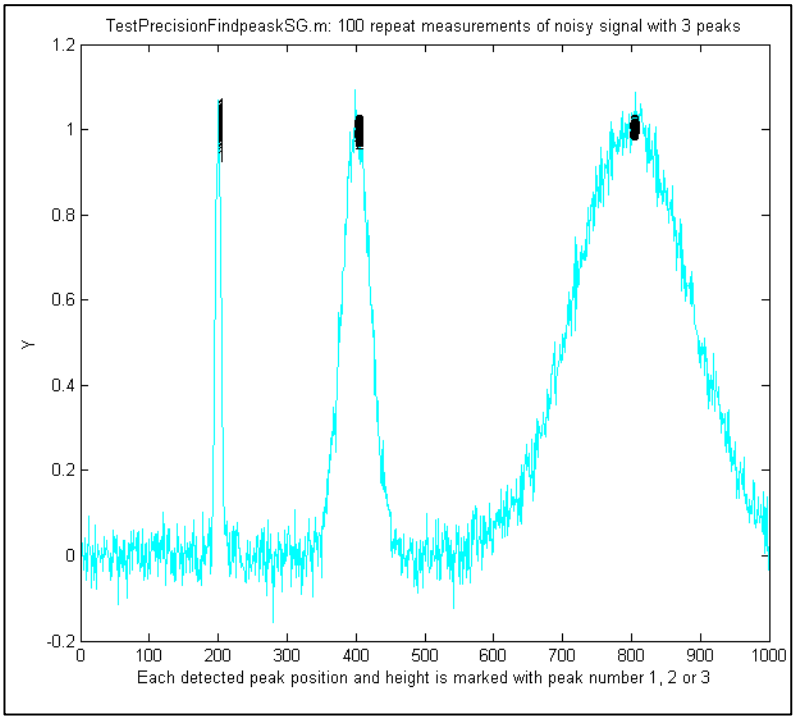
The function findpeaksplot.m is a simple variant of findpeaksG.m that also *plots* the x,y data and numbers the peaks on the graph (if any are found). The function findpeaksplotL.m does the same thing optimized for Lorentzian peak.

findpeaksSG.m is a *segmented* variant of the findpeaksG function, with the same syntax, except that the four peak detection parameters can be *vectors*, dividing up the signal into regions that you can optimize for peaks of different widths. You can declare any number of segments, based on the length of the third (SlopeThreshold) input argument. (Note: you only need to enter vectors for those parameters that you want to vary between segments. To allow any of the other peak detection parameters to remain *unchanged* across all segments, simply enter a *single scalar value* for that parameter. Only the SlopeThreshold must be a vector). (FindpeaksSL.m is the same thing for Lorentzian peaks.) The following example declares two segments, with AmpThreshold remaining the same in both segments.

```
SlopeThreshold=[0.001 .0001];
AmpThreshold=.2;
SmoothWidth=[5 10];
FitWidth=[10 20];
P=findpeaksSG(x, y, SlopeThreshold, AmpThreshold, SmoothWidth, FitWidth,3);
```

In the graphic shown below on the right, the demonstration script TestPrecisionFindpeaksSG.m creates a noisy signal with three peaks of widely different widths, detects and measures the peak positions, heights and widths of each peak using findpeaksSG, then prints out the percent relative standard

deviations of parameters of the three peaks in 100 measurements with independent random noise. With 3-segment peak detection parameters, findpeaksSG reliably detects and accurately measures all three peaks. In contrast, findpeaksG, when tuned to the middle peak (using line 26 instead of line 25), measures the first and last peaks poorly, because the peak detection parameters are far from optimum for those peak widths. You can also see that the *precision* of peak height measurements gets progressively *better* (smaller relative standard deviation) the *larger* the peak widths, simply because there are *more data points* in wider peaks. (You can change any of the variables in lines 10-18).



A related function is **findpeaksSGw.m** which is like the above except that is uses *wavelet denoising* (page 136) instead of smoothing (requires the Wavelet Toolbox). It takes the wavelet "level" rather than the smooth width as an input argument. The script TestPrecisionFindpeaksSGvsW.m compares the precision and accuracy for peak position and height measurement for both the regular findpeaksSG.m and the wavelet-based findpeaksSGw.m functions, finding that there is little to be gained in most cases by using the wavelet denoise instead of smoothing. That is mainly because in either case the peak parameter measurements are based on least-squares fitting to the *raw*, not the *smoothed*, data at each detected peak location, so the usual wavelet denoising advantage of avoiding smoothing distortion does not apply here.

One inconvenience with the above peak finding functions: it is annoying to have to estimate the values of the peak detection parameters that you need to use for your signals. A quick way to estimate these is to use **autofindpeaks.m**, which is similar to findpeaksG.m except that *you can optionally leave out the peak detection parameters* and just write "autofindpeaks(x, y)" or "autofindpeaks(x, y, *n*)", where *n* is the "peak capacity", roughly the number of peaks that would fit into that signal record (greater *n* looks for many narrow peaks. Smaller *n* looks for fewer wider peaks and neglects the fine structure). Simply, *n* allows you to *quickly adjust all the peak detection parameters at once* just by changing a single number. In addition, if you do leave out the explicit peak detection parameters, autofindpeaks will *print out the numerical input argument list* that it uses in the command window, so you can copy, paste, and edit for use with any of the findpeaks… functions. If you call autofindpeaks with the *output* arguments [P,A]=autofindpeaks(x,y,*n*), it returns the calculated peak detection parameters as a 4-element row vector A, which you can then pass on to other functions such as measurepeaks, effectively giving that function the ability to calculate the peak detection parameters from a single number *n*. For example, in the following signal, a visual estimate indicates about 20 peaks, so you use 20 as the 3rd argument:

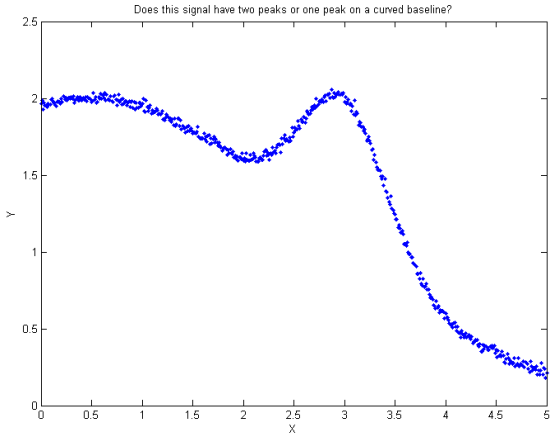
```
x=[0:.1:50];
y=10+10.*sin(x).^2+randn(size(x));
[P,A]=autofindpeaks(x,y,20);
```

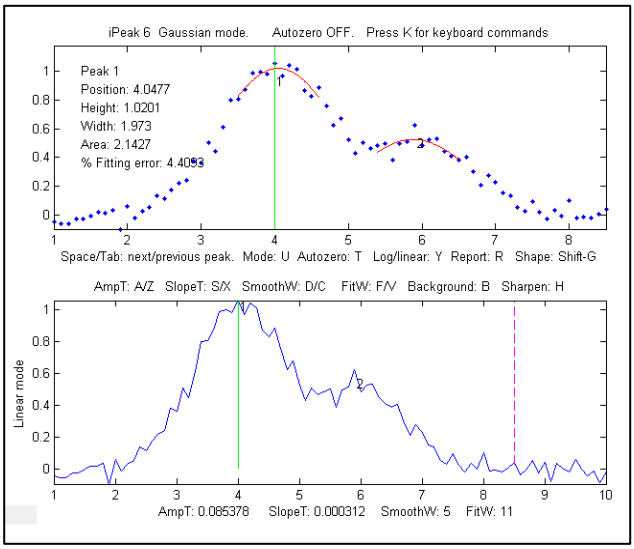
Then you can use A as the peak detection parameters for other peak detection functions, such as **`P=findpeaksG(x,y,A(1),A(2),A(3),A(4),1)`** or **`P=measurepeaks(x,y,A(1),A(2),A(3),A(4),1)`**. You will probably want to fine-tune the *amplitude* threshold A(2) manually for your own needs, but that is the one that is easiest to know.

Type "help autofindpeaks" and run the examples there. (The function autofindpeaksplot.m is the same but also plots and numbers the peaks). The script testautofindpeaks.m runs all the examples in the help file, plots the data and numbers the peaks (like autofindpeaksplot.m), with a 1-second pause between each example (If you are reading this online, click for the animated graphic).

Optimization of peak finding

Finding peaks in a signal depends on distinguishing between legitimate peaks and other features like noise and baseline changes. Ideally, a peak detector should detect all the legitimate peaks and ignore all the other features. This requires that a peak detector be "tuned" or optimized for the desired peaks. For example, the Matlab/Octave demonstration script OnePeakOrTwo.m creates a signal (shown on the right) that might be interpreted as either *one peak* at x=3 on a curved baseline or as *two peaks* at x=5 and x=3, depending on context. The peak finding algorithms described here have input arguments that allow some latitude for adjustment. In this example script, the "SlopeThreshold" argument is adjusted to detect just one or both of those peaks. The findpeaks… functions allow *either* interpretation, depending on the peak detection parameters. The optimum values of the input arguments for findpeaksG and related functions depend on the signal and on which features of the signal are important for your work. Rough values for these parameters can be estimated based on the width of the peaks that you wish to detect, as described above, but for the greatest control it will be best to fine-tune these parameters for your particular signal. A simple way to do that is to use autopeakfindplot(x, y, *n*) and adjust *n* until it finds the peak you want. It will print out the numerical input argument list so you can copy, paste, and edit for use with *any* of the findpeaks… functions. A more flexible way, if you are using Matlab, is to use the *interactive peak detector iPeak* (page 253), which allows you to adjust all of these parameters individually by simple keypresses and displays the results graphically and instantly. The script FindpeaksComparison shows how findpeakG compares to the other peak detection functions when applied to a computer-generated signal with multiple peaks with variable types and amounts of baseline and random noise. By itself, autofindpeaks.m, findpeaksG and findpeaksL do *not* correct for a non-zero baseline. If your peaks are superimposed on a baseline, you should subtract the baseline first



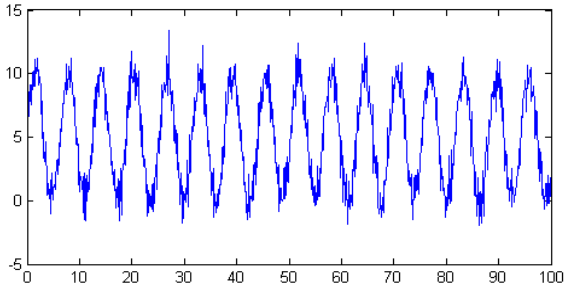


or use the other peak detection functions that do correct for the baseline. In the example shown on the left (using the interactive peak detector iPeak program described on page 253), suppose that the important parts of the signal are two broad peaks at x=4 and x=6, the second one half the height of the first. The small, jagged features are just random noise. You want to detect the two peaks but ignore the noise. (The detected peaks are numbered 1,2,3,…in the lower panel of this graphic). This is what it looks like if the *AmpThreshold* is too small or too large, if the *SlopeThreshold* is too small or too large, if the *SmoothWidth* is too small or too large, and if the *FitWidth* is too small or too large. If these parameters are within the optimum range for this measurement objective, the findpeaksG functions will return something like this (although the exact values will vary with the noise and with the value of FitWidth):

Peak#	Position	Height	Width	Area
1	3.9649	0.99919	1.8237	1.94
2	5.8675	0.53817	1.6671	0.955

How does 'findpeaksG' differ from 'max' in Matlab or 'findpeaks' in the *Signal Processing Toolkit*?

The 'max' function simply returns the largest *single* value in a vector. Findpeaks in the *Signal Processing Toolbox* can be used to find the values and indices of all the peaks that are higher than a specified peak height and are separated from their neighbors by a specified minimum distance. My version of findpeaks (findpeaksG) accepts both an independent variable (x) and dependent variable (y) vectors, finds the places where the average curvature over a specified region is concave down, fits that region with a least-squares fit and returns the peak position (in x units), height, width, and area, of any peak that exceeds a specified height. For example, to create a noisy series of peaks (plotted on the right) and apply both findpeaks functions to the resulting data:



```
x=[0:.1:100];
y=5+5.*sin(x)+randn(size(x));
plot(x,y)
```

Now, most people just looking at this plot of data would count *16 peaks*, with peak heights averaging about 10 units. Every time the statements above are run, the random noise is different, but you would still count the 16 peaks because the signal-to-noise ratio is 10, which is not that bad. But the findpeaks function in the *Signal Processing Toolbox* counts anywhere from 11 to 20 peaks, with an average height (PKS) of 11.5.

```
[PKS,LOCS]=findpeaks(y,'MINPEAKHEIGHT',5,'MINPEAKDISTANCE',11)
```

In contrast, my findpeaksG function **`findpeaksG(x,y,0.001,5,11,11,3)`** counts 16 peaks every time, with an average height of 10 ±0.3, which is much more reasonable. It also measures the width and area, assuming the peaks are Gaussian (or Lorentzian, in the variant findpeaksL). To be fair, findpeaks in the *Signal Processing Toolbox*, or my even faster findpeaksx.m function, works better for peaks that have only 1-3 data points on the peak. FindpeaksG is better for peaks that have more data points.

The demonstration script FindpeaksSpeedTest.m compares the speed of the four peak detectors on the same large test signal: Signal Processing Toolkit findpeaks, peaksat, findpeaksx, and findpeaksG:

```
                    Number     Elapsed     Peaks per
 Function         of peaks    time, sec    second
findpeaks (SPT)      160      0.012584      12715
peaksat              999      0.0012912     773699
findpeaksx           158      0.001444      109418
findpeaksG           157      0.011005       14267
```

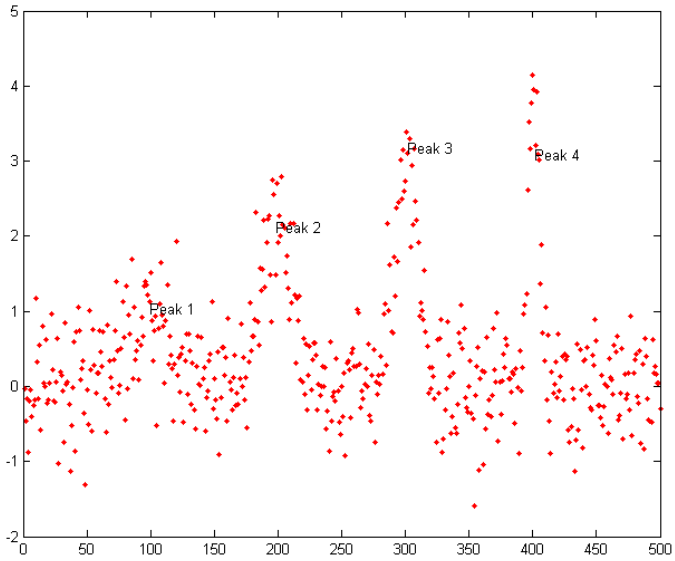
Finding valleys

There is also a similar function for finding *valleys* (minima), called **findvalleys.m**, which works the same way as findpeaksG.m, except that it locates *minima* instead of *maxima*. Only valleys above the AmpThreshold (that is, more positive or less negative) are detected. If you wish to detect valleys that have negative minima, then AmpThreshold must be set more negative than that.

```
>> x=[0:.01:50];y=cos(x);P=findvalleys(x,y,0,-1,5,5)
P =
    1.0000    3.1416   -1.0000    2.3549         0
    2.0000    9.4248   -1.0000    2.3549         0
    3.0000   15.7080   -1.0000    2.3549         0
    4.0000   21.9911   -1.0000    2.3549         0....
....
```

Accuracy of peak measurements

The accuracy of the measurements of peak position, height, width, and area by the findpeaksG function depends on the shape of the peaks, the extent of peak overlap, the strength of the background, and the signal-to-noise ratio. The width and area measurements particularly are strongly influenced by peak overlap, noise, and the choice of FitWidth. Isolated peaks of Gaussian shape are measured most accurately. For *Lorentzian* peaks, use findpeaksL.m instead (the only difference is that the reported peak heights, widths, and areas will be more accurate if the peaks are Lorentzian). See "ipeakdemo.m" below for an accuracy trial for Gaussian peaks. For highly overlapping peaks that do not exhibit distinct maxima, use peakfit.m or the Interactive Peak Fitter (ipf.m, page 416). For a direct comparison of the accuracy of findpeaksG vs peakfit, run the demonstration script peakfitVSfindpeaks.m. This



script generates four very noisy peaks of different heights and widths, then measures them in two different ways: first with findpeaksG.m (figure on the left) and then with peakfit.m, and compares the results. The peaks detected by findpeaksG are labeled "Peak 1", "Peak 2", etc. If you run this script several times, it will generate the same peaks but with *independent samples of the random noise each time*. You will find that both methods work well most of the time, with peakfit giving smaller errors in most cases (because it uses *all* the points in each peak, not just the top part), but occasionally findpeaksG will miss the first (lowest) peak and rarely it will detect a 5th peak that is not there. On the other hand, in this case peakfit.m is constrained to fit 4 and only 4 peaks each time.

The demonstration script FindpeaksComparison compares the accuracy of findpeaksG and findpeaksL to several peak detection functions when applied to signals with multiple peaks and variable types and amounts of baseline and random noise.

Peak finding combined with iterative curve fitting.

findpeaksb.m is a variant of findpeaksG.m that more accurately measures peak parameters by using iterative least-square curve fitting based on my peakfit.m function. This yields better peak parameter values than findpeaksG alone for three reasons.

(1) it can be set for different peak shapes with the input argument 'PeakShape'.

(2) it fits the *entire* peak, not just the top part.

(3) it has provision for background subtraction (when the input argument "BaselineMode" is set to 1, 2, or 3 - linear, quadratic, or flat, respectively).

This function works best with isolated peaks that do not overlap. For version 3, the syntax is **`P = findpeaksb(x,y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, smoothtype, windowspan, PeakShape, extra, BASELINEMODE)`**. The first seven input arguments are the same as for the findpeaksG.m function. If you have been using findpeaksG or *iPeak* to find and measure peaks in your signals, you can use those same input argument values for findpeaksb.m. The remaining four input arguments are for the peakfit function:

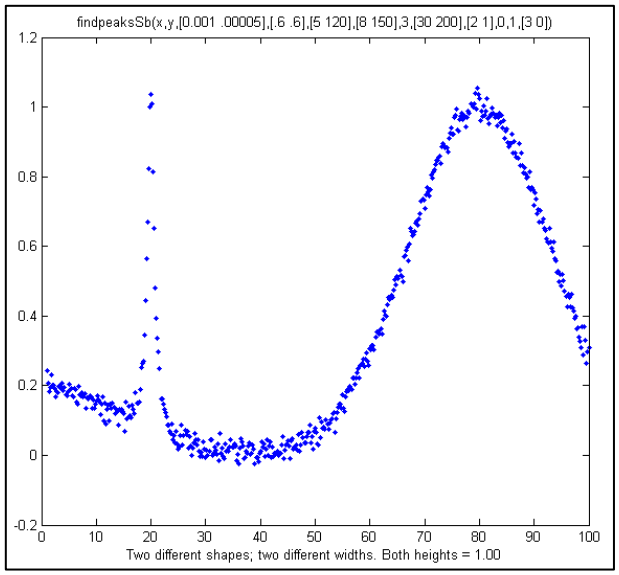
- "`windowspan`" specifies the number of data points over which each peak is fit to the model shape (This is the hardest one to estimate. In BaselineMode 1 and 2, 'windowspan' must be large enough to cover the entire single peak and get down to the background on both sides of the peak, but not so large as to overlap neighboring peaks);"`PeakShape`" specifies the model peak shape: 1=Gaussian, 2=Lorentzian, etc (type 'help findpeaksb' for a list),

- "extra" is the shape modifier variable that is used for the Voigt, Pearson, exponentially broadened Gaussian and Lorentzian, Gaussian/Lorentzian blend, and bifurcated Gaussian and Lorentzian shapes to fine-tune the peak shape;
- "BASELINEMODE" is 0, 1, 2, or 3 for no, linear, quadratic, or flat background subtraction.

The peak table returned by this function has a 6th column listing the percent fitting errors for each peak. Here is a simple example with three Gaussians on a linear background, comparing plain *findpeaksG, findpeaksb* without background subtraction (BASELINEMODE=0), and to *findpeaksb* with background subtraction (BASELINEMODE=1).

```
x=1:.2:100;Heights=[1 2 3];Positions=[20 50 80];Widths=[3 3 3];
y=2-(x./50)+modelpeaks(x,3,1,Heights,Positions,Widths)+.02*randn(size(x));
plot(x,y);
disp('          Peak      Position      Height      Width        Area
   % error')
PlainFindpeaks=findpeaksG(x,y,.00005,.5,30,20,3)
NoBackgroundSubtraction=findpeaksb(x,y,.00005,.5,30,20,3,150,1,0,0)
LinearBackgroundSubtraction=findpeaksb(x,y,.00005,.5,30,20,3,150,1,0,1)
```

The demonstration script DemoFindPeaksb.m shows how findpeaksb works with multiple peaks on a curved background, and FindpeaksComparison shows how findpeaksb compares to the other peak detection functions when applied to signals with multiple peaks and variable types and amounts of baseline and random noise.



Segmented peak finder. What if the peak widths vary greatly over the signal? **findpeaksSb.m** is a *segmented* variant of findpeaksb.m. It has the same syntax as findpeaksb.m, except that the input arguments SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, window, width, PeakShape, extra, NumTrials, BaselineMode, and fixedparameters, can all optionally be scalars *or vectors with one entry for each segment*, in the same manner as findpeaksSG.m. It returns a matrix P listing the peak number, position, height, width, area, percent fitting error and "R2" of each detected peak. In the example on the right, the two peaks have the same height above baseline (1.00) but different shapes (the first Lorentzian and the second Gaussian), very different widths, and different baselines. So, using findpeaksG or findpeaksL or findpeaksb, it would be impossible to find one set of input arguments that would be optimum for both peaks. However, using findpeaksSb.m, different settings can apply to different regions of the signal. In this simple example, there are only *two* segments, defined by SlopeThreshold with 2 different values, and the other input arguments are either the same or are different in those two segments. The result is that the peak height of both peaks is measured accurately. First, you define the values of the peak detection parameters, then call findpeaksSb.

```
>> SlopeThreshold=[.001 .00005]; AmpThreshold=.6; smoothwidth=[5
120]; peakgroup=[5 120];smoothtype=3; window=[30 200]; PeakShape=[2 1];
extra=0; NumTrials=1; BaselineMode=[3 0];

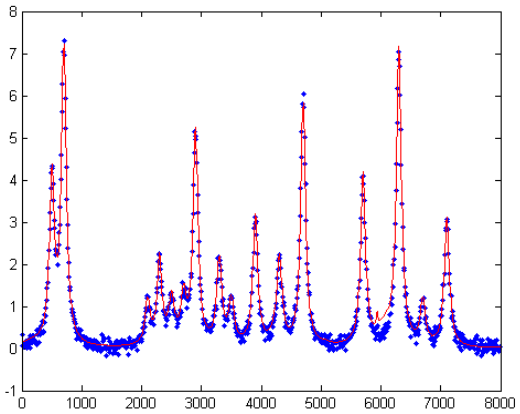
>> findpeaksSb(x,y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup,
smoothtype, window, PeakShape, extra, NumTrials, BaselineMode)

Peak #    Position    Height    Width     Area
   1       19.979     0.9882     1.487     1.565
   2       79.805     1.0052    23.888    25.563
```

DemoFindPeaksSb.m demonstrates the findpeaksSG.m function by creating a random number of Gaussian peaks whose widths increase by a factor of 25-fold over the x-axis range and that are superimposed on a curved baseline with random white noise that increases gradually. Four segments are used in this example, changing the peak detection and curve fitting values so that all the peaks are measured accurately. Graphic. Printout.

findpeaksb3.m is a more ambitious variant of findpeaksb.m that fits each detected peak *along with the previous and following peaks* found by findpeaksG.m, to deal better with the *overlap of the adjacent overlapping peaks*. The syntax is

```
FPB=findpeaksb3(x,y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup,
smoothtype, PeakShape, extra, NumTrials, BASELINEMODE, ShowPlots).
```



The demonstration script DemoFindPeaksb3.m shows how findpeaksb3 works with irregular clusters of overlapping Lorentzian peaks, as in the example on the left (type "help findpeaksb3") for more. The demonstration script FindpeaksComparison shows how findpeaksb3 compares to the other peak detection functions when applied to signals with multiple peaks and variable types and amounts of baseline and random noise.

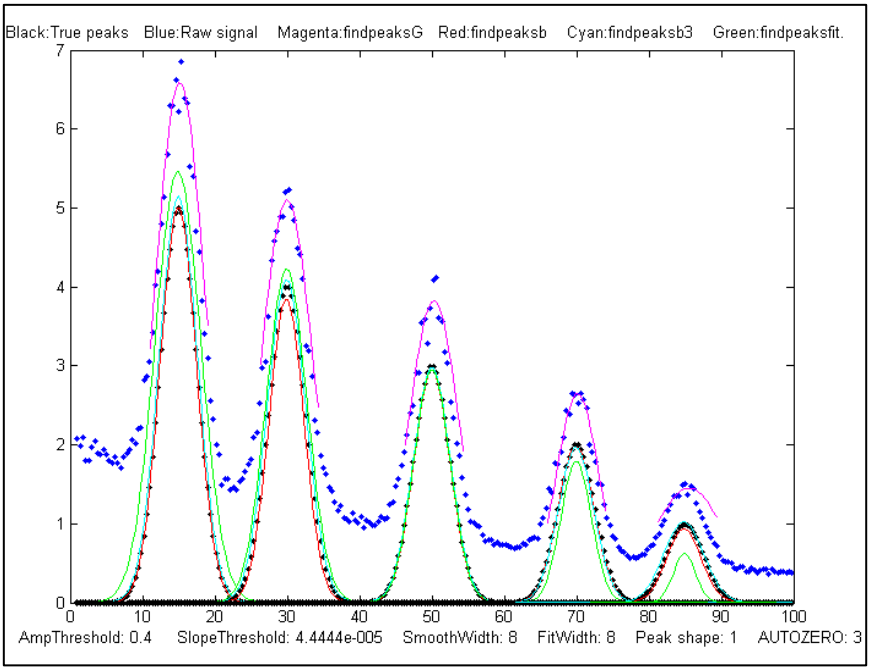
findpeaksfit.m is essentially a serial combination of findpeaksG.m and peakfit.m. It uses the number of peaks found and the peak positions and widths that are the output of the findpeaksG function as the input for the peakfit.m function, which then fits the *entire signal* with the specified peak model. This combination yields better values than findpeaksG alone, because peakfit fits the entire peak, not just the top part, and it deals with non-Gaussian and overlapped peaks. However, it fits only those peaks that are found by findpeaksG, so you must make sure that all the peaks in the data are found. The syntax is:

```
function [P, FitResults, LowestError, BestStart, xi, yi] = findpeaksfit (x,
y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, smoothtype,
peakshape, extra, NumTrials, BaselineMode, fixedparameters, plots)
```

The first seven input arguments are exactly the same as for the findpeaksG.m function. If you have been using findpeaksG or iPeak to find and measure peaks in your signals, you can use those same input argument values for findpeaksfit.m. The remaining six input arguments of findpeaksfit.m are for

the peakfit function. If you have been using peakfit.m or ipf.m (page 416) to fit the peaks in your signals, then you can use those same input argument values for findpeaksfit.m. The demonstration script findpeaksfitdemo.m was used to generate a GIF animation that shows findpeaksfit automatically finding and fitting the peaks in a set of 150 signals, each of which may have 1 to 3 noisy Lorentzian peaks in variable locations, *artificially slowed down* with the "pause" function so you can see it better. (This requires the findpeaksfit.m and lorentzian.m functions installed. Type "help findpeaksfit" for more information).

Comparison of peak finding functions



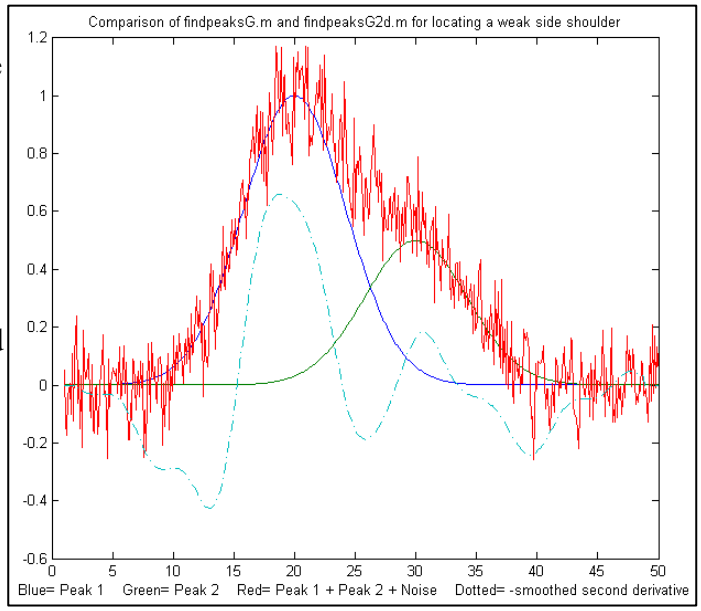
The demonstration script FindpeaksComparison.m compares the peak parameter accuracy of findpeaksG/L, findpeaksb, findpeaksb3, and findpeaksfit applied to a computer-generated signal with multiple peaks plus variable types and amounts of baseline and random noise. (Requires those four functions, plus gaussian.m, lorentzian.m, modelpeaks.m, findpeaksG.m, findpeaksL.m, pinknoise.m, and propnoise.m, in the Matlab/ Octave search path). Results are displayed graphically in figure windows 1, 2, and 3 and printed out in a table of parameter accuracy and elapsed time for each method, as shown below. You may change the lines in the script marked by <<< to modify the number and character and amplitude of the signal peaks, baseline, and noise. (Make the signal like yours to discover which method works best for your type of signal). The best method depends mainly on the shape and amplitude of the baseline and on the extent of peak overlap. Type "help FindpeaksComparison" for details. (Elapsed times for Matlab 2020 running on Dell XPS i7 3.5Ghz).

Average absolute percent errors of all peaks

	Position error	Height error	Width error	Elapsed time, sec
findpeaksG	0.35955%	38.573%	25.797%	0.005768
findpeaksb	0.38828%	8.5024%	14.329%	0.069061
findpeaksb3	0.27187%	3.7445%	3.0474%	0.49538
findpeaksfit	0.51930%	8.0417%	24.035%	0.27363

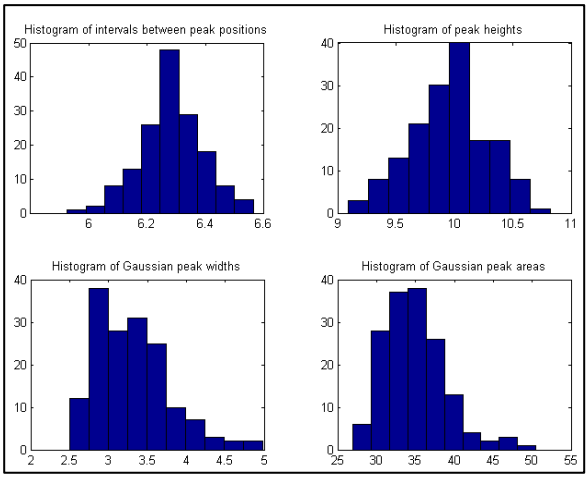
Note: **findpeaksfit.m** differs from **findpeaksb.m** in that findpeaksfit.m *fits all the found peaks at one time* with a single multi-peak model, whereas findpeaksb.m *fits each peak separately* with a single-peak model, and findpeaksb3.m fits each detected peak *along with the previous and following peaks.* As a result, findpeaksfit.m works better with a relatively small number of peak that all overlap, whereas findpeaksb.m works better with a large number of isolated non-overlapping peaks, and findpeaksb3.m works for large numbers of peaks that overlap at most one or two adjacent peaks. FindpeaksG/L is simple and fast, but it does not perform baseline correction. Findpeaksfit can perform flat, linear, or

quadratic baseline correction, but it works only over the entire signal at once. In contrast, findpeaksb and findpeaksb3 perform *local* baseline correction, which often works well if the baseline is curved or irregular.



findpeaksG2d.m is a variant of findpeaksG that can be used to locate the positive peaks *and shoulders* in a noisy x-y time series data set. Detects peaks in the *negative of the second derivative* of the signal, by looking for downward slopes in the *third* derivative that exceed SlopeThreshold. See TestFindpeaksG2d.m.

[M,A]=autopeaks.m is a peak detector for peaks of arbitrary shape. It is basically a combination of autofindpeaks.m and measurepeaks.m. It has similar syntax to measurepeaks.m, except that the peak detection parameters (SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, and smoothtype) can be omitted and the function will calculate trial values in the manner of autofindpeaks.m. Using the syntax [M,A]=autopeaks(x, y) works well in some cases, but if not try [M,A]=autopeaks(x, y, n), using different values of n (roughly the number of peaks that would fit into the signal record) until it detects the peaks that you want to measure. Like measurepeaks, it returns a table M containing the peak number, peak position, absolute peak height, peak-valley difference, perpendicular drop area (page 146), and tangent skim area of each peak it detects (page 142), but is also can optionally return a vector A containing the peak detection parameters that it calculates (for use by other peak detection and fitting functions). For the most precise control over peak detection, you can specify all the peak detection parameters by typing M=autopeaks(x,y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup). [M,A]=autopeaksplot.m is the same but it also plots the signal and the individual peaks in the manner of measurepeaks.m (shown above). The script testautopeaks.m runs all the examples in the autopeaks help file, with a 1-second pause between each one, printing out results in the command window and additionally plotting and numbering the peaks (Figure window 1) and each individual peak (Figure window 2). It requires gaussian.m and fastsmooth.m in the search path.



Peak statistics. The function **peakstats.m** uses the same algorithm as findpeaksG, but it computes and returns a table of summary statistics of the peak intervals (the x-axis interval between adjacent detected peaks), heights, widths, and areas, listing the maximum, minimum, average, and percent standard deviation of each, and optionally plotting the x,y data with numbered peaks in figure window 1, printing the table of peak statistics in the command window, and plotting the histograms of the peak intervals, heights, widths, and areas in the four quadrants of figure

window 2. Type "help peakstats". The syntax is the same as findpeaksG, with the addition of an 8th input argument to control the display and plotting. Example:

```
x=[0:.1:1000];y=5+5.*cos(x)+randn(size(x));
PS=peakstats(x,y,0,-1,15,23,3,1);
```

```
Peak Summary Statistics
158 peaks detected
           Interval     Height     Width        Area
Maximum    6.6428      10.9101     5.6258      56.8416
Minimum    6.0035      9.1217      2.5063      28.2559
Mean       6.283       9.9973      3.3453      35.4737
% STD      1.8259      3.4265      15.1007     12.6203
Median     6.2719      10.0262     3.2468      34.6473
Mode       6.0035      9.1217      2.5063      28.2559
```

With the last input argument omitted or equal to zero, the plotting and printing in the command window are omitted. The numerical values of the peak statistics table are returned as a 4x4 array, in the same order as the example above.

tablestats.m (`PS=tablestats(P,displayit)`) is similar to peakstats.m except that it accepts as input a peak table P such as generated by findpeaksG.m, findvalleys.m, findpeaksL.m, findpeaksb.m, findpeaksplot.m, findpeaksnr.m, findpeaksGSS.m, findpeaksLSS.m, or findpeaksfit.m - any of the functions that return a table of peaks with at least 4 columns listing peak number, height, width, and area. Computes the peak intervals (the x-axis interval between adjacent detected peaks) and the maximum, minimum, average, and percent standard deviation of each, and optionally displaying the histograms of the peak intervals, heights, widths, and areas in figure window 2. Set the optional last argument displayit = 1 if the histograms are to be displayed, otherwise not. Example:

```
x=[0:.1:1000];y=5+5.*cos(x)+.5.*randn(size(x));
figure(1);P=findpeaksplot(x,y,0,8,11,19,3);tablestats(P,1);
```

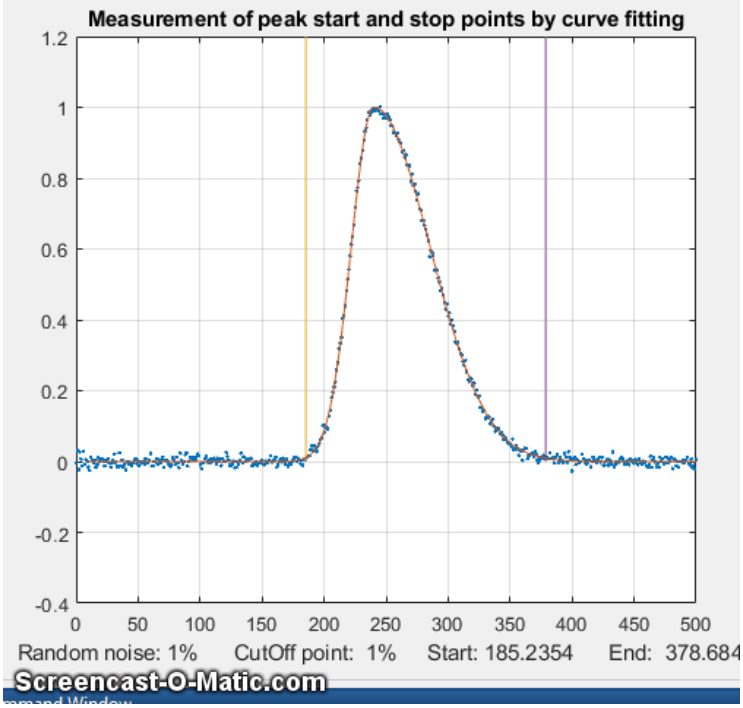
FindpeaksE.m is a variant of findpeaksG.m that additionally estimates the percent relative fitting error of each peak (assuming a Gaussian peak shape) and returns it in the 6th column of the peak table.

Example:

```
>> x=[0:.01:5];
>> y=x.*sin(x.^2).^2+.1*whitenoise(x);
>> P=findpeaksE(x,y,.0001,1,15,10)
P =
    1    1.3175     1.3279     0.25511    0.36065    5.8404
    2    1.4245     1.2064     0.49053    0.62998   10.476
    3    2.1763     2.1516     0.65173    1.4929     3.7984
    4    2.8129     2.8811     0.2291     0.70272    2.3318...
```

Peak start and end

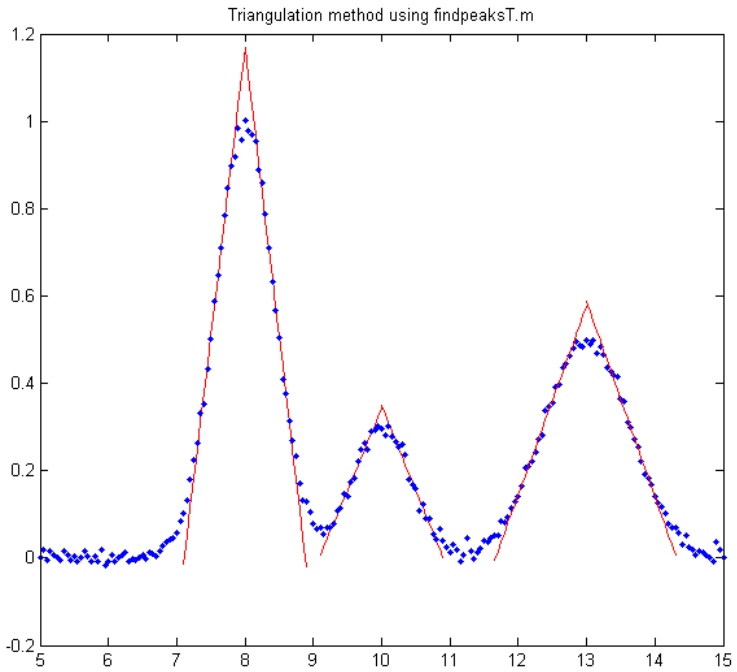
Defining the "start" and "end" of the peak (the x-values where the peak begins and ends) is a bit arbitrary because typical peak shapes approach the baseline *asymptotically* far from the peak maximum. You might define the peak start and end points as the x values where the y value is some small fraction, say 1%, of the peak height, but then the random noise on the baseline will be a larger fraction of the signal amplitude at that point. Smoothing to reduce noise is likely to distort and broaden peaks, effectively changing their start and end points. Overlap of peaks also greatly complicates the issue. One solution is to fit each peak to a model shape (page 173), then calculate the peak start and end from the model expression. That method minimizes the noise problem by fitting the data over the entire peak, and it can handle overlapping peaks, but it works only if the peaks can be modeled by available fitting programs. For example, Gaussian peaks can be shown to reach a fraction ***a*** of the peak height at x = p ± sqrt(w^2 log(1/***a***))/(2 sqrt(log(2))) where p is the peak position and w is the peak width (full width at half maximum). So, for example if ***a*** = .01, x =p ± w*sqrt((log(2) + log(5))/(2 log(2))) = 1.288784*w. Lorentzian peaks can be shown to reach a fraction ***a*** of the peak height at x = p ± sqrt[(w^2 - ***a*** w^2)/a]/2. If ***a*** = .01, x = p ± (3/2 sqrt(11)*w) = 4.97493*w. The findpeaksG variants **findpeaksGSS.m** and **findpeaksLSS.m**, for Gaussian and Lorentzian peaks respectively, compute the peak start and end positions in this manner and return them in the 6th and 7th columns of the peak table **P**. Uncertainty in the measured peak position p and especially in the peak width w will make the results less certain.



The problem with this method is that it requires an analytical peak model, expressed as a closed-form expression that can be solved algebraically for their start and end points. A more versatile method is to fit a model to the peak data by iterative curve fitting (page 199), and then use the best-fit model to locate the start and stop points by interpolation. For complex peak shapes, the model *need not be limited to a single peak*. Complex, asymmetrical peak shapes can often be modeled as the sum of simple shapes, such as Gaussians. An example of this method is demonstrated in the script StartAndEnd.m, which simulates a noisy, asymmetrical peak and then applies this method using my peakfit.m function (page 397). You can select the start/stop cut-off point as a fraction of the peak height in line 8 of this script, the amount of random noise in line 7, and the number of model peaks in line 9. At the cut-off points, the signal-to-noise ratio is very poor, so a direct measurement of x where y equals the cut-off is impractical. Nevertheless, the start and end points can be calculated surprisingly precisely by computing a least-squares best-fit model (contained in the output arguments xi and yi of

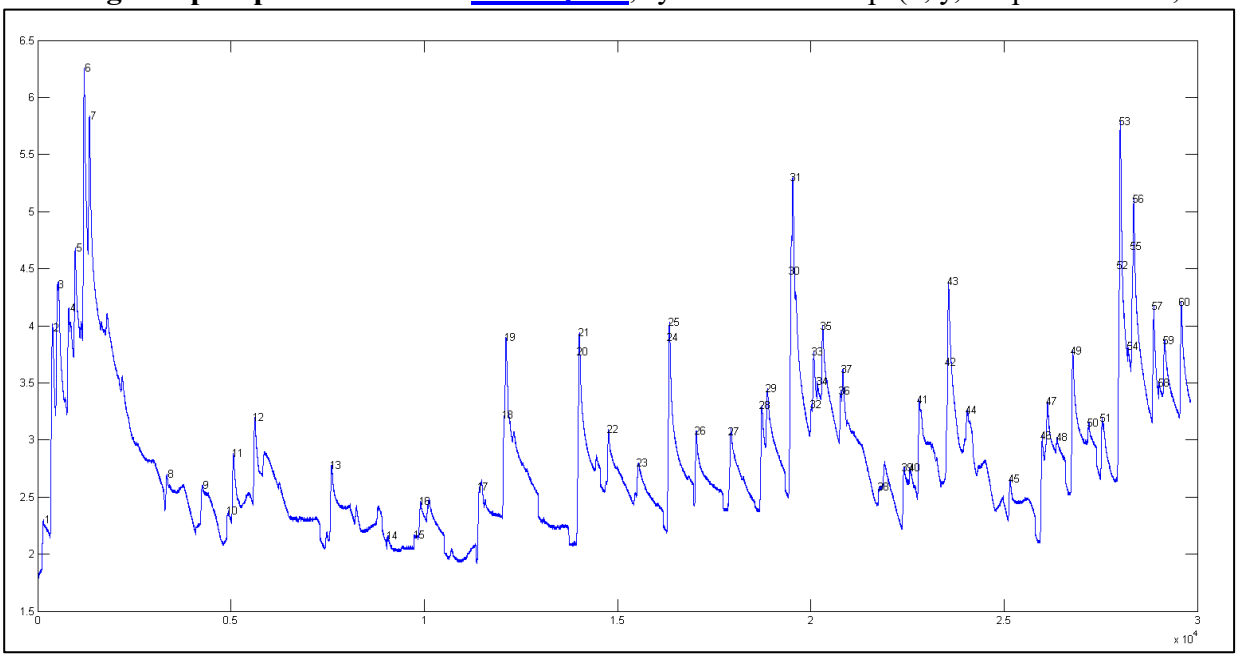
the peakfit function), which averages out the noise over the entire signal (the more data points the better). The graphic on the previous page shows the method in operation for 50 repeat measurements with different random noise samples, first with 1% noise and then with 10% noise. (If the animation is not visible, click this link). Despite the poor signal-to-noise ratio at the cut-off points, the relative standard deviation of the measured start and end points (marked by the vertical lines) is only about 0.2%. Even when the noise is increased 10-fold (line 7), the relative standard deviation is still under 1%. (If you have a different number of data points per peak, the precision will be inversely proportional to the square root of the number of points).

Triangle construction method. Before the age of computers and electronics, peak parameters were sometimes measured by *constructing a triangle around each peak* with its long sides tangent to the sides of the peak, as shown below. This old method is mimicked by the functions **findpeaksT.m** and

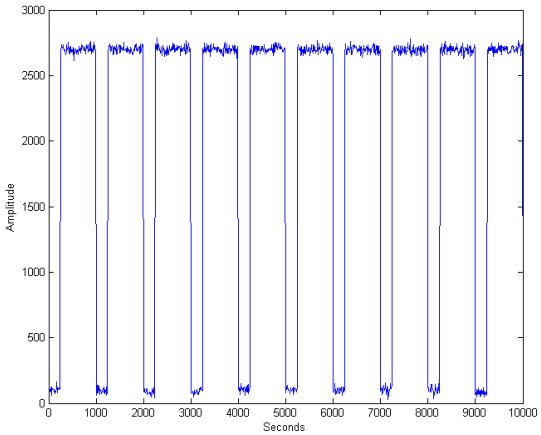


findpeaksTplot.m, which are demonstrated by the script that generated this graphic). In this method the peak height is taken as the apex of the triangle, which is slightly higher than the peak of the underlying curve. It turns out that the performance of this method is poor when the signals are very noisy or if the peaks overlap, but in a *few special circumstances*, the triangle construction method can be more accurate for the measurement of peak area than the Gaussian method if the peaks are *asymmetric* or of *uncertain shape.* (For some specific examples, see the demo function triangulationdemo.m: click for the graphic).

Locating sharp steps. The function **findsteps.m**, syntax: P=findsteps(x, y, SlopeThreshold,



AmpThreshold, SmoothWidth, peakgroup), locates positive transient steps in noisy x-y time series data, by computing the first derivative of y that exceed “SlopeThreshold”, computes the step height as the difference between the maximum and minimum y values over a number of data point equal to "Peakgroup", and returns list P with step number, x position, y position, and the step height of each step detected. "SlopeThreshold" and "AmpThreshold" control step sensitivity. Higher values will neglect smaller features. Increasing "SmoothWidth" reduces small sharp false steps caused by random noise or by "glitches" in the data acquisition. The figure above shows a real example of experimental data. The related function findstepsplot.m also plots the data and numbers the peaks.



Rectangular pulses (square waves) require a different approach, based on amplitude discrimination rather than differentiation. The function "findsquarepulse.m" (syntax **`S=findsquarepulse(t,y, threshold)`**) locates the rectangular pulses in the signal t,y that exceed a y-value of "threshold" and determines their start time, average height (relative to the baseline) and width. DemoFindsquare.m creates a test signal (with a true height of 2636 and a width of 750) and calls findsquarepulse.m to demonstrate. If the signal is very noisy, some preliminary rectangular smoothing (e.g. using fastsmooth.m) before calling findsquarepulse.m may be helpful to eliminate false peaks.

NumAT(m,threshold): "**Num**bers **A**bove **T**hreshold": Counts the number of adjacent elements in the vector “m” that are greater than or equal to the scalar value 'threshold'. It returns a matrix listing each group of adjacent values, their starting index, the number of elements in each group, the sum of each group, and the average (mean) of each group. Type "`help NumAT`" and try the example.

Using the peak table

All these peak finding functions return a peak table as a matrix, with one row for each peak detected and with several columns listing, for example, the peak number, position, height, width, and area in columns 1 - 5 (with additional columns included for the variants measurepeaks.m, findpeaksnr.m, findpeaksGSS.m, and findpeaksLSS.m). You can assign this matrix to a variable (e.g., **P**, in these examples) and then use Matlab/Octave notation and built-in functions to extract specific information from that matrix. *The powerful combination of functions and Matlab's "colon" notation allows you to construct compact expressions that extract the very specific information that you need*. Here are several examples:

`[P(:,2) P(:,3)]` is the time series of peak heights (peak position in the first column and peak height in the second column).

`mean(P(:,3))` returns the average peak height of all peaks (because peak height is in column 3). This also works with "`median`".

`max(P(:,3))` returns the maximum peak height of all the peaks. This also works with `min`.

`hist(P(:,3))` displays the histogram of peak heights (using built-in "hist" function).

`std(P(:,4))./mean(P(:,4))` returns the relative standard deviation of the peak widths (column 4).

`P(:,3)./max(P(:,3))` returns the ratio of each peak height (column 3) to the height of the highest peak detected.

`100.*P(:,5)./sum(P(:,5))` returns the percentage of each peak area (column 5) of the total area of all peaks detected.

`sortrows(P,2)` sorts **P** by peak position; `sort rows(P,3)` sorts **P** by peak height (small to large).

To create "d" as the vector of x-axis (position) differences between adjacent peaks (because peak position is in column 2):

```
for n=1:length(P)-1;d(n)=max(P(n+1,2)-P(n,2));end
```

(In Matlab/Octave, multiple statements can be placed on one line, separated by semicolons.)

The val2ind function. My downloadable function val2ind.m (syntax **`[index,closestval] = val2ind(v,val)`**) is a simple function that returns the index and the value of the element of vector 'v' that is closest to 'val' (download this function and place in the Matlab search path). This simple function is very useful in working with peak tables: **`val2ind(P(:,3),7.5)`** returns the peak number whose height (column 3) is closest to 7.5. **`P(val2ind(P(:,2),7.5),3)`** returns the peak height (column 3) of the peak whose position (column 2) is closest to 7.5. **`P(val2ind(P(:,3),max(P(:,3))),:)`** returns the row vector of peak parameters of the highest peak in peak table **P**. The three statements **`j=P(:,4)<5.8; k=val2ind(j,1); P(k,:)`** return the matrix of peak parameters of all peaks in **P** whose halfwidths (in column 4) are less than a specified number (5.8 in this case). **Note:** In Matlab and Octave, multiple statements can be placed on one line, separated by semicolons.

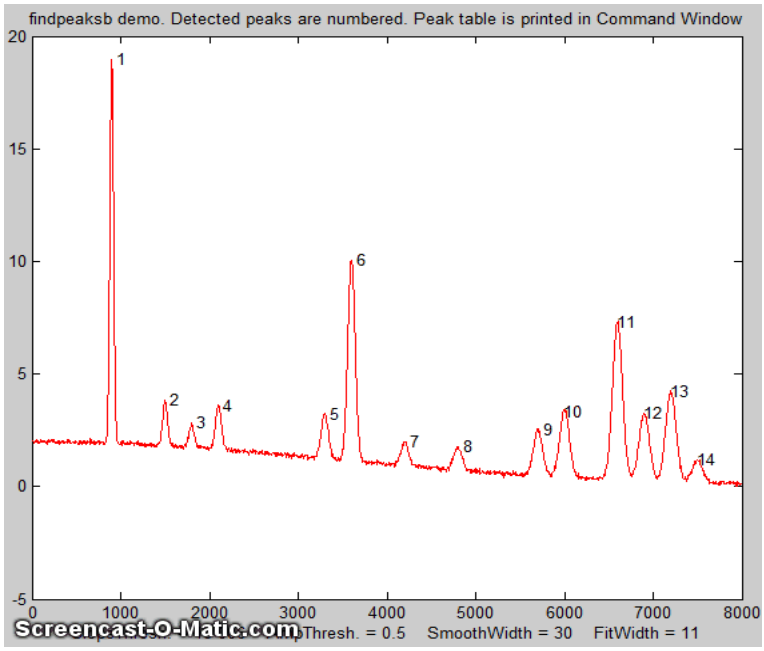
Finding peaks in multi-column data. The script FindingPeaksInMultiColumnData.m shows how to read a *multi-column* dataset from an Excel file and detect the peaks *in each column*, returning the peak data in a 3-dimentional table of results, PP. Works with any of my peak finding and/or fitting functions.

Demo scripts

DemoFindPeak.m is a simple demonstration script using the findpeaksG function on noisy synthetic data. The function numbers the peaks and prints out the peak table in the Matlab command window:

```
Peak #   Position     Height     Width     Area
Measuredpeaks =
1        799.95      6.9708      51.222    380.12
2        1199.4      3.9168      50.44     210.32
3        1600.6      2.9955      49.683    158.44
4        1800.4      2.0039      50.779    108.33
 ......etc.
```

DemoFindPeakSNR is a variant of DemoFindPeak.m that uses findpeaksnr.m to compute the signal-to-noise ratio (SNR) of each peak and returns it in the 5th column (click for a graphic).



DemoFindPeaksb.m is a similar demonstration script that uses the findpeaksb function on noisy synthetic data consisting of variable numbers of Gaussian peaks *superimposed on a variable curved background*. (The findpeaksG function would not give accurate measurements of peak height, width, and area for this signal, because it does not correct for the background). Click for animation.

```
Relative Percent Errors

   Position     Height     Width      Area
 -0.002246     0.54487    1.4057     1.9429
 -0.02727      5.0091     8.9204     13.483
  0.008429    -1.1224    -1.4923    -2.6315 …etc.

% Root mean square errors

ans =
0.044428    2.2571      3.8253      5.850
```

Peak Identification

The command line function idpeaks.m is used for identifying peaks *according to their x-axis maximum positions*, which is useful whenever the identification of a peak depends on its x-axis position, for example in atomic spectroscopy and in chromatography. The syntax is

`[IdentifiedPeaks, AllPeaks]=idpeaks(DataMatrix, AmpT, SlopeT, SmoothWidth, FitWidth, maxerror, Positions, Names)`

It finds peaks in the signal "DataMatrix" (x-values in column 1 and y-values in column 2), according to the peak detection parameters "AmpT", "SlopeT", "SmoothWidth", "FitWidth" (see the "findpeaksG" function above), then compares the found peak positions (x-values) to a database of known peaks, in the form of an array of known peak maximum positions ('Positions') and matching cell array of names ('Names'). If the position of a peak found in the signal is closer to one of the known peaks by less than the specified maximum error ('maxerror'), that peak is considered a match and its peak position, name, error, and peak amplitude (height) are entered into the output cell array "IdentifiedPeaks". The full list of detected peaks, identified or not, is returned in "AllPeaks". Use "cell2mat" to access numeric elements of IdentifiedPeaks, e.g., **`cell2mat(IdentifiedPeaks(2,1))`** returns the position of the first identified peak, **`cell2mat(IdentifiedPeaks(2,2))`** returns its name, etc. Obviously, for your own applications, it is up to you to provide your own array of known peak maximum positions ('Positions') and matching cell array of names ('Names') for your particular types of signals. The related function idpeaktable.m does the same thing for a peak table P returned by any of my peak finder or peak fitting functions, having one row for each peak and columns for peak number, position, and height as the first three columns. The syntax is **`[IdentifiedPeaks] = idpeaktable(P, maxerror, Positions, Names).`** The interactive iPeak function described in the next section has this function built-in as one of the keystroke commands (page 188).

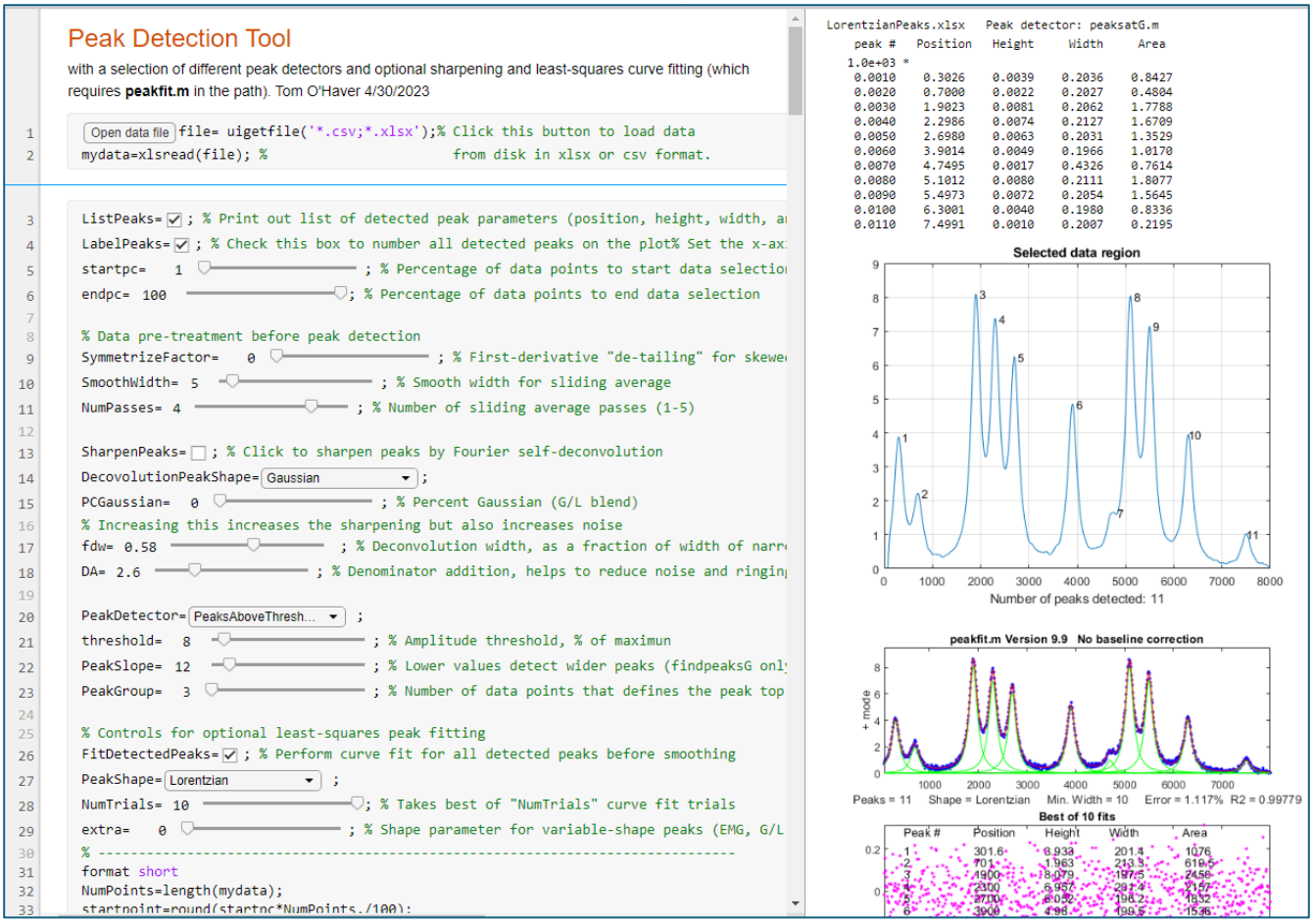
Example: Download idpeaks.zip, extract it, and place the extracted files in the Matlab or Octave search path. This contains a high-resolution atomic emission spectrum of copper ('spectrum', x = wavelength in nanometers; y = amplitude) and a data table of known Cu I and Cu II atomic lines ('DataTable') containing the positions and names of many copper lines. The idpeaks function detects and measures the peak locations of all the peaks in "spectrum", then looks in 'DataTable' to see if any of those peaks are within .01 nm of any entry in the table and prints out the peaks that match.

```
>> load DataTable
>> load spectrum
>> idpeaks(Cu,0.01,.001,5,5,.01,Positions,Names)

ans=
  'Position'    'Name'            'Error'          'Amplitude'
  [  221.02]    'Cu II 221.027'   [ -0.0025773]    [ 0.019536]
  [  221.46]    'Cu I 221.458'    [ -0.0014301]    [   0.4615]
  [  221.56]    'Cu I 221.565'    [-0.00093125]    [  0.13191] …
```

Live Script Peak detection tool

PeakDetection.mlx is an interactive Live Script (page 368) for peak detection and measurement. It collects into one easy-to-use tool several of the peak-related functions previously described, including a selection of peak detectors, data smoothing, symmetrization, peak sharpening, and curve fitting, with interactive sliders and drop-down menus to control them interactively.



Clicking the **OpenDataFile** button in line 1 opens a file browser, allowing you to navigate to your data file (in .csv or .xlsx format). Start by de-selecting the **FitDetectedPeaks** check box on line 26 to make peak detection adjustments faster. The **startpc** and **endpc** sliders in lines 5 and 6 allow you to set the start and end of the region to focus on (expressed as a percentage of the total data length). You can set controls to smooth the data (lines 10 and 11) or to "de-tail" or symmetrize the peaks (line 9). You can choose a peak detector using the **PeakDetector** drop-down menu in line 20. The **ListPeaks** and **LabelPeaks** check boxes in lines 3 and 4 allow you to number the peaks on the graph and/or to display a list of peak parameters of the detected peaks. You can optionally try to sharpen the peaks, to enable detection of weak side peak or shoulders, by clicking the **SharpenPeaks** check box in line 13.

You can optionally apply iterative least-square curve fitting, by clicking the **FitDetectedPeaks** check box on line 26 and selecting the desired fitting function shape from the **PeakShape** drop-down menu on line 27. The position and width of the peaks estimated by the peak detectors is used as the first-guess starting point for the iterative fit, so *only detected peaks will be included in the fit*. This function requires that peakfit.m be in the Matlab path. (Normally, curve fitting uses only the unsmoothed data. However, if peak sharpening or symmetrization is applied in line 9 or 13, it uses the processed data). The function of each of the controls is described in the associated comment lines.

Examples of its application. For illustrations of its application to peaks with different shapes and overlap, see the PDF file **PeakDetector.pdf**, which references a set of .csv data files that are downloadable from the same address. (To see the graphs shown on the right of the script as above,

right-click on the right panel and select "Disable synchronous scrolling").

iPeak: Keypress-operated interactive peak detector

iPeak (ipeak.m or ipeakoctave.m) is an interactive peak finder for time series data, based on the "findpeaksG.m" and "findpeaks**L**.m" functions. The interactive keypress operation works on your computer, even if you run Matlab in a web browser, but not on Matlab Mobile. Its basic operation is similar to **iSignal** and **ipf.m.** It accepts data in a single vector, a pair of vectors, or a matrix with the independent variable in the first column and the dependent variable in the second column. If you call *iPeak* with *only* those one or two input arguments, it estimates a default initial value for the peak detection parameters (AmpThreshold, SlopeThreshold, SmoothWidth, and FitWidth) based on the formulas below and displays those values at the bottom of the screen.

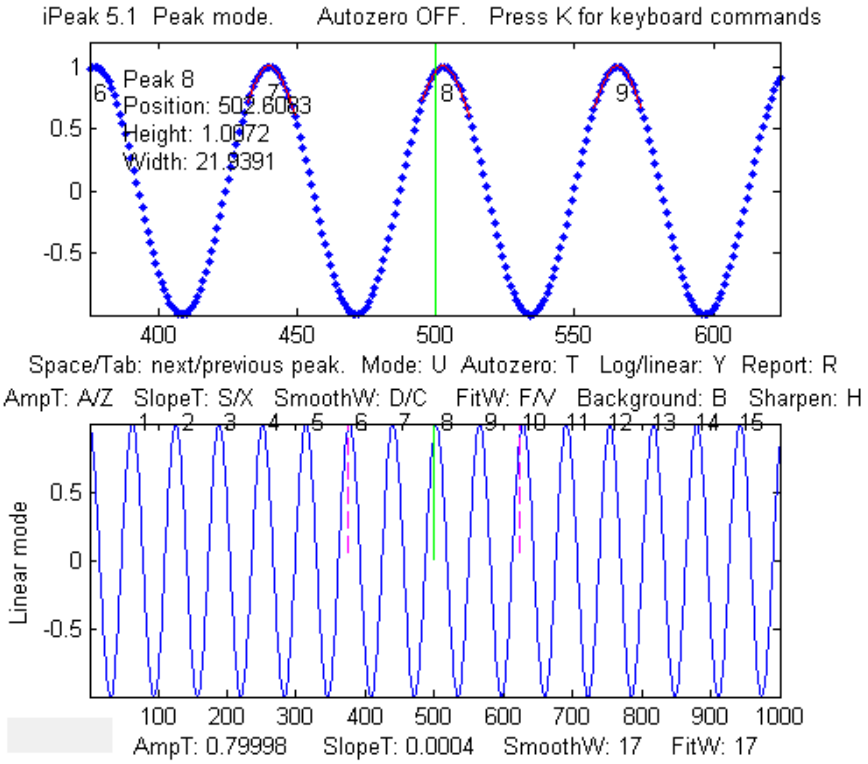
```
WidthPoints=length(y)/20;
SlopeThreshold=WidthPoints^-2;
AmpThreshold=abs(min(y)+0.1*(max(y)-min(y)));
SmoothWidth=round(WidthPoints/3);
FitWidth=round(WidthPoints/3);
```

You can then fine-tune the peak detection up/down by using these pairs of adjacent keys:

Amplitude threshold: **A/Z**
Slope threshold: **S/X**
Smooth width: **D/C**
Fit width: **F/V**.

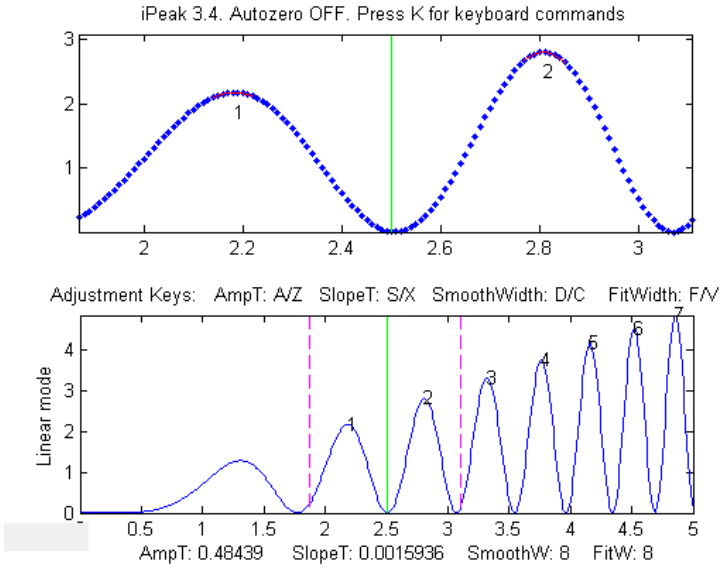
Example 1: One input argument, data in single vector:

```
>> y=cos(.1:.1:100);
>> ipeak(y)
```



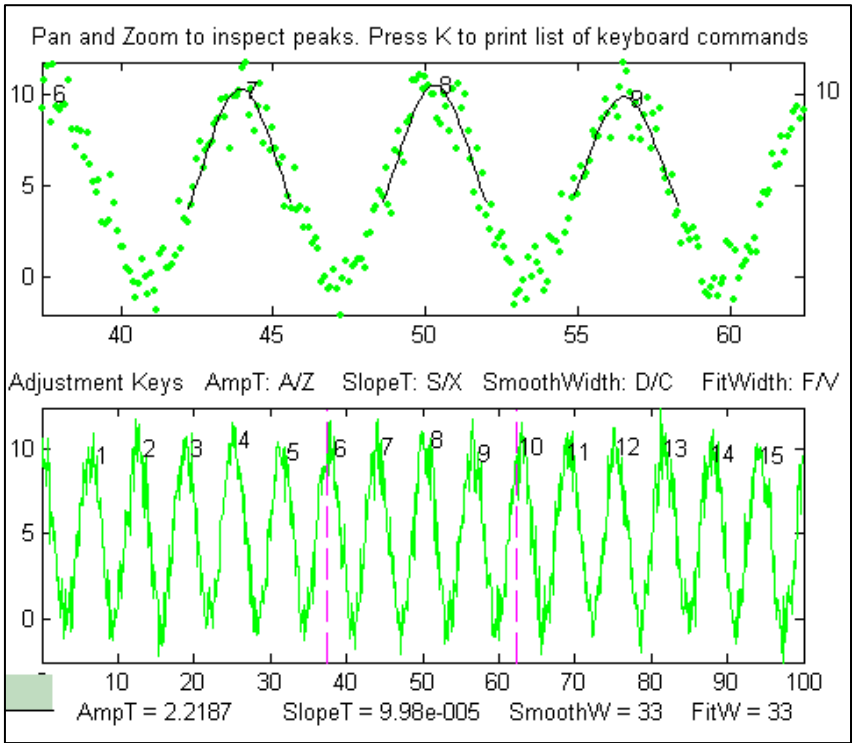
Example 2: One input argument, data in two columns of a matrix:

```
>> x=[0:.01:5]';
>> y=x.*sin(x.^2).^2;M=[x y];
>> ipeak(M)
```



Example 3: Noisy data. Two input arguments, data in separate x and y vectors:

```
>> x=[0:.1:100];
>> y=(x.*sin(x)).^2;
>> ipeak(x,y);
```



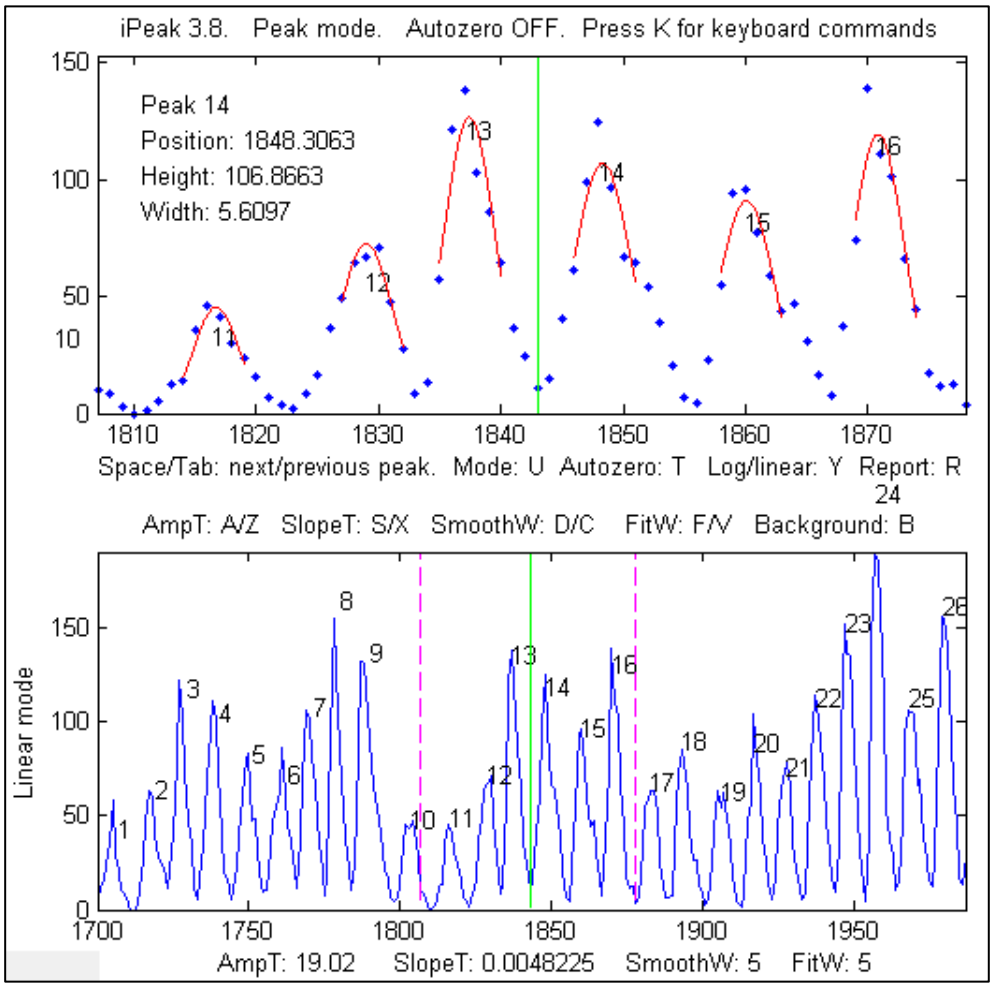
Double-click the Matlab figure window title bar to expand it to full screen. Double-click again to return the figure window to its former size and position.

Example 4: When you start *iPeak* using the simple syntax as illustrated above, the initial values of the peak detection parameters are calculated by the program, but if it starts off by picking up far *too many* or *too few* peaks, you can add an additional input argument (after the data) to control *peak sensitivity*.

```
>> x=[0:.1:100];y=5+5.*cos(x)+randn(size(x));ipeak(x,y,10);
   or  >> ipeak([x;y],10);
   or  >> ipeak(humps(0:.01:2),3)
   or  >> x=[0:.1:10];y=exp(-(x-5).^2);ipeak([x' y'],1)
```

This additional numeric argument is an estimate of *maximum peak density* (PeakD), the ratio of the typical peak width to the length of the entire data record. Small values detect fewer peaks. Larger values detect more peaks. It affects only the *starting* values for the peak detection parameters. (It is just a quick way to set reasonable initial values of the peak detection parameters, so you will not have so much adjusting to do. You can still fine-tune the peak detection parameters individually).

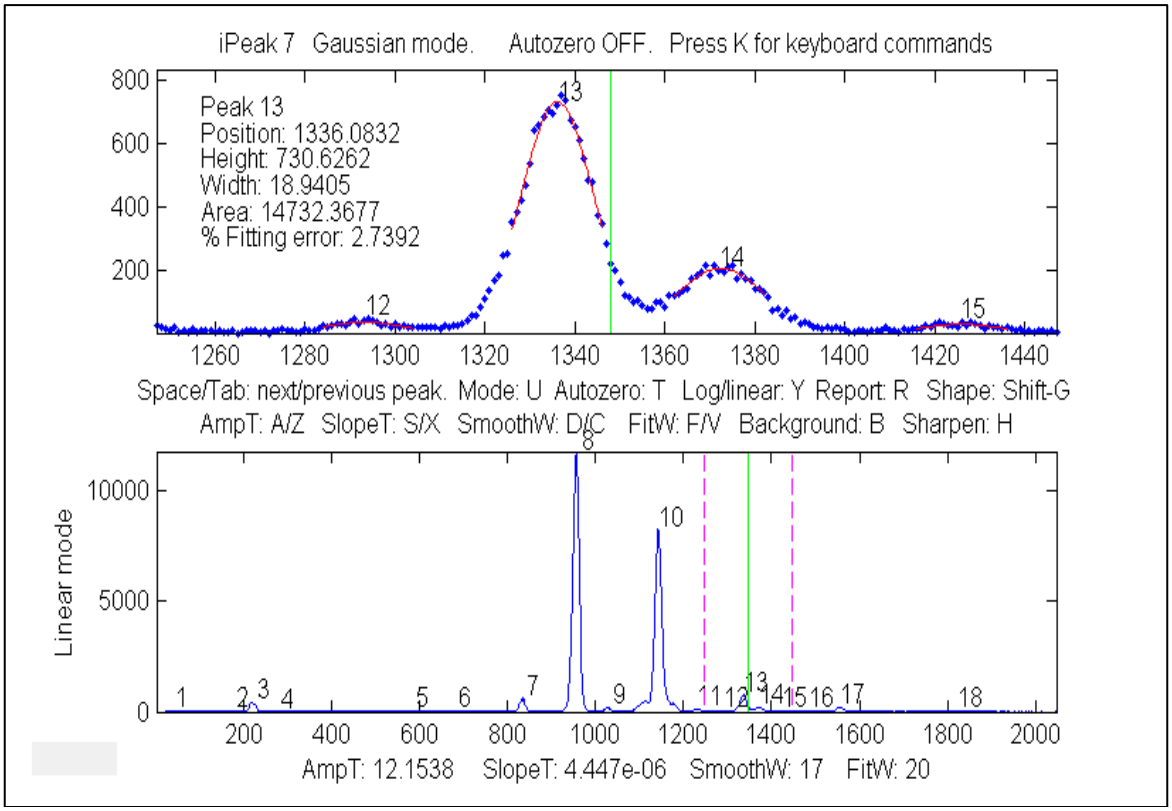
```
>> load sunspots
>> ipeak(year,number,20)
```



Peaks in annual sunspot numbers from 1700 to 2009 (download the datafile). Sunspot data downloaded from NOAA

iPeak displays the entire signal in the lower half of the Figure window and an adjustable zoomed-in

section in the upper window. Pan and zoom the portion in the upper window using the cursor arrow keys. The peak closest to the center of the upper window is labeled in the upper left of the top window, and its peak position, height, and width are listed. The **Spacebar/Tab** keys jump to the next/previous detected peak and display it in the upper window at the current zoom setting (use the up and down cursor arrow keys to adjust the zoom range). Or you can press the **J** key to jump to a specified peak number. Double-click the figure window title bar to expand to full screen for a closer view.



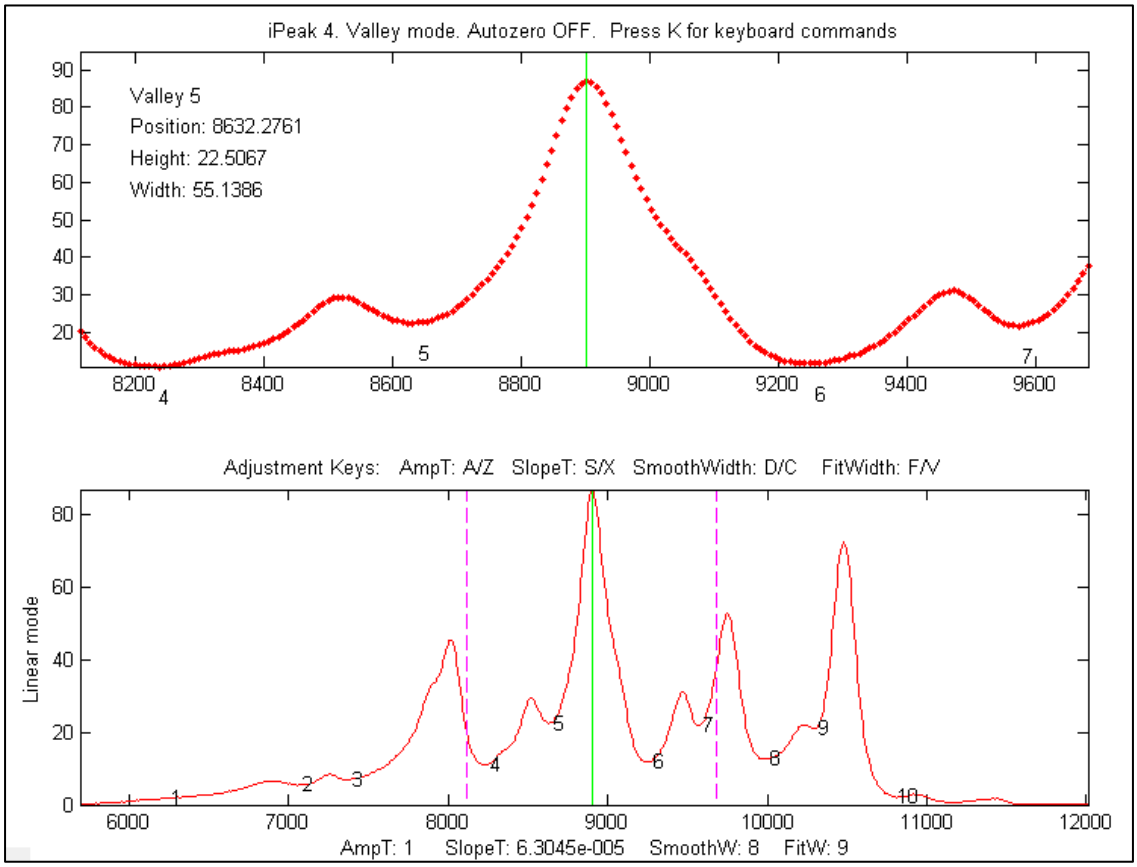
Adjust the peak detection parameters AmpThreshold (**A/Z** keys), SlopeThreshold (**S/X**), SmoothWidth (**D/C**), FitWidth (**F/V**) so that it detects the desired peaks and ignores those that are too small, too broad, or too narrow to be of interest. You can also type in a specific value of AmpThreshold by pressing **Shift-A** or a specific value of SlopeThreshold by pressing **Shift-S**. Detected peaks are numbered from left to right.

Press **P** to display the peak table of all the detected peaks (Peak #, Position, Height, Width, Area, and percent fitting error)**:**

```
Gaussian shape mode (press Shift-G to change)
Window span: 169 units
Linear baseline subtraction
   Peak#     Position     Height      Width       Area       Error
     1       500.93       6.0585      34.446      222.17      9.5731
     2       767.75       1.8841     105.58       211.77     25.979
     3       1012.8       0.20158     35.914        7.7      269.21
          ..............
```

Press **Shift-G** to cycle between Gaussian**,** Lorentzian, and flat-top shape modes. Press **Shift-P** to save peak table as disc file. Press **U** to switch between peak and valley mode. Do not forget that only valleys *above* (that is, more positive or less negative than) the AmpThreshold are detected. If you wish to

detect valleys that have negative minima, then AmpThreshold must be set more negative than that. Note: to speed up the operation for signals over 100,000 points in length, the lower window is refreshed only when the number of detected peaks changes or if the **Enter** key is pressed. Press **K** to see all the keystroke commands.



*The Valley mode. Press **U** key to switch between peak and valley mode.*

If the density of data points on the peaks is *too low* - less than about 4 points - the peaks may not be reliably detected. You can improve reliability by using the interpolation command (**Shift-I**) to re-sample the data by linear interpolation to a *larger* number of points. Conversely, if the density of data points on the peaks of interest is very high - say, more than 100 points per peak - then you can speed up the operation of *iPeak* by re-sampling to a *smaller* number of points.

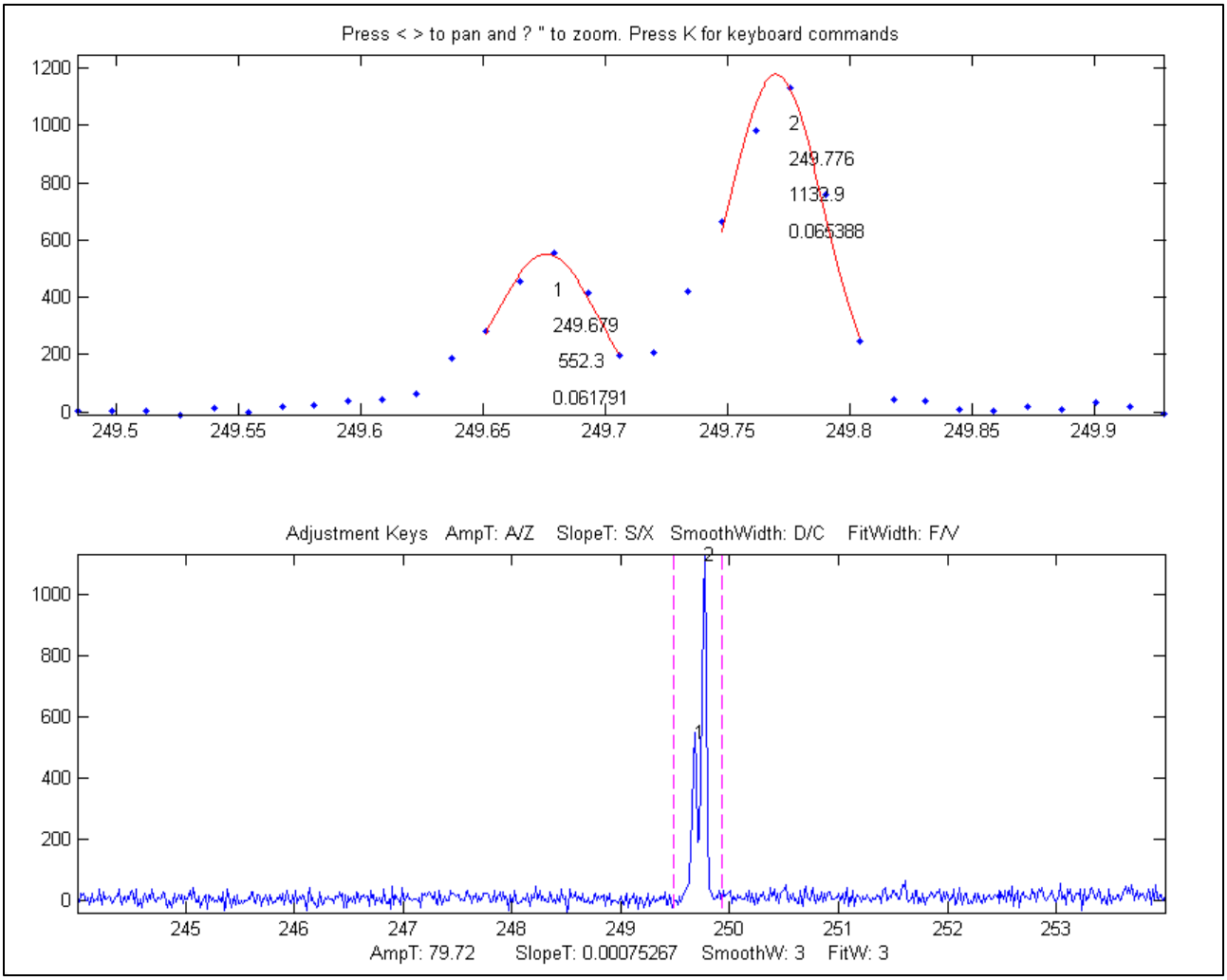
Peak Summary Statistics in iPeak. The **E** key prints a table of summary statistics of the peak intervals (the x-axis interval between adjacent detected peaks), heights, widths, and areas, listing the maximum, minimum, average, and percent standard deviation, and displaying the *histograms* of the peak intervals, heights, widths, and areas in figure window 2.

Peak Summary Statistics
149 peaks detected
No baseline correction

	Interval	Height	Width	Area
Maximum	1.3204	232.7724	0.33408	80.7861
Minimum	1.1225	208.0581	0.27146	61.6991
Mean	1.2111	223.3685	0.31313	74.4764
% STD	2.8931	1.9115	3.0915	4.0858

Example 5: Six input arguments. As above, but input arguments 3 to 6 directly specifies initial values of AmpThreshold (AmpT), SlopeThreshold (SlopeT), SmoothWidth (SmoothW), FitWidth (FitW). PeakD is ignored in this case, so just type a '0' as the second argument after the data matrix).

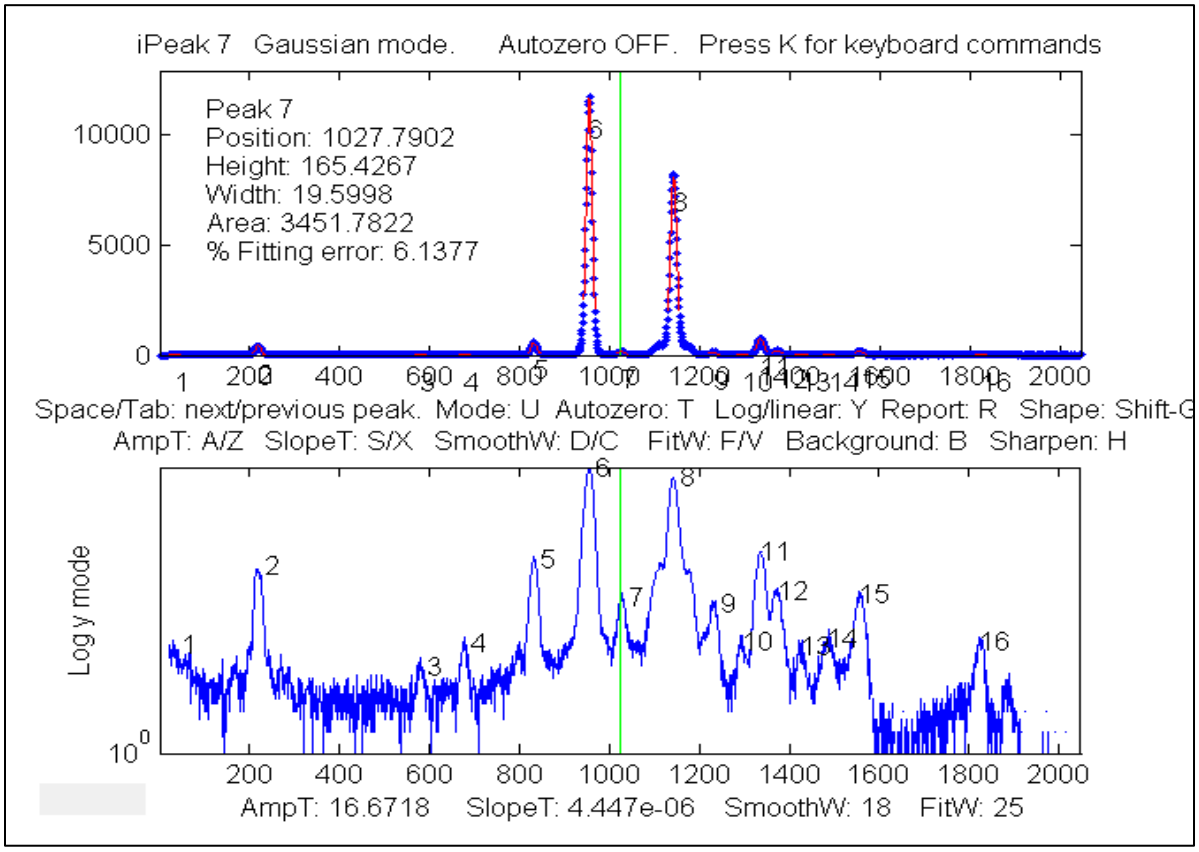
```
>> ipeak(datamatrix,0,.5,.0001,20,20);
```



Pressing 'L' toggles ON and OFF the peak labels in the upper window.

Keystrokes allow you to pan and zoom the upper window, to inspect each peak in detail if desired. You can set the initial values of pan and zoom in optional input arguments 7 ('xcenter') and 8 ('xrange'). See example 6 below.

The **Y** key toggles between linear and log y-axis scale in the *lower window* (a log axis is good for inspecting signals with high dynamic range). It effects only the lower window display and *has no effect on the data* itself or on the peak detection and measurements.



Log scale (Y key) makes the smaller peaks easier to see in the lower window.

Example 6: Eight input arguments. As above, but input arguments 7 and 8 specify the initial pan and zoom settings, 'xcenter' and 'xrange', respectively. In this example, the x-axis data are wavelengths in nanometers (nm), and the upper window zooms in on a very small 0.4 nm region centered on 249.7 nm. (These data, provided in the ZIP file, are from a high-resolution atomic spectrum).

```
>> load ipeakdata.mat
>> ipeak(Sample1,0,100,0.05,3,4,249.7,0.4);
```

Baseline correction modes. The **T** key cycles the *baseline correction mode* from *off, linear, quadratic*, *flat*, *linear mode(y), flat mode(y)*, and then back to *off*. The current mode is displayed above the upper panel. When the baseline correction mode is OFF, peak heights are measured relative to zero. (Use this mode when the baseline is zero or if you have previously subtracted the baseline from the entire signal using the **B** key). In the *linear* or *quadratic* modes, peak heights are automatically measured relative to the local baseline interpolated from the points at the ends of the segment displayed in the upper panel. Use the zoom controls to isolate a group of peaks so that the signal returns to the local baseline at the beginning and end of the segment displayed in the upper window. The peak heights, widths, and areas in the peak table (**R** or **P** keys) will be automatically corrected for the baseline. The *linear* or *quadratic* modes will work best if the peaks are well separated so that the signal returns to the local baseline between the peaks. (If the peaks are highly overlapped, or if they are not Gaussian in shape, the best results will be obtained by using the curve fitting function - the **N** or **M** keys. The *flat mode* is used only for curve fitting function, to account for a flat baseline offset *without* reference to the edges of the signal segment being fit). The *mode(y)* method subtracts

the *most common y value* from all the points in the selected region. For peak-type signals where the peaks usually return to the baseline between peaks, this is usually the baseline even if the signal does not return to the baseline at the ends like modes 2 and 3 (graphic example).

Example 7: Nine input arguments. As example 6, but the 9th input argument sets the background correction mode (equivalent to pressing the **T** key): 0=OFF, 1=linear, 2=quadratic, 3=flat, 4=mode(y). If not specified, it is initially OFF.

```
>> ipeak(Sample1,0,100,0.00,3,4,249.7,0.4,1);
```

Converting to command-line functions. To aid in writing your own scripts and functions to automate processing performed with iPeak, the '**Q**' key prints out the findpeaksG, findpeaksb, and findpeaksfit commands for the segment of the signal in the upper window and for the entire signal, with the input arguments in place, which you can then Copy and Paste into your own scripts. The '**W**' key similarly prints out the peakfit.m and ipf.m commands.

Shift-Ctrl-S transfers the current signal to i**S**ignal.m (page 376) and **Shift-Ctrl-P** transfers the current signal to **I**nteractive **P**eak detector (iPeak.m), if those functions are installed in your Matlab path.

Ensemble averaging in *iPeak*. For signals that contain repetitive waveform patterns occurring in one continuous signal, with nominally the same shape except for noise, the ensemble averaging function (**Shift-E**) can compute the average of all the repeating waveforms. It works by detecting a single peak in each repeat waveform to synchronize the repeats (and therefore does not require that the repeats be equally spaced or synchronized to an external reference signal). To use this function, first adjust the peak detection controls to detect *only one peak in each repeat pattern*, and then zoom in to isolate any one of those repeat patterns, and then press **Shift-E**. The average waveform is displayed in Figure 2 and saved as "EnsembleAverage.mat" in the current directory. See the script iPeakEnsembleAverageDemo.m.

De-tailing peaks with exponential tails in *iPeak*. If your signal has peaks that tail to the right or left because they have been exponentially broadened, you can remove the tails by the first-derivative addition technique (page 83): press **Shift-Y,** enter an estimate of the exponential time constant and then use the **1** and **2** keys to adjust it by 10% per keystroke (or **Shift-1** and **Shift-2** to adjust by 1% per keystroke). Click for animation. Increase the factor until the baseline after the peak goes negative, then increase it slightly so that it is *as low as possible but not negative.* This results in narrower, taller peaks, but has no effect on the peak areas. The effect is like deconvoluting the exponential function from the broadened peak, but it is faster and simpler. (It the peaks tail to the *left* rather than the right, use a *negative* factor).

Normal and Multiple Peak fitting in *iPeak*: The **N** key applies iterative curve fitting to the *detected peaks that are displayed in the **upper window*** (referred to here as "**N**ormal" curve fitting). The use of the iterative least-squares function can result in more accurate peak parameter measurements than the normal peak table (**R** or **P** keys), especially if the peaks are non-Gaussian in shape or are highly overlapped. (If the peaks are superimposed on a background, first select the **baseline correction** mode using the **T** key, then use the pan and zoom keys to select a peak or a group of overlapping peaks in the upper window, with the signal returning all the way to the local baseline at the

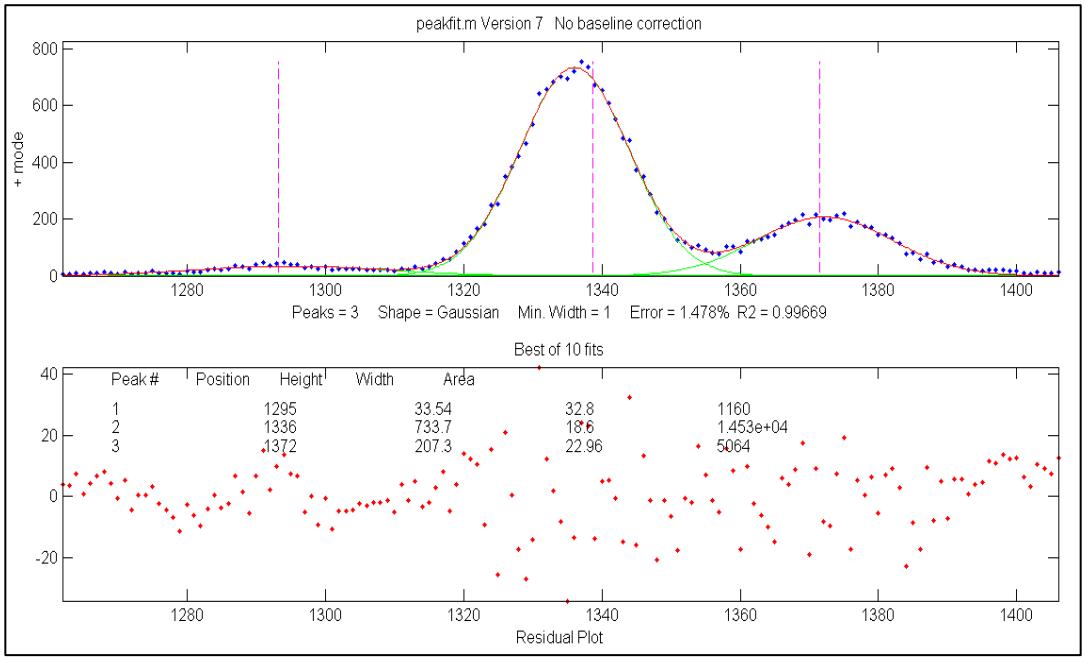
ends of the upper window if you are using the linear or quadratic baseline modes. See page 219). Make sure that the AmpThreshold, Slope-Threshold, SmoothWidth are adjusted so that each peak is numbered once. Only numbered peaks are fitted. Then press the **N** key, which will display this **menu of peak shapes** (graphic on page 424):

```
Gaussians: y=exp(-((x-pos)./(0.6005615.*width)) .^2)
  Gaussians with independent positions and widths.................1
(default)
  Exponentially--broadened Gaussian (equal time constants).........5
  Exponentially--broadened equal-width Gaussian....................8
  Fixed-width exponentially-broadened Gaussian....................36
  Exponentially--broadened Gaussian (independent time constants)...31
  Gaussians with the same widths...................................6
  Gaussians with preset fixed widths..............................11
  Fixed-position Gaussians........................................16
  Asymmetrical Gaussians with unequal half-widths on both sides....14
Lorentzians: y=ones(size(x))./(1+((x-pos)./(0.5.*width)).^2)
  Lorentzians with independent positions and widths................2
  Exponentially--broadened Lorentzian.............................18
  Equal-width Lorentzians..........................................7
  Fixed-width Lorentzian.........................................12
  Fixed-position Lorentzian......................................17
Gaussian/Lorentzian blend (equal blends).........................13
  Fixed-width Gaussian/Lorentzian blend..........................35
  Gaussian/Lorentzian blend with independent blends)..............33
Voigt profile with equal alphas..................................20
  Fixed-width Voigt profile with equal alphas.....................34
  Voigt profile with independent alphas...........................30
Logistic: n=exp(-((x-pos)/(.477.*wid)).^2); y=(2.*n)./(1+n).......3
Pearson: y=ones(size(x))./(1+((x-pos)./((0.5.^(2/m)).*wid)).^2).^m..4
  Fixed-width Pearson.............................................37
  Pearson with independent shape factors, m.......................32
Breit-Wigner-Fano................................................15
Exponential pulse: y=(x-tau2)./tau1.*exp(1-(x-tau2)./tau1).......9
Alpha function: y=(x-spoint)./pos.*exp(1-(x-spoint)./pos);.......19
Up Sigmoid (logistic function): y=.5+.5*erf((x-tau1)/sqrt(2*tau2)).10
Down Sigmoid y=.5-.5*erf((x-tau1)/sqrt(2*tau2))..................23
Triangular......................................................21
```

Type the number for the desired peak shape from this table and press **Enter**, then type in a number of repeat trial fits and press **Enter** (the default is 1. Start with that and then increase if necessary). If you have selected a variable-shape peak (e.g., numbers 4, 5, 8 ,13, 14, 15, 18, 20, 30-33), the program will ask you to type in a number that fine-tunes the shape. The program will then perform the fit, display the results graphically in Figure window 2, and print out a table of results in the command window, e.g.:

```
Peak shape (1-8): 2
Number of trials: 1

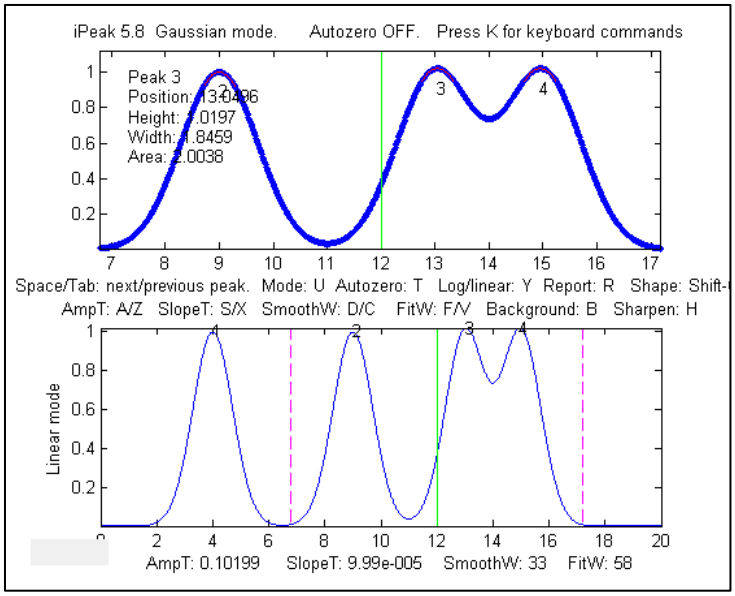
Least-squares fit to Lorentzian peak model
Fitting Error 1.1581e-006%
  Peak#   Position   Height   Width   Area
    1        100        1       50    71.652
    2        350        1      100    146.13
    3        700        1      200    267.77
```



*Normal Peak Fit (**N** key) applied to a group of three overlapping Gaussians peaks*

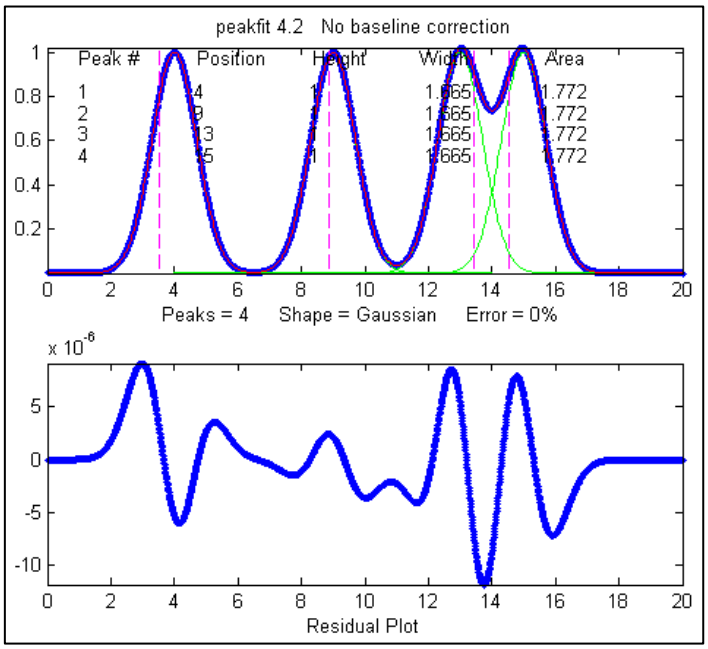
There is also a "Multiple" peak fit function (**M** key) that will attempt to apply iterative curve fitting to ***all the detected peaks*** *in the signal simultaneously*. Before using this function, it is best to turn ***off*** the automatic baseline correction (**T** key) and use the *multi-segment baseline correction function* (**B** key) to remove the background (because the baseline correction function will probably not be able to subtract the baseline from the entire signal). Then press **M** and proceed as for the normal curve fit. A multiple curve fit may take a minute or so to complete if the number of peaks is large, possibly longer than the Normal curve fitting function on each group of peaks separately.

The **N** and **M** key fitting functions perform non-linear iterative curve fitting using the peakfit.m function. The number of peaks and the starting values of peak positions and widths for the curve fit function are automatically supplied by the findpeaksG function, so it is essential that the peak detection variables in *iPeak* be adjust so that all the peaks in the selected region are detected and numbered once. (For more flexible curve fitting, use ipf.m, page 416, which allows manual optimization of peak groupings and start positions).



Example 8. This example generates four Gaussian peaks, all with the exact same peak height (1.00) and area (1.773). The first peak (at x=4) is isolated, the second peak (x=9) is slightly overlapped with the third one, and the last two peaks (at x= 13 and 15) are strongly overlapped.

```
x=[0:.01:20];
y=exp(-(x-4).^2)+exp(-(x-9).^2)+exp(-(x-13).^2)+exp(-(x-15).^2);
ipeak(x,y)
```



By itself, *iPeak* does a fairly good job of measuring peaks positions and heights by fitting just the top part of the peaks, because in this example the peaks are Gaussian. However, the areas and widths of the last two peaks (which should be 1.665 like the others) are quite a bit too large because of the overlap:

Peak#	Position	Height	Width	Area
1	4	1	1.6651	1.7727
2	9	1	1.6651	1.7727
3	13.049	1.02	1.8381	1.9956
4	14.951	1.02	1.8381	1.9956

In this case, curve fitting (using the **N** or **M** keys) does a much better job, even if the overlap is even greater, but *only if the peak shape is known*:

Peak#	Position	Height	Width	Area
1	4	1	1.6651	1.7724
2	9	1	1.6651	1.7725
3	13	1	1.6651	1.7725
4	15	0.99999	1.6651	1.7724

Note 1: If the peaks are too overlapped to be detected and numbered separately, try pressing the **H** key to activate the even-derivative sharpen function before pressing **M** (version 4.0 and above only). This affects only the peak detection, not the signal itself.

Note 2: If you plan to use a variable-shape peak (numbers 4, 5, 8 ,13, 14, 15, 18, or 20) for the **M**ultiple peak fit, it is a good idea to obtain a reasonable value for the requested "extra" shape parameter by performing a **N**ormal peak fit on an isolated single peak (or small group of partly-overlapping peaks) of the same shape, then use that value for the **M**ultiple curve fit of the entire signal.

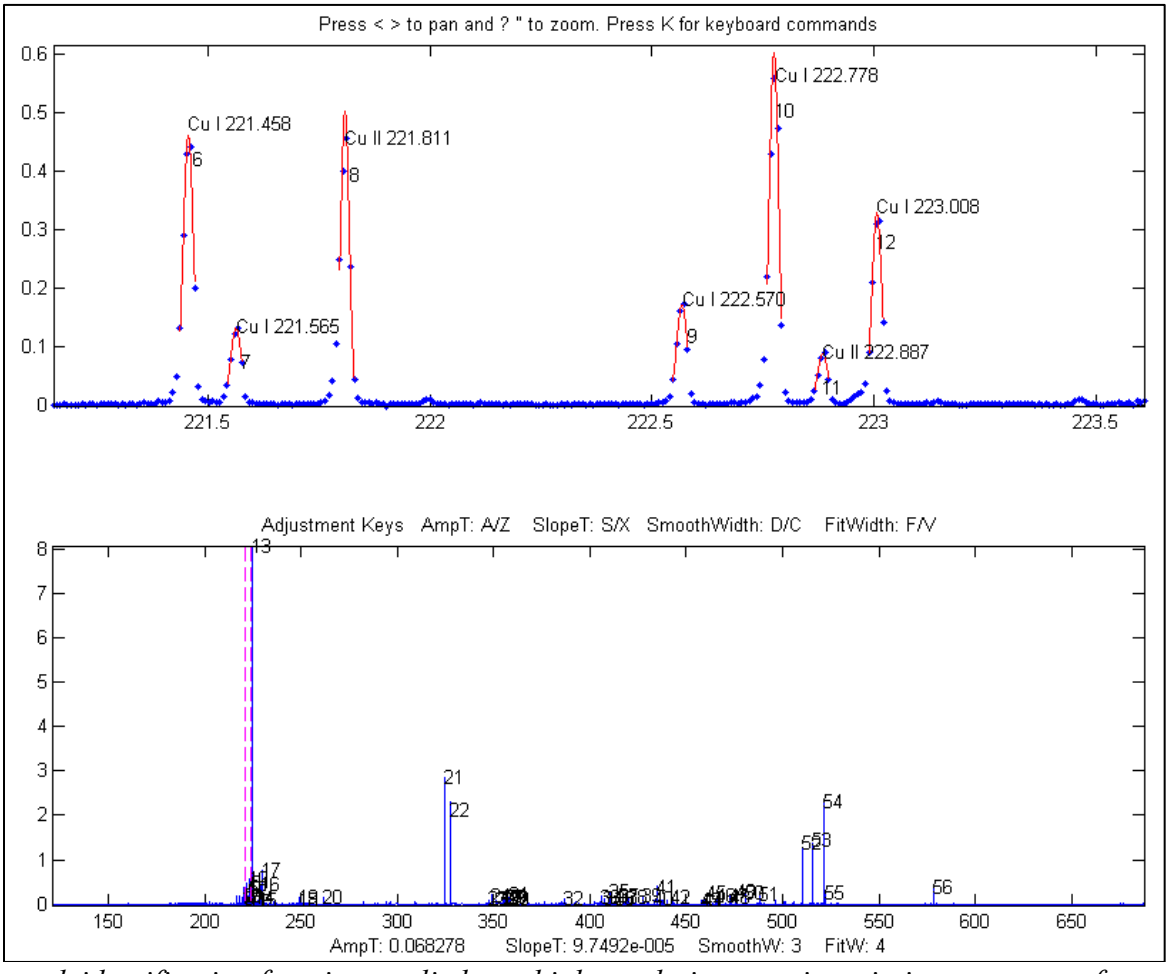
Note 3: If the peak shape varies across the signal, you can either use the **N**ormal peak fit to fit each section with a different shape rather than the **M**ultiple peak fit, or you can use the *unconstrained* shapes

that fit the shape individually for each peak: Voigt (30), ExpGaussian (31), Pearson (32), or Gaussian/Lorentzian blend (33).

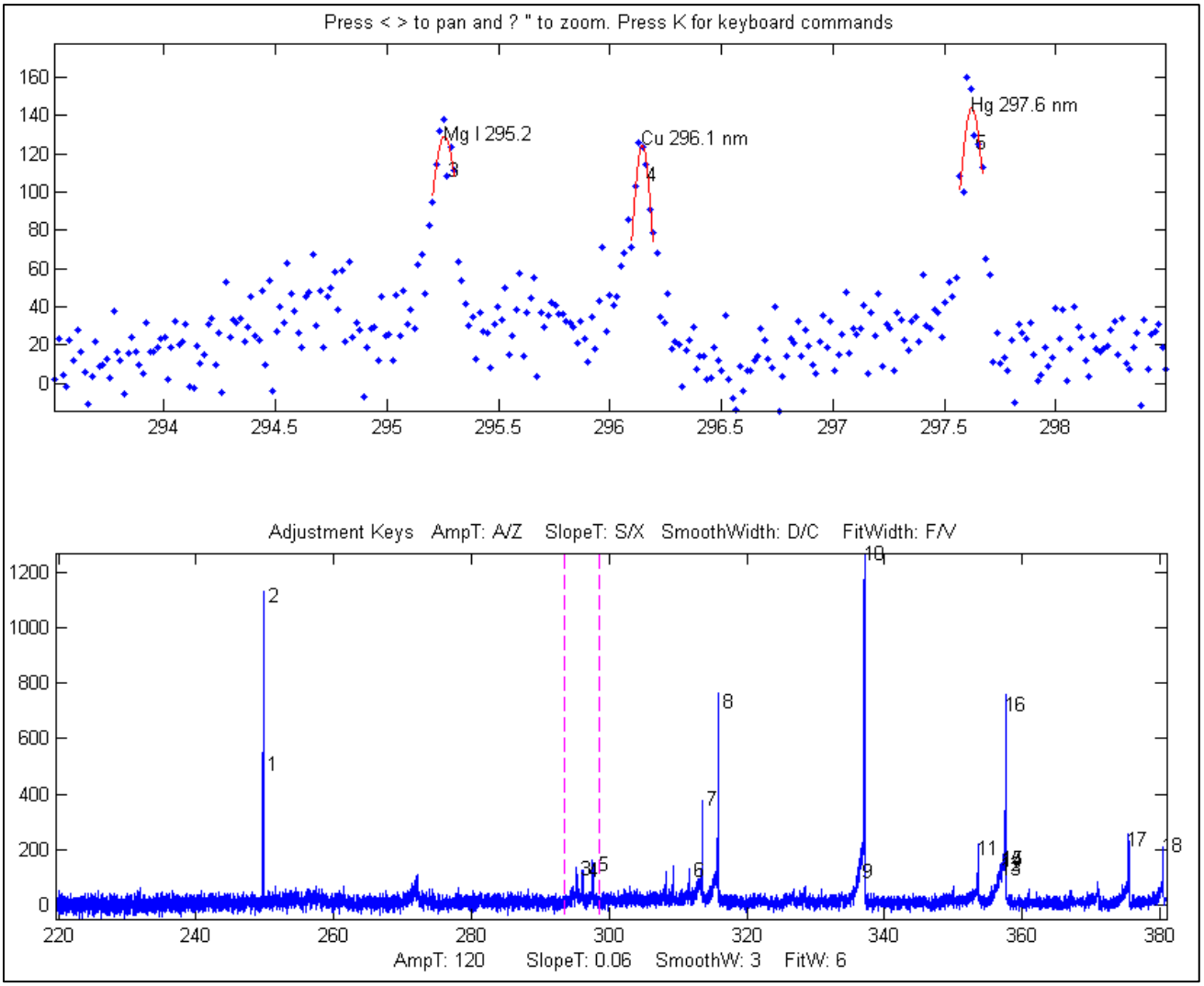
Peak identification in *iPeak*. There is an optional "peak identification" operation if the optional input arguments 9 ('MaxError'), 10 ('Positions'), and 11 ('Names') are included. The "**I**" key toggles this function ON and OFF. This function compares the found peak positions (maximum x-values) to a *reference database of known peaks* that is supplied in input arguments 10 and 11 in the form of arrays of known peak maximum positions ('Positions') and matching cell array of names ('Names'). If the position of a found peak in the signal is closer to one of the known peaks by less than the specified maximum error ('MaxError'), then that peak is considered a match and its name is displayed next to the peak in the upper window. When the '**O**' key is pressed (the letter 'O'), the peak positions, names, errors, and amplitudes are printed out in a table in the command window.

Example 9: Application to atomic spectra. Eleven input arguments. As above, but also specifies 'MaxError', 'Positions', and 'Names' in optional input arguments 9, 10, and 11, for peak identification function. Pressing the '**I**' key toggles off and on the peak identification labels in the upper window. These data (provided in this ZIP file) are from a high-resolution atomic).

```
>> load ipeakdata.mat
>> ipeak(Sample1,0,100,0.05,3,6,296,5,0.1,Positions,Names);
```



The peak identification function applied to a high-resolution atomic emission spectrum of copper.



An atomic emission spectrum of a sample containing trace amounts of several elements. iPeak is used to zoom in to three small peaks near 296 nm. iPeak identified and labeled three peaks based on the atomic line data in `ipeakdata.mat`. *Press the* ***I*** *key to display the peak ID names. Double-click the figure window title bar to expand to full screen for an even better view.*

Pressing "O" prints the peak positions, names, errors, and amplitudes in a table in the command window.

Name	Position	Error	Amplitude
'Mg I 295.2'	[295.2]	[0.058545]	[129.27]
'Cu 296.1 '	[296.1]	[0.045368]	[124.6]
'Hg 297.6 '	[297.6]	[0.023142]	[143.95]

Here is another example, from a large atomic emission spectrum with over 10,000 data points and many hundreds of peaks. The reference table of known peaks in this case is taken from Table 1 of ASTM C1301 - 95(2009)e1. With the settings I was using, ten peaks were identified, shown in the table below. You can see that some of these elements have more than one line identified. Obviously, the lower the settings of the AmpThreshold, SlopeThreshold, and SmoothWidth, the more peaks will be detected. And the higher the setting of "MaxError", the more peaks will be close enough to be considered identified. In this example, the element names in the table below are hot-linked to the screen

image of the corresponding peak detected as identified by *iPeak*. Some of these lines, especially Nickel 231.66nm, Silicon 288.18nm, and Iron 260.1nm, are rather weak and thus have poor signal-to-noise ratios, so their identifications might be in doubt (especially Iron, because its wavelength error is greater than the rest). It is up to you to decide which peaks are strong enough to be significant. In this example, I used an *independently published table of element wavelengths*, rather than data acquired on that *same* instrument, so these results really do depend on the accurate wavelength calibration of the instrument. The results suggest that the wavelength calibration is in fact excellent, based on the small errors for the *two well-known and relatively strong sodium lines* at 589 and 589.59 nm. I set the wavelength matching requirement (MaxError) to 0.2 nm in this example. (The identification of iron at 260.1 nm might be doubted, based on its relatively large wavelength error and low amplitude).

'Name'	'Position'	'Error'	'Amplitude'
'Cadmium'	[226.46]	[-0.039471]	[44.603]
'Nickel'	[231.66]	[0.055051]	[26.381]
'Silicon'	[251.65]	[0.041616]	[45.275]
'Iron'	[260.1]	[0.156]	[38.04]
'Silicon'	[288.18]	[0.022458]	[27.214]
'Strontium'	[421.48]	[-0.068412]	[41.119]
'Barium'	[493.35]	[-0.057923]	[72.466]
'Sodium'	[589]	[0.0057964]	[405.23]
'Sodium'	[589.57]	[-0.015091]	[315.2]
'Potassium'	[766.54]	[0.051585]	[61.987]

Note: The ZIP file contains the latest version of the *iPeak* function as well as some sample data to demonstrate peak identification (Example 8). Obviously for your own applications, it is up to you to provide your own array of known peak maximum positions ('Positions') and matching cell array of names ('Names') for your particular types of signals.

iPeak keyboard Controls (version 8.1):

```
Pan signal left and right...Coarse pan: < or >
                            Fine pan: left or right cursor arrow keys
                            Nudge one point left or right: [ and ]
Zoom in and out.............Coarse zoom: / or '
                            Fine zoom: up or down cursor arrow keys
Resets pan and zoom.........ESC
Select entire signal........Ctrl-A
Refresh entire plot.........Enter (Updates cursor position in lower plot)
Change plot color...........Shift-C (cycles through standard colors)
Adjust AmpThreshold.........A,Z  (Larger values ignore short peaks)
Type in AmpThreshold........Shift-A  (Type value and press Enter)
Adjust SlopeThreshold.......S,X  (Larger values ignore broad peaks)
Type in SlopeThreshold......Shift-S  (Type value and press Enter)
Adjust SmoothWidth..........D,C  (Larger values ignore sharp peaks}
Adjust FitWidth.............F,V (Adjust to cover top part of peaks)
Toggle sharpen mode ........H  Helps detect overlapped peaks.
Enter de-tailing factor.....Shift-Y. Removes exponential peak tails
Adjust de-tailing 10%.......1,2  Adjust peak de-tailing factor 10%.
Adjust de-tailing 1%........Shift-1,Shift-2  Adjust de-tailing 1%.
Baseline correction.........B, then click baseline at multiple points
Restore original signal.....G  to cancel previous background subtraction
Invert signal...............-  Invert (negate) the signal (flip + and -)
```

```
Set minimum to zero.........0 (Zero) Sets minimum signal to zero
Interpolate signal..........Shift-I  Interpolate (re-sample) to N points
Toggle log y mode OFF/ON....Y  Plot log Y axis on lower graph
Cycles baseline modes.......T 0=none; 1=linear; 2=quad; 3=flat;
                              4=linear mode(y); 5=flat mode(y)
Toggle valley mode OFF/ON...U  Switch to valley mode
Gaussian/Lorentzian mode....Shift-G Cycle between Gaussian,
                                    Lorentzian, and flat-top modes
Print peak table............P  Prints Peak #, Position, Height, Width
Save peak table.............Shift-P  Saves peak table as disc file
Step through peaks..........Space/Tab  Jumps to next/previous peak
Jump to peak number.........J  Type peak number and press Enter
Normal peak fit.............N  Fit peaks in upper window with peakfit.m
Multiple peak fit...........M  Fit all peaks in signal with peakfit.m
Ensemble average all peaks..Shift-E  (Read instructions first)
Print keyboard commands.....K  Prints this list
MeasUre peak areas .........Shift-U  Areas by perp. drop and tan. skim.
Print findpeaks arguments...Q  Prints findpeaks function with arguments.
Print ipeak arguments.......W  Prints ipeak function with all arguments.
Print report................R  Prints Peak table and parameters
Print peak statistics.......E  prints mean, std of peak intervals,
                                heights, etc.
Peak labels ON/OFF......... L  Label all peaks detected in upper window.
Peak ID ON/OFF..............I  Identifies closest peaks in
                               'Names' database.
Print peak IDs..............O  Prints table of peaks IDs
Switch to ipf.m.............Shift-Ctrl-F  Transfer current signal to
                                           Interactive Peak Fitter
Switch to iSignal...........Shift-Ctrl-S  Transfer current signal
                                         to iSignal.m
Expand to full screen.......Double-click figure window title bar
```

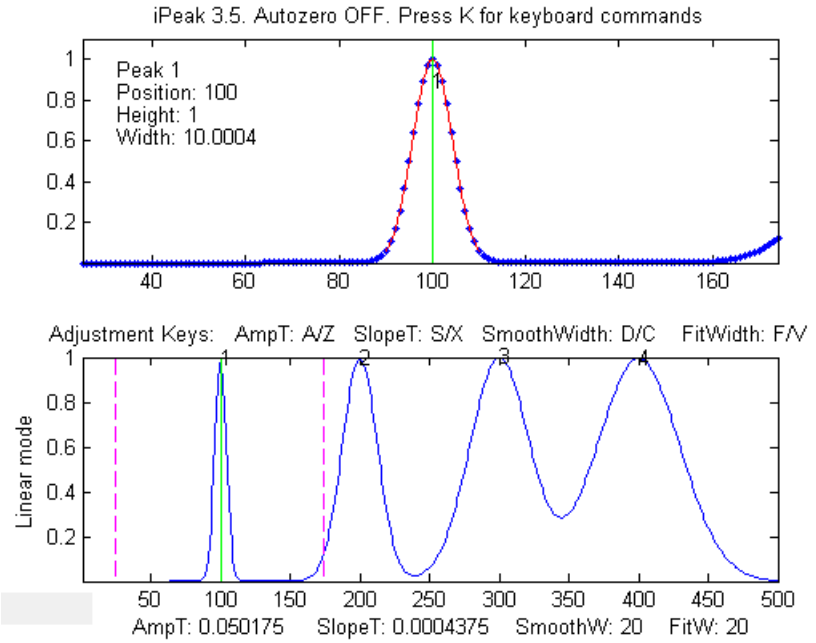
Click for Animated step-by-step instructions

iPeak Demo functions

demoipeak.m (or demoipeakoctave) is a demonstration function that generates a noisy signal with peaks, calls *iPeak*, and then prints out a table of the actual peak parameters and a list of the peaks detected and measured by *iPeak* for comparison. Before running this demo, ipeak.m must be downloaded and placed in the Matlab search path. The ZIP file contains several demo functions (ipeakdemo.m, ipeakdemo1.m, etc.) that illustrate various aspects of the *iPeak* function and how it can be used effectively. Download the zip file, right-click and select "Extract all", then put the resulting files in the Matlab path and run them by typing their names at the Matlab command window prompt. To test for the proper installation and operation of *iPeak*, run "testipeak.m".

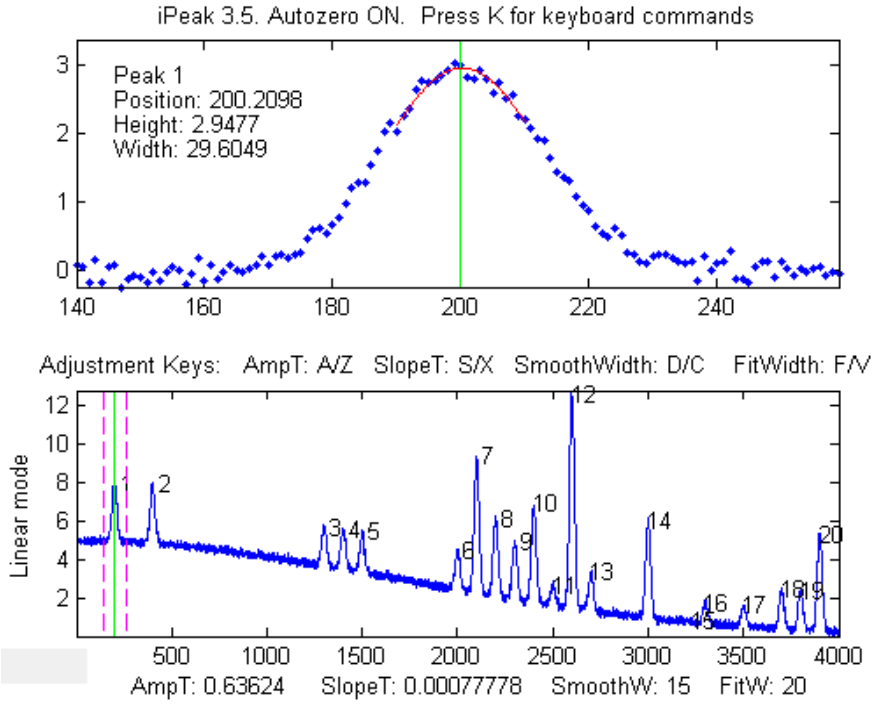
PeakAreaMethods.m is a script that compares the three methods are available in iPeak.m for peak area measurements, using a computer-generated signal of 5 overlapping peaks with unequal heights and widths but equal areas. The peak areas are measured by Gaussian estimation (GE), perpendicular drop (PD), and iterative curve fitting (Fa), which are taken as the correct values. You can control the peak shape and widths in lines 18 and 19.

ipeakdemo: effect of the peak detection parameters



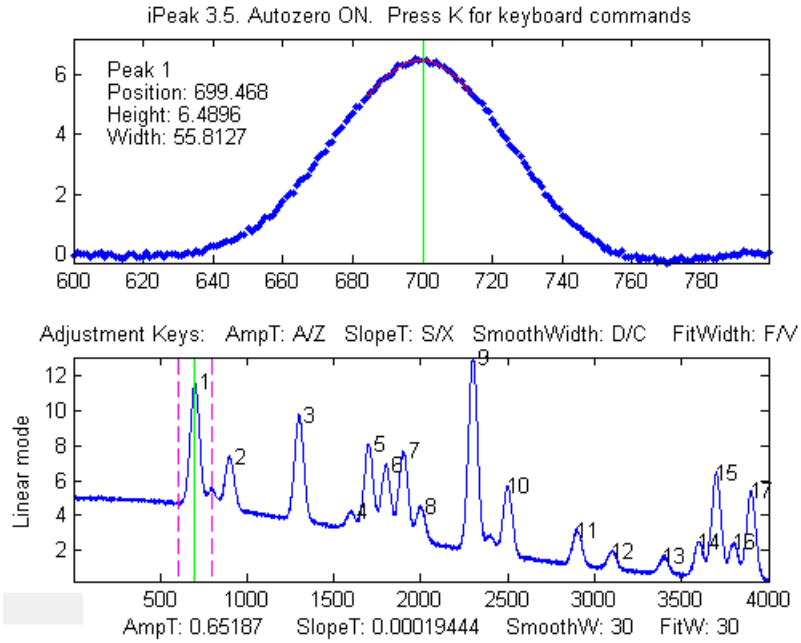
Four Gaussian peaks with the same heights but different widths (10, 30, 50 and 70 units). This demonstrates the effect of SlopeThreshold and SmoothWidth on peak detection. Increasing SlopeThreshold (S key) will discriminate against the broader peaks. Increasing SmoothWidth (D key) will discriminate against the narrower peaks and noise. FitWidth (adjusted by the F and V keys) controls the number of points around the "top part" of the (unsmoothed) peak that are taken to estimate the peak heights, positions, and widths. A reasonable value is ordinarily about equal to 1/2 of the number of data points in the half-width of the peaks. In this case, where the peak widths are quite different, set it to about 1/2 of the number of data points in the narrowest peak.

ipeakdemo1: the baseline correction mode. Demonstration of background correction, with separated, narrow peaks on a large baseline.



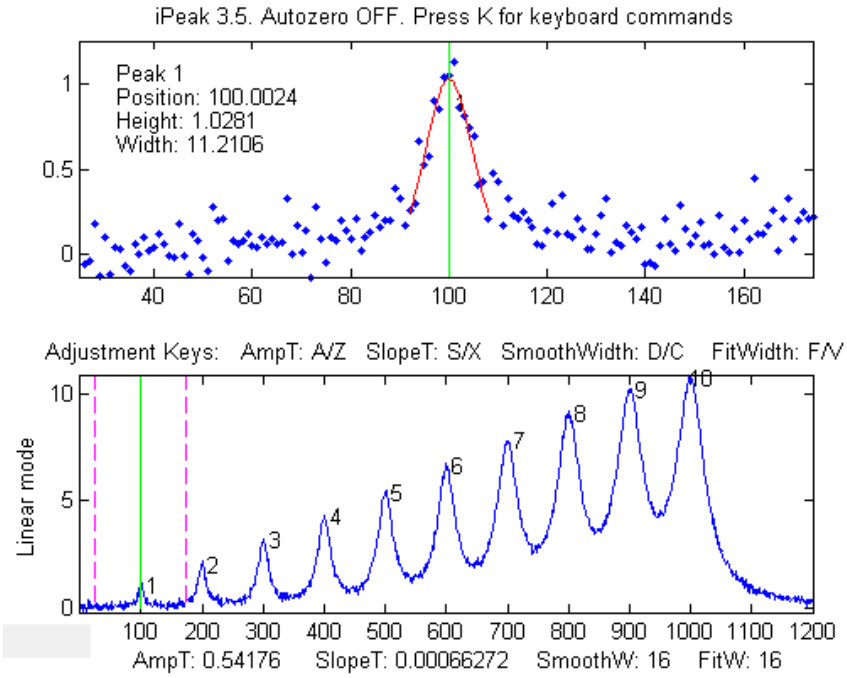
Each time you run this demo, you will get a different set of peaks and noise. A table of the actual peak positions, heights, widths, and areas is printed out in the command window. Jump to the next/ previous peaks using the **Spacebar/Tab** keys. Hint: Select the linear baseline correction mode (**T** key), adjust the zoom setting so that the peaks are shown one at a time in the upper window, then press the **P** key to display the peak table. For peak signals like this, the mode(y) baseline correction modes 4 and 5 are often useful, subtracting the *most common y value* (the statistical *mode)* from the points in the selected region, which usually removes the baseline and does not need the signal to return to the baseline to the edges of the selected region like modes 2 and 3.

ipeakdemo2: peak overlap and the curve fitting functions.



Demonstration of error caused by overlapping peaks on a large offset baseline. Each time you run this demo, you will get a different set of peaks and noise. A table of the actual peak positions, heights, widths, and areas is printed out in the command window. (Jump to the next/ previous peaks using the Spacebar/ Tab keys). Hint: Use the **B** key and click on the baseline points, then press the **P** key to display the peak table. Or use background correction modes 2-4 and use the Normal curve fit (**N** key) with peak shape 1 (Gaussian).

[ipeakdemo3](): **Baseline shift caused by overlapping peaks**

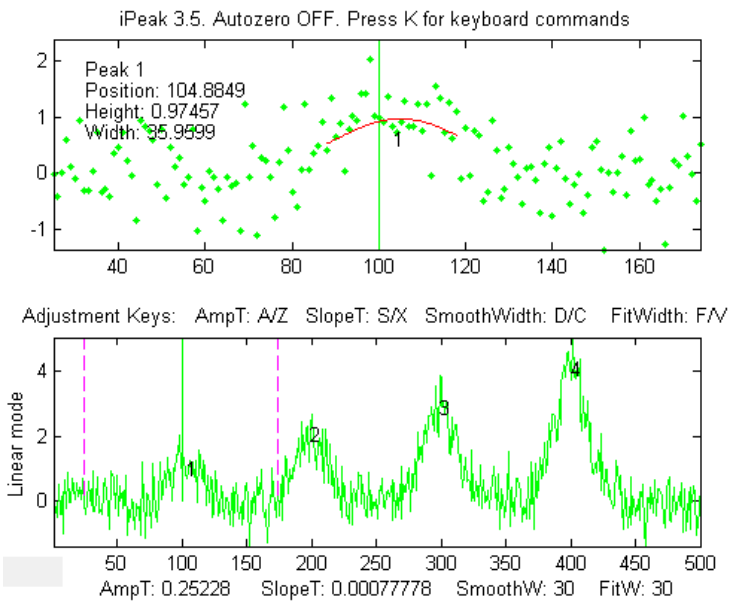


Demonstration of overlapping Lorentzian peaks, *without* an added background. A table of the actual peak positions, heights, widths, and areas is printed out in the command window. In this example, the true peak heights are 1,2 3,...10. Overlap of peaks can cause significant errors in measuring peak parameters, especially for Lorentzian peaks, because they have gently sloping sides that contribute to the baseline of any peaks in the region. In this example, there is no actual background signal. The apparent baseline shift is caused purely by peak overlap.

Hint: turn OFF the background correction mode (**T** key) and use the Normal curve fit (**N** key) to fit small groups of 2-5 peaks numbered in the upper window, with peak shape 2 (Lorentzian). For the greatest accuracy in measuring a particular peak, include one or two additional peaks on either side, to help account for the baseline shift caused by the sides of those neighboring peaks. Alternatively, if the

total number of peaks is not too great, you can use the Multiple curve-fit (**M** key) to fit the entire signal in the lower window.

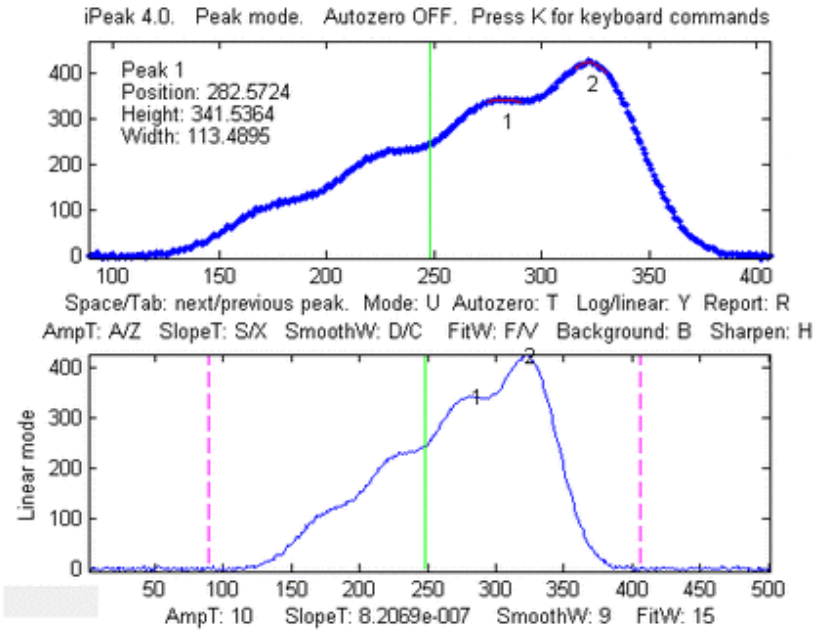
ipeakdemo4: dealing with very noisy signals



Detection and measurement of four peaks in a very noisy signal. The signal-to-noise ratio of the first peak is 2. Each time you run this demo, you will get a different set of noise. A table of the actual peak positions, heights, widths, and areas is printed out in the command window. Jump to the next/previous peaks using the **Spacebar/ Tab** keys. The peak at x=100 is usually detected, but the accuracy of peak parameter measurement is poor because of the low signal-to-noise ratio. **ipeakdemo6** is similar but has the option of different kinds of noise (white, pink, proportional, etc.)

Hint: With very noisy signals it is usually best to increase *SmoothWidth* and *FitWidth* to help reduce the effect of the noise.

ipeakdemo5: dealing with highly overlapped peaks

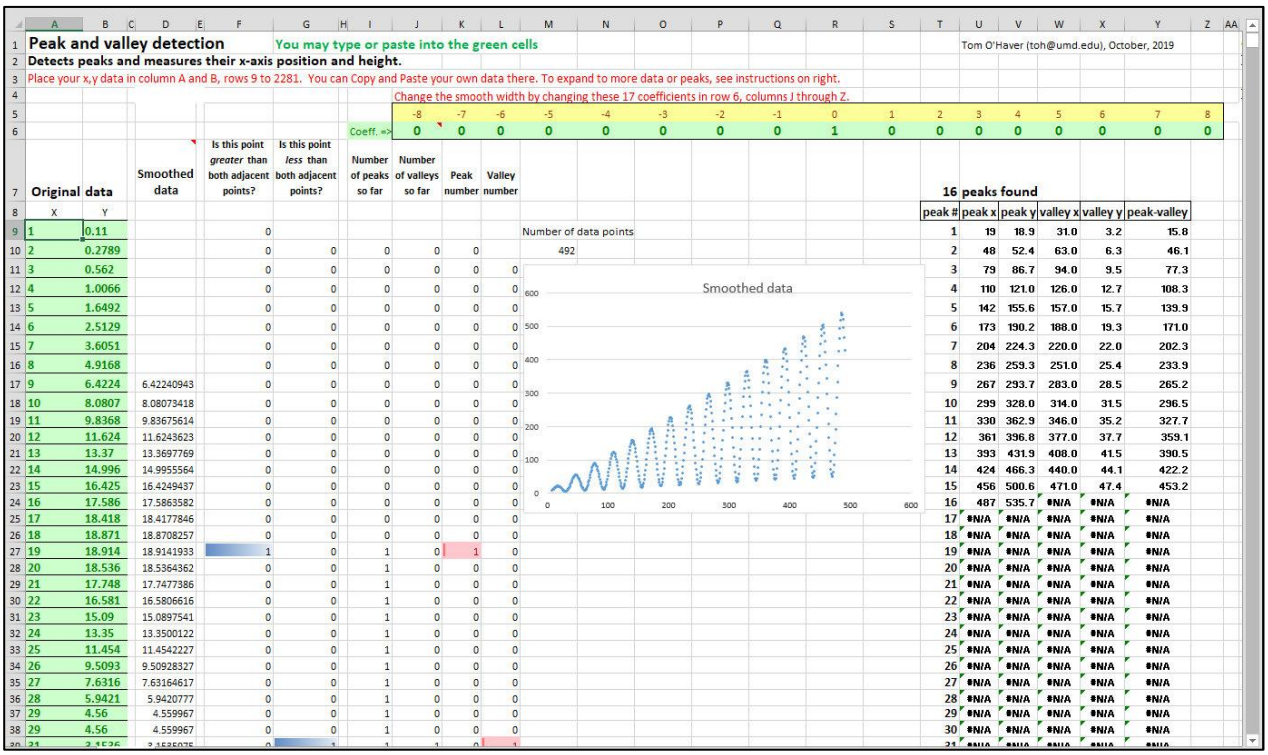


In this demo, the peaks are so highly overlapped that only one or two of the highest peaks yield distinct maxima that are detected by iPeak. The height, width, and area estimates are highly inaccurate because of this overlap. The normal peak fit function ('**N**' key) would be useful in this case, but it depends on iPeak for the number of peaks and for the initial guesses, and so it would fit only the peaks that were found and numbered.

To help in this case, pressing the '**H**' key will activate the peak sharpen function that decreases peak width and increases the peak height of all the peaks, making it easier to detect and number all the peaks for use by the peakfit function (**N** key). Note: peakfit fits the original unmodified peaks. The sharpening is used only to help locate the peaks to provide peakfit with suitable starting values.

If you are using **Python**, a basic peak finding algorithm is described on page 451, and on https://terpconnect.umd.edu/~toh/spectrum/Python.html

Spreadsheet Peak Finder Templates for Excel



Simple peak and valley detection.

The spreadsheet pictured above, PeakAndValleyDetectionTemplate.xlsx (or PeakAndValleyDetectionExample.xlsx with sample data), is a simple peak and valley detector that *defines a peak as any point with lower points on both sides and a valley as any point with higher points on both sides.* Peaks and valleys are marked by colored cells in columns F through L and tabulated in columns T through Y with their x and y measured values, based on the use of the INDIRECT, ADDRESS, and MATCH functions as described on page 353. The raw data can be optionally smoothed by entering a smooth width (a positive odd integer) in cell E6 to suppress false detection caused by random noise. Directions for expanding the template are included.

Selective peak detection. The spreadsheet PeakDetectionTemplate.xls (or PeakDetectionExample.xls with sample data) implements the first-derivative zero-crossing peak detection method with amplitude and slope thresholds as described on page 235. The input x, y data are contained in Sheet1, column **A** and **B**, rows 9 to 1200. (You can type or paste your own data there). The amplitude threshold and slope threshold are set in cells **B4** and **E4**, respectively. Smoothing and differentiation are performed by the convolution technique used by the spreadsheets DerivativeSmoothing.xls described previously on page 75. The Smooth Width and the Fit Width are both controlled by the number of non-zero convolution coefficients in row 6, columns **J** through **Z**. (In order to compute a symmetrical first derivative, the coefficients in columns **J** to **Q** must be the negatives of the positive coefficients in columns **S** to **Z**). The original data and the smoothed derivative

are shown in the two charts. To detect peaks in the data, a series of three conditions are tested for each data point in columns **F**, **G**, and **H**, corresponding to the three nested loops in findpeaksG.m:

1. Is the signal greater than Amplitude Threshold? (line 45 of findpeaksG.m, column **F** in the spreadsheet)
2. Is there a downward directed zero crossing (page 67) in the smoothed first derivative? (line 43 of findpeaksG.m, column **G** in the spreadsheet)
3. Is the slope of the derivative at that point greater than the Slope Threshold? (line 44 of findpeaksG.m, column **H** in the spreadsheet)

If the answer to *all three* questions is *yes* (highlighted by blue cell coloring), a peak is registered at that point (column **I**), counted in column **J**, and assigned an index number in column **K**. The peak index numbers, X-axis positions, and peak heights are listed in columns **AC** to **AF**. Peak heights are computed *two* ways: "Height" is based on slightly smoothed Y values (more accurate if the peaks are broad and noisy, as in PeakDetectionDemo2b.xls) and "Max" is the highest individual Y value near the peak (more accurate if the data are smooth or if the peaks are very narrow, as in PeakDetectionDemo2a.xls). PeakDetectionDemo2.xls/xlsx is a demonstration with a user-controlled computer-generated series of four noisy Gaussian peaks with known peak parameters. PeakDetectionSineExample.xls is a demo that generates a sinusoidal signal with an adjustable number of peaks.

You can extend these spreadsheets to *longer columns of data* by dragging the last row of columns **A** through **K** as needed, then select and edit the data in the graphs to include all the points in the data (Right-click, Select data, Edit). You can extend the spreadsheet to a *greater number of peaks* by dragging the last row of columns **AC** and **AD** as needed. Edit **R7** and the data range in the equations of cells in row 9, columns **U**, **V**, **W**, **X**, **AE**, and **AF** to include all the rows containing data, then copy-drag them down to cover all expected peaks.

Peak detection with least-squares measurement. An extension of that method is made in the spreadsheet template PeakDetectionAndMeasurement.xlsx (screen image), which makes the assumption that the peaks are Gaussian and measures their height, position, and width more precisely using a *least-squares technique*, just like "findpeaksG.m". For the first 10 peaks found, the x,y original unsmoothed data are **copied** to Sheets 2 through 11, respectively, where that segment of data is subjected to a Gaussian least-squares fit, using the same technique as GaussianLeastSquares.xls. The best-fit Gaussian parameter results are copied back to Sheet1, into the table in columns **AH-AK**. (In its present form, the spreadsheet is limited to measuring 10 peaks, although it can detect any number of peaks. Also, it is limited in Smooth Width and Fit Width by the 17-point convolution coefficients).

The spreadsheet is available in OpenOffice (ods) and in Excel (xls and xlsx) formats. They are functionally equivalent and differ only in minor cosmetic aspects. There are two example spreadsheets (A, and B) with calculated noisy waveforms that you can modify. *Note: To enter data into the .xls and .xlsx versions, click the "Enable Editing" button in the yellow bar at the top.*

If the peaks in the data overlap too much, they may not make sufficiently distinct maxima to be detected reliably. If the noise level is low, the peaks can be sharpened artificially by the technique

described previously. This is implemented by PeakDetectionAndMeasurementPS.xlsx and its demo version with example data already entered PeakDetectionAndMeasurementDemoPS.xlsx.

Expanding the PeakDetectionAndMeasurement.xlsx spreadsheet to *larger numbers of measured peaks* is more difficult. You will have to drag down row 17, columns **AC** through **AK**, and adjust the formulas in those rows for the required number of additional peaks, then copy all Sheet11 and paste it into a series of new sheets (Sheet12, Sheet13, etc.), one for each additional peak, and finally adjust the formulas in columns **B** and **C** in each of these additional sheets to refer to the appropriate row in Sheet1. Modify these additional equations in the same pattern as those for peaks 1-10. (In contrast to these spreadsheets, the Matlab/Octave findpeaks functions adapt automatically to *any* length of data and *any* number of peaks).

Spreadsheet vs Matlab/Octave. A comparison of this spreadsheet to its Matlab/Octave equivalent findpeaksplot.m is instructive. On the positive side, the spreadsheet literally "spreads out" the data and the calculations spatially over many cells and sheets, breaking down the discrete steps in a very graphic way. In particular, the use of conditional formatting in columns **F** through **K** makes the peak detection decision process more evident for each peak, and the least-squares sheets 2 through 11 lay out every detail of those calculations. This has significant *instructional* value. Spreadsheet programs have many flexible and easy-to-use formatting options to make displays more attractive. On the downside, a spreadsheet as complicated as PeakDetectionAndMeasurement.xlsx is far more difficult to construct than its Matlab/Octave equivalent. Much more seriously, the spreadsheet is *less flexible* and harder to expand to larger signals and to larger number of peaks. In contrast, the Matlab/Octave equivalent is easy to use, faster in execution, much more flexible, and *can easily handle signals and smooth/fit widths of any size*. Spreadsheets become cumbersome with very large data sets. Moreover, a Matlab/ Octave *function* can readily be employed as an element in your own custom Matlab/Octave programs to perform even larger tasks. It is harder to do that in a spreadsheet.

To compare the computation speed of this spreadsheet peak finder to the Matlab/Octave peak finder, you can take as an example the spreadsheet PeakDetectionExample2.xls, or PeakDetectionExample2.ods, which computes and plots a test signal consisting of a noisy sine-squared waveform with 300 data points and then detects and measures 10 peaks in that waveform and displays a table of peak parameters. This is equivalent to the Matlab/Octave script:

```
x=1:300;
y(1:99)=randn(size(1:99));
y(100:300)=10.*sin(.16.*x(100:300)).^2. + randn(size(x(100:300)));
P=findpeaksplot(x,y,0.005,5,7,7,3);
disp(P)
drawnow
```

The table below compares the elapsed times measured for Matlab 2020 and for Octave 6.1.0 running on a Dell XPS i7 3.5Ghz desktop. The speed advantage of Matlab is clear. Python (page 451) can do as well as Matlab and at the same speed.

Method	Elapsed time
Excel spreadsheet	~ 1 sec
Calc spreadsheet	~ 1 sec
Matlab or Python	0.035 sec
Octave	0.5 sec

This is a rather small test. Many real-world applications have many more data points and many more peaks, in which the speed advantage of Matlab/Python would be more significant. Moreover, those would be the tools of choice if you have many separate data sets to which you need to apply a peak detection/measurement algorithm completely automatically (page 345). See also Excel VS Matlab.

Hyperlinear Quantitative Absorption Spectrophotometry

Specific knowledge of instrumentation design is often useful in designing an effective signal-processing regimen. This example, taken from my own research in the 1980s, shows how a custom signal processing procedure for an optical measurement system can be constructed by combining several of the methods and concepts introduced in this book. The result is a technique that *expands the classical limits of measurement* in optical absorption spectroscopy. This is an alternative computational method for quantitative analysis by multiwavelength absorption spectroscopy, which I have called the transmission-fitting or "TFit" method. It is based on fitting a model of the instrumentally-broadened transmission spectrum to the observed transmission data, and it is an alternative to the usual method of calculating the absorbance by the simple "classical" definition of log10(Io/I). The method is described in T. C. O'Haver and J. Kindervater, *J. of Analytical Atomic Spectroscopy* **1**, 89 (1986); T. C. O'Haver and Jing-Chyi Chang, *Spectrochim. Acta* **44B**, 795-809 (1989); T. C. O'Haver, *Anal. Chem.* **68**, 164-169 (1991). I include this here as an example of combining different signal processing techniques because it combines many of the important concepts that have been covered in this book, namely: signal-to-noise ratio (page 24), Fourier convolution (page 109), multicomponent spectroscopy (page 187), iterative least-squares fitting (page 199), and calibration for quantitative analysis (page 337).

Advantages of the TFit method compared to conventional methods are:

(a) much wider *dynamic range* (i.e., the concentration range over which one calibration curve can be expected to give good results),

(b) greatly improved calibration linearity ("hyperlinearily"), which reduces the labor and cost of preparing and running large numbers of standard solutions and safely disposing of them afterward, and

(c) the ability to operate under conditions that are optimized for signal-to-noise ratio rather than for Beer's Law ideality, that is, using small spectrometers with shorter focal length, lower

dispersion and larger slit widths to *increase light throughput* and reduce the effect of photon and detector noise (assuming, of course, that the detector is not saturated or overloaded).

Just like the classical multilinear regression (classical least-squares) methods that are conventionally used in absorption spectroscopy, the Tfit method

(a) requires an accurate reference spectrum of each analyte,

(b) utilizes multiwavelength data such as would be acquired on diode-array, Fourier transform, or automated scanning spectrometers,

(c) applies both to single-component and multi-component mixture analysis, and

(d) requires that the absorbances of the components *vary with wavelength*, and, for multi-component analysis, that the absorption spectra of the components be sufficiently different. Black or grey absorbers do not work with this method.

The disadvantages of the Tfit method are:

(a) It makes the *computer* work harder than the multilinear regression methods (but, on a typical personal computer, calculations take only a fraction of a second, even for the analysis of a mixture of several components).

(b) It requires knowledge of the instrument function, i.e., the slit function or the resolution function of an optical spectrometer. (However, this is a *fixed characteristic* of the instrument and can be measured beforehand by scanning the spectrum of a narrow atomic line source such as a hollow cathode lamp). It changes only if you change the slit width of the spectrometer.

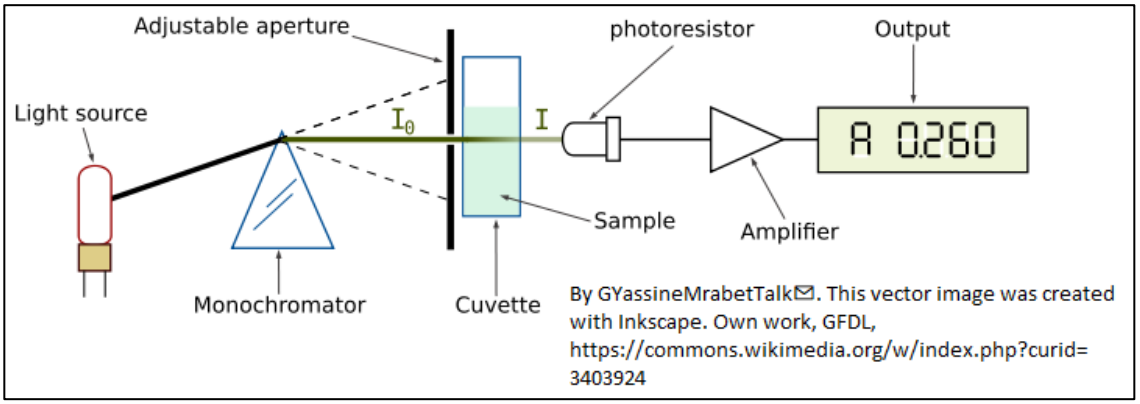
(c) It is an iterative method that can under unfavorable circumstances converge on an incorrect local optimum, but this is handled by proper selection of the starting value, which in this case can be calculated by the traditional log (Io/I) method, and

(d) It will not work for gray-colored substances whose absorption spectra do not vary over the spectral region measured.

You can perform the required calculations in a spreadsheet or in Matlab or Octave, using the software described below.

The following sections give the background of the method and a description of the main function and demonstration programs and spreadsheet templates:

Background



The figure above illustrates optical absorption spectroscopy, where the intensity "I" of monochromatic light passing through an absorbing sample is given (in Matlab notation) by the Beer-Lambert Law:

```
I = Io.*10^-(alpha*L*c)
```

where "I_O" (pronounced "eye-zero") is the intensity of the light incident on the sample, "alpha" is the absorption coefficient of the absorber, "L" is the distance that the light travels through the material (the optical path length), and "c" is the concentration of absorber in the sample. The variables I, Io, and alpha are all functions of wavelength. L and c are scalar.

Traditionally, measured values of I and Io are used to compute a quantity called "absorbance", defined as

$$A = \log10(Io/I)$$

Absorbance is defined in this way so that, when you combine this definition with the Beer-Lambert law, you get:

$$A = alpha*L*c$$

In this way, absorbance is proportional to concentration, ideally, which simplifies analytical calibration. This works for monochromatic light beams. However, any real spectrometer has a *finite spectral resolution*, meaning that the light beam passing through the sample is not truly monochromatic, with the result that an intensity reading at one wavelength setting is an average over a small spectral interval, which is determined by the "adjustable aperture" in the above diagram, also called the "slit width". More exactly, what is measured is a convolution of the true spectrum of the absorber and the *instrument function*, the instrument's response as a function of the wavelength of the light. Ideally, the instrument function is infinitely narrow (a "delta" function), but practical spectrometers have a *non-zero* slit width, the width of the adjustable aperture in the diagram above, which passes a small spectral interval of wavelengths of light from the dispersing element (prism) onto the sample and detector. If the absorption coefficient "alpha" varies over that spectral interval, then the traditionally calculated absorbance will no longer be linearly proportional to the concentration (this is called the "polychromicity" error). The effect is most noticeable at high absorbances. In practice, many instruments will become non-linear starting at an absorbance of 2 (~1% Transmission). As the

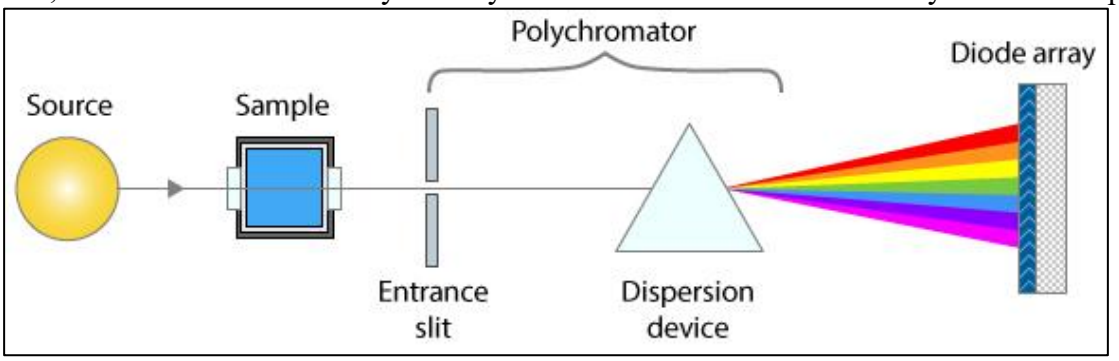
absorbance increases, the effect of unabsorbed stray light and instrument noise becomes more significant.

The theoretical best signal-to-noise ratio and absorbance precision for a photon-noise limited optical absorption instrument can be shown to be close to an absorbance close to 1.0 (see "Is there an optimum **absorbance** for best signal-to-noise ratio?"). The range of absorbances *below* 1.0 is easily accessible down to at least 0.001, but the range *above* 1.0 is limited. Even an absorbance of 10 is unreachable on most instruments and the direct measurement of an absorbance of 100 is unthinkable, as it implies the measurement of light attenuation of 100 powers of ten - no real measurement system has a dynamic range even close to that. In practice, it is difficult to achieve a dynamic range even as high as 5 or 6 absorbance units, so that most of the theoretically optimum absorbance range *above* 1.0 is unusable. (c.f. http://en.wikipedia.org/wiki/Absorbance). So, in conventional practice, greater sample dilutions and shorter optical path lengths are required to force the absorbance range to lower values, *even if this means poorer signal-to-noise ratio and measurement precision at the low end*.

It is true that the non-linearity caused by polychromicity can be reduced by operating the instrument at the highest resolution setting (reducing the instrumental slit width). However, this has a serious undesired side effect: in dispersive instruments, reducing the slit width to increase the spectral resolution degrades the signal-to-noise substantially. It also reduces the number of atoms or molecules that are measured. Here is why: UV/visible absorption spectroscopy is based on the absorption of photons of light by molecules or atoms resulting from transitions between electronic energy states. It is well known that the absorption peaks of molecules are wide *bands*, not monochromatic *lines*, because the molecules are undergoing vibrational and rotational transitions as well as electronic ones and are also under the perturbing influence of their environment. This is the case also in atomic absorption spectroscopy: the absorption "lines" of gas-phase free atoms, although much narrower that molecular bands, nevertheless have a finite non-zero width, mainly due to their velocity (*temperature* or *Doppler* broadening) and due to collisions with the matrix gas (*pressure* broadening). A macroscopic collection of molecules or atoms, therefore, presents to the incident light beam a *distribution* of energy states and absorption wavelengths. Absorption results from the collective interaction of many *individual* atoms or molecules with *individual* photons. A purely *monochromatic* incident light beam would have photons all the *same* energy, ideally corresponding to the maximum of the energy distribution of the collection of atoms or molecules being measured. But in fact, many of the atoms or molecules in the optical light path would have an energy slightly *greater or less than* the average and *would thus not be measured*. If the bandwidth of the incident beam is increased, more of those non-average atoms or molecules would be available to be measured and more transmitted photons would reach the detector, but then the simple calculation of absorbance as log10(Io/I) no longer results in a linear response to concentration.

Numerical simulations show that the optimum signal-to-noise ratio is typically achieved when *the spectral resolution of the instrument matches the width of the analyte absorption*, but under those conditions using the conventional log10(Io/I) absorbance would result in very substantial non-linearity over the higher range of absorbance values because of the "polychromicity" error. This non-linearity has its origin in the *spectral domain* (intensity vs wavelength), not in the *calibration domain* (absorbance vs concentration). Therefore, it should be no surprise that curve fitting in the calibration domain, for example fitting the calibration data with a quadratic or cubic fit, would not be the best

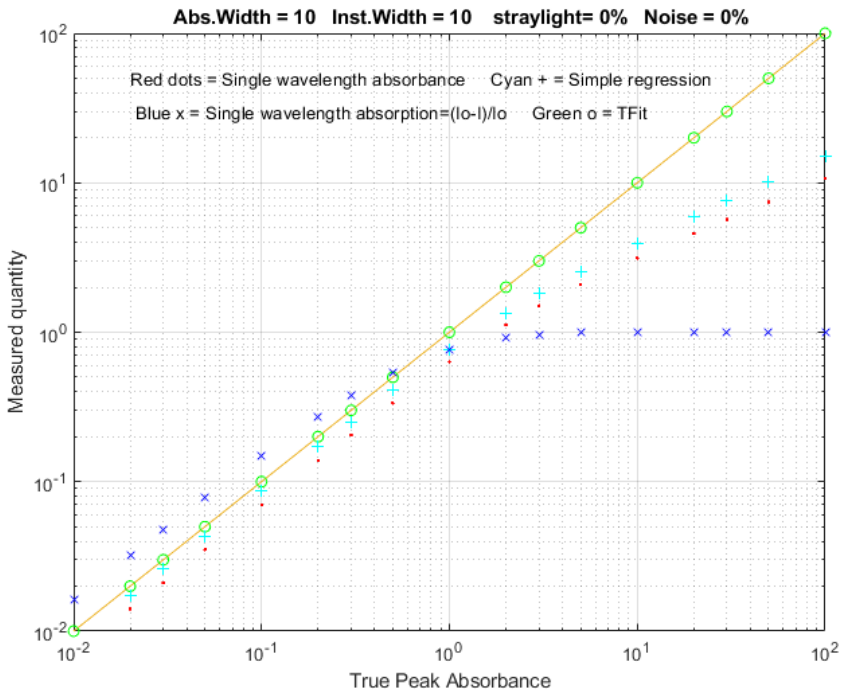
solution, because there is no theory that says that the deviations from linearity would be expected to be



exactly quadratic or cubic. A more rigorous theory-based approach would be to perform the curve fitting in the *spectral domain,* where the source of the non-linearity arises. This is possible with modern absorption spectrophotometers that use *array detectors,* which have many tiny detector elements that slice up the spectrum of the transmitted beam into many small wavelength segments, rather than detecting the sum of all those segments with one big photo-tube detector, as older instruments do. An instrument with an array detector typically uses a slightly different optical arrangement, as shown by the simplified diagram above: the light travels *first* through the sample and *then* to the polychromator and array detector. The spectral resolution is determined by both the entrance slit width and by the width of optical detector elements (or by the number of those elements that are added up to determine the transmitted intensity).

The TFit method sidesteps the above problems by calculating the absorbance in a completely different way. It starts with the reference spectra (an accurate absorption spectrum for each analyte, also required by the multilinear regression methods). It normalizes the reference spectra to unit height, multiplies each by an adjustable coefficient – usually starting with the conventional log10(Io/I) absorbance as a first guess - adds them up, computes the antilog, and convolutes it with the previously-measured slit function. The result, representing the instrumentally broadened transmission spectrum, is compared to the observed transmission spectrum. The program adjusts the coefficients (one for each unknown component in the mixture) until the computed transmission model is a least-squares best fit to the observed transmission spectrum. The best-fit coefficients are then equal to the absorbances that would have been determined under ideal optical conditions. The program also compensates for unabsorbed stray light and changes in background intensity (background absorption). These calculations are performed by the function fitM, which is used as a fitting function for Matlab's iterative non-linear fitting function fminsearch. It sounds complicated but, in fact, takes only a fraction of a second to compute. The Fit method gives measurements of absorbance that are much closer to the "true" peak absorbance that would have been measured in the absence of stray light and polychromatic light errors. More importantly, it allows linear and wide dynamic range measurements to be made even if the slit width of the instrument is increased to optimize the signal-to-noise ratio. From a historical perspective, by the time Pierre Bouguer formulated what became to be known as the *Beer-Lambert law* in 1729, the *logarithm* was already well known, having been introduced by John Napier in 1614. The additional mathematical work needed to compute the *absorbance*, log(Io/I), rather than the simpler relative *absorption*, (Io-I)/Io, was justified because of the better linearity of absorbance with respect to concentration and optical path length. Moreover, the log calculation could easily be performed simply

even in those days by a slide-rule type graduated scale. Certainly, by today's standards, the calculation of a logarithm is considered completely routine. In contrast, the TFit method presented here is admittedly more mathematically complex than computing a logarithm and cannot be done without the aid of software (at least a spreadsheet) and at least some low-end miniaturized computational hardware. However, it offers a further improvement in linearity beyond that achieved by the logarithmic calculation of absorbance, and it additionally *allows the small slit width limitation to be loosened*. The figure on the right above compares the dynamic range of simple relative absorption (blue x), single-wavelength logarithmic absorbance (red dots), multilinear regression or CLS method (cyan +) based on absorbance, and the TFit method (green o) over a 10^4 range of absorbances. (This plot was created by the Matlab/Octave script TFitCalCurveAbs.m).



Bottom line: It's important to understand that *the TFit method does not reject the Beer-Lambert Law*. In fact, it is *based* on that law. The TFit method simply *calculates the absorbance in a different way* that does not require the assumption that stray light and polychromatic radiation effects are zero. It allows larger slit widths and shorter focal lengths to be used, resulting in greater light intensity and signal-to-noise ratios while still achieving a much wider linear dynamic range than previous methods, thus requiring fewer standards to properly define the calibration curve and avoiding the need for non-linear calibration models. Keep in mind that the log(Io/I) absorbance calculation is a 165-year-old simplification that was driven by the need for *mathematical convenience* (and limited also by the mathematical skills of the college students to whom this subject is typically first presented), *not* by the desire to optimize linearity and signal-to-noise ratio. It dates from the time before electronics and computers, when the only computational tools were pen and paper and slide rules, and when a method such as described here would have been completely unimaginable. That was then; this is now. *Tfit is the 21st century way to do quantitative absorption spectrophotometry.*

Note 1: The TFit method compensates for the non-linearity caused by unabsorbed stray light and the polychromatic light effect, but other potential sources of non-linearity remain, specifically, *chemical effects*, such as photolysis, equilibrium shifts, temperature and pH effects, binding, dimerization, polymerization, molecular phototropism, fluorescence, etc. A well-designed quantitative analytical method is designed to minimize those effects.

Note 2: In practical applications, the information required by the Tfit method may not be known

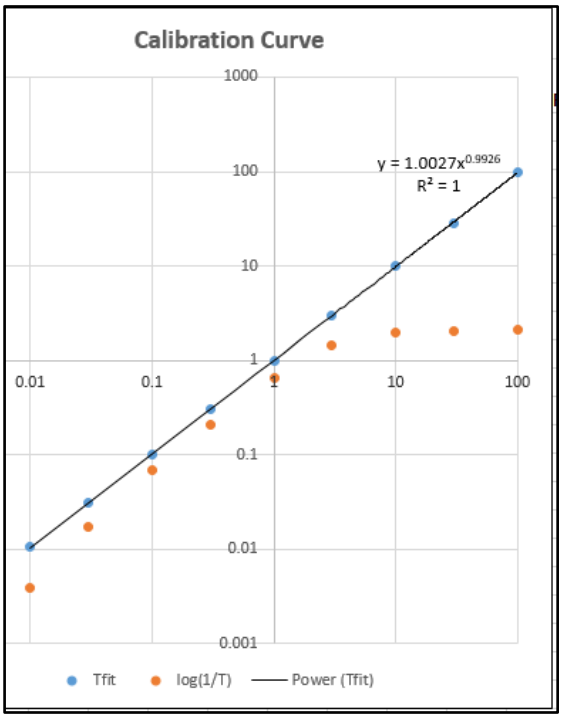
exactly, but a reasonable estimate is better than nothing. For example, even an imperfect estimate of the instrumental bandwidth and/or stray light is better than simply assuming that they are *zero*.

Spreadsheet implementation of the Tfit method

The Tfit method can also be implemented in an *Excel* or *Calc* spreadsheet. It is a bit more cumbersome that the Matlab/Octave implementation, but it works. The shift-and-multiply method is used for the convolution of the reference spectrum with the slit function, and the "Solver" add-in for Excel and Calc is used for the iterative fitting of the model to the observed transmission spectrum. It is very handy, but not essential, to have a "macro" capability to automate the process. (See http://peltiertech.com/Excel/SolverVBA.html#Solver2 for more info about setting up macros and solver on your version of Excel).

TransmissionFittingTemplate.xls (screen image) is an empty template. All you have to do is to enter the data in the cells marked by a gray background: wavelength (Column **A**), observed absorbance of the sample (Column **B**), the high-resolution reference absorbance spectrum (Column **D**), the stray light (**A6**) and the slit function used for the observed absorbance of the sample (**M6-AC6**). (TransmissionFittingTemplateExample.xls (screen image) is the same template with example data entered.

TransmissionFittingDemoGaussian.xls (screen image) is a demonstration with a simulated Gaussian absorption peak with a variable peak position, width, and height, plus added stray light, photon noise, and detector noise, as viewed by a spectrometer with a triangular slit function. You can vary all the parameters and compare the best-fit absorbance to the true peak height and to the conventional log(1/T) absorbance.



All of these spreadsheets include a macro, activated by pressing **Ctrl-f,** that uses the Solver function to perform the iterative least-squares calculation (see page 315). However, if you prefer not to use macros, you can do it manually by clicking the **Data** tab, **Solver**, **Solve**, and then **OK**.

TransmissionFittingCalibrationCurve.xls (screen image) is a demonstration spreadsheet that includes another Excel macro that constructs calibration curves comparing the TFit and conventional log(1/T) methods for a series of 9 standard concentrations that you can specify. To create a calibration curve, enter the standard concentrations in AF10 - AF18 (or just use the ones already there, which cover a 10,000-fold concentration range from 0.01 to 100), then press **Ctrl-f** to run the macro. In this spreadsheet the macro does a lot more than in the previous example: it automatically goes through the first row of the little table in AF10 - AH18, extracts each concentration value in turn, places it in the concentration cell A6, recalculates the spreadsheet, takes the resulting conventional absorbance (cell J6) and places it as the first guess in cell I6, brings up the Solver to compute the best-fit absorbance for that peak height, places both the conventional absorbance and the best-fit

absorbance in the table in AF10 - AH18, then goes to the next concentration and repeats for each concentration value. Then it constructs and plots the log-log calibration curve (shown on the right) for both the TFit method (blue dots) and the conventional (red dots) and computes the trend-line equation and the R^2 value for the TFit method, in the upper right corner of the graph. Each time you press **Ctrl-f** it repeats the whole calibration curve with another set of random noise samples. (Note: you can also use this spreadsheet to compare the precision and reproducibility of the two methods by entering the *same* concentration 9 times in AF10 - AF18. The result should ideally be a straight flat line with zero slope).

Matlab/Octave implementation: The fitM.m function

```
function err = fitM(lam, yobsd, Spectra, InstFun, StrayLight)
```

fitM is a fitting function for the Tfit method, for use with Matlab's or Octave's fminsearch function. The input arguments of fitM are:

absorbance= vector of absorbances that are calculated to give the best fit to the transmission spectrum.
yobsd = observed transmission spectrum of the mixture sample over the spectral range (column vector)
Spectra = reference spectra for each component, over the same spectral range, one column/component, normalized to 1.00.
InstFun = Zero-centered instrument function or slit function (column vector)
StrayLight = fractional stray light (scalar or column vector, if it varies with wavelength)

Note: **yobsd**, **Spectra**, and **InstFun** must have the same number of rows (wavelengths). **Spectra** must have one column for each absorbing component.

Typical use:

```
absorbance = fminsearch(@(lambda)(fitM(lambda, yobsd, TrueSpectrum,
InstFun, StrayLight)), start);
```

where "`start`" is the first guess (or guesses) of the absorbance(s) of the analyte(s). It is convenient to use the conventional log10(Io/I) estimate of absorbance for `start`. The other arguments (described above) are passed on to FitM. In this example, the fminsearch function returns the value of absorbance that would have been measured in the absence of stray light and polychromatic light errors (which is either a single value or a vector of absorbances, if it is a multi-component analysis). The absorbance can then be converted into concentration by any of the usual calibration procedures (Beer's Law, external standards, standard addition, etc.).

Here is a very simple numerical example, where the true absorbance is 1.00, using only *4-point spectra* for illustrative simplicity (Naturally, array-detector systems would acquire *many* more wavelengths than that, but the principle is the same). In this case the instrument width ("InstFun") is *twice* the absorption width, the stray light is constant at 0.01 (1% relative). The conventional single-wavelength estimate of absorbance is far too low:

`log10(1/.38696)=0.4123`. In contrast, the TFit method using fitM is set up like this:

```
yobsd=[0.56529 0.38696 0.56529 0.73496]';
TrueSpectrum=[0.2 1 0.2 0.058824]';
InstFun=[1 0.5 0.0625 0.5]';
straylight=.01;
start=.4;
absorbance=fminsearch(@(lambda)(fitM(lambda,yobsd,TrueSpectrum,InstFun,
straylight)),start)
```

This returns the correct absorbance value of 1.000. The "start" value is not critical in this case and can be just about any value you like, but I like to use the conventional log10(Io/I) absorbance, which is easily calculated and is useful as a *rough but reasonable estimate* of the correct value.

For a ***multiple-component*** **measurement**, the only difference is that the variable "`TrueSpectrum`" would be a *matrix* rather than a *vector*, with one column for each absorbing component. The resulting "`absorbance`" would then be a *vector* rather than a *single number*, with one absorbance value for each component. (See TFit3.m below for an example of a 3-component mixture).

Iterative least-squares methods are ordinarily considered to be more difficult and less reliable than the more common non-iterative multilinear regression methods. This can be true if there is more than one nonlinear variable that must be iterated, especially if those variables are correlated. However, in the TFit method, there is only *one* iterated variable (absorbance) per measured component, and reasonable first guesses are readily available from the conventional single-wavelength absorbance calculation or standard multiwavelength regression methods. As a result, the iterative least-squares method works very well in this case. The expression for absorbance given above for the TFit method can be compared to that for the *weighted regression method*:

```
absorbance=([weight weight].*[Background RefSpec])\(-log10(yobsd).*weight)
```

where RefSpec is the matrix of reference spectra of all the pure components. You can see that, in addition to the RefSpec and observed transmission spectrum (yobsd), the TFit method also requires a measurement of the Instrument function (spectral bandpass) and the stray light (which the linear regression methods assume to be zero), but these are characteristics of the *spectrometer* and need be done only once for a given spectrometer. Finally, the computation time on a typical laboratory personal computer is only a fraction of a second (roughly 25 µsec per spectral data point per component analyzed), using Matlab as the computational environment. The cost of the computational hardware need not be burdensome. The method can be performed in Python, or in *Octave* (with some loss in speed), or even on a $35 single-board computer (see page 344) which comes with Python installed.

Demo function for Octave or Matlab

The tfit.m function is a self-contained command-line Matlab demonstration function that compares the Tfit method to the single-wavelength (SingleW), simple regression (SimpleR), and weighted regression (WeightR) methods. The syntax is **tfit(absorbance)**, where 'absorbance' is the *true underlying peak absorbance* (True A) of an absorber with a Lorentzian spectral profile of width 'width' (line 29), measured with a spectrometer with a Gaussian spectral bandpass of width 'InstWidth' (line 30), fractional unabsorbed stray light level of 'straylight' (line 32), photon noise level of 'noise' (line 31) and a random Io shift of 'IzeroShift' (line 33). Plots the spectral profiles and prints the measured

absorbances of each method in the command window. Examples:

```
>> tfit(1)

width = 10
InstWidth = 20
noise = 0.01
straylight = 0.01
IzeroShift = 0.01

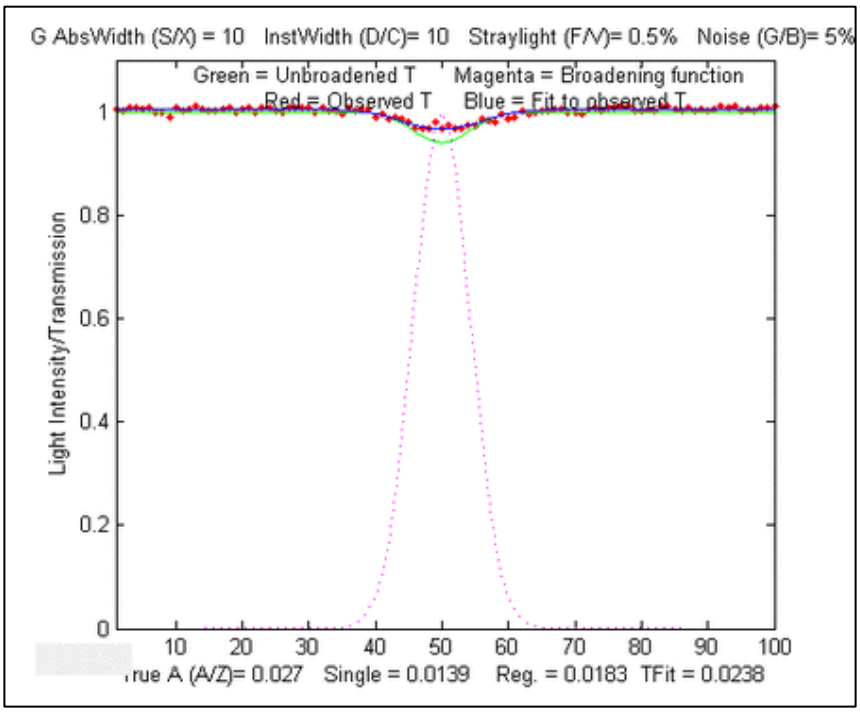
          True A     SingleW     SimpleR      WeightR     TFit
            1       0.38292     0.54536      0.86839     1.0002
>> tfit(10)
            10      1.4858      2.2244       9.5123      9.9979

>> tfit(100)
           100      2.0011      3.6962       57.123      99.951

>> tfit(200)
           200      2.0049      3.7836       56.137     200.01

>> tfit(.001)
          0.001    0.00327      0.00633     0.000520     0.000976
```

TFitDemo.m: Interactive demo for the Tfit method



TFitDemo.m is a keypress-operated interactive explorer for the Tfit method (for Matlab or Octave), applied to the measurement of a single component with a Lorentzian (or Gaussian) absorption peak, with controls that allow you to adjust the true absorbance (Peak A), spectral width of the absorption peak (AbsWidth), spectral width of the instrument function (InstWidth), stray light, and the photon noise level (Noise) continuously while observing the effects graphically and numerically. (If the animation is not visible, click this link). The demo simulates the effect of photon noise, unabsorbed

stray light, and random background intensity shifts (light source flicker). Compares observed absorbances by the single-wavelength, weighted multilinear regression (sometimes called Classical Least-squares in the chemometrics literature), and the TFit methods. To run this file, right-click TFitDemo.m click "Save link as...", save it in a folder in the Matlab search path, then type "TFitDemo" at the Matlab command prompt. With the figure window topmost on the screen, press **K** to get a list of the keypress functions. Version 2.1, November 2011, adds signal-to-noise ratio calculation and uses the **W** key to switch between Transmission and Absorbance display.

In the example shown above, the true peak absorbance is shown varying from 0.0027 to 57 absorbance units, the absorption widths and instrument function widths are equal (which results in the optimum signal-to-noise ratio), the unabsorbed stray light is 0.5%, and the photon noise is 5%. (For demonstration purposes, the lowest 6 absorption peak shapes are *Gaussian* and the highest 3 are *Lorentzian*). The results displayed below the graphs show that, at every absorbance and for either a Gaussian or a Lorentzian peak shape, the TFit method gives much more accurate measurements than either the single-wavelength method or weighted multilinear regression method.

```
KEYBOARD COMMANDS
Peak shape....Q      Toggles between Gaussian and Lorentzian
                     absorption peak shape
True peak A...A/Z    True absorbance of analyte at peak center, without
                     instrumental broadening, stray light, or noise.
AbsWidth......S/X    Width of the absorption peak
SlitWidth.....D/C    Width of instrument function (spectral bandpass)
Straylight....F/V    Fractional unabsorbed stray light.
Noise.........G/B    Random noise level
Re-measure....Spacebar   Re-measure signal with another random noise sample
Switch mode...W      Switch between Transmission and Absorbance display
Statistics....Tab    Prints table of statistics of 50 repeats
Cal. Curve....M      Displays analytical calibration curve in Fig. window 2
Keys..........K      Print this list of keyboard commands
```

Why does the noise on the graph change if I change the instrument function (slit width or InstWidth)? In an absorption spectrometer using a continuum light source and a dispersive spectrometer, there are two adjustable apertures or slits, one before the dispersing element, which controls the physical width of the light beam, and one after, which controls the wavelength range of the light measured (and which, in an array detector, is controlled by the software reading the array elements). A spectrometer's spectral bandwidth ("*InstWidth*") is changed by changing both of those, which also affects the light intensity measured by the detector and thus the signal-to-noise ratio. Therefore, in all these programs, when you change *InstWidth*, the photon noise is automatically changed accordingly just as it would in a real spectrophotometer. The detector noise, in contrast, remains the same. I am also assuming that the detector does not become saturated or overloaded if the slit width is increased.

Statistics of methods compared (TFitStats.m, for Matlab or Octave)

This is a script that computes the statistics of the TFit method compared to single- wavelength (SingleW), simple regression (SimpleR), and weighted regression (WeightR) methods. It simulates photon noise, unabsorbed stray light, and random background intensity shifts, estimates the precision and accuracy of the four methods by repeating the calculations 50 times with different random noise samples, and computes the mean, relative percent standard deviation, and relative percent deviation from true absorbance. You can easily change the parameters in lines 19 - 26. The program displays its results in the MATLAB command window.

In the sample output shown below, the program has computed results for true absorbances of 0.001 and 100, demonstrating that the accuracy and the precision of the TFit method are superior to the other methods over a 10,000-fold range.

This statistics function is included as a keypress command (**Tab** key) in TFitDemo.m.

Results for true absorbances of 0.001

```
   True A    SingleW   SimpleR   WeightR   TFit
MeanResult =
    0.0010    0.0004    0.0005    0.0006    0.0010

PercentRelativeStandardDeviation =
  1.0e+003 *
    0.0000    1.0318    1.4230    0.0152    0.0140

PercentAccuracy =
    0.0000  -60.1090  -45.1035  -38.6300    0.4898
```

Results for true absorbances of 100

```
MeanResult =
  100.0000    2.0038    3.7013   57.1530   99.9967

PercentRelativeStandardDeviation =
         0    0.2252    0.2318    0.0784    0.0682

PercentAccuracy =
         0  -97.9962  -96.2987  -42.8470   -0.0033
```

As you can see, the Tfit method offers improved accuracy *and* precision.

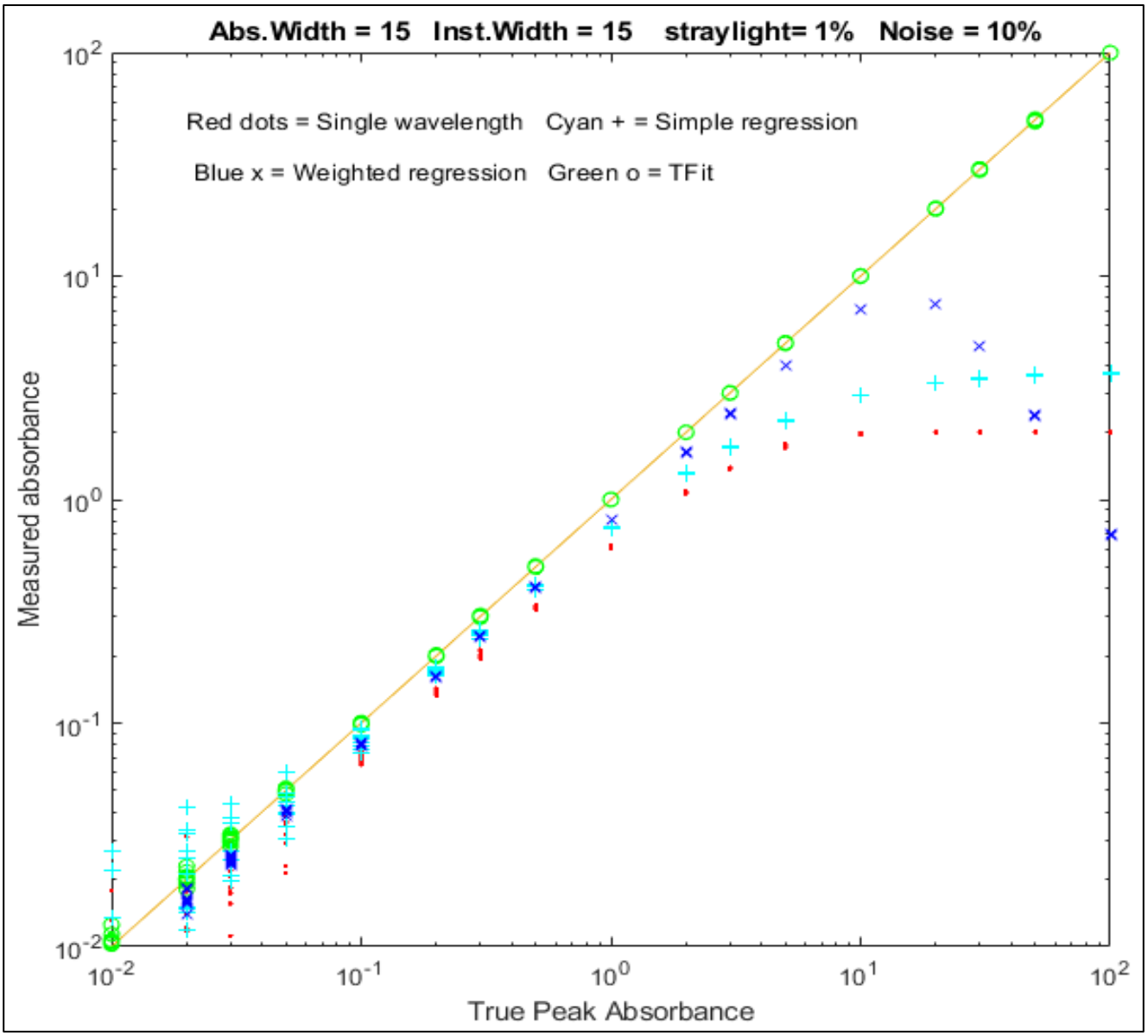
Comparison of analytical curves (TFitCalDemo.m, for Matlab or Octave)

TFitDemo.m is a demonstration function that compares the analytical curves for single-wavelength, simple regression, weighted regression, and the TFit method over any specified absorbance range (specified by the vector "absorbancelist" in line 20). Simulates photon noise, unabsorbed stray light, and random background intensity shifts. Plots a log-log scatter plot with each repeat measurement plotted as a separate point (so you can see the scatter of points at low absorbances). The parameters can be changed in lines 20 - 27.

In the sample result shown below, analytical curves for the four methods are computed over a 10,000-fold range, up to a peak absorbance of 100, demonstrating that the TFit method (shown by the green

circles) is much more nearly linear over the whole range than the single-wavelength, simple regression, or weighted regression methods. *The wide linearity range of Tfit is especially important in regulated laboratories where* anything but linear least-squares fits to the calibration curve are discouraged.

This calibration curve function is included as a keypress command (**M** key) in TFitDemo.m.



Comparison of the simulated analytical curves for single-wavelength, simple regression, weighted regression, and TFit methods over a 10,000-fold absorbance range, created by TFitCalDemo.m.

Application to a three-component mixture

The application of absorption spectroscopy to *mixtures* of absorbing components requires the adoption of one additional assumption: that of additivity of absorbances, meaning that the measured absorbance of a mixture is equal to the sum of the absorbances of the separate components. In practice, this requires that the absorbers do not interact chemically and that they do not react with themselves or with the other components or modify any property of the solution (e.g., pH, ionic strength, density, etc.) that might affect the spectra of the other components. These requirements apply equally to the conventional

multi-component methods (page 188) as well as to the T-Fit method.

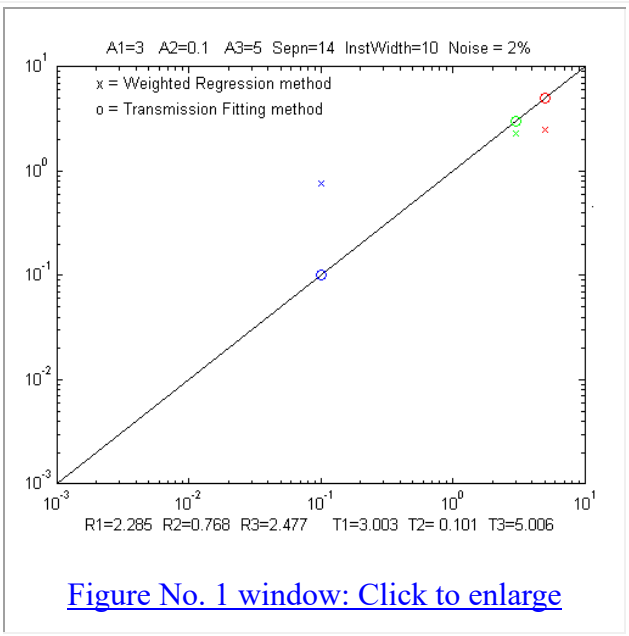


Figure No. 1 window: Click to enlarge

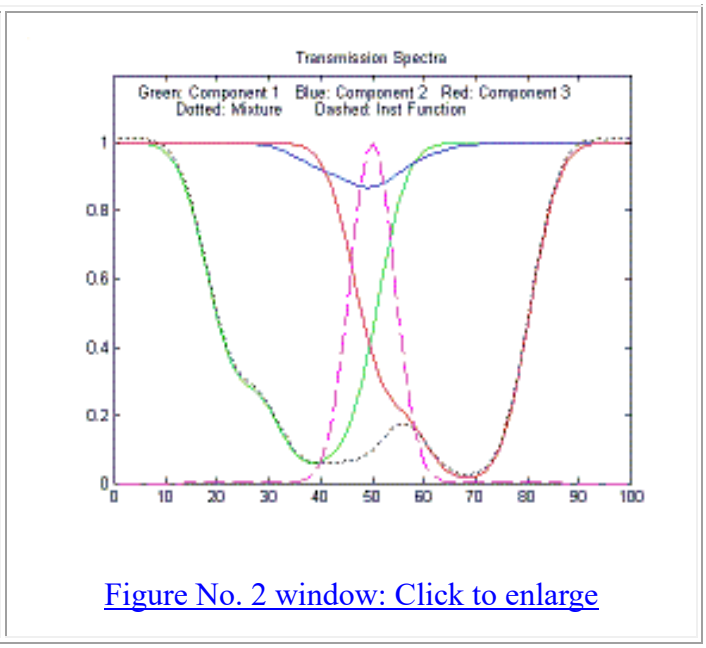


Figure No. 2 window: Click to enlarge

TFit3.m is a Matlab/Octave demonstration of the T-Fit method applied to the multi-component absorption spectroscopy of a mixture of three absorbers. The adjustable parameters are the absorbances of the three components (A1, A2, and A3), the spectral overlap between the component spectra ("Sepn"), the width of the instrument function ("InstWidth"), and the noise level ("Noise"). TFit3 compares quantitative measurement by weighted regression and TFit methods and simulates photon noise, unabsorbed stray light, and random background intensity shifts. Note: After executing this m-file, slide the "Figure No. 1" and "Figure No.2" windows side-by-side so that they do not overlap. Figure window 1 shows a log-log scatter plot of the true vs. measured absorbances, with the three absorbers plotted in different colors and symbols. Figure window 2 shows the transmission spectra of the three absorbers plotted in the corresponding colors. As you use the keyboard commands (below) in Figure No. 1, both graphs change accordingly.

In the sample calculation shown above, component 2 (the small dip shown in blue) *is almost completely buried* by the stronger absorption bands of components 1 and 3 on either side and has a much weaker peak absorbance (0.1) than the other two components (3 and 5, respectively). Even in this challenging case, the TFit method gives a result (T2=0.101) within 1% of the correct value (A2=0.1). In fact, over *most* combinations of the three concentrations, the TFit method works better (although, of course, *nothing* works if the spectral difference between the components is too small).

Note: in this program, as in all the above, when you change InstWidth, the photon noise automatically changes accordingly just as it would in a real variable-slit dispersive spectrophotometer with a white light source. You can also download the newer self-contained keyboard-operated version (TFit3Demo.m) that works in Matlab or in recent versions of Octave:

KEYSTROKE COMMANDS (Matlab or Octave)

A1	**A**/**Z**	Increase/decrease true absorbance of component 1
A2	**S**/**X**	Increase/decrease true absorbance of component 2
A3	**D**/**C**	Increase/decrease true absorbance of component 3
Sepn	**F**/**V**	Increase/decrease spectral separation of the components
InstWidth	**G**/**B**	Increase/decrease width of instrument function (spectral bandpass)
Noise	**H**/**N**	Increase/decrease random noise level when InstWidth = 1
Peak shape	**Q**	Toggles between Gaussian and Lorentzian absorption peak shape
Table	**Tab**	Print table of results
	K	Print this list of keyboard commands

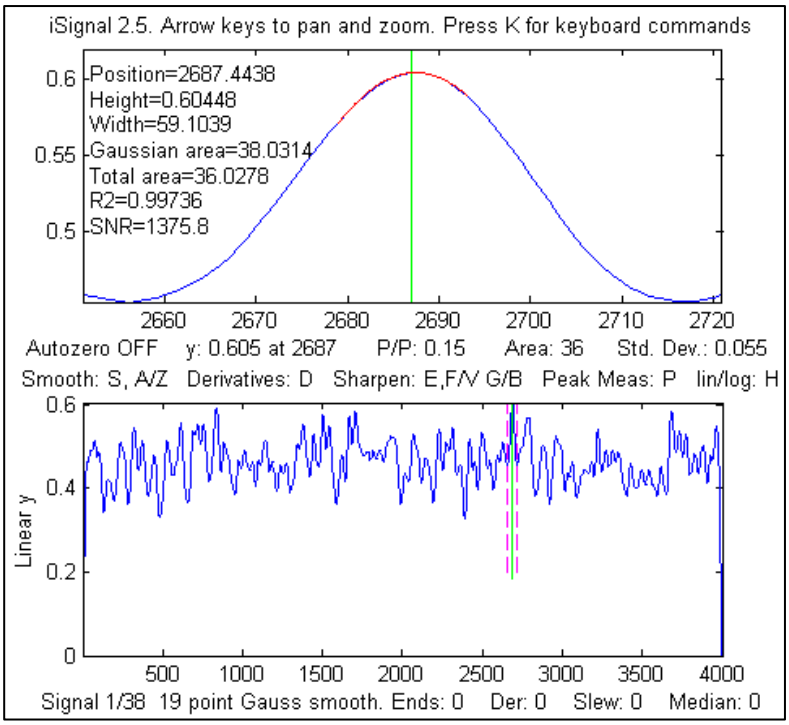
Sample table of typical results (displayed by pressing the **Tab** key):

	True Absorbance	Weighted Regression	TFit method
Component 1	**3**	**2.06**	**3.001**
Component 2	**0.1**	**0.4316**	**0.09829**
Component 3	**5**	**2.464**	**4.998**

Clearly, the Tfit method gives results that are much closer to the true values.

Case Studies and Numerical Experiments.

Can smoothed noise be mistaken for an actual signal?



Here are two examples that show that the answer to this question is *yes*. The first example is shown on the left. This shows iSignal (page 376) displaying a computer-generated 4000-point signal consisting of pure random noise that has been smoothed with a 19-point P-spline smooth. The upper window shows a tiny slice of this signal that looks like a Gaussian peak with a calculated SNR over 1000. Only by looking at the entire signal (bottom window) do you see the true picture. That "peak" is just part of the noise, smoothed so that it looks nice. Do not fool yourself.

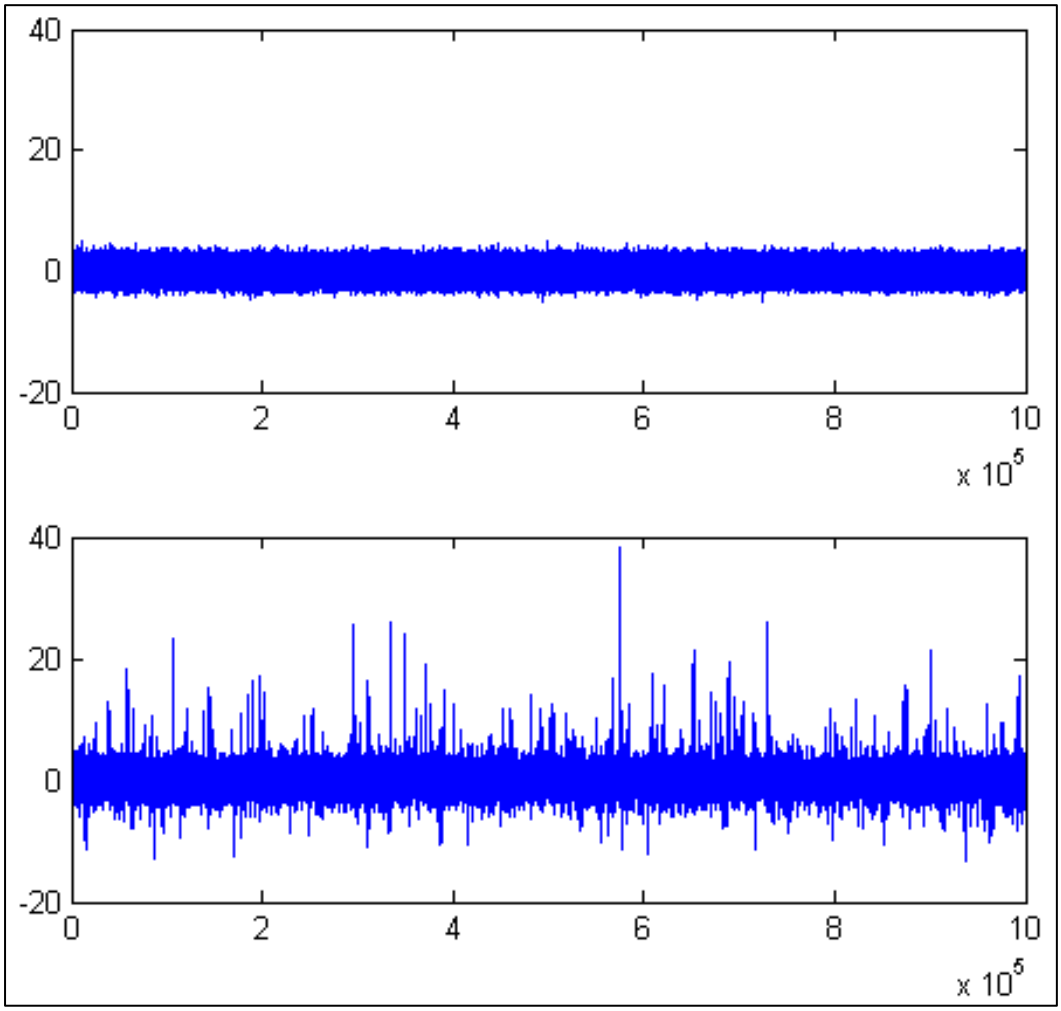
The second example is a simple series of three Matlab commands that use the 'randn' function to generate a 10000-point data set containing only normally-distributed white noise. Then it uses 'fastsmooth.m' to smooth that noise, resulting in a 'signal' with a standard deviation of about 0.3 and a maximum value around 1.0. That signal is then submitted to the iPeak function (page 253). If the peak detection criteria (e.g., AmpThreshold and SmoothWidth) are set too low, many peaks will be found. But setting the AmpThreshold to 3 times the standard deviation (3 x 0.3 = 0.9) will greatly reduce the incidence of these false peaks.

```
>> noise=randn(1,10000);
>> signal=fastsmooth(noise,13);
>> iPeak([1:10000;signal],0,0.6,1e-006,17,17)
```

The peak identification function, which identifies peaks based on their exact x-axis peak position and a stored table of previously identified peak positions, is even *less* likely to be fooled by random noise, because in addition to the peak detection criteria of the findpeaks algorithm, any detected peak must *also* match closely to a peak position in the table of known peaks.

Signal or Noise?

In this case study, the client's experimental signal was unusual because it was very large (over 1,000,000 points) and did not look like a typical signal when plotted. In fact, it just looked like a lot of *noise,* at first glance. The figure below compares the raw experimental signal (bottom panel) with the same number of points of normally-distributed computer-generated white noise (top panel) with a mean of zero and a standard deviation of 1.0, obtained from the Matlab/Octave 'randn' function.

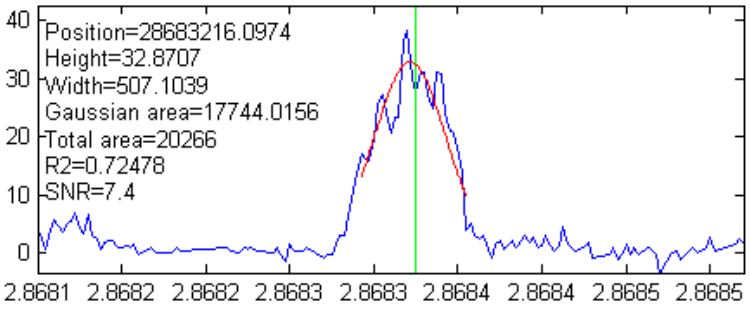


As you can see, the main difference is that the signal has more large 'spikes', especially in a positive direction. This difference is evident when you look at the descriptive statistics of the signal and the randn function:

DESCRIPTIVE STATISTICS	Raw signal	random noise (randn function)
Mean	0.4	0
Maximum	38	about 5 - 6
Standard Deviation (STD)	1.05	1.0
Inter-Quartile Range (IQR)	1.04	1.3489
Kurtosis	38	3
Skewness	1.64	0

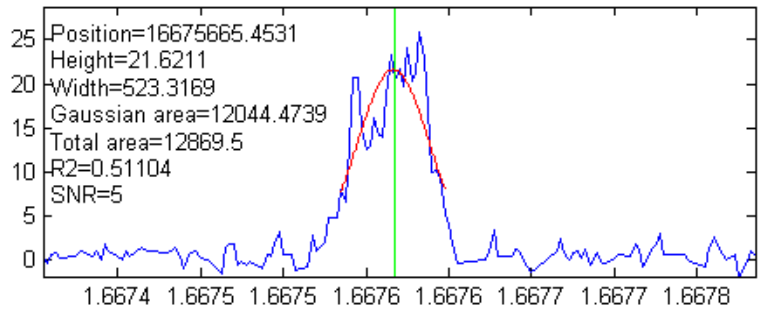
You can see that the *standard deviations of these two are nearly the same*, but the other statistics (especially the kurtosis and skewness) indicate that the probability distribution of the signal is *far*

from normal. There are far more positive spikes in the signal than expected for pure noise. These spikes turned out to be the peaks of interest for this signal. They look like spikes only because the length of

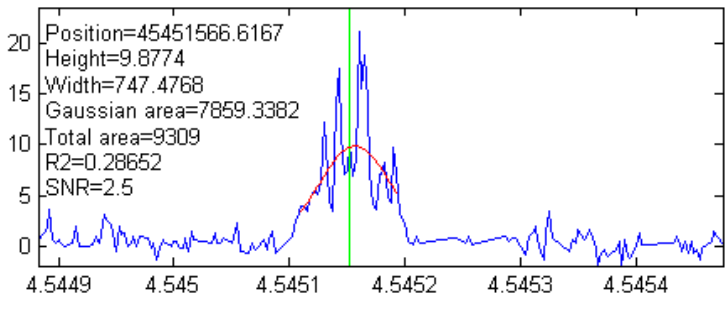


the signal (over 1,000,000 points) causes the peaks to be compressed into one screen pixel or less when the entire signal is plotted on the screen. In the figures on the left, iSignal (page 376) is used to "*zoom in*" on some of the larger of these peaks (using the cursor arrow keys). The peaks are very sparsely separated (by an average of 1000 half-widths between peaks) and are well above the level of background noise (which has a standard deviation of roughly 0.9 throughout the signal).

The researcher who obtained this signal told me that a 'good' peak was expected to be 'bell-shaped', with an amplitude above 5 and a width of 500-1000 x-axis units. So that means that you can expect the signal-to-background-noise ratio to be at least 5/0.9 = 5.5. You can see in the three example peaks in



the figures on the left. The peak widths do indeed meet those expectations. The interval between adjacent x-axis points is 25, so that means that you can expect the peaks to have about 20 to 40 points in their widths. Based on that, you can expect that the positions, heights, and widths of the peaks should be able to be measured fairly accurately using least-squares methods (which reduce the uncertainty of measured parameters by about the square root of the number of points used - about a factor of 5 in this case). However, *the noise appears to be signal-dependent*. The noise on the top of the peaks is distinctly greater than the noise on the baseline. The result is that the actual signal-to-noise (S/N) ratio of peak parameter measurement for the larger peaks will not be as good as might be expected based on the ratio of the peak height to the noise on the background. Most likely, the total noise in this signal is the sum of two major components, one with a fixed standard deviation of 0.9 and the other roughly equal to 10% of the peak height.



To automate the detection of large numbers of peaks, you can use the findpeaksG or iPeak (page 416) functions. Reasonable values of the input arguments *AmplitudeThreshold*, *SlopeThreshold*, *SmoothWidth*, and *FitWidth* for those functions can be estimated based on the expected peak height (5) and width (20 to 40 data points) of the "good" peaks. For example, using *AmplitudeThreshold*=5, *SlopeThreshold*=.001, *SmoothWidth*=25, and *FitWidth*=25, these functions detect and measure 76 peaks above an amplitude of 5 and with an average peak width of 523. The interactive iPeak function (page 416) is especially convenient for

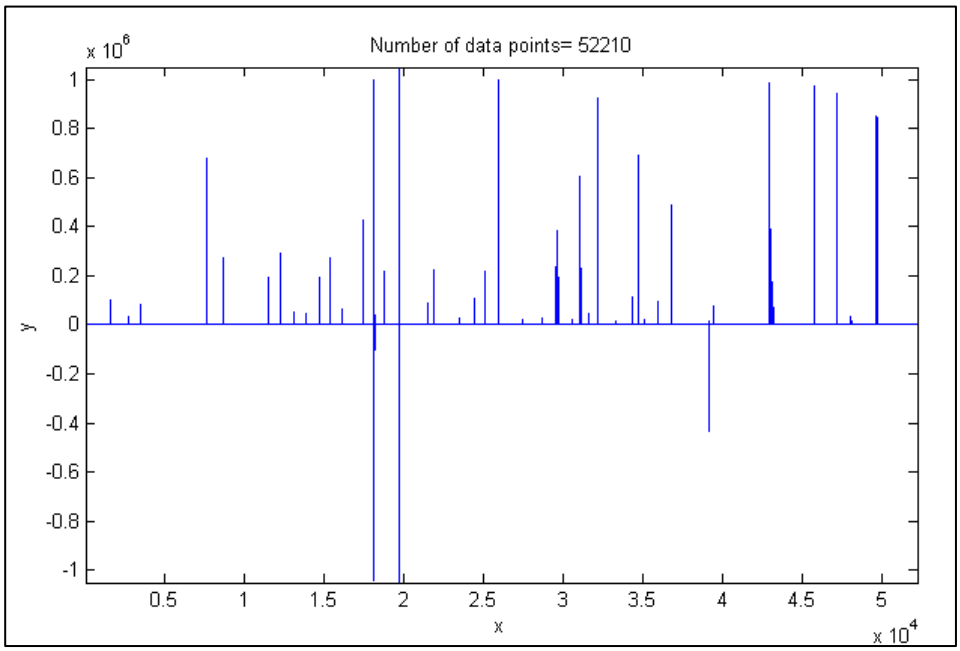
exploring the effect of these peak detection parameters and for graphically inspecting the peaks that it finds. Ideally, the objective is to find a set of peak detection arguments that detect and accurately measure all the peaks that you would consider 'good' and skip all the 'bad' ones. But the criteria for good and bad peaks is at least partly subjective, so it is usually best to err on the side of caution and avoid skipping 'good' peaks at the risk of including a few 'bad' peaks in the mix, which can be weeded out manually based on unusual position, height, width, or appearance.

Of course, it must be expected that the values of the peak position, height, and width given by the findpeaksG or iPeak functions will only be approximate and will vary depending on the exact setting of the peak detection arguments. The noisier the data, the greater the uncertainty in the peak parameters. In this regard, the peak-fitting functions peakfit.m and ipf.m usually give more accurate results, because they make use of *all* the data across the peak, not just the top of the peak as do findpeaksG and *iPeak*. For example, compare the results of the peak near x=3035200 measured with iPeak (click to view) and with peakfit (click to view). Also, the peak fitting functions are better for dealing with overlapping peaks and for estimating the uncertainty of the measured peak parameters, using the bootstrap options of those functions. For example, the largest peak in this signal has an x-axis position of 2.8683e+007, a height of 32, and a width of 500. The bootstrap method determines that the standard deviations are 4, 0.92, and 9.3, respectively.

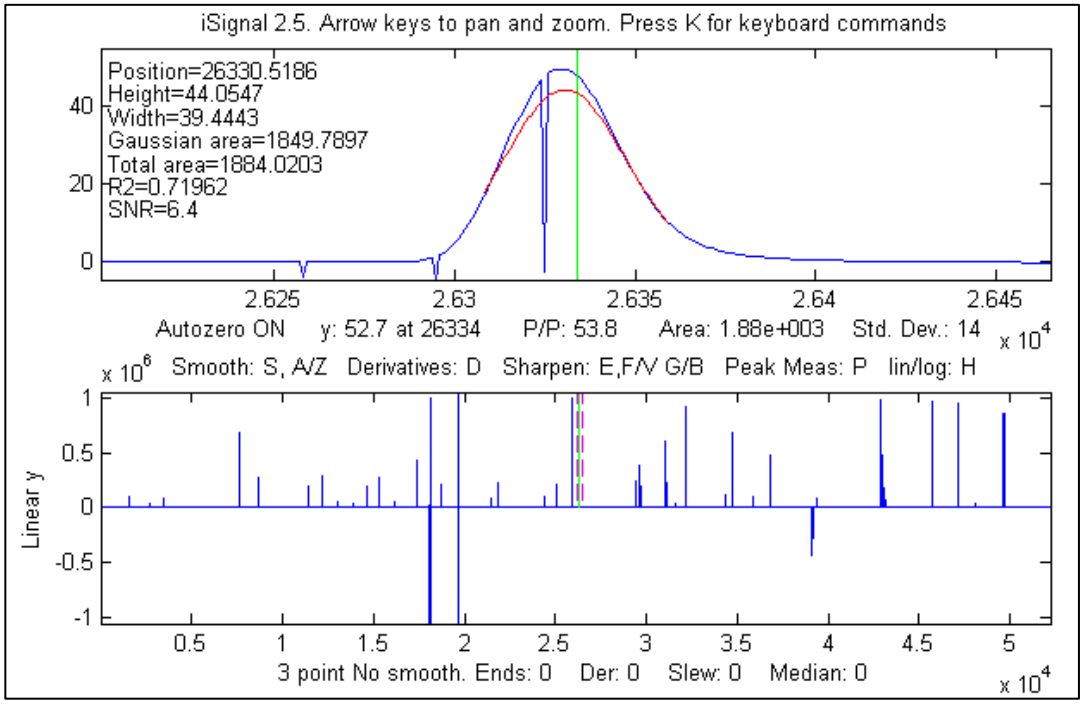
Because the signal in the case study was so large (over 1,000,000 points), the interactive programs such as iPeak, iSignal, and ipf might be sluggish in operation, especially if your computer is not fast computationally or graphically. If this is a serious problem, it may be best to break the signal up into two or more segments and deal with each segment separately, then combine the results. Alternatively, you can use the condense function to average the entire signal into a smaller number of points by a factor of 2 or 3 (at the risk of slightly reducing peak heights and increasing peak widths), but then you should reduce *SmoothWidth* and *FitWidth* by the same factor to compensate for the reduced number of data points across the peaks. Run testcondense.m for a demonstration of the condense function.

Buried treasure

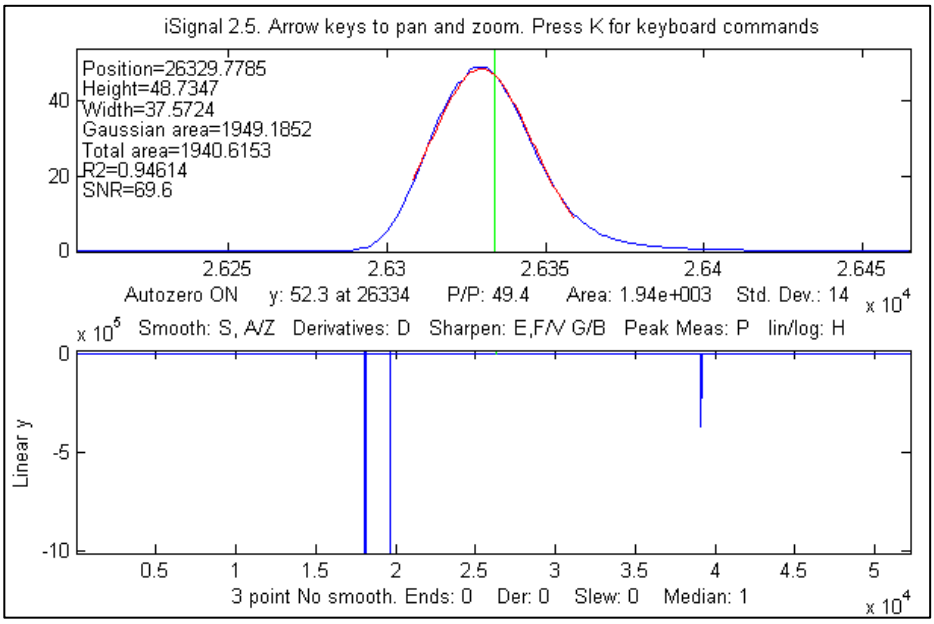
The experimental signal in this example had several narrow spikes rising above a *seemingly flat baseline*. The experimenter said that the spikes were artifacts.



Using iSignal (page 376) to investigate the signal, I found that the visible positive spikes were *single points* of large amplitude ($1x10^6$), whereas the regions *between* the spikes were not really flat but contained many bell-shaped peaks that were so much smaller (by a factor of 1000) that they were not even visible at first. For example, using iSignal to zoom in to the region around x=26300, you can see one of those bell-shaped peaks with a small single-point negative-going spike artifact near its peak.

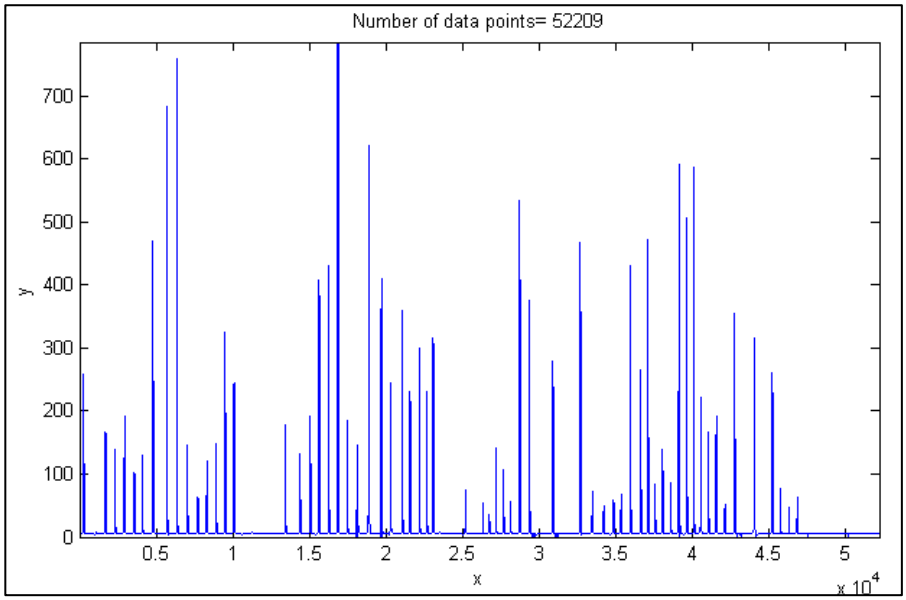


Most of the peaks in this signal had narrow spikes like this. Such artifacts are occasionally seen in experimental signals. They are easy to eliminate by using a median filter. The iSignal function (page 376) has such a filter, activated by the "**M**" key. The result (on the next page) shows that the single-point spike artifacts have been eliminated, with little effect on the character of the bell-shaped peak.

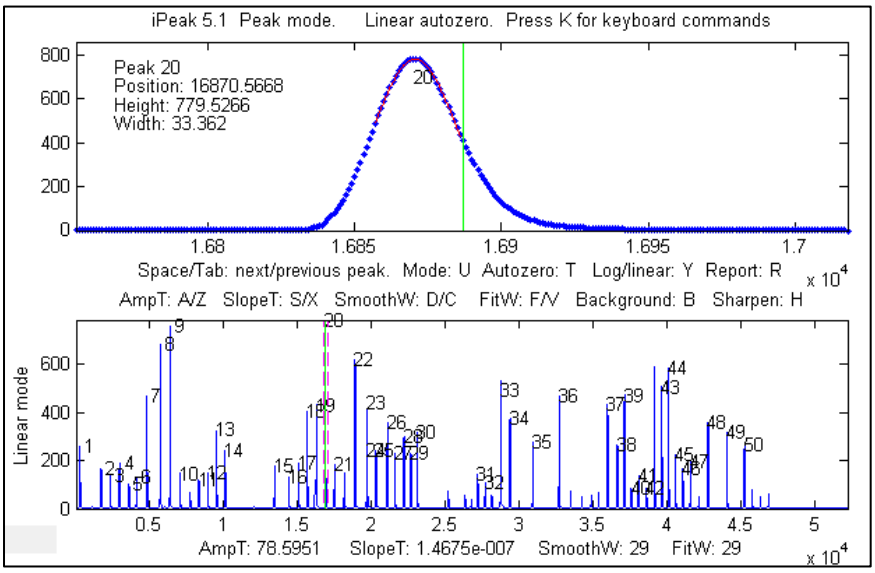


Other filter types, like most forms of smoothing (page 42), would be far less effective than a median filter for this type of artifact and would distort the peaks.

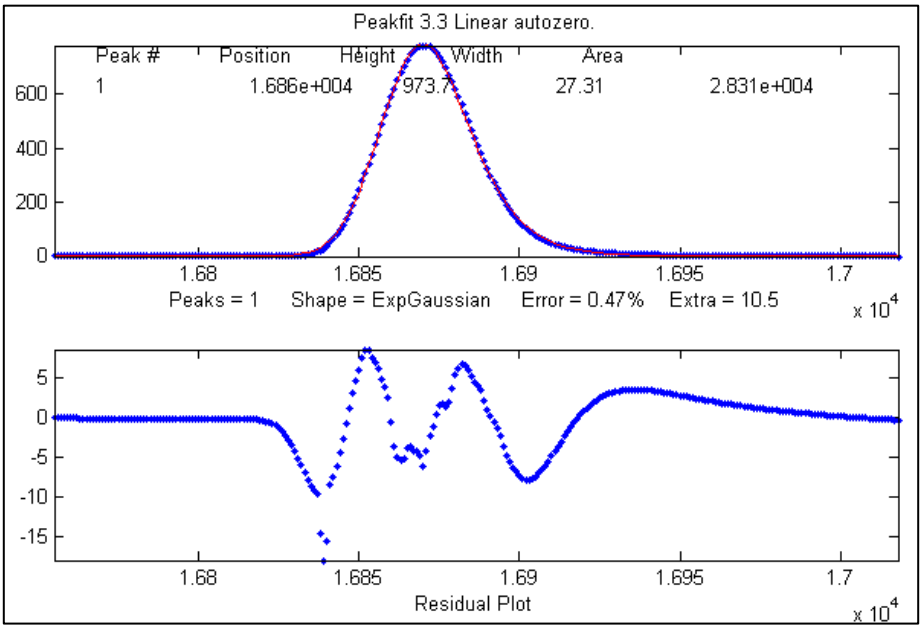
The negative spikes in this signal turned out to be steep *steps*, which can either be reduced by using iSignal's slew-rate limit function (the ` key) or manually eliminated by using the semicolon key (;) to set the selected region between the dotted red cursor lines to zero. Using the latter approach, the entire cleaned-up signal is shown below. The remaining peaks are all positive, bell-shaped and have amplitudes from about 6 to about 750.



iPeak (page 416) can automate measurements of all the peak positions and heights for the entire signal at once, using the peak detection settings shown at the bottom of the screen shot below.



If required, individual peaks can be measured more accurately by fitting the whole peak with *iPeak*'s "**N**" key (page 416) or with peakfit.m or ipf.m (page 416). The peaks are all slightly asymmetrical. They fit an exponentially-broadened Gaussian model (page 228) to a fitting error less than about 0.5%. The smooth residual plots suggest that the signal was smoothed *before* the spikes were introduced and that the noise increases with the signal amplitude (because these is little or noise on the baseline).



Note that fitting with an exponentially-broadened Gaussian model gives the peak parameters of the Gaussian before broadening. iSignal (page 376) and *iPeak* (page 416) estimate the peak parameters of

the broadened peak. As before, the effect of the broadening is to shift the peak position to larger values, reduce the peak height, and increase the peak width.

```
Position  Height  Width   Area  error
isignal  16871  788.88  32.881  27612          S/N Ratio = 172
ipeak    16871  785.34  33.525  28029
peakfit  16871  777.9   33.488  27729  1.68%  Gaussian model
peakfit  16863  973.72  27.312  28308  0.47%  Exponentially-broadened Gaussian
```

The Battle Rounds: a comparison of methods

This numerical experiment demonstrates the application of several techniques described in this book to the quantitative measurement of a peak that is buried in an unstable background, a situation that commonly occurs in the quantitative analysis applications of various forms of spectroscopy, process monitoring, and remote sensing. The objective is to derive a measure of peak amplitude that varies linearly with the actual peak amplitude but that is *minimally affected by the changes in the background and by the random noise*. In this example, the peak to be measured is located at a fixed location in the center of the recorded signal, at x=100 and has a fixed shape (Gaussian) and width (30). The background, on the other hand, is highly variable, both in amplitude *and* in shape. The numerical simulation shows six superimposed recordings of the signal with six different peak amplitudes and with randomly varying background amplitudes and shapes (top row left in the following figures). The methods that are compared here include smoothing (page 42), differentiation (page 63), classical least-squares multicomponent method (page 188), and iterative non-linear curve fitting (page 199).

CaseStudyC.m is a self-contained Matlab/Octave demo function that demonstrates this case. To run it, download it, place it in the search path, and type "CaseStudyC" at the command prompt. Each time you run it, you will get the same series of true peak amplitudes (set by the vector "SignalAmplitudes", in line 12) but a different set of background shapes and amplitudes. The background is modeled as a Gaussian peak of randomly varying amplitude, position, and width. You can control the average *amplitude* of the background by changing the variable "BackgroundAmplitude" and the *average change* in the background by changing the variable "BackgroundChange".

The five methods compared in the figures that follow are:

1: Top row center. A simple zero-to-peak measurement of the smoothed signal, which assumes that the background is *zero*.

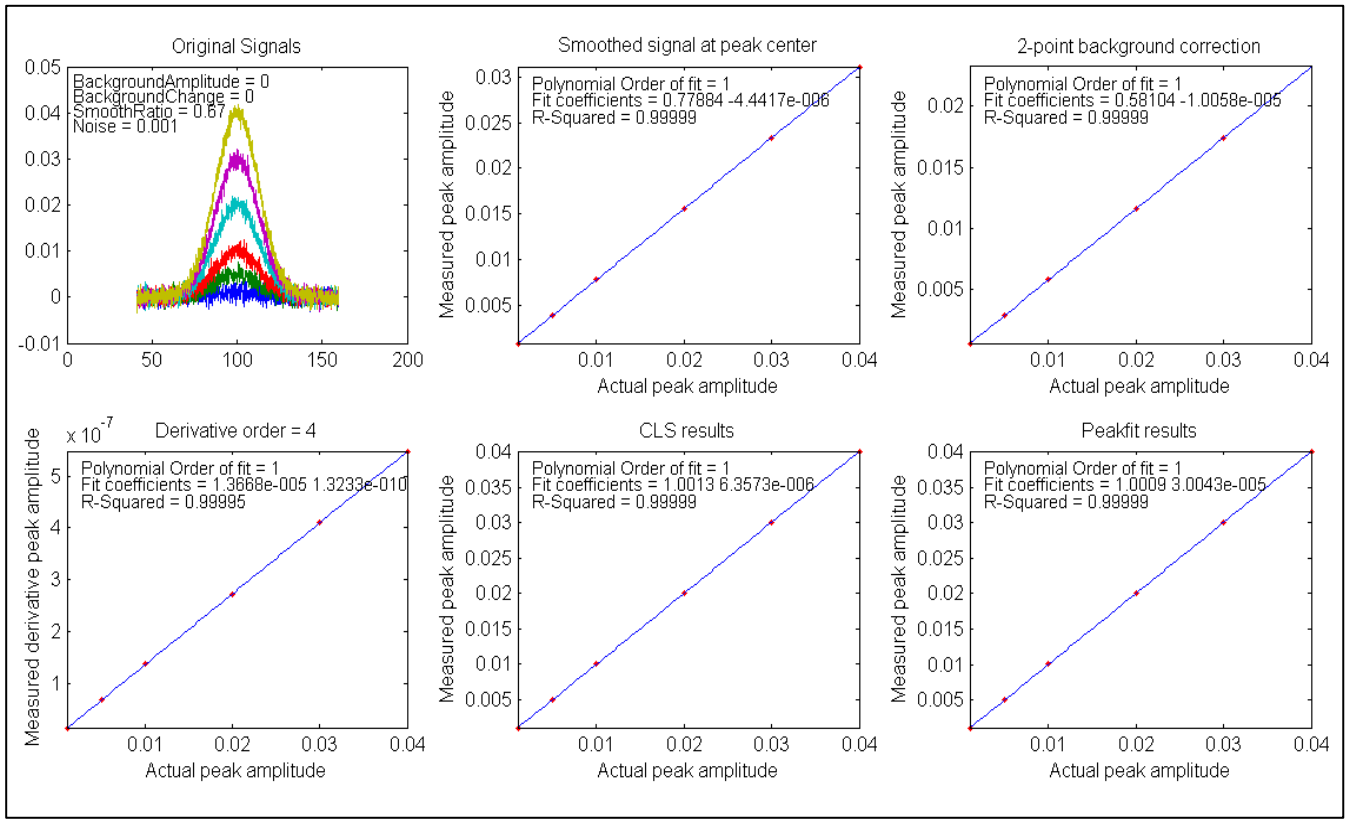
2: Top row right. The difference between the peak signal and the average background on both sides of the peak (both smoothed), which assumes that the background is *flat*.

3: Bottom row left. A derivative-based method, which assumes that the background is *very broad* compared to the measured peak.

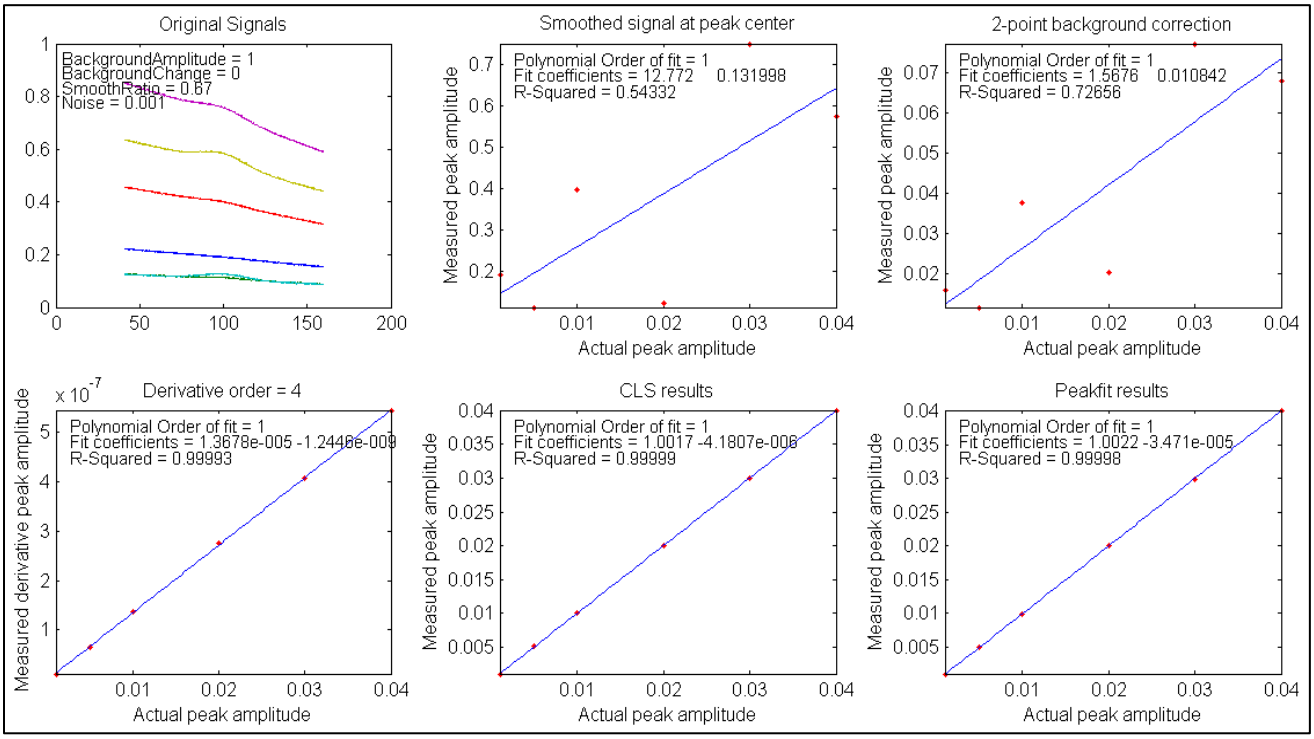
4: Bottom row center. Classical least-squares (CLS), which assumes that the background is a peak of *known shape, width, and position* (the only unknown being the *height*).

5: Bottom row right. iterative non-linear curve fitting (INLS), which assumes that the background is a peak of *known shape* but unknown width and position. This method can track changes in the background peak position and width (within limits) if the measured peak and the background *shapes* are independent of the concentration of the unknown.

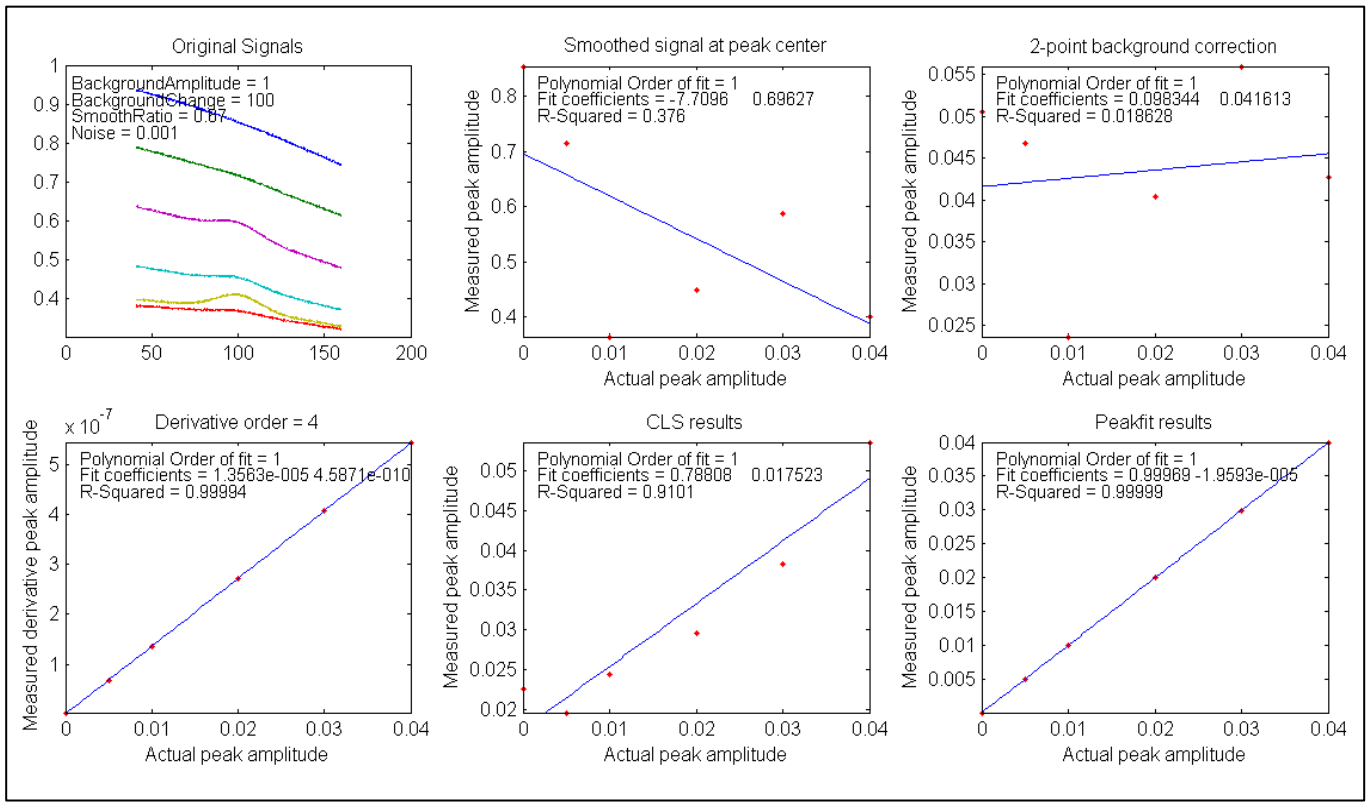
These five methods are listed roughly in order of increasing mathematical and geometrical complexity. They are compared below by plotting the actual peak heights (set by the vector "SignalAmplitudes") against the quantity derived from that method, fitting those data to a straight line, and computing the *coefficient of determination,* R^2, which of course is 1.0000 for a perfectly linear plot.



Each plot is labeled at the top with the method of peak amplitude measurement. For the first test (shown in the figure above), both "BackgroundAmplitude" and "BackgroundChange" are set to zero, so that only the random noise is present. In that case, all the methods work well, with R^2 values all very close to 0.9999. With a 10x higher noise level (click to view), all methods still work about equally well, but with a lower coefficient of determination R^2, as might be expected.



For the second test (shown in the figure immediately above), "BackgroundAmplitude"=1 and "BackgroundChange"=0, so the background has significant amplitude variation but a fixed shape, position, and width. In that case, the first two methods fail, but the derivative, CLS, and INLS methods work well.



For the third test, shown in the figure above, “BackgroundAmplitude”=1 and “BackgroundChange” =100, so in this case the background varies in position, width, and amplitude (but remains broad compared to the signal). Here, the CLS method fails as well, because it assumes that the background varies only in amplitude. However, if you go one step further (click to view) and set “BackgroundChange”= 1000, the background shape is now so unstable that even the INLS method fails, but the derivative method still remains effective as long as the background is broader than the measured peak, no matter what its shape. On the other hand, if the width and position of the *measured* peak changes from sample to sample, the derivative method will fail and the INLS method is more effective (click to view), as long as the fundamental shape of both measured peak and the background are both known (e.g. Gaussian, Lorentzian, etc.)

Not surprisingly, the more mathematically complex methods perform better, on average. Fortunately, software can "hide" that complexity

---

Ensemble averaging patterns in a continuous signal

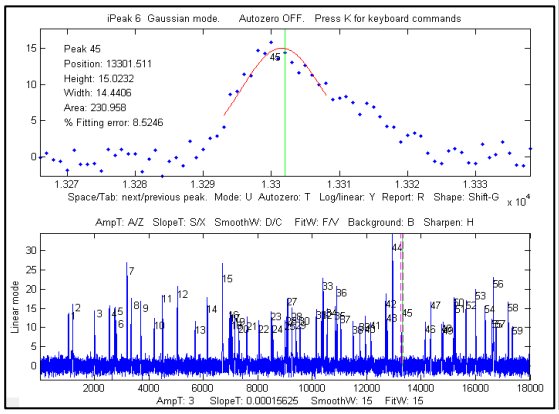
Ensemble averaging is a powerful method of reducing the effect of random noise in experimental signals when it can be applied. The idea is that the signal is repeated, preferably many times, and all the repeats are averaged. The signal builds up, and the noise gradually averages towards zero, as the number of repeats increases.

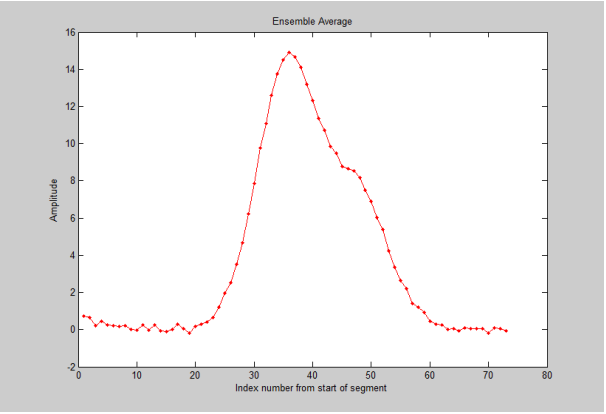
An important requirement is that the repeats be aligned or synchronized so that in the absence of random noise, the repeated signals would line up exactly. There are two ways of managing this:

(a) the signal repeats are triggered by some external event and the data acquisition can use that trigger to synchronize the signals, or
(b) the signal itself has some feature that can be used to detect each repeat, whenever it occurs.

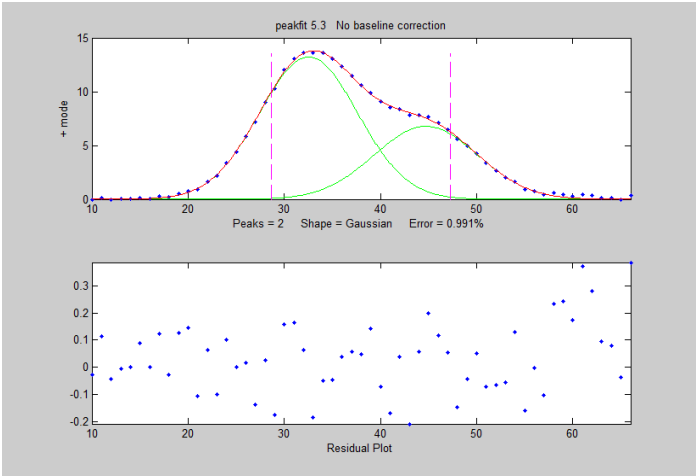
The first method (a) has the advantage that the signal-to-noise (S/N) ratio can be arbitrarily low and the average signal will still gradually emerge from the noise if the number of repeats is large enough. However, not every experiment has a reliable external trigger.

The second method (b) can be used to average repeated patterns in one continuous signal without an external trigger that corresponds to each repeat, but the signal must then contain some feature (for example, a peak) with a signal-to-noise ratio large enough to detect reliably in each repeat. This method can be used even when the signal patterns occur at random intervals when the timing of the repetitions is not of interest. The interactive peak detector iPeak 6 (page 416) has a built-in ensemble averaging function (Shift-E) that can compute the average of all the repeating waveforms. It works by detecting a single peak in each repeat to synchronize the repeats.





The Matlab script iPeakEnsembleAverageDemo.m (on http://tinyurl.com/cey8rwh) demonstrates this idea, with a simulated signal that contains a repeated underlying pattern of two overlapping Gaussian peaks, 12 points apart, with a 2:1 height ratio, both of width 12. These patterns occur at random intervals, and the noise level is about 10% of the average peak height. Using *iPeak* (page 416) shown above left), you adjust the peak detection controls to detect only one peak in each repeat pattern, zoom in to isolate any one of those repeat patterns, and press Shift-**E**. In this case, there are about 60 repeats, so the expected signal-to-noise (S/N) ratio improvement is sqrt(60) = 7.7. You can save the averaged pattern (above right) into the Matlab workspace as "EA" by typing




```
>> load EnsembleAverage; EA=EnsembleAverage;
```

then curve-fit this averaged pattern to a 2-Gaussian model using the peakfit.m function (figure on the right):

```
peakfit([1:length(EA);EA],40,60,2,1,0,10)

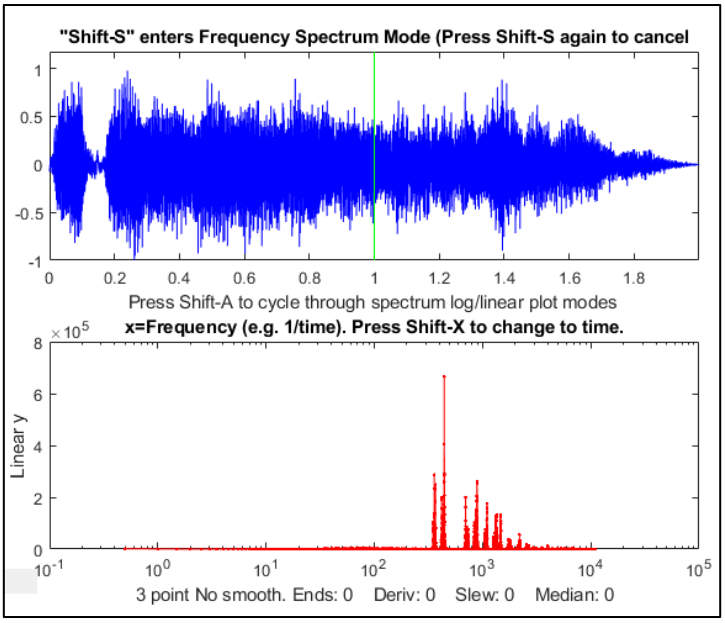
Position    Height     Width     Area
 32.54      13.255     12.003    169.36
 44.72       6.7916    12.677     91.69
```

You will see a big improvement in the accuracy of the peak separation, height ratio, and width, compared to fitting a *single* pattern in the original x,y signal:

```
>> peakfit([x;y],16352,60,2,1,0,10)
```

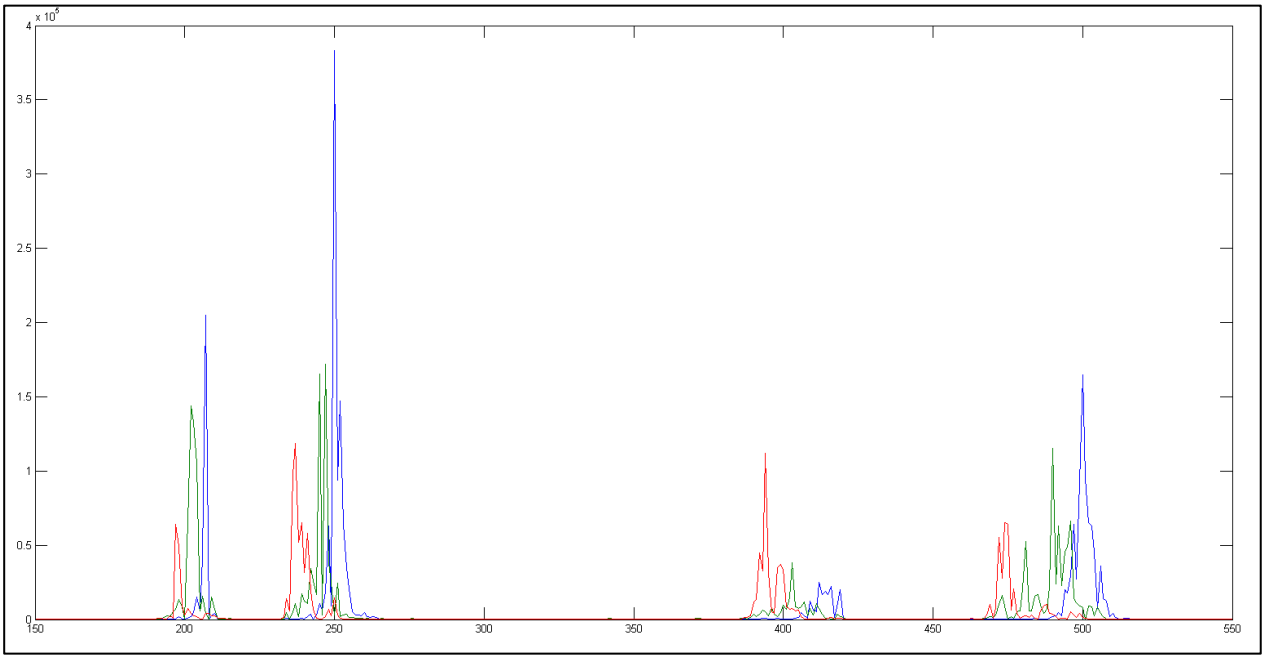
Harmonic Analysis of the Doppler Effect

The wav file "horngoby.wav" (**Ctrl-click** to open) is a 2-second recording of the sound of a passing automobile horn, exhibiting the familiar Doppler effect. The sampling rate is 22000 Hz. Download this file and place it in your Matlab path. You can then load this into the Matlab workspace as the variable "doppler" and display it using iSignal (page 376):




```
t=0:1/21920:2;
load horngoby.mat
isignal(t,doppler);
```

Within iSignal, you can switch to frequency spectrum mode by pressing **Shift-S** and zoom in on different portions of the waveform using the cursor keys, so you can observe the *downward frequency shift* and measure it quantitatively. (Actually, it is much easier to *hear* the frequency shift - press **Shift-P** to play the sound - than to *see* it graphically. The frequency shift is rather small on a percentage basis, but human hearing is very sensitive to small pitch (frequency) changes). It's easier to see if you re-plot the data to stretch out the frequency region around the fundamental frequency or one of the harmonics. I used iSignal to zoom in on three slices of this waveform and then I plotted the frequency spectrum (**Shift-S**) near the *beginning* (shown in blue), *middle* (green), and *end* (red) of the sound. The frequency region between 150 Hz and 550 Hz are plotted in the figure below:



The group of peaks near 200 is the fundamental frequency of the lowest note of the horn and the group

of peaks near 400 is the second harmonic. (Pitched sounds have a harmonic structure of 1, 2, 3... times a fundamental frequency). The group of peaks near 250 is the fundamental frequency of the next higher note of the horn and the group of peaks near 500 is its second harmonic. (Car and train horns often have two or three harmonious notes sounded together). In each of these groups of harmonics, you can clearly see that the blue peak (the spectrum measured at the *beginning* of the sound) has a *higher* frequency than the red peak (the spectrum measured at the *end* of the sound). The green peak, taken in the middle, has an intermediate frequency. *The peaks are ragged because the amplitude and frequency vary over the sampling interval*, but you can still get good quantitative measures of the frequency of each component by curve fitting to a Gaussian peak model using peakfit.m or ipf.m (page 416):

Peak	Position	Height	Width	Area
Beginning	206.69	3.0191e+005	0.81866	2.4636e+005
Middle	202.65	1.5481e+005	2.911	4.797e+005
End	197.42	81906	1.3785	1.1994e+005

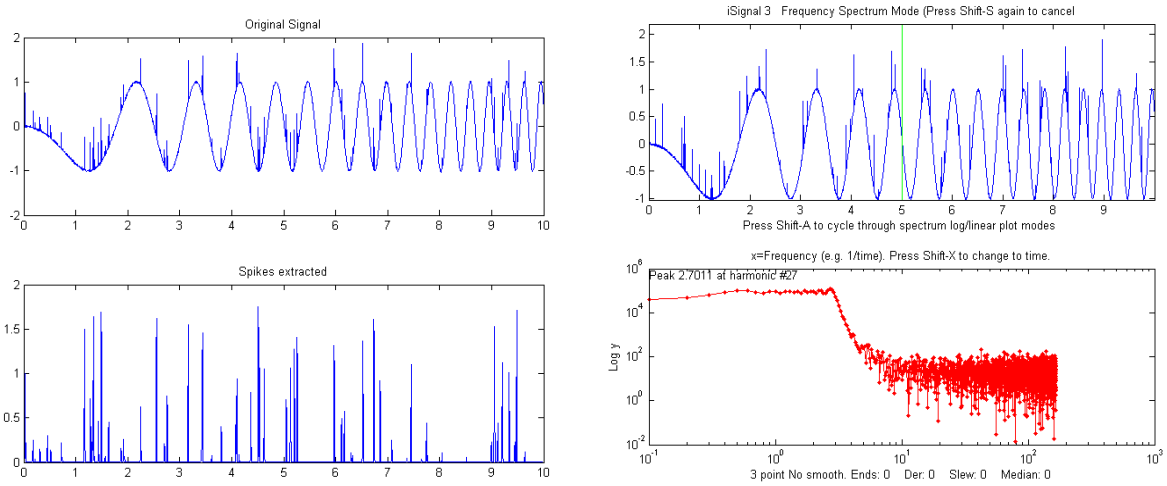
The estimated precision of the peak position (i.e. frequency) measurements is about 0.2% relative, based on the bootstrap method, good enough to allow accurate calculation of the frequency shift (about 4.2%) and of the speed of the vehicle. This also shows that the measured ratio of the second harmonic to the fundamental for these data is 2.0023, which is very close to the theoretical value of 2.

Measuring spikes

Spikes, narrow pulses with a width of only one or a few points, are sometimes encountered in signals as a result of an electronic "glitch" or stray pickup from nearby equipment, and they can easily be eliminated by the use of a "median" filter.

But it is possible that in some experiments the spikes *themselves* might be the important part of the signal and that it is required to count or measure them. This situation was encountered in a research application by one of my clients, and it brings up some interesting twists on the usual procedures. In that application the spikes were caused by grains of wind-blown beach sand striking the diaphragm of a microphone element, accompanied by more-or-less sinusoidal waveforms, probably caused by wind whistling through the equipment or by the calls of shore birds. The objective was to count the sand grain impacts and ignore the other, possibly louder, sounds.

As a numerical simulation of this situation, the Matlab/Octave script SpikeDemo1.m creates a waveform (top panel of the figure below) in which a series of spikes are randomly distributed in time, contaminated by two types of noise: white noise and a large-amplitude oscillatory interference simulated by a swept-frequency sine wave. The objective is to count the spikes and locate their position on the x (time) axis. Direct application of findpeaks or *iPeak* (page 416) to the raw signal does not work well.



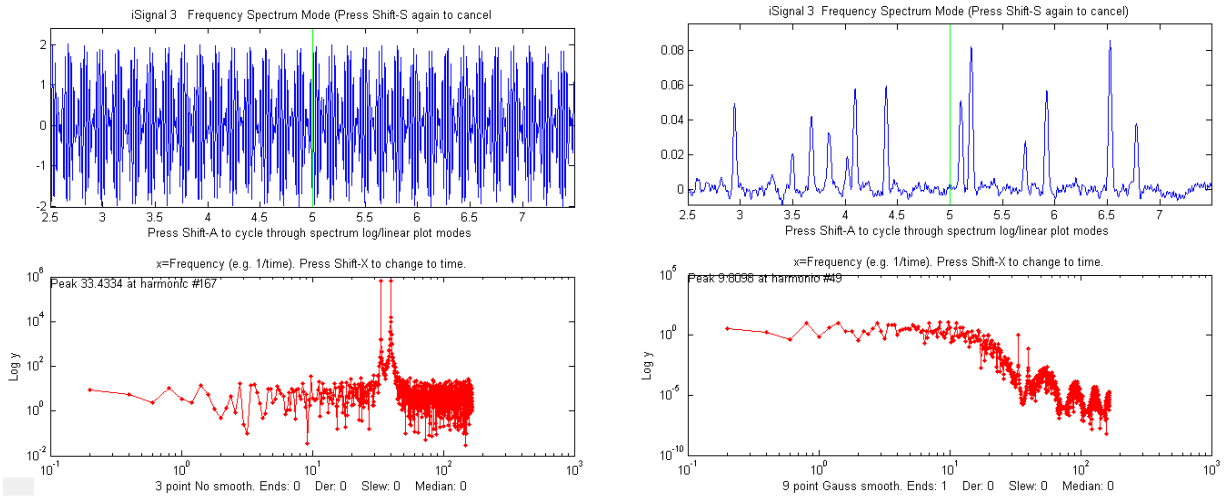
A single-point spike, called a *delta function* in mathematics, has a power spectrum that is *flat*. Which means that it has *equal power at all frequencies*, just like white noise. But the oscillatory interference, in this case, is in a *specific range of frequencies*, which opens some interesting possibilities. One approach would be to use a Fourier filter, for example, a notch or band-reject filter, to remove the troublesome oscillations selectively. But if the objective of the measurement is only to count the spikes and measure their times, a simpler approach would be to (1) compute the second derivative (which greatly amplifies the spikes relative to the oscillations), (2) smooth the result (to limit the white noise amplification caused by differentiation), then (3) take the absolute value (to yield positive-pointing peaks). This can be done in a *single line* of nested Matlab/Octave code:

```
y1=abs(fastsmooth((deriv2(y)).^2,3,2));
```

The result, shown the lower panel of the figure on the left above, is an almost complete extraction of the spikes, which can then be counted with findpeaksG.m or peakstats.m or iPeak.m (page 416):

```
P=ipeak([x;y1],0,0.1,2e-005,1,3,3,0.2,0);
```

The second example, SpikeDemo2.m, is even more difficult. In this case the oscillatory interference is caused by *two* fixed-frequency sine waves at a higher frequency, which *completely obscures* the spikes in the raw signal (top panel of the left figure below). In the power spectrum (bottom panel, in red), the oscillatory interference shows as two sharp peaks that dominate the spectrum and reach to $y=10^6$, whereas the spikes show as the much lower broad flat plateau at about y=10. In this case, use can be made of an interesting property of sliding-average smooths, such as the boxcar, triangular, and Gaussian smooths. Their frequency responses exhibit a series of deep cusps at frequencies that are inversely proportional to their filter widths. So, this opens the possibility of suppressing specific frequencies of oscillatory interference by adjusting the filter widths until the cusps occur at or near the frequency of the oscillations. Since the signal, in this case, consists of spikes that have a flat power spectrum, they are simply smoothed by this operation, which will reduce their heights and increase their widths, but will have little or no effect on their number or x-axis positions. In this case, a 9-point P-spline smooth puts the first (lowest frequency) cusp right in between the two oscillatory frequencies.



In the figure on the right, you can see the effect of applying this filter. The spikes, which were *not even visible* in the original signal, are now cleanly extracted (upper panel), and you can see in the power spectrum (right lower panel, in red) that the two sharp peaks of oscillatory interference are *reduced by about a factor of 1,000,000!* (Note: the frequency spectra are plotted on a *log-log scale*). This operation can be performed by a single command-line function, adjusting the smooth width (the second input argument, here a 9) by trial and error to minimize the oscillatory interference:

```
y1=fastsmooth(y,9,3);
```

If the interference varies in frequency across the signal, you could use a segmented smooth rather than the standard fastsmooth. Alternatively, the segmented Fourier Spectrum (page 104) could be used to visualize this signal, and a Fourier filter (page 127) in "notch" mode could be employed to specifically eliminate the interfering frequencies. The extracted peaks could then be counted using any of the peak finding functions, such as:

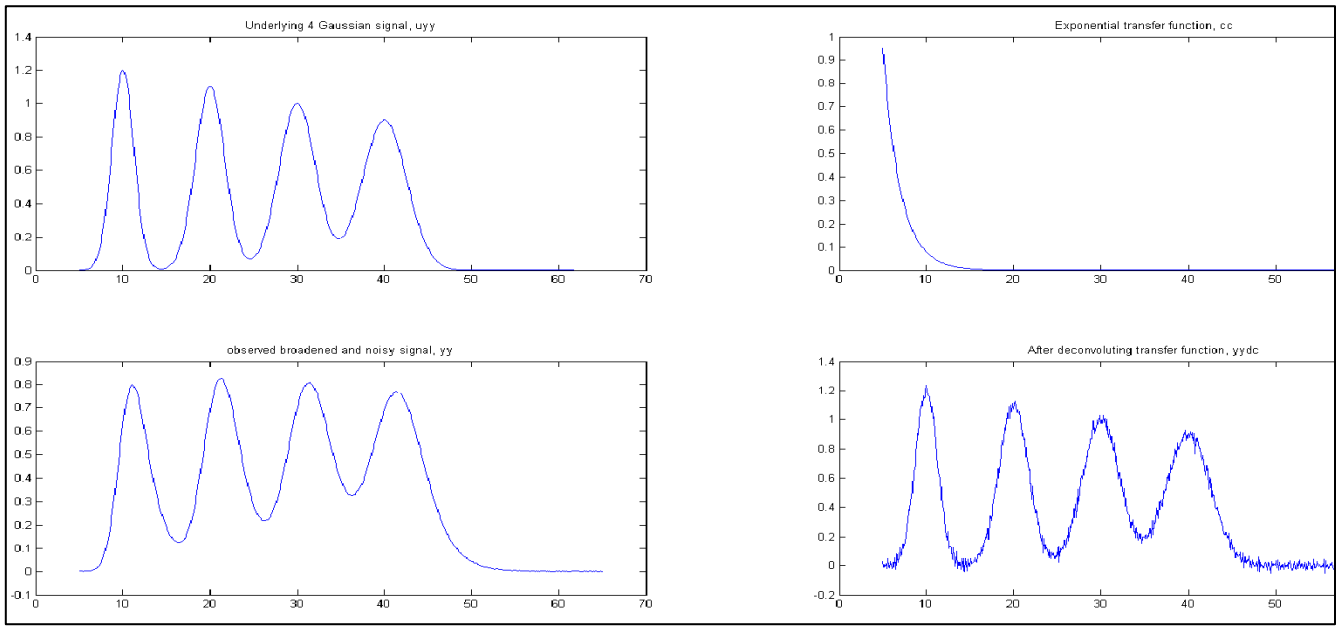
```
P=findpeaksG(x,y1,2e-005,0.01,2,5,3);
or
P=findpeaksplot(x,y1,2e-005,0.01,2,5,3);
or
PS=peakstats(x,y1,2e-005,0.01,2,5,3,1);
```

The simple script "iSignalDeltaTest" demonstrates the power spectrum of the smoothing and differentiation functions of iSignal (page 376) by applying them to a delta function. Change the smooth type, smooth width, derivative order and other functions to see how the power spectrum changes.

Fourier deconvolution vs curve fitting (they are *not* the same)

Some experiments produce peaks that are distorted by being convoluted by processes that make peaks less distinct and modify peak position, height, and width. Exponential broadening is one of the most common of these processes. Fourier deconvolution and iterative curve fitting are two methods that can help to measure *the true underlying peak parameters*. Those two methods are conceptually distinct

because, in Fourier deconvolution, the underlying peak shape is unknown but the nature and width of the broadening function (e.g., exponential) is assumed to be known. In iterative least-squares curve fitting it is just the reverse: the underlying peak *shape* (i.e., Gaussian, Lorentzian, etc.) must be known,



but the width of the broadening process is initially unknown.

In the script shown below and the resulting graphic shown above (Download this script), the underlying signal (uyy) is a set of four Gaussians with peak heights of 1.2, 1.1, 1, 0.9 located at x=10, 20, 30, 40 and peak widths of 3, 4, 5, 6, but in the *observed* signal (yy) these peaks are broadened exponentially by the exponential function cc, resulting in shifted, shorter, and wider peaks, and then a little constant white noise is added *after* the broadening. The deconvolution of cc from yy successfully removes the broadening (yydc), but at the expense of a substantial noise increase. However, the extra noise in the deconvoluted signal is high-frequency weighted ("blue") and so is easily reduced by smoothing and *has less effect on least-square fits than does white noise*. (For a greater challenge, try adding more noise in line 6 or use a bad estimate of time constant in line 10). To plot the recovered signal overlaid with the underlying signal: **`plot(xx, uyy, xx, yydc)`**. To plot the observed signal overlaid with the underlying signal: **`plot(xx,uyy,xx,yy)`**. `Excellent values for the original underlying peak positions, heights, and widths can be obtained by` curve-fitting the recovered signal to four Gaussians: **`[FitResults,FitError]= peakfit([xx;yydc],26,42,4,1,0,10)`**.

With *ten times* the previous noise level (Noise=.01), the values of peak parameters determined by curve fitting are still quite good, and even with *100x more noise* (Noise=0.1) the peak parameters are more accurate than you might expect for that amount of noise (because that noise is *blue*). Visually, the noise is so great that the situation looks *hopeless*, but the curve fitting works well.

```
xx=5:.1:65;
% Underlying Gaussian peaks with unknown heights, positions, and widths.
uyy=modelpeaks2(xx,[1 1 1 1],[1.2 1.1 1 .9],[10 20 30 40],[3 4 5 6],...
[0 0 0 0]);
% Observed signal yy, with noise added AFTER the broadening convolution
Noise=.001; % <---Try more noise to see how this method handles it.
yy=modelpeaks2(xx,[5 5 5 5],[1.2 1.1 1 .9],[10 20 30 40],[3 4 5 6],...
[-40 -40 -40 -40])+Noise.*randn(size(xx));
% Compute transfer function, cc,
cc=exp(-(1:length(yy))./40); % <---Change exponential factor here
% Attempt to recover original signal uyy by deconvoluting cc from yy
% It is necessary to zero-pad the observed signal yy as shown here.
yydc=deconv([yy zeros(1,length(yy)-1)],cc).*sum(cc);
% Plot and label everything
subplot(2,2,1);plot(xx,uyy);title('Underlying signal, uyy');
subplot(2,2,2);plot(xx,cc);title('Exponential transfer function, cc')
subplot(2,2,3);plot(xx,yy);title('observed broadened, noisy signal, yy');
subplot(2,2,4);plot(xx,yydc);title('Recovered signal, yydc')
```

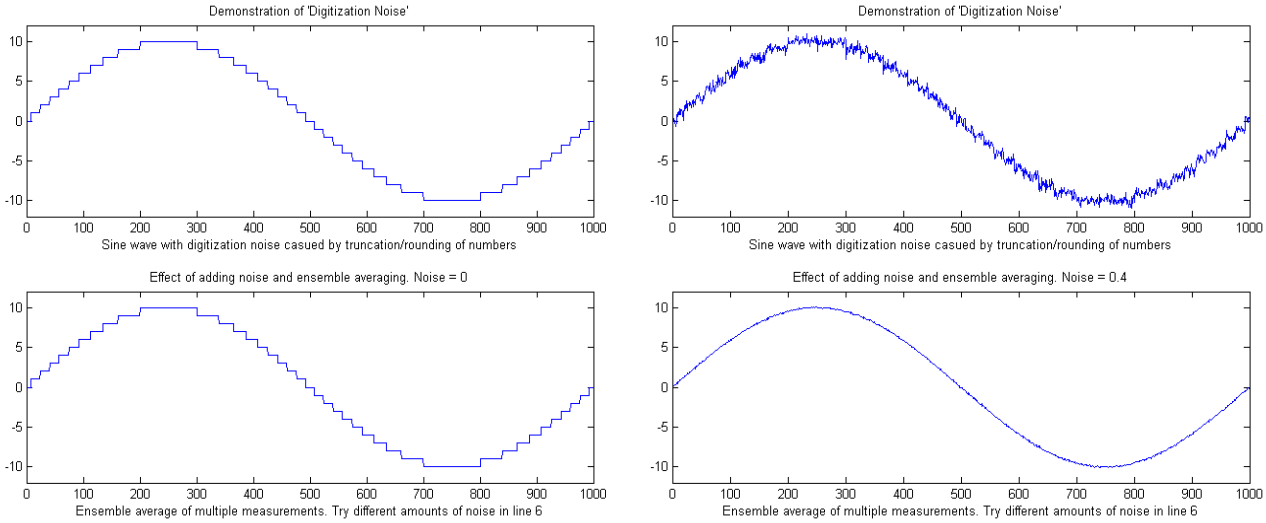
An alternative to the above deconvolution approach is to curve fit the observed signal directly with an exponentially broadened Gaussian (shape number 5):

[FitResults,FitError] = peakfit([xx;yy],26,50,4,5,40,10).

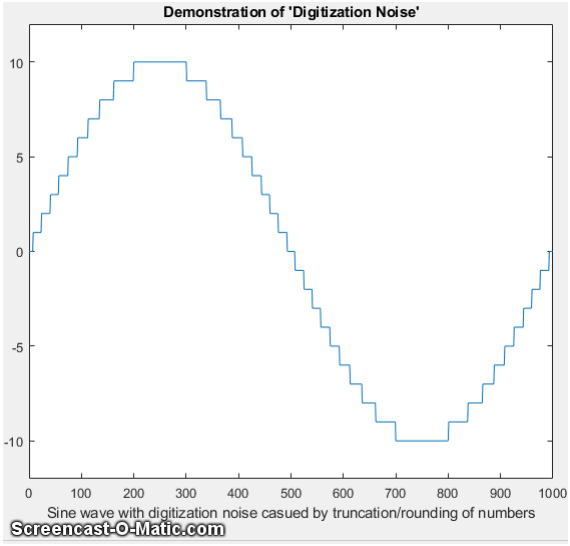
Both methods give good values of the peak parameters, but the deconvolution method is considerably faster because curve fitting with a simple Gaussian model is faster than fitting with an exponentially broadened peak model, especially if the number of peaks is large. Also, if the exponential factor is not known, it can be determined by curve fitting one or two of the peaks in the observed signal, using ipf.m (page 416), adjusting the exponential factor interactively to get the best fit. Note that *you must give peakfit a reasonably good value for the time constant* ('extra'), the input argument right after the peakshape number. If the value is too far off, the fit may fail completely, returning all zeros. A little trial and error suffice. Alternatively, you could try to use peakfit.m version 7 with the *independent variable* exponent broadened Gaussian shape number 31 or 39, to measure the time constant as an iterated variable (to understand the difference, see example 39). If the time constant is expected to be the *same* for all peaks, better results will be obtained by using shape number 31 or 39 initially to measure the time constant of an isolated peak (preferably one with a good S/N ratio), then apply that fixed time constant in peak shape 5 to all the other groups of overlapping peaks.

Digitization noise - can adding noise really help?

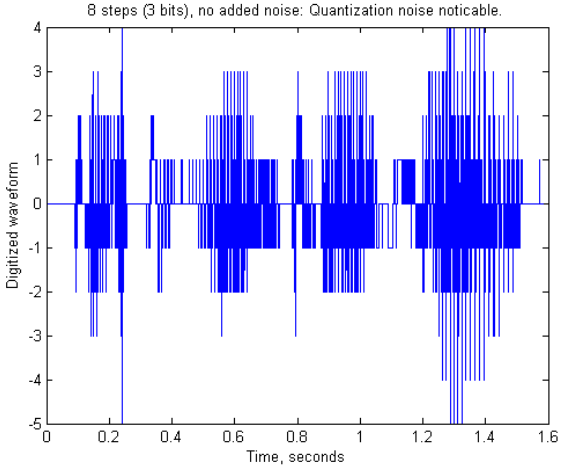
Digitization noise, also called quantization noise, is an artifact caused by the rounding or truncation of numbers to a fixed number of figures. It can originate in the analog-to-digital converter that converts an analog signal to a digital one, or in the circuitry or software involved in transmitting the digital signal to a computer, or even in the process of transferring the data from one program to another, as in copying and pasting data to and from a spreadsheet. The result is a series of non-random steps of equal height. The frequency distribution is white, because of the sharpness of the steps, as you can see by observing the power spectrum.



The figure on the left, top panel, shows the effect of integer digitization on a sine wave with an amplitude of +/- 10. Ensemble averaging, which is usually the most effective of noise reduction techniques, does not reduce this type of noise (bottom panel) because it is non-random.



Interestingly, if additional random noise is present in the signal, then ensemble averaging becomes effective in reducing *both* the random noise *and* the digitization noise. In essence, the added noise *randomizes the digitization*, allowing it to be reduced by ensemble averaging. Moreover, if the noise already in the signal is too low, it can be beneficial to *add additional noise artificially*! Seriously. The script RoundingError.m illustrates this effect, as shown the animation on the right above, which shows the digitized sine wave with gradually increasing amounts of added random noise in line 8 (generated by the randn.m function) followed by ensemble averaging of 100 repeats (in lines 17-20). Look closely at the waveform in this animation as it changes in response to the random noise addition shown in the title. You can clearly see how the noise starts out mostly quantization noise but then quickly decreases as small but increasing amounts of random noise are added before the ensemble averaging step, then eventually increases as too much noise is added. The optimum standard deviation of random noise is about 0.36 times the quantization size, as you can demonstrate by adding lesser or greater amounts via the variable *Noise* in line 6 of this script. Note that this works only for ensembled averaged signals where the random



noise is added *before* the quantization.

An *audible* example of this idea is illustrated by the Matlab/Octave script DigitizedSpeech.m, which starts with an audio recording of the spoken phrase "Testing, one, two, three", previously recorded at 44000 Hz and saved in WAV format (TestingOneTwoThree.wav) and in .mat format (testing123.mat), rounds off the amplitude data progressively to 8 bits (256 steps; sound link), shown on the previous page, 4 bits (16 steps; sound link), and 1 bit (2 steps; sound link), and then the 2-step case again *with random white noise added* before the rounding (2 steps + noise; sound link), plots the waveforms and plays the resulting sounds, demonstrating both the degrading effect of rounding and the remarkable improvement caused by adding noise. (Click on these sound links to hear the sounds on your computer). Although the computer program, in this case, does *not* perform an *explicit* ensemble averaging operation as does RoundingError.m, it is likely that the neurons of the hearing center of your brain provide that function by virtue of their response time and memory effect.

How low can you go? Performance with very low signal-to-noise ratios.

This is a numerical experiment comparing several techniques described in this book applied to the quantitative measurement of a peak that is *buried in an excess of random noise*, where the signal-to-noise (S/N) ratio is *below 2*. (Ordinarily, an S/N ratio of 3 or better is desired for reliable detection).

The Matlab/Octave script LowSNRdemo.m performs the simulations and calculations and compares the results graphically, focusing on the behavior of each method as the S/N ratio approaches zero. Four methods are compared:

(1) smoothing, followed by the peak-to-peak measure of the smoothed signal (page 42),

(2) a peak finding method based on findpeakG (page 236),

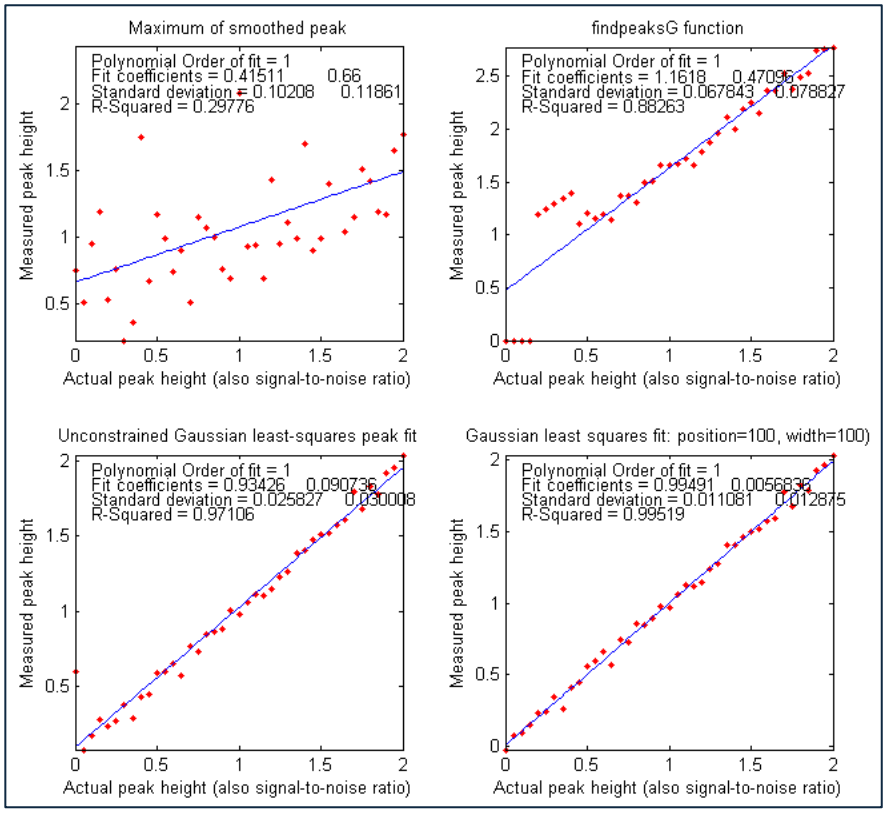
(3) unconstrained iterative least-squares fitting (INLS) based on peakfit.m (page 199),

(4) constrained classical least-squares fitting (CLS) based on the cls2.m function (page 188).

The measurements are carried out over a range of peak heights for which the S/N ratio varies from 0 to 2. The noise is random, constant, and white. Each time you run the script, you get the same set of underlying signals but independent samples of the random noise.

Results for the initial values in the script are shown in the plots and in the table on the next page, both of which are created by the script LowSNRdemo.m. The graphs on the left show correlation plots of the measured peak height vs the real peak height, which should ideally be a straight line with a slope of 1, an intercept of zero, and an R-squared of 1. As you can see, the simplest smoothed-peak method (upper left) is completely inadequate, with a low slope (because smoothing reduces peak height) and a high intercept (because even smoothed noise has a non-zero peak-to-peak value). The findpeaks function (upper right) works OK for height for higher peak heights but fails completely below an S/N ratio of 0.5 because the peak height falls below the amplitude threshold setting. In comparison, the two least-squares techniques work much better, reporting much better values of slope, intercept of zero, and R-squared. But if you look closely at the low end of the peak height range, near x=zero, you can see

that the values reported by the unconstrained fit (lower left) occasionally stray from the line, whereas the constrained fit (lower right) decrease gracefully all the way to zero every time you run the script. Essentially the reason why it is even possible to make measurements at such low S/N ratios is that *the data density is very high*: that is, there are many data points in each signal (about 1000 points across the half-width of the peak with the initial script values). The results are summarized in the table below.




```
Number of points in half-width of peak: 1000
Method          Height Error      Position Error
Smoothed peak     21.2359%          120.688%
findpeaksG.m      32.3709%          33.363%
peakfit.m         2.7542%           4.6466%
cls2.m            1.6565%
```

The height errors are reported as a percentage of the maximum height (initially 2). (For the first three methods, the peak position is also measured, and its relative accuracy is reported. The constrained classical least-squares fitting does not measure peak position but rather assumes that it remains fixed at the initial value of 100). You can see that the CLS method has a slight advantage in the accuracy, but you must consider also that this method works well only if the peak shape, position, and width are known. The unconstrained iterative method can track changes in peak position and width.

You can change several of the factors in this simulation to test the robustness of these methods. Search for the word 'change' in the comments for values that can be changed. Reduce MaxPeakHeight (line 8) to make the problem harder. Change peak position and./or width (lines 9 and 10) to show how the CLS method fails. As usual, the more you know, the better your results. Change the increment (line 4) to

change the data density. More data is always better.

(Surprisingly, as you will see on page 337, Measurement Calibration, *it is not even necessary to have an accurate peak shape model* in order to get a good correlation between measured and actual height).

LowSNRdemo.m also computes the power spectrum of the signal and the amplitude (square root of the power) of the fundamental, where most of the power of a broad Gaussian peak falls, and plots it in Figure window(2). The correlation to peak height is like the CLS method, but *the intercept is higher* because there is a non-zero quantity of noise even in that one frequency slice of the power spectrum.

We are now, in the 21st century, into the era of "big data", where high-speed automated data systems can acquire, store, and process greater quantities of data than ever before. As this little example shows, greater quantities of data allow researchers to probe deeper and measure smaller effects than previously.

Signal processing in the search for extraterrestrial intelligence

The signal detection problems facing those who search the sky for evidence of extraterrestrial civilizations or interesting natural phenomena are enormous. Among those problems are the fact that we do not know much about what to expect: we do not know exactly where to look in the sky, or what frequencies might be used, or the possible forms of the transmissions. Moreover, astronomers do not want to confuse the many powerful sources of natural and *terrestrial* sources of interfering signals for genuine *extraterrestrial* ones. There is also the massive computer power required, which drove the development of specialized hardware and software as well as, until recently, distributed computation over thousands of Internet-connected personal computers across the world using the former SETI@home computational screen-saver. Although some of the computational techniques used in this search are more sophisticated than those covered in this book, they begin with the basic concepts covered here.

One of the reoccurring themes of this book has been that the more you know about your data, the more likely you are to obtain a reliable measurement. In the case of possible extraterrestrial signals, we do not know much, but we do know a few things.

We do know that electromagnetic radiation over a wide range of frequencies is used for long-distance transmission on earth and between earth and satellites and probes far from earth. Astronomers already use radio telescopes to receive natural radiation from vast distances. To look at different frequencies simultaneously, Fourier transforms of the raw telescope signals can be computed over multiple time segments. On page 97, I showed a simulation that demonstrated how hard it is to see a periodic component in the presence of an equal amount of random noise and yet how easy it is to pick it out in the frequency spectrum.

Also, transmissions from extraterrestrial civilizations might be in the form of groups of pulses, so their detection and verification are also part of SETI signal processing. Interestingly, triplets and other groups of equally spaced pulses appear in the Fourier transforms of high-frequency carrier waves that

are amplitude or frequency modulated (like AM or FM radio). Of course, there is no reason to assume, nor to reject, that extraterrestrial civilizations might use the same methods of communication as ourselves.

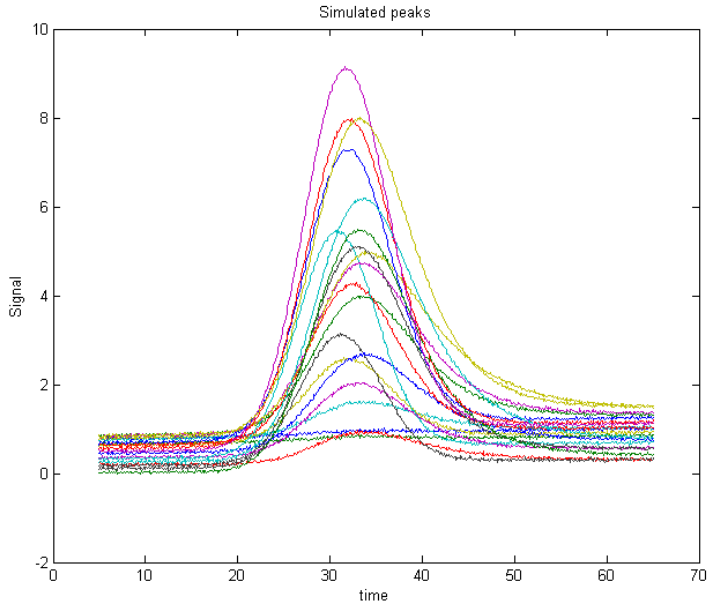
One thing that we know for sure is that the earth rotates around its axis once a day and that it revolves around the sun once a year. So, if we look at a fixed direction out from the earth, the distant stars will seem to move in a predictable pattern, whereas terrestrial sources will remain fixed on earth. The huge Arecibo Observatory dish in Puerto Rico (sadly no longer operational) was fixed in position and was often used to look in one selected direction for extended periods of time. The field of view of this telescope is such that a point source at a distance takes 12 seconds to pass, as the earth rotates. As SETI says:

“Radio signals from a distant transmitter should get stronger and then weaker as the telescope's focal point moves across that area of the sky. Specifically, the power should increase and then decrease with a bell-shaped curve (a Gaussian curve). Gaussian curve-fitting is an excellent test to determine if a radio wave was generated 'out there' rather than a simple source of interference somewhere here on Earth since signals originating from Earth will typically show constant power patterns rather than curves”.

Also, any observed 12-second peaks can be re-examined with another focal point shifted towards the west to see if it repeats with the expected time and duration.

It is well known that there will be a Doppler shift in the frequencies observed if the source is moving relative to the receiver. This is observed with sound waves as well as with electromagnetic waves like radio or light. Because the earth is rotating and revolving at a known and constant speed, we can accurately predict and compensate for the Doppler shift caused by earth's motion (this is called “de-chirping” the data).

Why measure peak area rather than peak height?



This simulation examines more closely the question of measuring peak area rather than peak height to reduce the effect of peak broadening, which commonly occurs in chromatography, for reasons that are discussed previously, and also in some forms of spectroscopy. Under what conditions might the measurement of peak area be better than peak height?

The Matlab/Octave script "HeightVsArea.m" simulates the measurement of a series of standard samples whose concentrations are given by the vector 'standards'. Each standard produces an isolated peak whose peak height is directly proportional to the corresponding value in 'standards' and whose *underlying* shape is a Gaussian with a constant peak position ('pos') and width ('wid'). To simulate the measurement of these samples under typical conditions, the script changes the shape of the peaks (by exponential broadening) and adds a variable baseline and random noise. You can control, by means of the variable definitions in the first few lines of the script, the peak beginning and end, the sampling rate 'deltaX' (increment between x values), the peak position and width ('pos' and 'wid'), the sequence of peak heights ('standards'), the baseline amplitude ('baseline') and its degree of variability ('vba'), the extent of shape change ('vbr'), and the amount of random noise added to the final signal ('noise').

The resulting peaks are shown in the figure above. The script prepares a series of "calibration curves" plotting the values of 'standard' against the measured peak heights or areas for each measurement method. The measurement methods include peak height in Figure window 2, peak area in Figure window 3, and curve fitting height and area in Figures 4 and 5, respectively. These plots should ideally have an intercept of zero and an R^2 of 1.000, but the *slope* is greater for the peak area measurements because the area has different units and is numerically greater than peak height. All the measurement methods are baseline corrected. They include code that attempts to compensate for changes in the baseline (controlled by the variable 'baseline').

With the initial values of 'baseline', 'noise', 'vba', and 'vbr', you can clearly see the advantage of peak area measurements (figure 3) compared to peak height (figure 2). This is primarily because of the variability of peak shape broadening ('vbr') and to the averaging out of random noise in the computation of area.

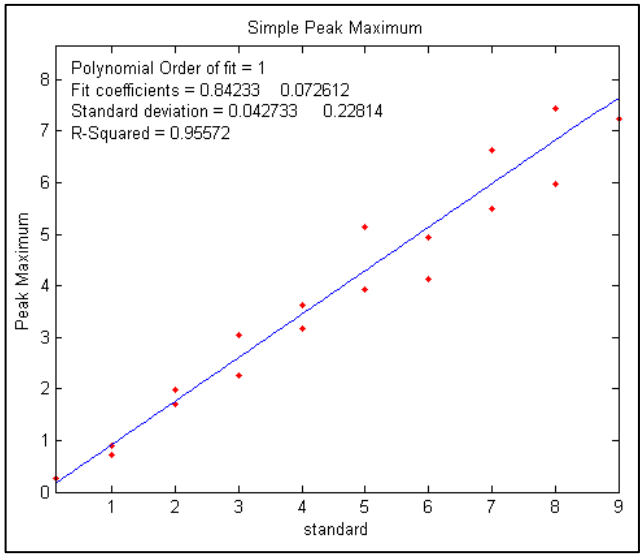


Figure window 2: measuring peak *height*

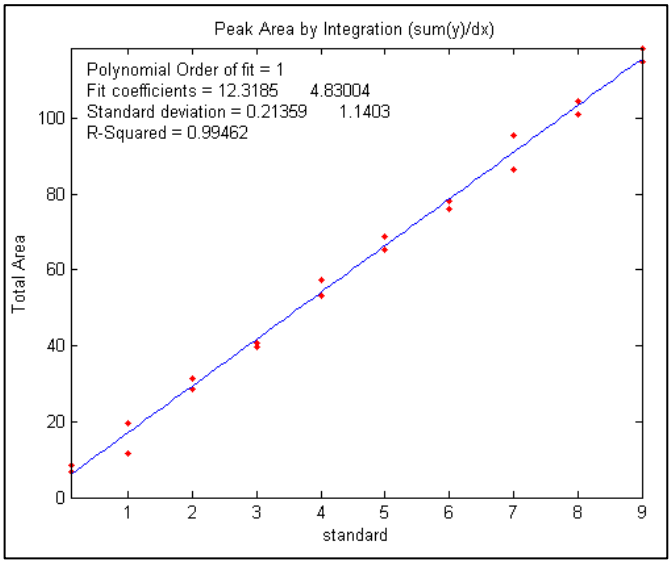


Figure window 3: measuring peak *area*

If you set 'baseline', 'noise', 'vba', and 'vbr' all to zero, you've simulated a perfect world in which all methods work perfectly.

Curve fitting can measure both peak height and area. It is not even absolutely necessary to use an accurate peak shape model. Using a simple Gaussian model in this example works much better for peak area (Figure window 5) than for peak height (Figure window 4) but is not significantly better than a simple peak area measurement (Figure window 3). The best results are obtained if an *exponentially broadened* Gaussian model (shape 31 or 39) is used, using the code in line 30, but that computation takes longer. Moreover, if the measured peak *overlaps* another peak significantly, curve fitting both of those peaks together can give much more accurate results that other peak area measurement methods.

Using macros to extend the capability of spreadsheets

Both Microsoft *Excel* and OpenOffice *Calc* can automate repetitive tasks using “macros”, which are saved sequences of commands or keystrokes that are stored for later use. Macros can be most easily created using the built-in “Macro Recorder”, which will literally watch all your clicks, drags, and keystrokes and record them for later playback. Or you can write or edit your macros in the macro language of that spreadsheet (VBA in *Excel*; Python or JavaScript in *Calc*). Or you can do both: use the macro recorder first, then edit the resulting code manually to modify it. That is what I usually do.

To enable macros in Excel, go to the **File** tab > **Options**. On the left-side pane, select **Trust Center**, and then click **Trust Center Settings**. In the Trust Center dialog box, click **Macro Settings** on the left, select **Enable all macros** and click OK. Then perform your spreadsheet operations, and when finished, click **Stop Recording** and save the spreadsheet. Thereafter, simply pressing your Ctrl-key shortcut will run the macro and perform all the spreadsheet operations that you recorded.

Here I will demonstrate two applications in Excel using macros with the Solver function. (See http://peltiertech.com/Excel/SolverVBA.html#Solver2 for information about setting up macros and

solver on your version of Excel).

A previous section (page 201) described the use of the "Solver" function applied to the iterative fitting of overlapping peaks in a spreadsheet. The steps listed there can easily be captured with the macro recorder and saved with the spreadsheet. However, a different macro will be needed for each different number of peaks, because the block of cells representing the "Proposed Model" will be different for each number of peaks. For example, the template "CurveFitter2Gaussian.xlsm" includes a macro named 'fit' for a 2-peak fit, activated by pressing **Ctrl-f**. Here is the text of that macro:

```
Sub fit()
'
' fit Macro
'
' Keyboard Shortcut: Ctrl+f
'
SolverOk
SetCell:="$C$12", MaxMinVal:=2, ValueOf:=0, ByChange:="$C$8:$D$9",
   Engine:=1, EngineDesc:="GRG Nonlinear"
SolverSolve
End Sub
```

You can see that the text of the macro uses only two macro instructions: "SolverOK" and "SolverSolve". SolverOK specifies all the information in the "Solver Parameters" dialog box in its input arguments: 'SetCell' sets the objective as the percent fitting error in cell C12, 'MaxMinVal' is set to the second choice (Minimum), and 'ByChange' specifies the table of cells representing the proposed model (C8:D9) whose values are to be changed to minimize the objective in cell C12. The last argument sets the Solver engine to 'GRC Nonlinear', the best Solver engine for iterative peak fitting. Finally, "SolverSolve" starts the Solver engine running. You could easily modify this macro for curve fitter templates with other numbers of peaks just by changing the cells referenced in the 'ByChange' argument, e.g., C8:E9 for a 3-peak fit. In this case, though, it is probably just as easy to use the macro recorder to record a macro for each curve fitter template.

A more elaborate example of a spreadsheet using a macro is demonstrated by the spreadsheet called TransmissionFittingCalibrationCurve.xls (screen image) that creates a calibration curve for a series of standard concentrations in the TFit method, which was previously described on page 282. Here's a portion of that macro:

```
    Range("AF10").Select
     Application.CutCopyMode = False
     Selection.Copy
     Range("A6").Select
     Selection.PasteSpecial Paste:=xlPasteValues, Operation:=xlNone,
SkipBlanks _
         :=False, Transpose:=False
     Calculate
     Range("J6").Select
     Selection.Copy
     Range("I6").Select
```

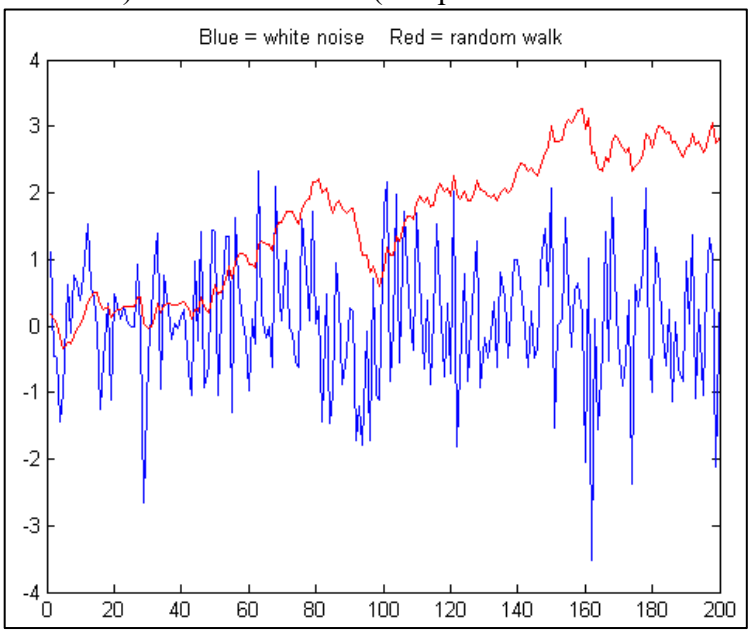
```
    Selection.PasteSpecial Paste:=xlPasteValues, Operation:=xlNone,
SkipBlanks _
        :=False, Transpose:=False
    Calculate
    SolverOk SetCell:="$H$6", MaxMinVal:=2, ValueOf:=0, ByChange:="$I$6",
Engine:=1 _
        , EngineDesc:="GRG Nonlinear"
    SolverOk SetCell:="$H$6", MaxMinVal:=2, ValueOf:=0, ByChange:="$I$6",
Engine:=1 _
        , EngineDesc:="GRG Nonlinear"
    SolverSolve userFinish:=True
    SolverSolve userFinish:=True
    SolverSolve userFinish:=True
    Range("I6:J6").Select
    Selection.Copy
    Range("AG10").Select
    Selection.PasteSpecial Paste:=xlPasteValues, Operation:=xlNone,
SkipBlanks _
        :=False, Transpose:=False
```

The macro in this spreadsheet repeats this chunk of code several times, once for each concentration in the calibration curve (changing only the "AF10" in the first line to pick up a different concentration from the "Results table" in column AF). This macro uses several additional instructions, to select ranges ("Range...Select"), copy ("Selection.Copy") and paste ("Selection.PasteSpecial Paste:= xlPasteValues") values from one place to another and re-calculate the spreadsheet ("Calculate"). Each click, menu selection, or keypress that you make creates one or more lines of macro text. The syntax is wordy but quite explicit and clear. You can learn quite a bit just by recording various spreadsheet actions and looking at the resulting macro text. Or at least that's the way I learned it.

It's worth noting that an additional way to create Excel functions and macros has become available since December of 2022: using an artificial intelligence conversational chatbot such as ChatGPT (*page 442).* Once you have a free account on https://chat.openai.com/, or on any of the more recently introduced chatbots, you can simply ask "Write an Excel macro to…", and describe clearly what you want, referring to cells and ranges in the usual way, and the chatbot will type out a solution that you can simply copy and paste into your Excel worksheet. More help.

Random walks and baseline correction

The *random walk* was mentioned in the section on signals and noise (page 24) as a type of low-frequency ("pink") noise. Wikipedia says: "A random walk is a mathematical formalization of a path that consists of a succession of random steps. For example, the path traced by a molecule as it travels in a liquid or a gas, the tracks of a foraging animal, superstring behavior, the price of a fluctuating stock, and the financial status of a gambler can all be modeled as random walks, although they may not be truly random in reality." Random walks describe and serve as a model for many kinds of unstable behavior. Whereas white, 1/f, and blue noises are tethered to a mean value to which they tend to return, random walks are more aimless and often drift off in one or another direction, possibly never to return. Numerically, a random walk can be modeled as the cumulative sum of some random process, for example the 'randn' function. The graph below compares a 200-point sample of white noise (computed as 'randn' and shown in blue) to a random walk (computed as a cumulative sum, often called 'cumsum',



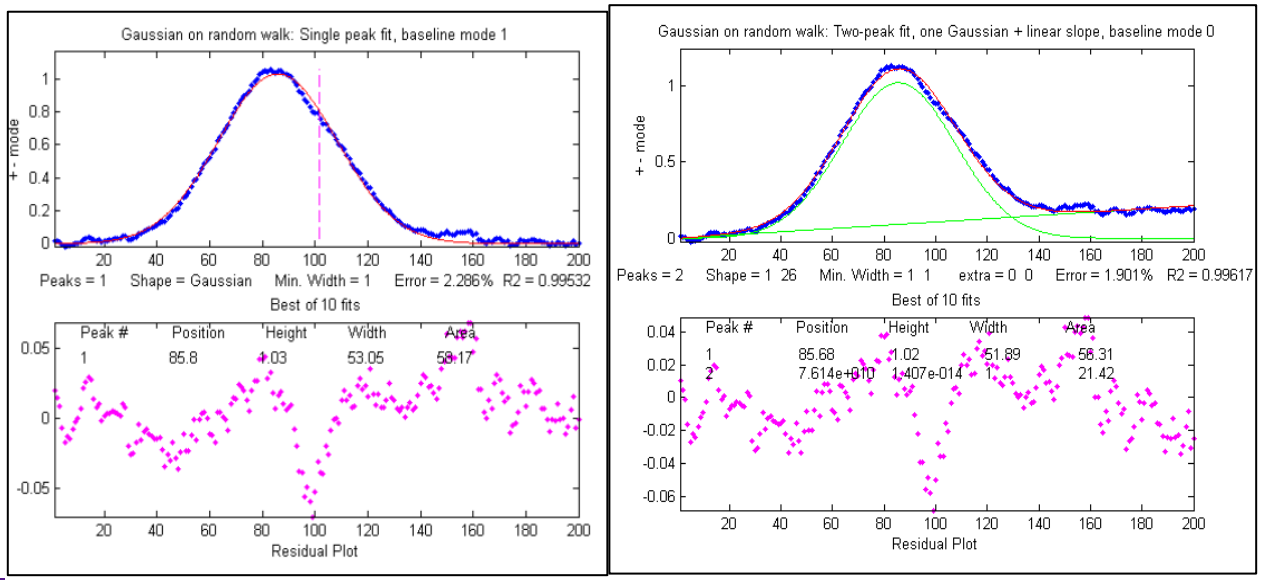
and shown in red). *Both samples are scaled to have the same standard deviation*, but you can see that their behavior is vastly different. The random walk has much more low-frequency behavior, and in this case it wanders off beyond the amplitude range of the white noise. This type of random behavior is very disruptive to the measurement process, distorting the shapes of peaks and causing baselines to shift and making them hard to define. Worse, *it cannot be reduced significantly by smoothing (*as shown by NoiseColorTest.m). In this particular example, the random walk has an overall positive slope and a "bump" near the middle that could be confused for a real signal peak (it is not; it is just one sample of random noise). But another sample might have very *different* behavior. Unfortunately, this behavior is sometimes observed in experimental signals.

To demonstrate the measurement difficulties, the script RandomWalkBaseline.m is a numerical experiment that creates a Gaussian peak with randomly variable position and width, on a random walk baseline, with a relatively good signal-to-noise ratio of 15. The peak is measured by least-squares curve fitting methods using peakfit.m with two different methods of baseline correction to handle the random walk:

Method *a*: a single-component Gaussian model (shape 1) with BaselineMode set to 1 (meaning a linear baseline is first interpolated from the edges of the data segment and subtracted from the signal): **`peakfit([x;y],0,0,1,1,0,10,1);`**

Method *b*: a 2-component model, the first being a Gaussian (shape 1) and the second a linear slope (shape 26), with BaselineMode set to 1: **`peakfit([x;y],0,0,2,[1 26],[0 0],10,0)`**.

In this case, the fitting error is lower for the second method, especially if the peak falls near the edges of the data range.



But the relative percent errors of the peak parameters show that the *first method* gives a lower error for position and width, at least in this case. On average, the peak parameters are about the same.

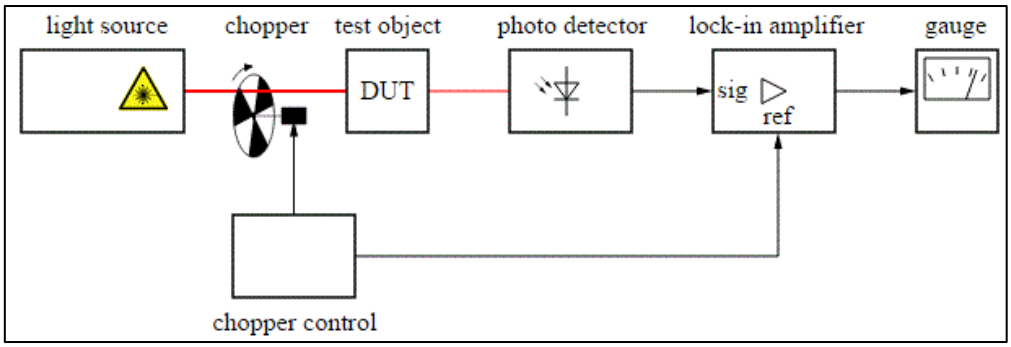
	Position Error	Height Error	Width Error
Method a:	**0.2772**	**3.0306**	**0.0125**
Method b:	**0.4938**	**2.3085**	**1.5418**

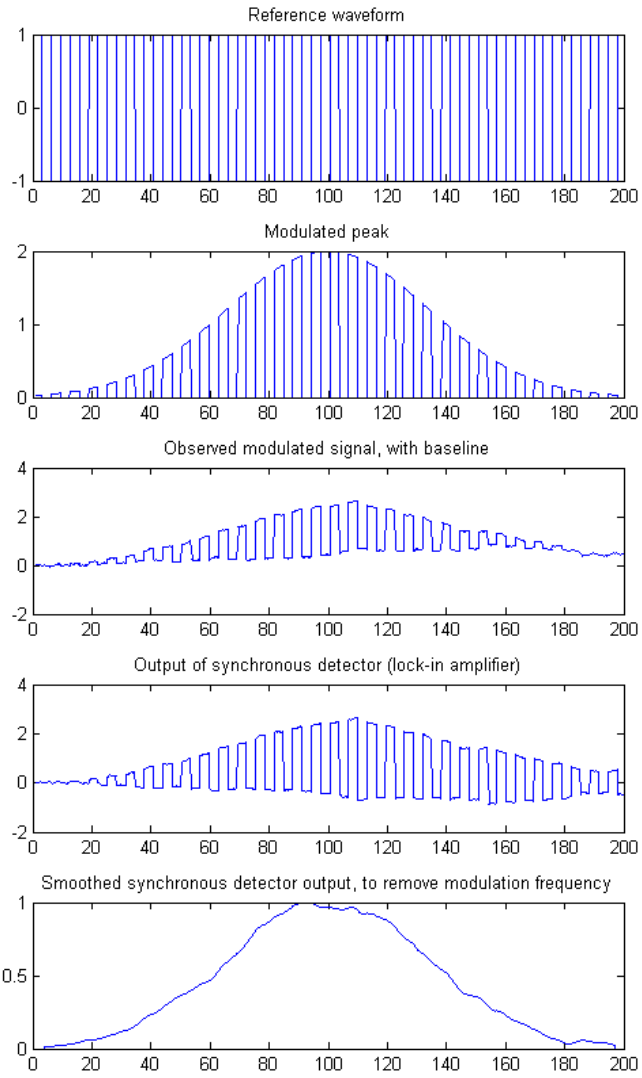
You can compare this to WhiteNoiseBaseline.m which has a similar signal and S/N ratio, except that the noise is *white*. Interestingly, the *fitting error* with white noise is *greater*, but the *parameter errors* (peak position, height, width, and area) are *lower,* and the residuals are more random and less likely to produce false noise peaks. This is because the random walk noise is very highly concentrated at low frequencies where the signal frequencies usually lie, whereas white noise also has considerable power at *higher frequencies*, which *increases the fitting error* but does comparatively little damage to signal measurement accuracy. This may be counter-intuitive, but it is important to realize that fitting error does not *always* correlate with peak parameter error. Bottom line: the random walk is troublesome.

Depending on the type of experiment, an instrumental design based on modulation techniques may help, and ensemble averaging multiple measurements can help with any type of unpredictable random noise, which is discussed in the very next section.

Modulation and synchronous detection.

In some experimental designs it may be beneficial to apply the technique of modulation, in which one of the controlled independent variables is oscillated in a periodic fashion, and then detecting the resulting oscillation in the measured signal. Such an instrumental design can reduce or eliminate some types of noise and drift.

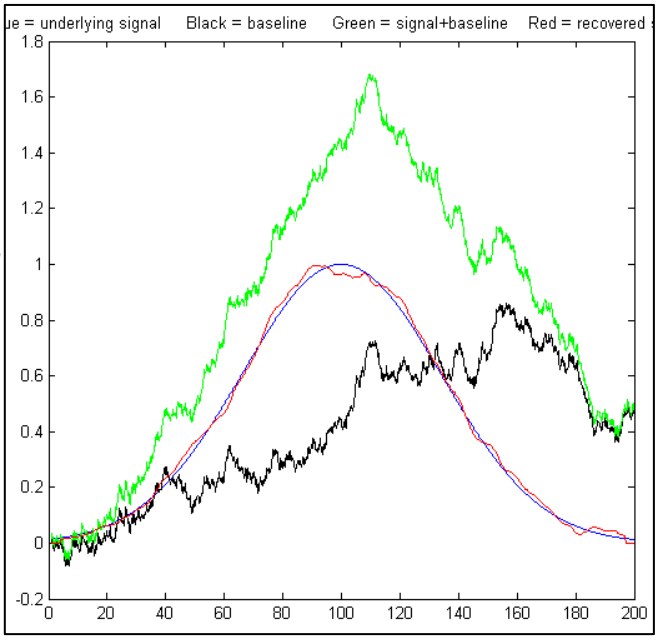




A simple example is the optical measurement systems pictured above. A light source illuminates a test object (DUT = "Device Under Test") and the resulting light from the test object is measured by the photodetector. Depending on the objective of the experiment and the arrangement of the parts, the detector might measure the light *transmitted by*, *reflected by*, *scattered by*, or *excited by* the light beam. A spinning optical chopper rapidly and repeatedly interrupts the light beam falling on the test object so that the photodetector sees an oscillating signal, and the following electronic system is designed to measure *only* the oscillating component and to ignore the constant component. The advantage of this arrangement is that any interfering signals introduced *after* the chopper - such as constant light emitted by the test object itself, or ambient light that leaks in from the outside, or any constant background signal generated by the photodetector itself - *are not oscillating in sync with the chopper* and are thus rejected. This works best if the electronics are *synchronized* to the chopper frequency. That's exactly the function of the lock-in amplifier, an electronic circuit that receives a synchronizing reference signal directly from the chopper to guarantee synchronization even if the chopper frequency were to vary. The lock-in amplifier is sometimes viewed as a "black box" with seemingly magical abilities, but in fact, it is performing a rather simple (but very useful) operation, as shown in this simulation and by the interactive simulation on this link and this link and described in T. C. O'Haver, "Lock-in Amplifiers," J. Chem. Ed. 49, March and April (1972).

AmplitudeModulation.m is a Matlab/Octave script that simulates modulation and synchronous detection, in which the signal created when the light beam scans the test sample is modeled as a Gaussian band ('y'), whose parameters are defined in the first few lines of the script. The graph on the previous page compares the signal at the different points in the apparatus. As the wavelength of the light beam hitting the test object is slowly scanned, the light beam is periodically interrupted by the spinning chopper, represented as a square wave defined by the bipolar vector 'reference', which switches between +1 and -1, shown in the top panel on the left. This is called *amplitude modulation*. The modulation frequency is many times faster than the rate at which the wavelength is scanned. The light emerging from the sample, therefore, shows a finely chopped Gaussian ('my'), shown in the second panel on the left. But the *total signal* seen by the detector also includes an unstable background introduced *after* the modulation ('omy'), such as lighted emitted by the sample itself or detector background, which in this simulation this is modeled as a random walk (page 318), which seriously distorts the signal, shown in the third panel. The detector signal is then sent to a lock-in amplifier that is synchronized to the reference waveform. The essential action of the lock-in is to multiply the signal by the bipolar reference waveform, *inverting the signal* when the light is *off* and *passing it unchanged* when the light is *on*. This causes the unmodulated background signal to be converted into a bipolar square wave, whereas the modulated signal is not affected because it is "off" when the reference signal is negative. The result ('dy') is shown in the 4th panel. Finally, this signal is low-passed filtered by the last stage in the lock-in amplifier to remove the modulation frequency, resulting in the recovered signal peak 'sdy' shown in the bottom panel. In effect, the modulation transforms the signal to a higher frequency ('frequency' in line 44) where low-frequency weighted noise on the baseline (line 50) is less intense.

These various signals are compared in the figure on the right. The underlying Gaussian signal peak ('y') is shown as the **blue line**, and the contaminating background ('baseline') is shown in **black**, in this case, modeled as a random walk. The total signal ('oy') that would have been seen by the detector if modulation were not used is shown in **green**. The signal distortion is evident, and any attempt to measure the signal peak in that signal would be greatly in error. The signal 'sdy' recovered by the modulation and lock-in system is shown in **red** and overlaid with the original signal peak 'y' in blue for comparison. The fact that the blue and red lines on the graph are so close to each other indicates the extent to which this method is successful.



To make a more quantitative comparison, this script also uses the peakfit.m function, which employs a least-squares method to measure the peak parameters in the original unmodulated total signal (green line) and in the modulated recovered signal (blue) and to compute the relative percent error in peak position, width, and width by both methods:

SignalToNoiseRatio = 4

Relative % Error:	**Position**	**Height**	**Width**
Original:	8.07	23.1	13.7
Modulated:	0.11	0.22	1.01

Each time you run it you will get the *same signal peak* but a very *different* random walk background. The S/N ratio will vary from about 4 to 9. It is not uncommon to see a *100-fold improvement* in peak height accuracy with modulation, as in the example shown here. (If you wish, you can change the signal peak parameters and the noise level in the first few code lines of this simulation. For an even greater challenge, change line 47 to "`baseline=10.*noise + cumsum(noise);`" to make the noise a mixture of white and random walk drift, which results in a really ugly raw signal. You can see that the white noise makes it through the synchronous detector but is reduced by the smoothing low pass filter in the last stage). In effect, the low-pass filter determines the frequency bandwidth of the lock-in system, but it also increases the response time to step changes (as in the Morse Code example).

This improvement in measurement accuracy works only because the dominant random error, in this case, is:

(a) introduced *after* the modulation, and
(b) a mostly *low*-frequency noise.

If the noise were *white*, there would be *no* improvement, because white noise is the same at all frequencies. In fact, there would be a slight reduction in precision because the chopper blocks half of the light on average. If the sample (device under test) generates an "absorption" peak that starts a some positive value and then dips down to a lower value before returning, the demodulated output will be a negative-going peak rather than a positive peak (see AmplitudeModulationAbsorption.m).

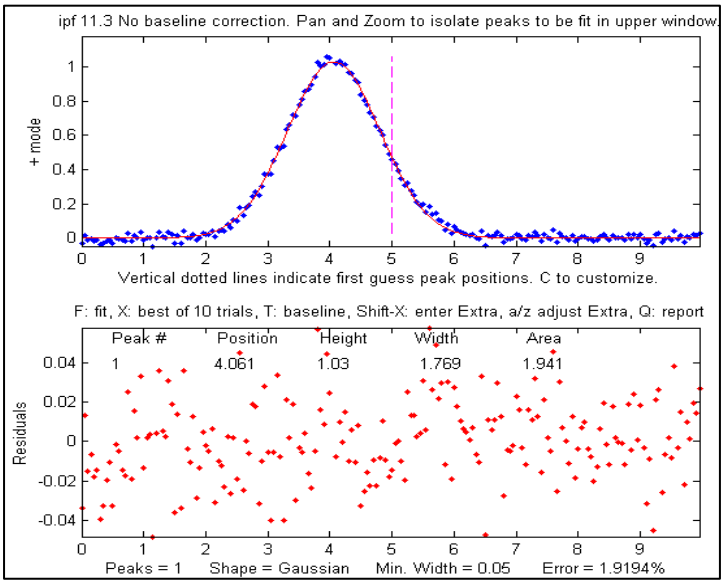
In a computer-interfaced experimental system, you may not actually need a physical lock-in amplifier. It is possible to simulate the effect in software, as is done in this numerical experiment. You would need two input channels of the analog-to-digital converter, one for the modulated sample signal and one for the modulation reference signal, and then, in software, invert the total modulated sample signal whenever the reference signal is "off".

In some spectroscopic applications another useful type of modulation is “wavelength modulation”, in which the *wavelength* of the light source oscillates over the wavelength region of an emission or absorption peak in the spectrum (reference 32). This has often been used in atomic emission and absorption spectroscopy (references 25, 26) and in tunable diode laser spectroscopy applied to the measurement of gases such as methane, water vapor, and carbon dioxide, especially in remote sensing, where the sample may be far from the detector. Less commonly, modulation techniques are also applied in “AC” (alternating current) electrochemistry and in spectroelectrochemistry.

Measuring a buried peak

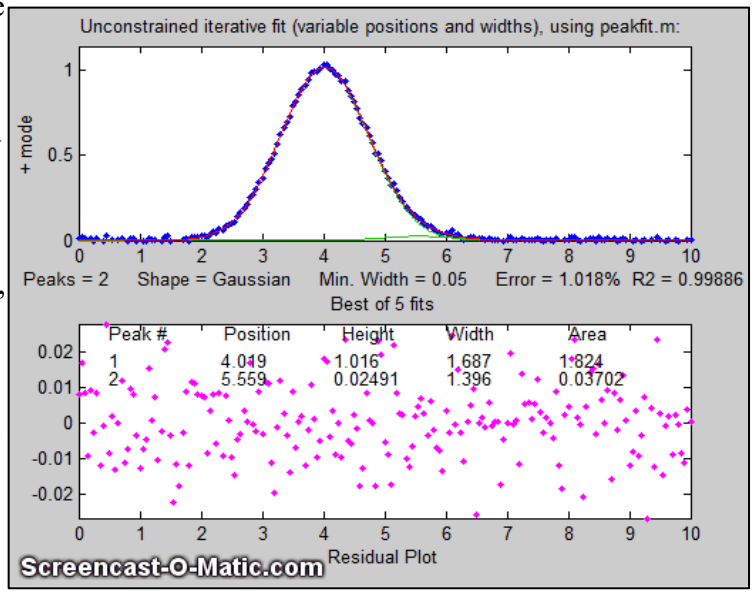
This simulation explores the problem of measuring the height of a small peak (a "child peak") that is buried in the tail of a much stronger overlapping peak (a "parent peak"), in the especially challenging case that the smaller peak *is not even visible* to the unaided eye. Three different measurement tools will be explored: iterative least-squares, classical least-squares regression, and peak detection, using the Matlab/Octave tools peakfit.m, cls.m, or findpeaksG.m, respectively. (Alternatively, you could use the corresponding spreadsheet templates).

In this example the larger peak is located at x=4 and has a height of 1.0 and a width of 1.66. The smaller measured peak is located at x=5 and has a height of 0.1. Both have a width of 1.66. Of course, for the purposes of this simulation, you pretend that you do not necessarily know all these facts and you will try to find methods that will extract as much information as possible from the data, even if the signal is noisy. The measured peak is small enough and close enough to the stronger overlapping peak (separated by less than the width of the peaks) that it *never forms a maximum* in the total signal. So, it *looks* like there is only *one* peak, as shown on the figure on the right. For that reason, the findpeaks.m function (which automatically finds peak maxima) will not be useful by itself to locate the smaller peak. Simpler methods for detecting the second peak also fail to provide a way to measure the smaller second peak, such as inspecting the derivatives of the signal (the smoothed fourth derivative shows some evidence of asymmetry, but that could just be due to the shape of the larger peak), or Fourier self-deconvolution to narrow the peaks so they are distinguishable, but that is unlikely to be successful with this much noise. Least-squares methods work better when the signal-to-noise ratio is poor, and they can be fine-tuned to make use of available information or constraints, as will be demonstrated below.

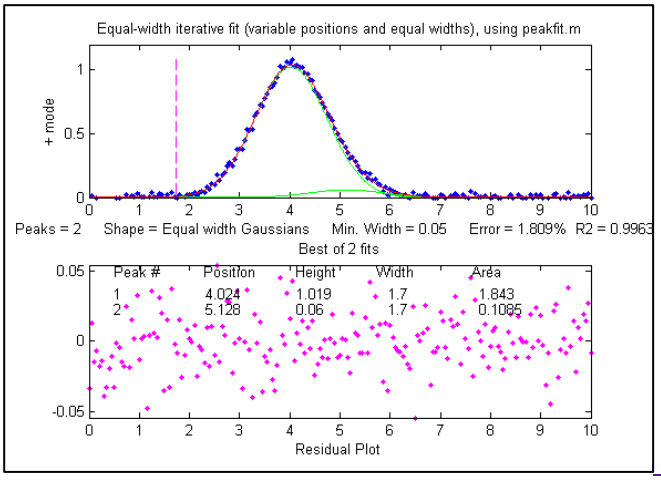


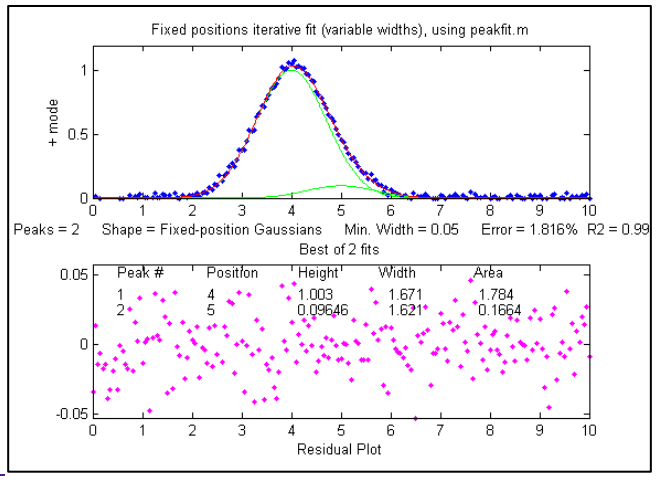
The selection of the best method will depend on what is known about the signal and the constraints that can be imposed. This will depend on your knowledge of your experimental signal. In this simulation (performed by the Matlab/Octave script SmallPeak.m), the signal is composed of two Gaussian peaks (although that can be changed if desired in line 26). The first question is: is there more than one peak there? If you perform an unconstrained iterative fit of a *single* Gaussian to the data, as shown in the figure on the right, it shows little or no evidence of a second peak - the residuals look random. (If you could reduce the noise, or if you ensemble-averaged even as few as 10 repeat signals, then the noise would be low enough to see evidence of a second peak). However, as it is, there is nothing that pops out at you suggesting a second peak.

But suppose you suspect that there *should* be another peak of the same Gaussian shape just on the right side of the larger peak. You can try fitting a *pair* of Gaussians to the data (figure on the right), but in this case, *there is so much random noise that the fit is not stable.* When you run SmallPeak.m, the script performs 20 repeat fits ("NumSignals" in line 20) with the same underlying peaks but with 20 different random noise samples, revealing the stability (or instability) of each measurement method. The fitted peaks in Figure window 1 bounce around all over the place as the script runs. (If the animation is not visible, click this link). The fitting error is on average *lower* than the single-Gaussian fit, but that by itself does not mean that the peak parameters so measured will be reliable. It could just be "fitting the noise". *If it were isolated all by itself*, the small peak would have an S/N ratio of about 5 and it could be measured to a peak height precision of about 3%, but the presence of the larger interfering peak makes the measurement much more difficult. (Hint: After running SmallPeak.m the first time, spread out all the figure windows so they can all be seen separately and do not overlap. That way you can compare the stability of the different methods more easily.)

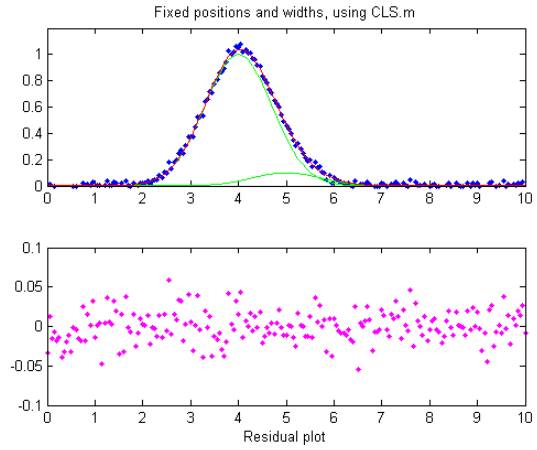


But suppose that you have reason to expect that the *two peaks will have the same width*, but you do not know what that width might be. You could try an *equal width* Gaussian fit (peak shape #6, shown in Matlab/Octave Figure window 2). The resulting fit is much more stable and shows that a small peak is located at about x=5 on the right of the bigger peak, shown below on the left. On the other hand, if you know the peak *positions* beforehand, but not the widths, you can use a *fixed-position* Gaussian fit (shape #16) shown on the right (Figure window 3). In the very common situation where the objective is to measure an *unknown concentration* of a *known* component, then it is possible to prepare standard samples where the concentration of the sought component is high enough for its position or width to be determined with certainty.





So far all these examples have used iterative peak fitting with at least one peak parameter (position and/or width) unknown and determined by measurement. If, on the other hand, *all* the peak parameters are known *except* the peak height, then the faster and more direct classical least-squares regression (CLS) can be employed (Figure window 4). In this case, you need to know the peak position and width of both the measured and the larger interfering peaks (the computer will calculate their heights). If the positions and the heights really are constant and known, then this method gives the best stability and precision of measurement. It is also computationally faster, which might be important if you have lots of data to process automatically.



The problem with CLS is that it fails to give accurate measurements if the peak position and/or width changes without warning, whereas two of the iterative methods (unconstrained Gaussian and equal-width Gaussian fits) can adapt to such changes. It some experiments it quite common to have small, unexpected shifts in the peak position, especially in chromatography or other flow-based measurements, caused by unexpected changes in temperature, pressure, flow rate or other instrumental factors. In SmallPeaks.m, such x-axis shifts can be simulated using the variable "xshift" in line 18. It is initially zero, but if you set it to something greater (e.g., 0.2) you will find that the equal-width Gaussian fit (Figure window 2) works better because it can keep up with the changes in x-axis shifts.

But with a greater x-axis shift (xshift=1.0) even the equal-width fit has trouble. Still, if you know the *separation* between the two peaks, it is possible to use the findpeaksG function to search for and locate the *larger* peak and to use that to calculate the position of the *smaller* peak. Then the CLS method, with the peak positions so determined for each separate signal, shown in Figure window 5 and labeled "findpeaksP" in the table below, works better. Alternatively, another way to use the findpeaks results is a variation of the equal-width iterative fitting method in which the first guess peak positions (line 82) are derived from the findpeaks results, shown in Figure window 6 and labeled "findpeaksP2" in the table below. That method does not depend on accurate knowledge of the peak widths, only their equality.

Each time you run SmallPeaks.m, *all these methods are computed* multiple times ("NumSignals", set in line 20) and compared in a table giving the average peak height accuracy of all the repeat runs:

```
xshift=0
Unconstr.  EqualW   FixedP  FixedP&W  findpeaksP  findpeaksP2
35.607     16.849   5.1375   4.4437    13.384      16.849

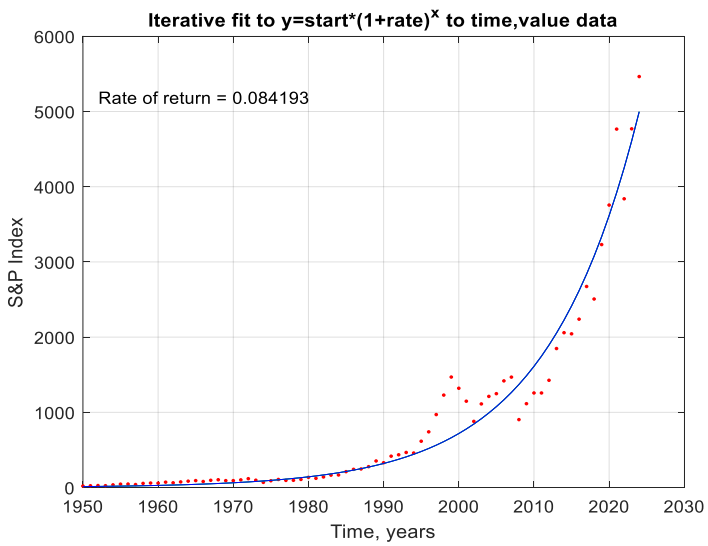
xshift=1
Unconstr.  EqualW   FixedP  FixedP&W  findpeaksP  findpeaksP2
31.263     44.107   22.794   46.18     10.607      10.808
```

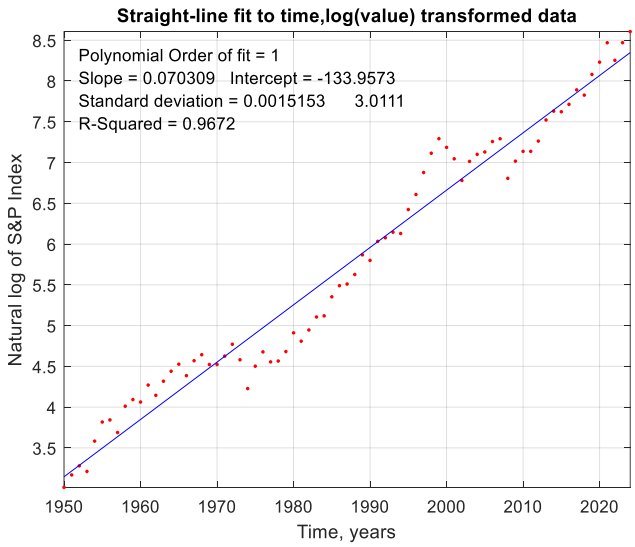
The bottom line is this: the more you know about your signals, the better you can measure them. A stable signal with *known* peak positions and widths is the most precisely measurable in the presence of random noise ("FixedP&W"), but if the positions or widths vary from measurement to measurement, different methods must be used, and precision is degraded because more of the available information is used to account for changes *other* than the ones you want to measure.

Signal and Noise in the Stock Market (updated)

From a signal-to-noise perspective, the stock market is an interesting example. A national or global stock market is an aggregation of large numbers of buyers and sellers of shares in publicly traded companies. They are described by stock market *indexes*, which are computed as the weighted average of many selected stocks. For example, the S&P 500 index is computed from the stock valuations of 500 large US companies. Millions of individuals and organizations participate in the buying and selling of stocks daily, so the S&P 500 index is a prototypical "big data" conglomerate, reflecting the overall value of 500 of the largest companies in the largest stock market on earth. Individual stocks can fail or fall drastically in value, but the market indexes average out the performance of hundreds of companies.

A plot of the yearly value, V, of the S&P 500 index vs time, T, for the 75-year period from 1950 to 2024 is shown in the following graphs.





Each plot contains 75 data points, one for each year, shown in red. The graph on the left plots the value V on linear coordinates and the graph on the right plots the *natural logarithm* of V, ln(V). There are considerable up-and-down fluctuations in the value over time that can be related to historical events: the oil crisis of the 1970s, the tech boom and bust of 2000, the subprime mortgage crisis of 2008, the trade wars of 2019 and the Coronavirus pandemic of 2020. Still, the *long-term* trend of the value is upwards – by 2025 the value was *over 300 times greater* than its value in 1950. This is basically why people invest in the stock market, because on average, *over the long run*, stock values eventually go up, almost always more than inflation (which has been averaging about 3.5% per year since 1950).

The most common way to model this overall long-term increase with time is based on the equation for compound interest that predicts the growth of investments that have a constant rate of return, such as savings accounts or certificates of deposit:

$$V = S*(1 + R)^T$$

where V is the value, S is the starting value, R is the annual rate of return, and T is time. By itself, this expression would yield a smooth upward curving exponential curve, without any peaks and dips. You can use curve fitting (pages 162 and 199) to determine how well this model fits the actual S&P. This can be done in two ways:

(1) directly, using the iterative curve fitting method, shown on the left above, or
(2) by taking the logarithm of the values and fitting a *straight line* to the transformed data, shown on the right above.

The best fits to the S&P data are shown in the graphs by the smooth blue lines. FitSandPto2024.m is a Matlab/Octave script that performs both calculations using the data in SandPfrom1950.mat or in SandPfrom1950.xlsx. When applied to the S&P 500 index data, the annual rate of return R is about 0.08 (or 8%), which is much greater than the 3.5% rate of inflation.

But interestingly these two methods give slightly *different results*, even though the *exact same data* are used for both. Moreover, *both* methods yield the *same* rate if they are applied to noiseless *synthetic data* calculated from the above expression. How can this be? This is caused by the *irregularities* in the stock data that deviate from a smooth line - in other words, the *noise* - and it is exacerbated by the large range of V over time (and by the fact that the average return from 1950 to 1983 is lower than that from 1983 to 2024).

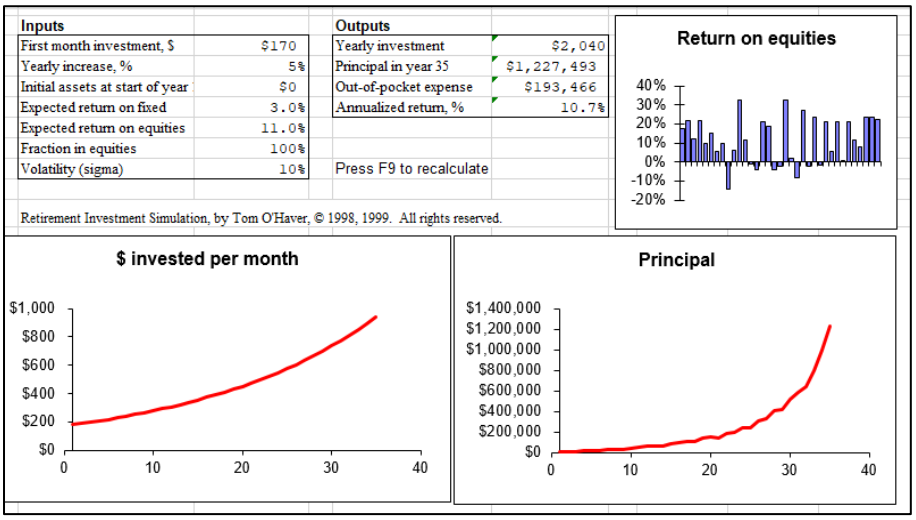
How good are those data at *predicting* the stock market trends? Even over the short term, such predictions are often not very accurate. For example, the trend line (blue line) in the left-hand plot on the previous page predicts that the value of the S&P by the end of 2024 should be 5000. In fact, the value was close to 6000, or about 20% higher than the prediction of the least-squares fit. By late 2025, its value had risen *to over 6800!* But this is not out of the ordinary. A glance at the graphs above will show that individual values have temporarily fallen at least that far above (or below) the best-fit line.

From the point of view of curve fitting, the deviation from a smooth curve described by the compound interest expression is just *noise*. But from the point of view of the stock market investor, those deviations can be both an *opportunity* and a *warning*. Naturally, most investors would like to know how

the stock market will behave in the future, but that requires *extrapolation beyond the range of the available data*, which is always uncertain and dangerous. But still, it is *most likely* (but not certain) that the *long-term* behavior of the market (say, over a period of 10 years or more) will be like the past - that is, growing exponentially at about the same rate as before but with unpredictable fluctuations similar to what has occurred in the past.

There are several notable features of this "noise". First, the *deviations are roughly proportional to* V and thus relatively equal when plotted on a log scale. Second, there are numerous instances over the years when there is a sharp dip followed by a slower recovery close to the previous value. And conversely, every peak is eventually followed by a dip. The conventional advice in investing is to "buy low" (on the dips) and "sell high" (on the peaks). But of course, the problem is that you cannot reliably determine *in advance* exactly where the peaks and dips will fall. You have only the past to guide you. Still, if the current market value is much *higher* than the long-term trend, it will likely fall, and if the market value is much *lower* than the long-term trend, it will likely rise, eventually. The only thing you can predict is that, in the long run, the market will eventually rise. This is the reason that it is so important to save for retirement by investing in the stock market, *starting as soon as possible*: over a 30-year working life, the market is essentially guaranteed to rise substantially. The most painless way to do this is with your employer's 401k or 403b automatic payroll withdrawal plan. You cannot invest in the stock market *as a whole*, but you *can* invest in *index mutual funds* or *exchange-traded funds* (ETFs), which are collections of stocks that are constructed to match or track the components of a market index. Such funds typically have *very low management fees*, an important factor in selecting an investment. Other mutual funds attempt to “beat the market” by carefully buying and selling stocks to create a return that is greater than the overall market indexes. Some are *temporarily* successful in doing that, but they charge higher management fees. Mutual funds and ETFs are much less risky investments than individual stocks. "Day traders", investors who buy and sell stocks and other securities multiple times over a single day, often do not perform well, because the market is propelled in the long run by business cycles, new businesses, and new technologies that do not change over a single day. Minute-to-minute changes are mostly noise.

Some companies periodically distribute payouts to investors called “dividends”. Those dividends are independent of the day-to-day variations in stock price, so even if the stock value drops temporarily, you still get the same dividend. For that reason, it is important that you set your investment account to “automatically reinvest dividends”, so when the share price drops, the dividends are buying shares at the *lower price*. The S&P 500 index values used above, called *price returns*, did *not* include dividend reinvestment. If you calculated the *total returns* with dividends reinvested, the returns would have



been substantially higher, closer to 11% (https://en.wikipedia.org/wiki/S%26P_500_Index#Versions). With an average total annual return of 11% (assuming dividend reinvestment), and starting with an investment of $170 the first month - that's less than $6 a day - and increasing that amount 5% each year, you could accumulate over $600,000 over a 30-year working life, or $1,000,000 if you continued investing an additional 5 years *or if you began 5 years earlier*, as shown by the spreadsheet graphic above. You might call this a "get rich slow" scheme. And that is with starting at just *$6 per day*, about the cost of a fancy coffee at Starbucks. Think about that the next time you see a line of young people waiting to order their daily coffee. The hard part is not so much giving up the coffee as is *finding a keeping a steady job* that allows you to make routine automatic contributions to your retirement account over the long haul. Becoming a millionaire by the time you retire is very possible, but it's not exciting. Rather, it is slow and plodding.

To illustrate how much influence stock market volatility fluctuation ("noise") has on the market gains, the Matlab/Octave script SnPsimulation.m adds proportional noise (page 27) to the compound interest calculation to mimic the S&P data, performs the two curve fitting methods described above, repeats the allocations over and over with independent samples of proportional noise, and then calculates the mean and the relative standard deviation (RSD) of the rates of return. A typical result is:

```
TrueRateOfReturn = 0.08

                          Measured Rate   RSD
Coordinate transformation:    0.078        3%
Iterative curve fitting:      0.077        6%
```

As you can see, the two methods do not agree perfectly. The return calculated by the iterative method is lower in this case, but it *could just have easily been the other way*. The fact is that the standard deviations are large, and the iterative method always has a higher standard deviation**,** because it weighs the higher values more heavily, where deviations from the line are higher, whereas the log transformation method weights the data more evenly. Even with this uncertainty, investing in a stock market index fund almost always performs better *in the long run* than more predictable investments such as savings accounts or certificates of deposit (CDs), which have much lower rates of return. When stocks drop, even for well-known reasons, some investors buy shares at reduced prices, and when

stocks rise, especially when they hit all-time highs, some investors sell shares to "lock in their gains". This behavior has been consistent throughout the decades and acts as a natural brake on the fluctuations of the market.

In investing in the stock market, it is important to focus on the long-term trends and not to be frightened by the short-term up and down fluctuations, even though most of the news coverage understandably emphasizes the short-term. This is similar to distinguishing between weather and climate. The large and dramatic short-term *weather* variations are newsworthy and tend to disguise the much smaller long term *climate* warming that is slowly melting the icecaps and raising the sea levels (whether it is caused by human activity or by natural causes alone or by a combination of both). Everyone talks about changes in the weather, but the climate changes so slowly that it is easy to conclude that it stays the same. The hour hand on the clock is never seen to move. If you are young and have many years ahead, keep your investment in stock funds, which have the best returns. As you get older, you can gradually shift to lower risk but lower return investments, such as high-yield savings accounts, certificates of deposit (CDs), money market accounts, Treasury securities and other bond funds. Stocks perform better because they profit from advances in infrastructure and technology.

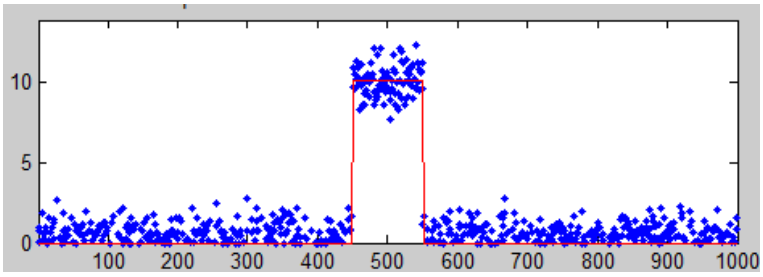
For a spreadsheet template that allows you to calculate the possible returns on long-term investments in stock market mutual funds, see https://terpconnect.umd.edu/~toh/simulations/Investment.html.

Measuring signal-to-noise ratio in complex signals

In the section "Signals and Noise" on page 24, I said: "The quality of a signal is often expressed as the signal-to-noise (S/N) ratio, which is the ratio of the true signal amplitude ... to the standard deviation of the noise." That is a simple enough statement but automating the measurement of signal and the noise in real signals is not always straightforward. Sometimes it is difficult to separate or distinguish between the signal and the noise, because it depends not only on the numerical nature of the data, but also on the objectives of the measurement.

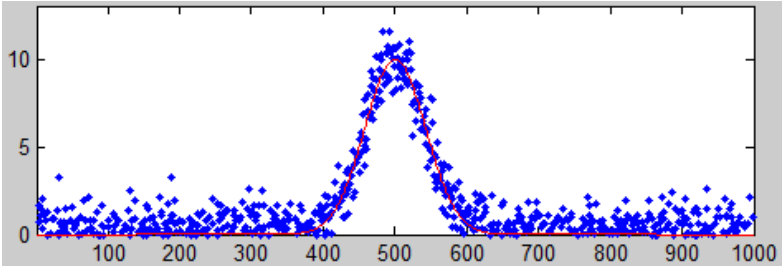
For a simple DC (direct current) signal, for example, measuring a fluctuating voltage, the signal is just the average voltage value and the noise is its standard deviation. This is easily calculated in a spreadsheet or in Matlab/Octave:

```
>> signal=mean(NoisyVoltage);
>> noise=std(NoisyVoltage);
>> SignalToNoiseRatio=signal/noise
```

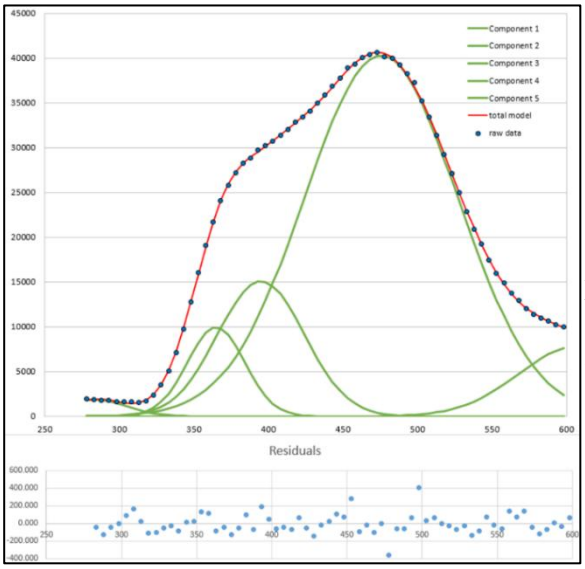


But usually, things are more complicated. For example, if the signal is a rectangular pulse (as in the figure on the right) with constant random noise, then the simple formulation above will not give accurate results. If the signal is stable enough that you can get two successive signal recordings *m1* and *m2* that are *identical except for the noise*, then you can *simply subtract the signal out*: the standard deviation of the noise is then given by sqrt((std(*m1*-*m2*)^2)/2), where "std" is the standard deviation function (because random noise adds quadratically). But not every signal source is

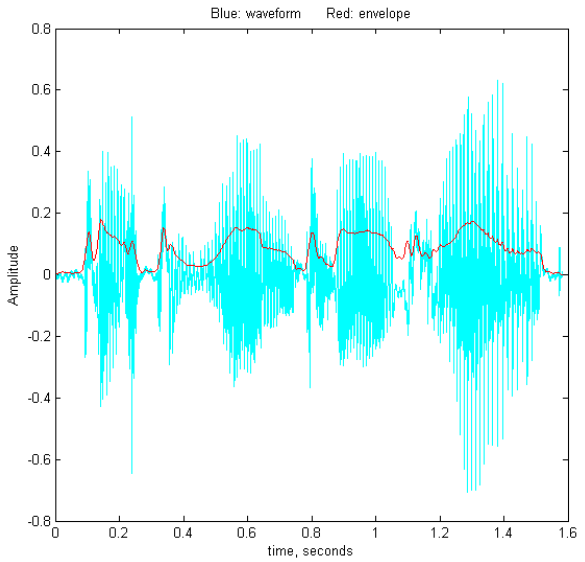
stable and repeatable enough for that to work perfectly. Alternatively, you can try to measure the average signal just over the top of the pulse and the noise only over the baseline interval before and/or after the pulse. That is not so hard to do by hand, but it is harder to automate with a computer, especially if the position or width of the pulse changes. It is basically the same for smooth peak shapes like the commonly encountered Gaussian peak (as in the figure on the right). You can estimate the height of the peak by smoothing it and then taking the maximum of the smoothed peak as the signal: **`max(fastsmooth(y,10,3))`**, but the accuracy would degrade if you chose too high or too low a smooth width. And clearly, all this depends on having a well-defined baseline in the data where there is only noise. It does not work if the noise varies with the amplitude of the peak.



In many cases, curve fitting can be helpful. For example, you could use peak fitting or a peak detector to locate multiple peaks and measure their peak heights and their S/N ratios on a peak-to-peak basis, by computing the noise as the standard deviation of the difference between the raw data and the best-fit line over the top part of the peak. That is how iSignal (page 376) measures S/N ratios of peaks. Also, iSignal has baseline correction capabilities that allow the peak to be measured relative to the baseline.



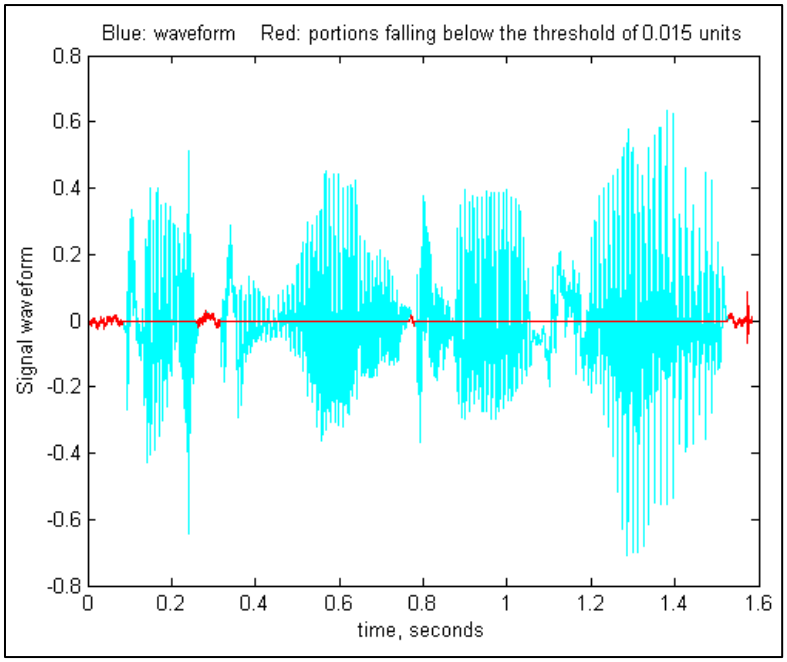
Curve fitting also works for complex signals of indeterminate shape that can be approximated by a high-order polynomial or as the sum of a number of basic functions such as Gaussians, as in the real-data example shown on the left. In this case, an increasing number of Gaussians are used to fit an experimental data set, to the point where the residuals are random and unstructured. The residuals (shown as blue dots below the graph) are then just the noise remaining in the signal, whose standard deviation is easily computed using the built-in standard deviation function in a spreadsheet ("STDEV") or in Matlab/Octave ("std"). In this example, the standard deviation of the residuals is 111 and the maximum signal is 40748, so the percent relative standard deviation of the noise is 0.27% and the S/N ratio is 367. (The positions, heights, and widths of the Gaussian components, usually the main results of the curve fitting, are not used in this case. Curve fitting is used only to obtain a measure the noise via the residuals). The advantage of this approach over simply subtracting two successive measurements of the signal is that it adjusts for slight changes in the signal from measurement to measurement. The only assumption is that the signal is a smooth, low-frequency waveform that can be fitted with a polynomial or a collection of basic peak shapes and that the noise is random and mostly high-frequency compared to the signal. But do not use too high a polynomial order or too many model peaks, otherwise you are just "fitting the noise".



With periodic signal waveforms, the situation is a bit more complicated. As an example, consider the audio recording of the spoken phrase "Testing, one, two, three" (click to download in .mat format or in WAV format). The Matlab/Octave script PeriodicSignalSNR.m loads the audio file into the vector variable named "waveform", then computes the average amplitude of the waveform (the "envelope") by smoothing (page 42) the absolute value of the waveform:

```
envelope = fastsmooth(abs(waveform), SmoothWidth, SmoothType);
```

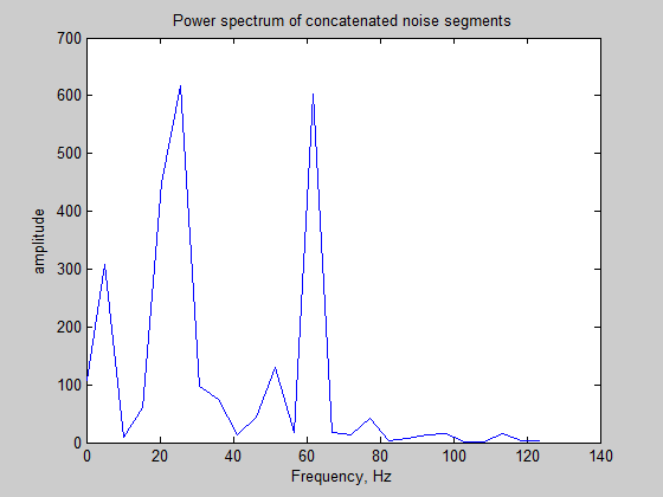
The result is plotted on the left, where the waveform is in blue and the envelope is in red. The signal is easy to measure as the maximum or perhaps the average of the waveform, but the noise is not so evident. The human voice is not reproducible enough to get a second identical recording to subtract out the signal. Still, there will often be gaps in the sound, during which the background noise will be dominant. In an audio (voice or music) recording, there will typically be such gaps at the beginning, when the recording process has already started but the sound has not yet begun, and possibly at other short periods when there are pauses in the sound. The idea is that, by monitoring the envelope of the sound and noting when it falls below some adjustable threshold value, you can automatically record the noise that occurs in those gaps, whenever they may occur in a recording.



In PeriodicSignalSNR.m, this operation is done in lines 26-32, and the threshold is set in line 12. The threshold value must be optimized for each recording. When the threshold value is set to 0.015 in the "Testing, one, two, three" recording, the resulting noise segments are located and marked in red in the plot on the right. The program determines the average noise level in this recording simply by computing the standard deviation of those segments (line 46), then computes and prints out the peak-to-peak S/N ratio and the RMS (root mean square) S/N ratio.

```
PeakToPeak_SignalToNoiseRatio = 143.7296
RMS_SignalToNoiseRatio = 12.7966
```

The frequency distribution of the noise is also determined (lines 60-61) and shown in the figure on the left, using the PlotFrequencySpectrum function, or you could have used iSignal (page 376) in the frequency spectrum mode (**Shift-S**). The spectrum of the noise shows a strong component very near 60 Hz, which is almost certainly due to *power line pickup* (the recording was made in the USA, where AC power is 60 Hz). This suggests that better shielding and grounding of the electronics might help to clean up future recordings. The peak at 20 Hz is harder to attribute: perhaps it is the low hum of a fan or an air conditioner. The lack of strong components at 100 Hz and above (where the human voice occurs) suggests that the vocal sounds have been effectively cut out at this amplitude threshold setting. The script can be applied to other sound recordings in WAV format simply by changing the file name and time axis in lines 8 and 9.



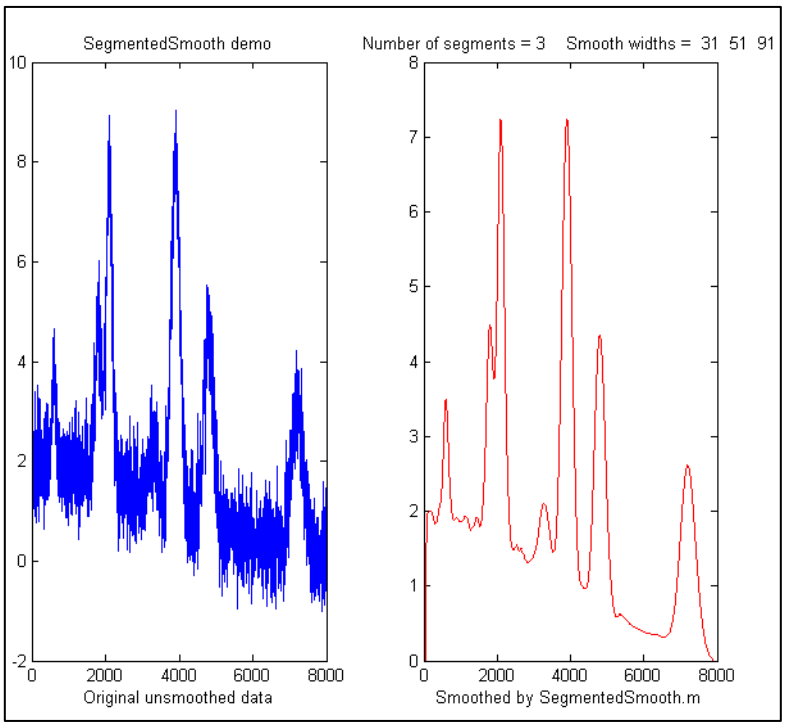
Dealing with wide-ranging signals: segmented processing

To facilitate the inspection of very large and complex signals, it is useful to be able to "zoom in" to different parts of the x-axis range, which can be done with the interactive Matlab tools *iSignal* (page 376), *iPeak* (page 255), and *ipf.m* (page 417) or by the Matlab/Octave function segplot.m (page 482). Sometimes an experimental signal will vary so much across its x-axis range that it is impossible to find a single setting for operations like smoothing or peak detection that is optimized for all regions of the signal. It is always possible to break up the signal into pieces and treat each separately, for example using segplot, but in some cases, it is easier to use a single *segmented* application over the entire signal. That's the idea behind the Matlab/Octave functions and the Excel spreadsheet templates in this section.

SegmentedSmooth.m, illustrated on the right on the next page, is a segmented variant of fastsmooth.m, which can be useful if the widths of the peaks or the noise level varies substantially across the signal. The syntax is that same as fastsmooth.m, except that the second input argument "smoothwidths" can be a *vector*:

```
[SmoothedSignal,SmoothMatrix] = SegmentedSmooth (Y, smoothwidths, type, ends).
```

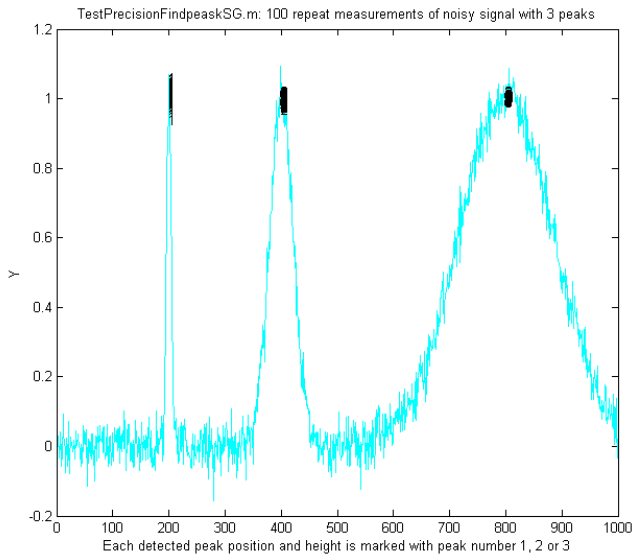
The function divides Y into several equal-length regions defined by the length of the vector 'smoothwidths', then smooths each region with a smooth of type 'type' and width defined by the elements of vector 'smoothwidths'. In the simple example in the figure on the right, `smoothwidths = [31 52 91]`, which divides up the signal into three regions and smooths the first region with smoothwidth 31, the second with 51, and the last with 91. *Any number and sequence of smooth widths can be used.* It optionally returns 'SmoothMatrix' consisting of all segments assembled into a matrix. Type "help SegmentedSmooth" for other examples.



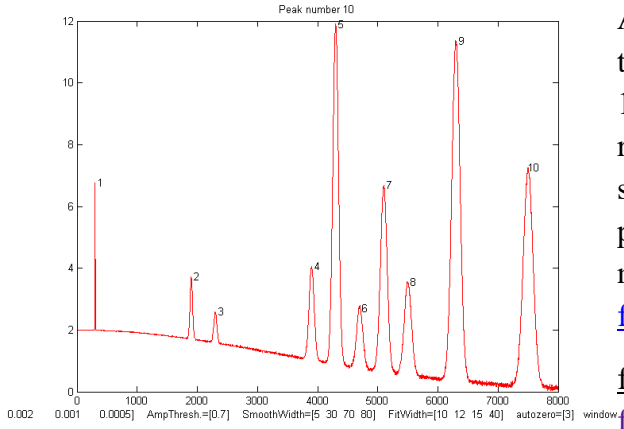
DemoSegmentedSmooth.m demonstrates the operation with different signals consisting of noisy variable-width peaks that get progressively wider, like the figure on the right. FindpeaksSL.m is the same thing for Lorentzian peaks.

Spreadsheet version. SegmentedSmoothTemplate.xlsx is a segmented multiple-width data smoothing spreadsheet template, which is functionally like SegmentedSmooth.m. In this version, there are 20 segments. SegmentedSmoothExample.xlsx is an example spreadsheet with data (graphic). A related spreadsheet GradientSmoothTemplate.xlsx (graphic) performs a linearly increasing (or decreasing) smooth width across the entire signal, given only the start and end values, automatically generating as many segments are necessary. Of course, as is usual with spreadsheets, you will have to modify these templates for your number of data points, usually by inserting rows somewhere in the middle and then drag-copying down from above the insert, plus you may have to change the x-axis range of the graph. (In contrast, the Matlab/Octave functions do that automatically).

SegmentedFouFilter.m is a segmented version of FouFilter.m that applies different center frequencies and widths to different segments of the signal. The syntax, **`[ffSignal,ffMatrix] = SegmentedFouFilter(y,samplingtime,centerFrequency,filterWidth,shape,mode),`** is like FouFilter.m except that the two input arguments centerFrequency and filterWidth must be vectors with the values of centerFrequency of filterWidth for each segment. The signal is divided equally into several segments determined by the length of centerFrequency and filterWidth, which must be equal in length. Optionally returns ffMatrix of all segments assembled into a matrix. Type help SegmentedFouFilter for help and examples. The figure on the left shows Example 2. It may help to visualize the signal by using a related function, PlotSegFreqSpect.m, which creates and displays a *time-segmented* Fourier power spectrum (see page 103 and following).



findpeaksSG.m is a variant of the findpeaksG function, with the same syntax, except that the four peak detection parameters can be *vectors*, dividing up the signal into regions that are optimized for peaks of different widths. Any number of segments can be declared, based on the length of the SlopeThreshold input argument. (Note: you need only enter vectors for those parameters that you want to vary between segments. To allow any of the other peak detection parameters to remain unchanged across all segments, simply enter a single scalar value for that parameter. Only the SlopeThreshold must be a vector). In the example shown on the left, the script TestPrecisionFindpeaksSG.m creates a noisy signal with three peaks of widely different widths, measures the peak positions, heights and widths of each peak using findpeaksSG, and prints out the percent relative standard deviations of parameters of the three peaks in 100 measurements with independent random noise. With 3-segment peak detection parameters, findpeaksSG reliably detects and accurately measures all three peaks. In contrast, findpeaksG, tuned to the middle peak (using line 26 instead of line 25), measures the first and last peaks poorly. You can also see that the precision of peak parameter measurements gets progressively better (smaller relative standard deviation) the larger the peak widths, simply because there are more data points in those peaks. (You can change any of the variables in lines 10-18).



A related function is **findpeaksSGw.m** which is like the above except that is uses *wavelet denoising* (page 136) instead of smoothing. It takes the wavelet level rather than the smooth width as an input argument. The script TestPrecisionFindpeaksSGvsW.m compares the precision and accuracy for peak position and height measurement for both the findpeaksSG.m and findpeaksSGw.m functions.

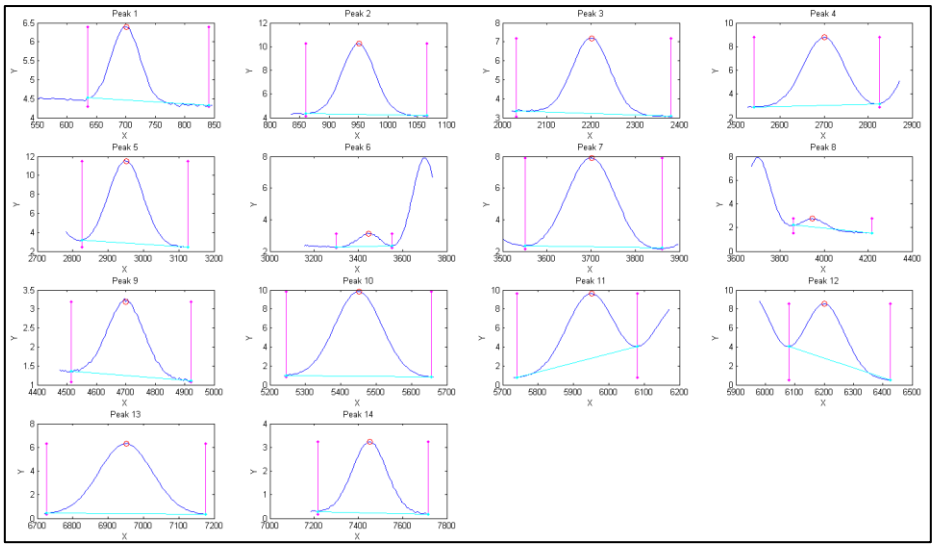
findpeaksSb.m is a segmented variant of the findpeaksb.m function. It has the same syntax as findpeaksSb, except that the arguments "SlopeThreshold", "AmpThreshold", "smoothwidth", "peakgroup", "window", "PeakShape", "extra", "NumTrials", "BaselineMode", and "fixedparameters" can all be optionally scalars or *vectors with one entry for each segment*. This allows the function to handle widely varying signals with peaks of very different shapes and widths and backgrounds. In the example on the right, the Matlab/Octave script DemoFindPeaksSb.m creates a series of Gaussian peaks whose widths increase *by a factor of 25* and that are superimposed in a curved baseline with random white noise that increases gradually across the signal. In this example, *four segments* are used, changing the peak detection and curve fitting values so that all the peaks are measured accurately.

```
SlopeThreshold = [.01 .005 .002 .001];
```

```
AmpThreshold = 0.7;
SmoothWidth = [5 15 30 35];
FitWidth = [10 12 15 20];
windowspan = [100 125 150 200];
peakshape = 1;
BaselineMode = 3;
```

The script also computes the relative percent error of the measurement of peak position, height, width, and area for each peak.

measurepeaks.m is an automatic peak detector peaks of arbitrary shape. It is based on findpeaksSG, with which it shares the first 6 input arguments. The four peak detection arguments can be *vectors* to accommodate signals with peaks of widely varying widths. It returns a table containing the peak number, peak position, absolute peak height, peak-valley difference, perpendicular drop area, and tangent skim area of each peak it detects. If the last input argument ('plots') is set to 1, it will plot the entire signal with numbered peaks and will also plot the individual peaks with the peak maximum, valley points, and tangent lines marked (as shown on the right). Type "help measurepeaks" and try the examples there or run testmeasurepeaks.m (graphic animation). The related functions autopeaks.m and autopeaksplot.m are similar, except that the four peak detection parameters *can be omitted* and the function will calculate estimated initial values.



The script HeightAndArea.m tests the accuracy of peak height and area measurement with signals that have multiple peaks with variable width, noise, background, and peak overlap. Generally, the values for absolute peak height and perpendicular drop area (page 146) are best for peaks that have no background, even if they are slightly overlapped, whereas the values for peak-valley difference and for tangential skim area are better for isolated peaks on a straight or slightly curved background. Note: this function uses smoothing (specified by the SmoothWidth input argument) only for peak detection. It performs its measurements on the raw unsmoothed y data. If the raw data are noisy, it may be best to smooth the y data yourself before calling measurepeaks.m, using any smooth function of your choice.

Other segmented functions. The same segmentation code used in SegmentedSmooth.m (lines 53-65) can be applied to other functions simply by editing the first line in the first for/end loop (line 59) to refer to the function that you want to apply in a segmented fashion. For example, segmented peak sharpening can be useful when a signal has multiple peaks that vary in width, and segmented deconvolution can be useful when a signal has multiple peaks that vary in width or tailing vary substantially across the signal: SegExpDeconv(x,y,tc) deconvolutes y with a vector of exponential functions whose time constants are specified by the vector tc. SegExpDeconvPlot.m is the same except

that it plots the original and deconvoluted signals and *shows the divisions between the segments by vertical magenta lines*.

Measurement Calibration

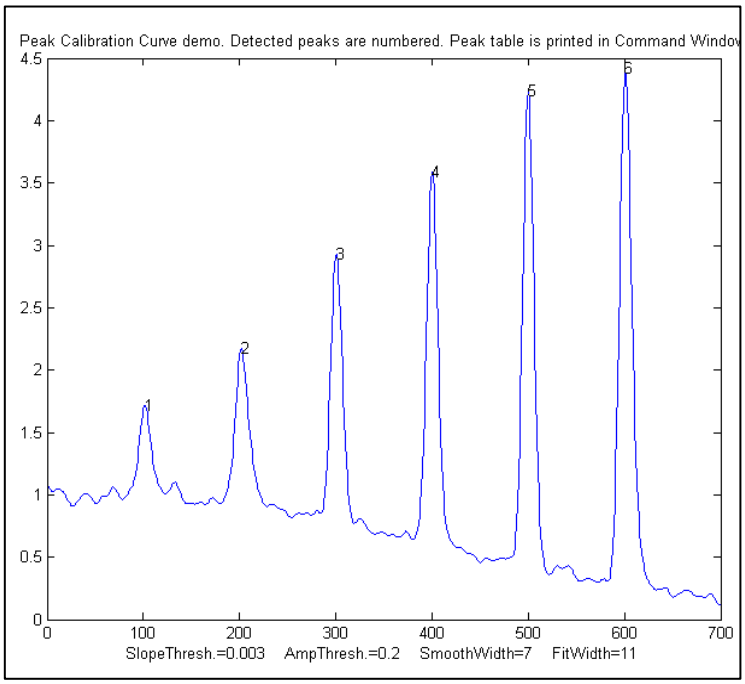
Most scientific measurements involve the use of instruments that measure something else and converts it to the desired measure. Examples are simple weight scales (which first measure the compression of a spring), common thermometers (which measure thermal expansion), pH meters (which measure a voltage), and most devices for measuring your heart rate, hemoglobin in blood, CO_2 in air, or sugar in wine grape juice (all of which measure a light beam). These instruments are *single-purpose*, designed to measure one quantity, and automatically convert what they first measure into the desired quantity and display it directly. But to ensure accuracy, such instruments can be calibrated, that is, used to measure one or more calibration standards of known accuracy, such as a standard weight or a sample that is carefully prepared to a known temperature, pH, or sugar content. Most are pre-calibrated at the factory for the measurement of a specific substance in a specific type of sample.

Analytical calibration. In contrast to single-purpose measurements, *general purpose* chemical analysis instruments are used to measure the quantity of many different chemical components in various types of samples. These methods include various kinds of spectroscopy, chromatography, and electro-chemistry, or combination techniques like "GC-mass spec". These must also be calibrated, but because those instruments can be used to measure a wide range of compounds or elements, they must be calibrated *by the user* for each substance and for each type of sample. Usually, this is accomplished by carefully preparing (or purchasing) one or more "standard samples" of known concentration, such as solution samples in a suitable solvent. Each standard is inserted or injected into the instrument, and the resulting instrument readings are plotted against the known concentrations of the standards, using least-squares calculations to compute the *slope* and *intercept*, as well as the standard deviation of the slope (*sds*) and intercept (*sdi*). Then the "unknowns" (that is, the samples whose concentrations are to be determined) are measured by the instrument and their signals are converted into concentrations with the aid of the calibration curve. If the calibration is linear, the sample concentration C of any unknown is given by (A - *intercept*) / *slope*, where A is the measured signal (height or area) of that unknown. The predicted standard deviation in the sample concentration is C*SQRT((*sdi*/(A-*intercept*))^2 + (*sds*/*slope*)^2) by the rules for propagation of error. All these calculations are done in the spreadsheet template CalibrationLinear.xls. In some cases the thing measured cannot be detected directly but must undergo a chemical reaction that makes it measurable. In that case, the exact same reaction must be carried out on all the standard solutions and unknown sample solutions.

Various calibration methods are used to compensate for problems such as random errors in standard preparation or instrument readings, interferences, drift, and non-linearity in the relationship between concentration and instrument reading. For example, the standard addition calibration technique can be used to compensate for multiplicative interferences. I have prepared a series of "fill-in-the-blanks" templates for various calibrations methods, with instructions, as well as a series of spreadsheet-based simulations of the error propagation in widely-used analytical calibration methods, including a step-by-step exercise.

Calibration and signal processing. Signal processing often intersects with calibration. For example, if you use smoothing or filtering (page 42) to reduce noise, or differentiation (page 63) to reduce the effect of background, or measure peak area to reduce the effect of peak broadening (page 140), or use modulation to reduce the effect of low-frequency drift (page 320), then you *must* use the exact same signal processing for both the standard samples and the unknowns. The choice of signal processing technique can have a big impact on the magnitude and even on the *units* of the resulting processed signal (e.g. in the derivative technique and in choosing between peak height and peak area).

PeakCalibrationCurve.m is a Matlab/Octave example of this. This script simulates the calibration of a flow injection system that produces signal peaks that are related to an underlying concentration or amplitude ('amp'). In this example, six known standards are measured sequentially, resulting in six separate peaks in the observed signal. (I assume that the detector signal is linearly proportional to the concentration at any instant). To simulate a more realistic measurement, the script adds four sources of "disturbance" to the observed signal:



a. *random* white noise added to all the signal data points, controlled by the variable "Noise";

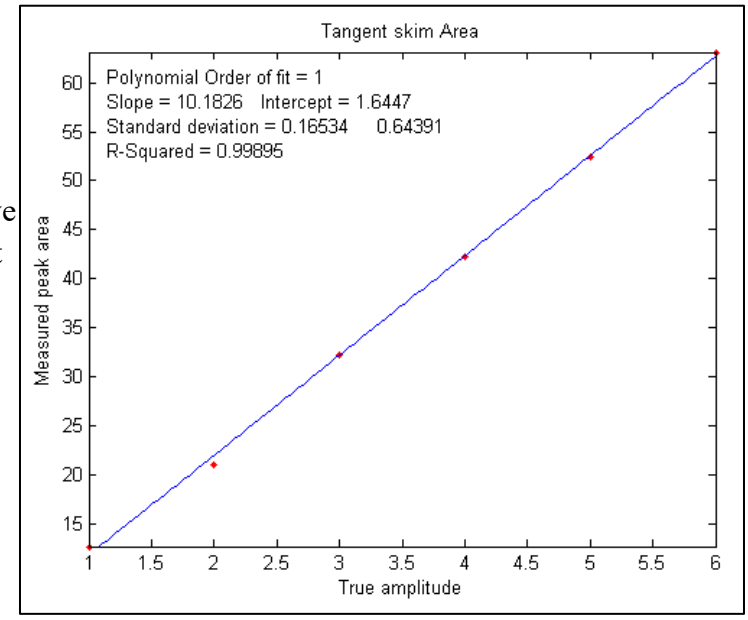
b. *background* - broad curved background of *random amplitude*, tilt, and curvature, controlled by "background";

c. *broadening* - exponential peak broadening that *varies randomly* from peak to peak, controlled by "broadening";

d. a final *smoothing* before the peaks are measured, controlled by "FinalSmooth".

The script uses measurepeaks.m as an internal function to determine the absolute peak height, peak-valley difference, perpendicular drop area, and tangent skim area (page 142). It plots separate calibration curves for each of these measures in Matlab Figure windows 2-5 against the true underlying amplitudes (in the vector "amp"), fitting the data to a straight line and computing the slope, intercept, and R2. (If the detector response were non-linear, a quadratic or cubic least squares would work better). The slope and intercept of the best-fit line are different for the different methods, but if the R^2 is close to 1.000, a successful measurement can be made. (If all the random disturbances are set to zero in lines 33-36, the R^2 values will all be 1.000. Otherwise, the measurements will not be perfect, and some methods will result in better measurements - R^2 closer to 1.000 - than others). Here is a typical result:

Peak	Position	PeakMax	Peak-val.	Perp drop	Tan skim
1	101.56	1.7151	0.72679	55.827	11.336
2	202.08	2.1775	1.2555	66.521	21.425
3	300.7	2.9248	2.0999	58.455	29.792
4	400.2	3.5912	2.949	66.291	41.264
5	499.98	4.2366	3.7884	68.925	52.459
6	601.07	4.415	4.0797	75.255	61.762
R2 values:		0.9809	0.98615	0.7156	0.99824



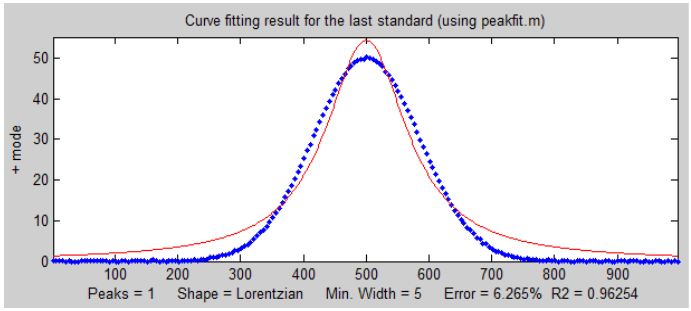
In this case, the tangent skim method works best, giving a linear calibration curve (shown on the right) with the highest R2.

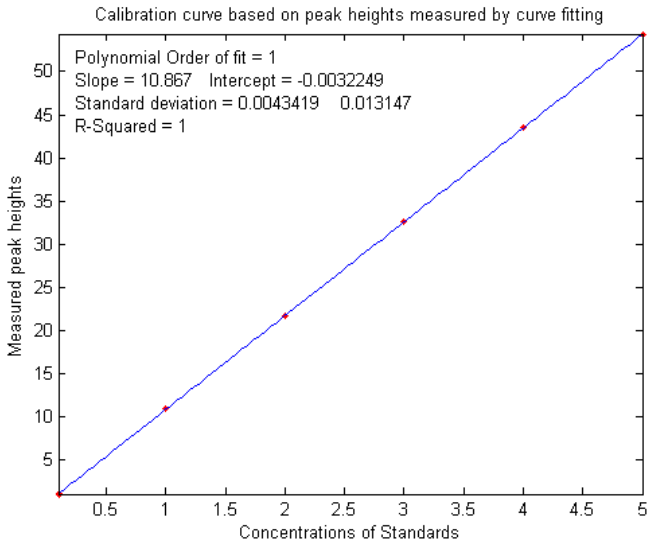
In this type of application, the peak heights and/or area measurements do not actually have to be *accurate*, but they must be *precise*. That is because the objective of an analytical method such as flow injection or chromatography is *not* to measure the peak *heights* and *areas*, but rather to measure *concentrations*, which is why calibration curves are used. Figure window 6 shows the correlation plot between the measured tangent skim areas and the actual true areas under the peaks in the signal shown above. The slope of this plot shows that the tangent skim areas are about 6% lower than the true areas, but that does not make a difference in this case because the standards and the unknown samples are measured the same way. In some *other* applications, you may need to measure the peak heights and/or areas accurately, in which case curve fitting is generally the best way to go.

If the peaks partly overlap, the measured peak heights and areas may be affected. To reduce the problem, it may be possible to reduce the overlap by using peak sharpening methods, for example the derivative method, deconvolution or the power transform method, as demonstrated by the self-contained Matlab/Octave function PowerTransformCalibrationCurve.m.

Curve fitting the signal data. Ordinarily in curve fitting methods, such as the classical least-squares (CLS) method and in iterative nonlinear least-squares, the selection of a model shape is very important. However, in the *quantitative analysis* applications of curve fitting, where the peak height or area measured by curve fitting is used only to determine the concentration of the substance that created the peak by constructing a calibration curve, having the exact model shape is *surprisingly*

uncritical. The Matlab/Octave script PeakShapeAnalyticalCurve.m shows that, for a single isolated peak whose shape is constant and independent of concentration, if the wrong model shape is used, the peak *heights* measured by curve fitting will be inaccurate, but that error will be *exactly the same* for the unknown samples and the known calibration standards, so the error will "cancel out" and the measured concentrations will still be accurate, provided you use the *same* inaccurate model for both the known standards and the unknown samples. In the example shown on the right above, the peak shape of the actual peak is Gaussian (blue dots), but the model used to fit the data is Lorentzian (red line). That is an intentionally bad fit to the signal data. The R^2 value for the fit to the signal data is only 0.962 (a poor fit by the standards of measurement science). The result of this is that the *slope* of the calibration curve (shown on the left) is *greater than expected*. It should have been 10 (because that's the value of the "sensitivity" in line 18), but it is 10.867 in the figure on the left. Nevertheless, the *calibration curve is linear* and its R^2 value is 1.000, meaning that the analysis should be accurate. (So, curve fitting is actually applied *twice* in this type of application, once using iterative curve fitting to fit the *signal data*, and then again using polynomial curve fitting to fit the *calibration data*).





Despite all this, it is still better to use an accurate model peak shape for the signal data, because the percent fitting error or the R^2 *of the signal fit* can be used as a warning that something unexpected is wrong, such as an increase in noise or the appearance of an interfering peak from a foreign substance.

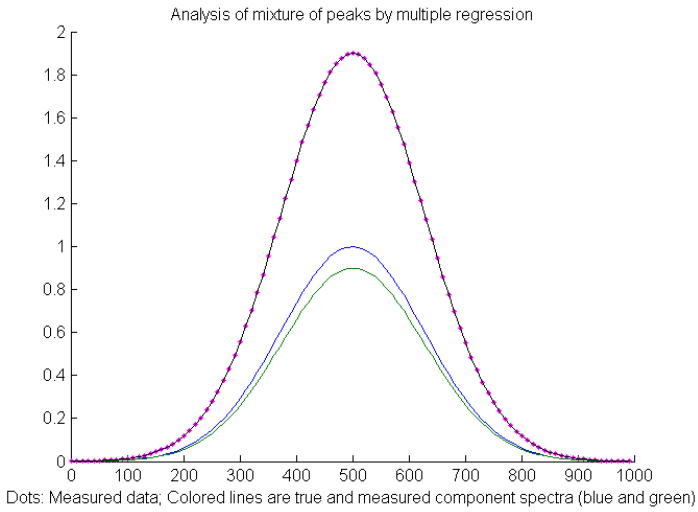
Numerical precision of computer software

Computations carried out by computer software with non-integer numbers have a natural limit to the precision with which they can be represented. For example, the number 1/3 is represented as 0.3333333..., using a large but finite number of "3"s, whereas theoretically there is an *infinite* string of "3"s in the decimal representation of 1/3. It is the same with irrational numbers such as "pi" and the square root of 2. They can never have an *exact* decimal representation. In principle, these tiny errors could accumulate in a very complex multiple-step calculation and could conceivably become a source of error. In most applications to scientific computation, these limits will be minuscule compared to the errors and random noise that is already present in most real-world measurements. But it is best to know what those numerical limits are, under what circumstances they might occur, and how to minimize them.

Multicomponent spectroscopy. Probably the most common calculation where numerical precision is an issue is in the matrix methods that are used in multicomponent spectroscopy. In the derivation of the

Classical Least-squares (CLS) method, the matrix inverse is used to solve large systems of linear equations. The matrix inverse is a standard function in programming languages such as *Matlab*, *Octave*, *Python*, *Mathematica*, and in spreadsheets. But if you use that function in Matlab, the function name ("inv") is *automatically flagged by the editor* with the following warning:

"For solving a system of linear equations, the inverse of a matrix is primarily of theoretical value. Never use the inverse of a matrix to solve a linear system Ax=b with x=inv(A)*b, because it is slow and inaccurate.... Instead of multiplying by the inverse, use matrix right division (/) or matrix left division (\). That is: Replace inv(A)*b with A\b ... [and]...replace b*inv(A) with b/A"

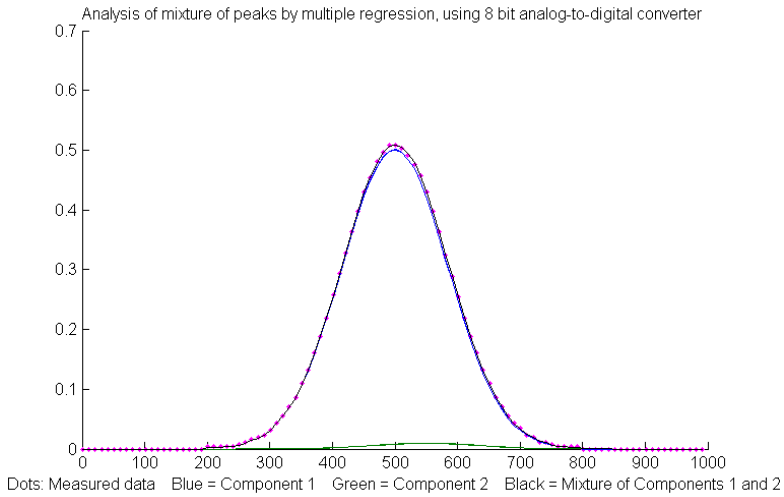


Dots: Measured data; Colored lines are true and measured component spectra (blue and green)

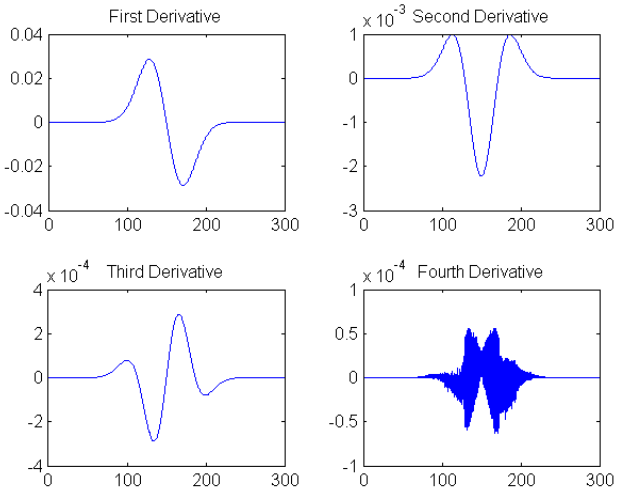
"Slow and inaccurate"? Scary words! But, really, how serious a problem is this in actual applications? To answer that question, the following numerical experiment, RegressionNumericalPrecisionTest.m, applies the CLS (Classical Least Squares) method to a mixture of two *very closely-spaced noiseless* overlapping Gaussian peaks (blue and green lines in the figure above) using *three different mathematical formulations* of the least-squares calculation that give different results. The difficulty of such a measurement depends on the ratio of the peak separation to the peak half-width. Small ratios mean very highly overlapped peaks which are hard to measure accurately. In this example the separation-to-width ratio is 0.0033, which is very small. This is equivalent to trying to measure a mixture of two absorption spectroscopy peaks that are 300 nm wide and *separated by only 1 nm*, a tiny difference that you would not even notice with the naked eye. The results of this script show that the matrix inverse ("inv") method does indeed have an *error thousands of times larger* than the method using matrix division, but even the matrix division error is still very small. Practically, the difference between these methods is unlikely to be significant when applied to *real* experimental data, because even the tiniest bit of signal instability would produce a far greater error (like that caused by small changes in the temperature of the sample or by random noise in the signal, which you can simulate in line 15).

So basically, that warning message is the voice of a mathematician or computer programmer, *not* that of an experimental scientist.

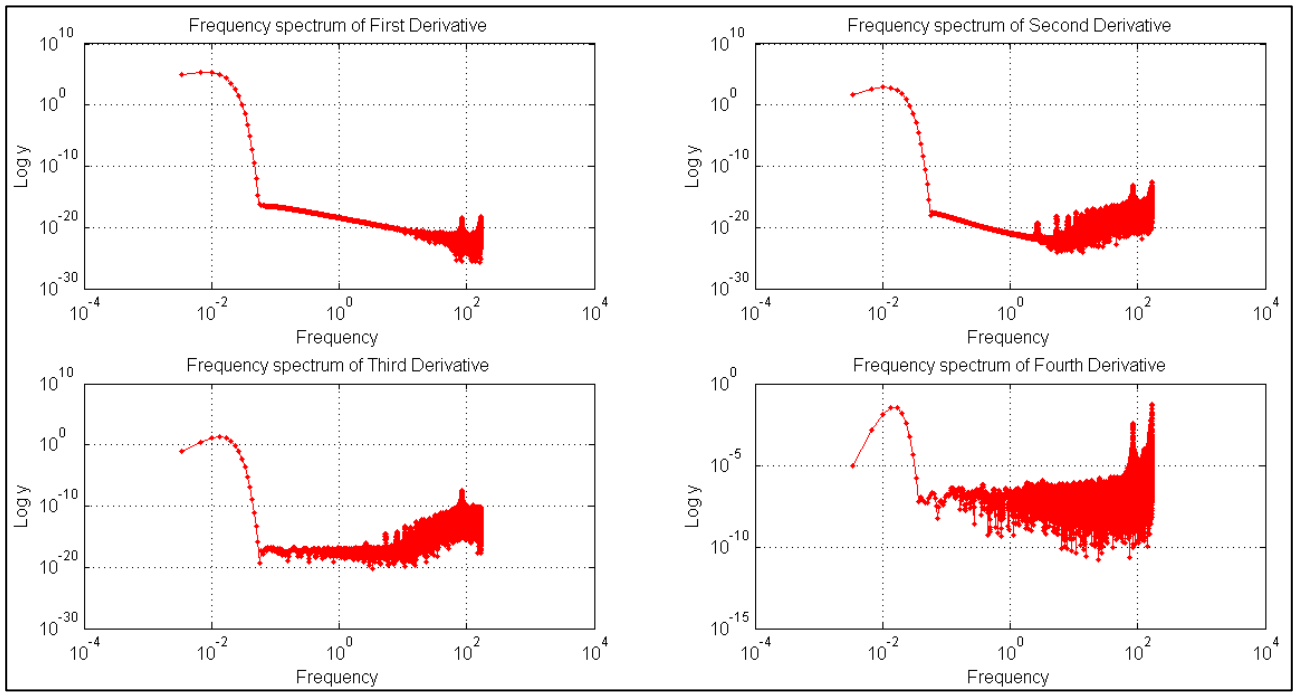
Analog-to-digital resolution. Potentially more significant than the computer's numerical resolution is the resolution of the *analog-to-digital converter* (ADC) that is used to convert analog signals (e.g., voltage) to a number. The numerical experiment RegressionADCbitsTest.m demonstrates this, with two slightly overlapping Gaussian bands with a *large* (*50-fold) difference in peak height* (blue and green lines in the figure on the right, peaking at 500 and 550 nm respectively). In this case the separation to width ratio is 0.25, much larger (i.e. easier) than the previous example. For this example, the simulation shows that the relative percent error of peak height measurement is 0.19% for the larger peak and 6.6% for the smaller peak. You can change the resolution of the simulated analog-to-digital converter in the *number of bits* (line 9). The amplitude resolution of an analog-to-digital converter is 2 raised to the power of the number of bits. Common ADC resolutions are 10, 12, and 14 bits, corresponding to resolutions of one part in 1024, 4096 and 16384, respectively. Of course, the effective resolution for the *smaller* peak, in this case, is 50 times less, and you cannot simply turn up the pre-amplifier on the smaller peak without overloading the ADC for the larger one. Surprisingly, if *most* of the noise in the signal is this kind of digitization noise, it may help to *add some additional random noise* (specified in line 10 in this script), as was seen on page 308.



Differentiation. Another application where you can see numerical precision noise is in differentiation, which involves the subtraction of very nearly equal adjacent numbers in a data series. The Matlab/Octave numerical experiment DerivativeNumericalPrecisionDemo.m shows how the numerical precision limits of the computer affect the first through fourth derivatives of a Gaussian band that is very finely sampled (over 16,000 points in the half-width in this case) and that has no added noise. The plot on the left shows the four waveforms on the right, and their frequency spectra are shown in the figure just below. The numerical precision limit of the computer creates random noise at very high frequencies, which are emphasized by differentiation. In the frequency spectra below, the big low-frequency bump near a frequency of 10^{-2} is the *signal* and everything above that is numerical *noise*. The lower-order derivatives are seldom a problem, but by the time you reach the fourth derivative, those noise frequencies approach the strength of the signal frequencies, as you can see in the frequency spectrum of the fourth derivative in the lower right. But this noise is only a very high-frequency noise, so smoothing with as little as a 3-point sliding average smooth removes most of it (click to view).



An alternative derivative method based on the Fourier Transform (page 84) has slightly lower numerical errors but is seldom used in practice (reference 88). The fundamental difference between the two is that the finite difference method works *locally*, on one small segment of the data at a time, whereas the FT method works *globally* because each frequency in the Fourier representation extends throughout the entire time domain. The Matlab/Octave script FDvsFTderivative.m compares the numerical errors of finite difference (FD) and Fourier transform (FT) methods of differentation. It creates a very broad, finely sampled Gaussian peak and then computes its fourth derivative both ways. (The noise is casued only by software numerical resolutiion limitations). The result (graphic) is that the



numerical errors are lower for the FT method near the peak but are greater far from the peak center.

Smoothing. Finally, there might potentially be a numerical problem with the fastsmooth algorithm, covered in the section on smoothing, because it is a *recursive* algorithm that uses the results of a previous step in the calculation to calculate the next step. The numerical precision of fastsmooth.m is shown by the numerical experiment script FastsmoothNumericalPrecisionTest.m. Even for 4000-point P-spline smooth applied to a 100,000-point signal, the numerical noise relative standard deviation is only 0.00027%, and most of that occurs in the edges of the signal (first 4% and last 4% of the points). The error over 90% of the signal is *orders of magnitude less*, a negligible problem in practical cases.

Miniaturized signal processing: Raspberry Pi and Adafruit



Signal processing does not necessarily require expensive computer systems. Raspberry Pi and Adafruit market a series of tiny but remarkably capable single-board computers and controllers that range in cost between $10 and $80 and range in size from a deck of cards to a pack of old-style chewing gum! Some models have several USB ports, general-purpose input-output pins, HDMI port, Ethernet port, audio jack and composite video, interfaces for a video camera, micro-SD card slot for mass storage, a graphics core, Wireless LAN, and Bluetooth. You can get them with various installed software, including a version of the Linux operating system, a Web browser, the complete LibreOffice suite, Wolfram's Mathematica (page 462, screenshot), several programming languages and various utilities. These are installed by default on the Raspberry Pi's operating system installer. (The smallest and cheapest models are ideal in situations where they might be damaged or lost, as in rocket or balloon-borne experiments). There are many starter packs available that bundle all the required bits and pieces.

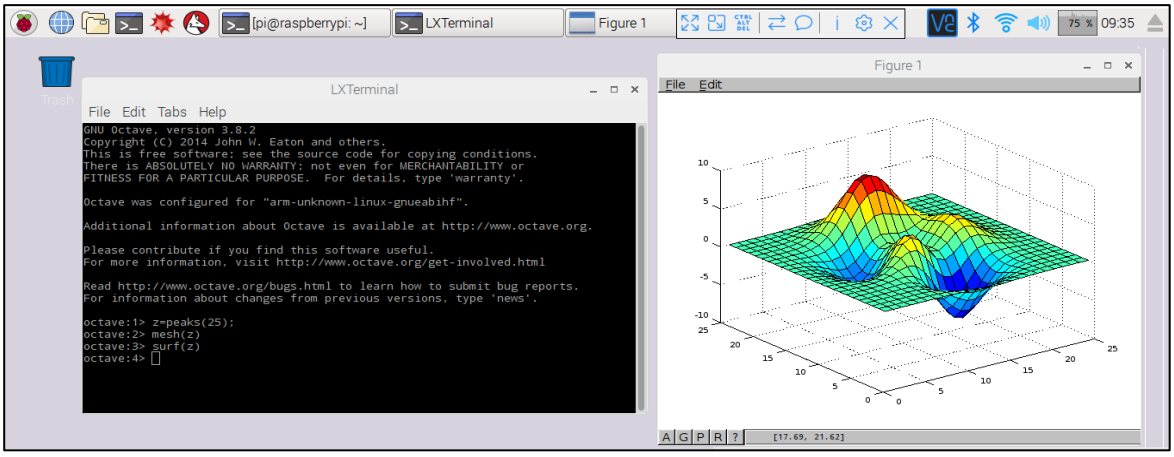
You can easily build a complete, low-cost Windows 10 computer from a Raspberry Pi version 4 ($70) which comes built into a full keyboard. You need only a USB-C 5 volt power supply, a TV/monitor with an HDMI input, and a mouse (all of which you might find at a second-hand shop), and a mini SD card (8 to 16 Gbytes) for mass storage (which you can buy with all the software already installed, or use a blank one to which you can download the free software yourself). For educational purposes, the Pi has been used as a low-cost alternative for school computer labs, using its included software for both Office-type applications (*Writer* word processor, *Calc* spreadsheet, etc.), and for programming instruction, such as Python (page 439), C, C++, Java, Scratch, Ruby, etc.

Single-board versions are ideal for "headless" applications (meaning without a monitor, keyboard, or mouse) which, after being set up, are *accessed only remotely* via WiFi or Bluetooth, using Putty (for command-line UNIX-style access) or using a graphical desktop sharing system such as RealVNC (free for Windows, Mac, IOS, and Android), which reproduces the entire graphical desktop on your local device, complete with a pop-up virtual keyboard. Typical applications are as a network file server, weather station, media center or as a networked security camera. It can also share files with Windows.

For scientific data acquisition and signal processing applications, the Pi version of Linux has all the "usual" UNIX terminal commands for data gathering, searching, cleaning and summarizing. In addition, there are many add-on libraries for Python, including SciPi, NumPy, and Matplotlib, all of which are free downloads (page 439). Allen B. Downey's PDF book "Think DSP" has many examples of Python code in traditional engineering applications. Some versions include video cameras and chips that read and displays sensor data from several built-in sensors: gyroscope, accelerometer, magnetometer, barometer, temperature, relative humidity, tec.

My signal-processing spreadsheets (page 507) run just fine on the version of *Calc* that comes with the

Pi, as do the calibration worksheets (page 465) and my analytical instrument models (page 355). For school applications, Element14 markets several different educational kits and activities based on the Raspberry Pi models.



Octave 3.6 can run directly on some Raspberry Pi versions. The screen above shows Octave 3.6 running within the Pi's built-in graphical user interface (showing off the 3D graphic "surf" function).

There are many laboratory and field applications, especially in combination with an Arduino micro-controller. But if none of the available Raspberry Pi models are sufficiently fast for your signal-processing needs, then you use the Pi only for data acquisition and transfer the data to a faster computer. (For MATLAB users, there is a MATLAB Support Package for Raspberry Pi Hardware that supports this. Alternatively, you could simply have the Raspberry Pi save data or results in a shared folder that is accessed via Wi-Fi from another computer).

Python (page 442) is the primary programming language that comes with the Raspberry Pi. Matlab programmers can use ChatGPT to convert their code to Python *(page 442)*. As a comparison, here is a real-time example of data acquisition and plotting on a Raspberry Pi, using the commercially available add-on Sense Hat board to measure its own temperature as it warms up, using a program written in Python. Click for real-time animation. (If you do not have a Sense Hat, here's a modification of the same Python program that plots the running average of random numbers, using the same autoscaling graphic technique, showing a result that gradually settles down closer and closer to the average the longer you let it run). This Matlab/Octave script does the *same thing at the same speed*, but in this particular case the Matlab/Octave script length is substantially shorter. (For a more extensive comparison of Python to Matlab for several different signal processing tasks, see page 439).

Other competing systems include the BeagleBoard and the LattePanda, a tiny $130 Windows-10 computer board with 2 Gbytes RAM and 32 Gbytes flash storage. Many similar products are available.

Batch processing

In situations where you have a large volume of similar types of data to process, it is useful to automate the process. For example, suppose that you have already acquired reams of data in the form of multiple text files or numerical data files of some standardized format that are stored in a known directory (folder) somewhere on your computer. They might be ASCII .txt (plain text) or .csv ("comma separated values") files with the independent variable ('x') in the first column and one or more dependent

variables ('y') in the other columns. There may be a variable number of data files, and their file names and length may be variable, but, crucially, the data *format* must be consistent from file to file. You could write a Matlab script or function that will process those files *one-by-one*. But wouldn't be nice if the computer could go through *all* the data files in that directory *automatically*, determine their file names, load each into the variable workspace, apply the desired processing operations (peak detection, deconvolution, curve fitting, wavelets, whatever), collect all the resulting terminal window output, each labeled with the file name, add those results to a growing "diary" file, and then go on to the next data file. Ideally, the program *should not stop* if it encounters any kind of fatal error. Rather, it should just *skip that file and go on to the next*. All that sounds complicated, but it is easier than it seems.

BatchProcess.m is a Matlab/Octave example of just such an automated process that you can use as a framework for your applications. To adapt this script to your own purposes, you need only change:

> **(a)** the directory name where your data are stored on your computer – this is ("DataDirectory") in line 11 of BatchProcess.m;
>
> **(b)** the directory name where the Matlab signal processing functions are stored on your computer - ("FunctionsDirectory") in line 12; and
>
> **(c)** the actual processing functions that you wish to apply to each file (which in this example perform peak fitting using the "peakfit.m" function in lines 34 – 41, but could be *anything*).

When it starts, the routine creates and opens a "diary" file in line 21, which will be placed in the FunctionsDirectory, with the file name "BatchProcess<date>.txt" (where <date> is the current date, e.g. 12-Jun-2022). This file captures all the terminal window output during processing - in this example, I am using the peakfit.m function that generates a FitResults matrix (with Peak#, Position, Height, Width, and Area of the best-fit model), and a Goodness of Fit (GOF) matrix containing the percent fitting error and R^2 value, for each data file in that directory. Subsequent runs of the program on the same date are appended to this file. On each subsequent day, a new file is begun for that day. You can also optionally save some of the variables in the workspace as data files. Add a "save" function after the processing and before the "catch me" statement (type "help save" at the command prompt for options).

This program uses some coding techniques that are especially useful in automated file processing. It uses the "function forms" of several commands -"ls" (line 13), "diary" (line 21), and "load" (line 29) - to allow then to accept variables computed within the program. It also uses the "try/catch/end" structure (lines 28, 47, 49), which prevents the program from stopping if it encounters an error on one of the data files. If an error occurs, it adds a line to the diary that reports the error for that file and skips to the next. (Note: not surprisingly, Python has a similar functions called "try..except..finally" and pydiary). After running this script, the "BatchProcess..." diary file will contain all the terminal output.

Here is an excerpt from a typical diary file. In this example, *the first two data files in the directory yielded errors*, but the third one ("`2016-08-05-RSCT-2144.txt`"), and all the following files, worked normally and reported the results of the peak fitting operations:

```
Error with file number 1.
Error with file number 2.
3: 2016-08-05-RSCT-2144.txt
         Peak#  Position  Height     Width       Area
            1   6594.2    0.1711    0.74403    0.13551
            2   6595.1    0.16178    0.60463     0.1041
   % fitting error    R2
       2.5735      0.99483
4: 2016-09-05-RSCT-2146.txt
         Peak#   Position  Height    Width      Area
            1    6594.7    0.11078   1.4432    0.17017
            2    6595.6    0.04243   0.38252   0.01727
   % fitting error    R2
       4.5342      0.98182

5: 2016-09-09-RSCT-2146.txt
         Peak#   Position  Height     Width      Area
            2    6594      0.05366    0.5515     0.0315
            1    6594.9    0.1068     1.2622     0.1435
   % fitting error    R2
        3.709      0.98743
6: .... Etc....
```

Note: You could optionally import the dairy file into Excel by opening an Excel worksheet, click on a cell, click **Data > From Text**, select the diary file, click to *specify that spaces are to be used as column separators*, and click **Import**. This will put all the collected terminal output into that spreadsheet. Additionally, you might want to save the workspace variables (e.g., as a .mat file).

Real-time signal processing

All the signal processing techniques covered so far assume that you have acquired and have stored the data in computer memory before beginning processing. In some cases, however, it is necessary to do the signal processing in "real-time", that is, point-by-point as the data are acquired from the sensor or instrument. That requires some modification of the software, but the main conceptual ideas still apply. In this section you will look at ways to perform real-time data plotting, smoothing, differentiation, peak detection, harmonic analysis (frequency spectra), and Fourier filtering. Because the data acquisition details vary with each individual experimenter and instrumental setup, these demonstration scripts will *simulate* real-time data so that you can run them immediately on your computer to see how they work, without additional hardware. I will do this in either of two ways:

(a) by using mouse-clicks to generate data points one-by-one, using Matlab's "ginput" function, or
(b) by pre-calculating some simulated data and then accessing it point-by-point in a loop.

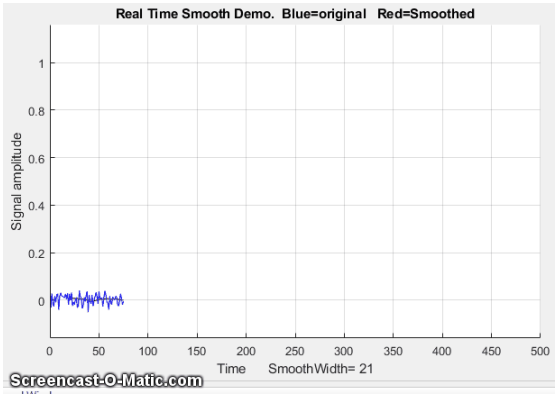
The first method is illustrated by the simple script realtime.m. When you run this script, it displays a graphical coordinate system. Position your mouse pointer along the *y* (vertical) axis and click to enter data points as you move the mouse pointer up and down. The "ginput" function waits for each click of the mouse button, then the program records the *y* coordinate position and counts the number of clicks.

Data points are assigned to the vector *y* (line 17), plotted on the graph as black points (line 18), and print out in the command window (line 19). The script realtimeplotautoscale.m is an expanded version that changes the graph scale as the data come in. If the number of data points exceeds 20 ('maxdisplay'), the *x*-axis maximum is re-scaled to twice that (line 32). If the data amplitude equals or exceeds ('maxy'), the *y*-axis is re-scaled to 1.1 times the data amplitude (line 36).

The second method is illustrated by the script realtimeplotautoscale2.m, which simulates real-time data by using pre-calculated data (loaded from your hard drive in line 13) that is accessed point-by-point in lines 25 and 26 (If the animation is not visible, click on the figure to open in a web browser). Another script, realtimeplotdatedtime.m, demonstrates one way to use Matlab's 'clock' function to record the date and time of each data point that is acquired by clicking. (You could also have the computer control the time of data acquisition by reading the clock in a loop until the desired time and date arrives, then take a data point). Of course, a Windows machine is hardly ideal for high-speed, precisely-timed data acquisition, because there are typically so many interrupts and other processes going on in the background, but it is adequate for low-speed applications. For higher speeds, specialized hardware and software are available, such as the new RP2350 Raspberry Pi microcontroller, which is designed specifically for real-time operation.



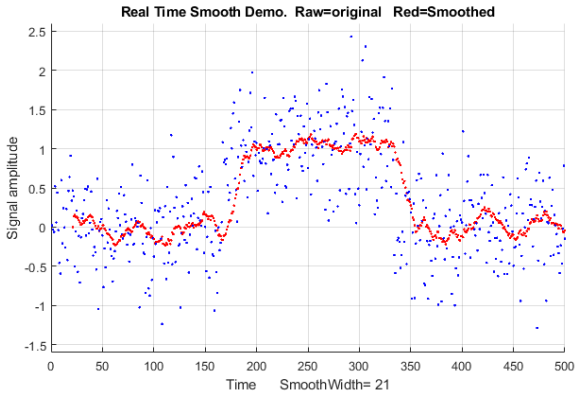
Smoothing. The script RealTimeSmoothTest.m demonstrates real-time smoothing (page 42), plotting the raw unsmoothed data as a blue line and the smoothed data in red. In this case, the script pre-calculates simulated data in line 28 and then accesses the data point-by-point in the processing 'for' loop (lines 30-51). The total number of data points is controlled by 'maxx' in line 17 (initially set to 1000) and the smooth width (in points) is controlled by 'SmoothWidth' in line 20. (To do this with real-time data from your sensor, comment out line 29 and replace line 32 with the code that acquires one data point from your sensor).



As you can see in the screen image on the left above (link to animation), the smoothed data (in red) is *delayed* compared to the raw data, because a smoothed data point cannot be computed until a number of data points equal to the smooth width have been acquired - 21 points in this example. (However, knowing the smooth width, you can correct the recorded y-axis positions of signal features, such as maxima, minima, peaks, or inflection points). This particular example implements a sliding average smooth, but other smooth shapes can be implemented simply by uncommenting line 24 (rectangular), 25 (triangular), or 26 (Gaussian), which requires that the functions 'triangle' and 'gaussian' be in the

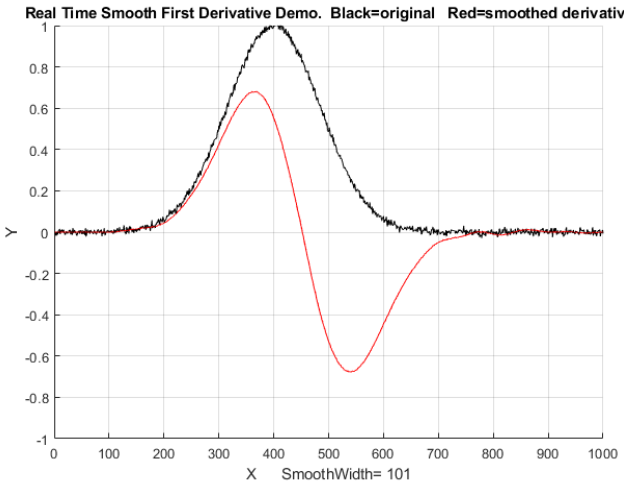
Matlab/Octave search path.

A practical application of a sliding average smooth like this is in a control system where a noisy signal turns on a valve, switch, or alarm signal whenever the signal exceeds a certain value. In the example shown in the figure on the right, the threshold value is 0.5 and the signal (blue dots) is so noisy that smoothing is required to prevent the signal from prematurely triggering the control. Too much smoothing, however, will cause an unacceptable delay in operation. In this case, the alarm is sounded at about t=160 and stops at 350.



On a standard desktop PC (Intel Core i5 3 Ghz) running Windows 10 home, the smooth operation adds about 2 microseconds per data point to the data acquisition time (without plotting, PlottingOn=0 in line 20) and 20 milliseconds per point (50 Hz max) with point-by-point plotting (PlottingOn=1). With plotting off, the script acquires, smooths, and stores the smoothed data in the variable "sy" in real-time, then plots the data only after data acquisition is complete, which is much faster than plotting in real-time.

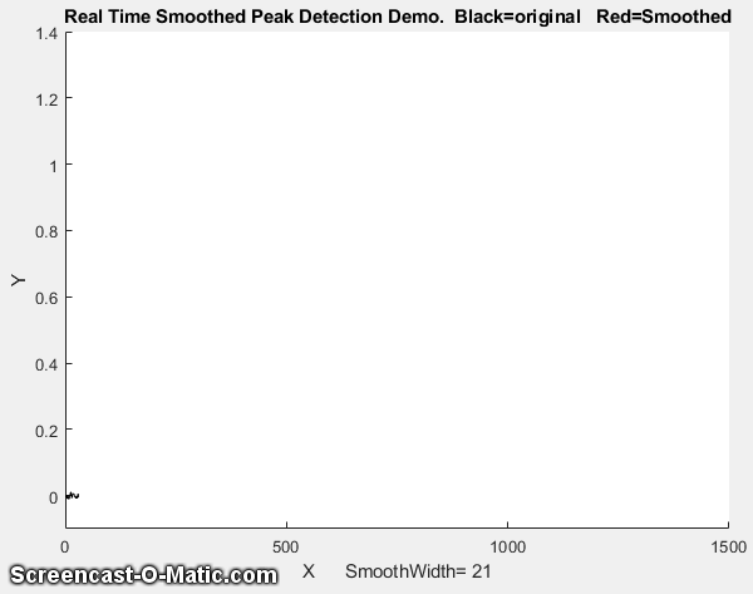
Differentiation. The script RealTimeSmoothFirstDerivative.m demonstrates real-time smoothed differentiation (page 63), using a simple adjacent-difference algorithm (line 47) and plotting the raw data as a black line and the first derivative data in red. The demonstration script RealTimeSmoothSecondDerivative.m computes the smoothed *second* derivative by using a central difference algorithm (line 47). Both scripts pre-calculate the simulated data in line 28 and then access the data point-by-point in the processing loop (lines 31-52). In both cases, the maximum number of points is set in line 17 and the smooth width is set in line 20. Again, the derivatives are delayed compared to the original signal. Any derivative order can be calculated this way using the derivative coefficients in the Matlab/Octave derivative functions listed on page 75.



Peak detection. The little script realtimepeak.m demonstrates simple real-time peak detection based on derivative zero-crossing (page 234), using mouse clicks to simulate data. Each time your mouse clicks form a peak (that is, go up and then down again), the program will register and label the peak on the graph and print out its *x* and *y* values.

```
Peak detected at x=13 and y=7.836
Peak detected at x=26 and y=1.707
```

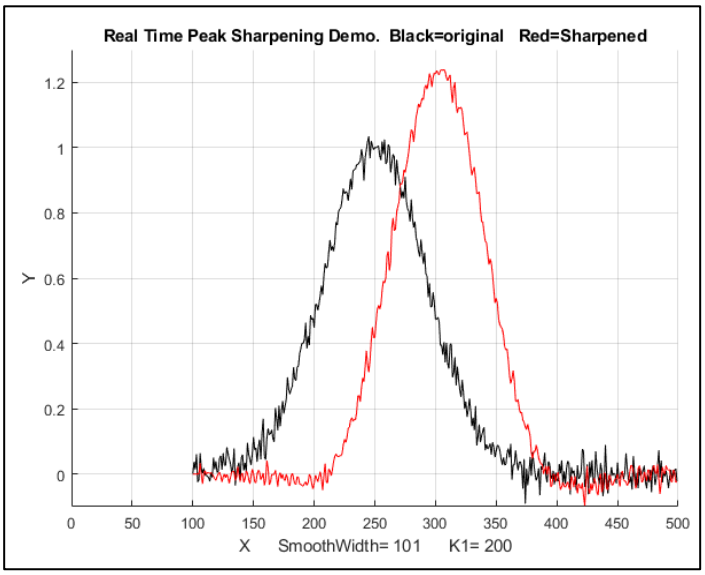
In this case, a peak is defined as any data point that has lower amplitude points adjacent to it on both sides, which is determined by the nested 'for' loops in lines 31-36. Of course, a peak cannot be registered until the point following the peak is recorded, because there is no way to predict ahead of time whether that point will be lower or higher than the previous point. If the data are noisy, it is better to smooth the data stream before detecting the peaks, which is exactly what the Matlab/ Octave script RealTimeSmoothedPeakDetection.m does, which reduces the chance of false peaks due to random noise but has the disadvantage of delaying the peak detection further. Even better, the Matlab/Octave script RealTimeSmoothedPeakDetectionGauss.m uses the technique described on page 236. It locates the positive peaks in a noisy data set that rise above a set amplitude threshold ("AmpThreshold" in line 55), performs a least-squares curve-fit of a Gaussian function to the *top part of the raw data peak* (in line 58), identifies each peak (line 59), computes the position, height, and width (FWHM) of each peak from that least-squares fit, and prints out each peak found in the command window. The peak parameters are measured on the raw data, so they are not distorted by smoothing. The "peak" label pops up next to each detected peak just a fraction of a second after the top of the peak, but *the peak times listed in the printed table are based on the raw data* and are *not delayed*. In this example, the actual peak times are x=500, 1000, 1100, 1200, 1400. (Also note that the first visible peak, at x=300, is not listed because it falls below the amplitude threshold, which is 0.1 in this case. However, if that peak is important, you could simply set that threshold to a lower value, e.g., 0.02). Link to animation.



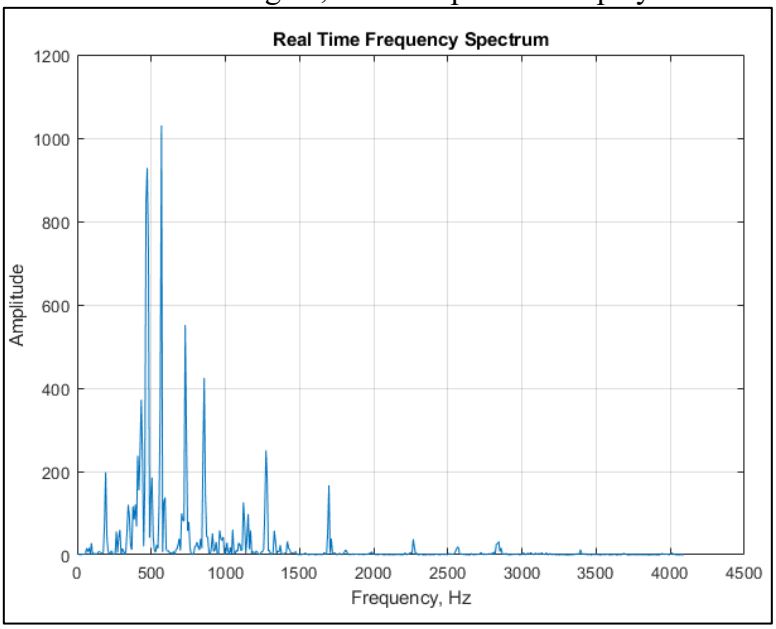

```
Peak detected at x=500.1705, y=0.42004, width= 61.7559
Peak detected at x=1000.074, y=0.18477, width= 61.8195
Peak detected at x=1100.033, y=1.28170, width= 60.1692
Peak detected at x=1199.849, y=0.36407, width= 63.8316
Peak detected at x=1400.147, y=0.26134, width= 58.9345
```

The script additionally numbers the peaks and saves the peak parameters of all the peaks in a matrix called PeakTable, which you can interrogate as each peak is encountered if you are looking for particular peak patterns. See page 251 for some ideas on using Matlab/Octave notation and functions to do this.

Peak sharpening. The Matlab/Octave script RealTimePeakSharpening.m demonstrates real-time peak sharpening (page 78) using the second derivative technique. It uses pre-calculated simulated data in line 30 and then accesses the data point-by-point in the processing loop (lines 33-55). In both cases the maximum number of points is set in line 17, the smooth width is set in line 20, and the weighting factor (K_1) is set in line 21. In the example on the right, the smooth width is 101 points, which accounts for the delay in the sharpened peak compared to the original.

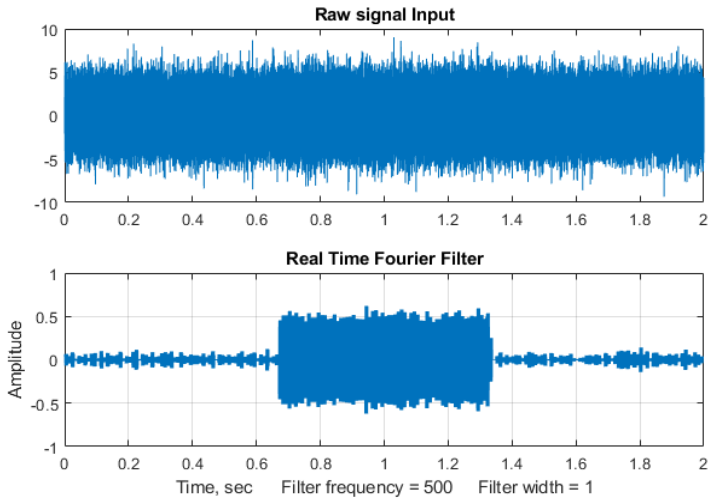


Real-Time Frequency Spectrum. The script RealTimeFrequencySpectrumWindow.m computes and plots the Fourier frequency spectrum of a signal (page 94). Like the scripts above, it loads the simulated real-time data from a ".mat file" and then accesses the data point-by-point in the processing loop. A critical variable in this case is "WindowWidth", the number of data points taken to compute each frequency spectrum. The larger this number, the fewer the number of spectra that will be generated, but the higher will be the frequency resolution. On an average desktop PC (Intel Core i5 3 Ghz running Windows 10 home), this script generates about 50 spectra per second with an average data rate (points per second) of about 50,000 Hz. Smaller spectra (i.e. lower values of WindowWidth) generate proportionally lower average data rates (because the signal stream is interrupted more often to calculate and graph a spectrum). If the data stream is an audio signal, it is also possible to play the sound through the computer's sound system synchronized with the display of the frequency spectra. To do this, set the variable `PlaySound` to a value of `1`. Each segment of the signal is played just before the spectrum of that segment is displayed, as shown on the right. (The sound reproduction will not be perfect, because of the slight delay while the computer computes and displays the spectrum before going on to the next segment). In this demonstration script, the data file is music - in fact it is an 8-second excerpt of an audio recording of the 'Hallelujah Chorus' from Handel's *Messiah,* with a sampling rate of 8192 Hz. This file is included as demonstration data in the Matlab distribution (handel.mat). The figure on the right shows one of the 70 spectra generated with a WindowWidth of



1024. You can adjust the argument of the 'pause' function for your computer to minimize this problem and to make the sound play smoothly at the correct pitch.

Real-Time Fourier Filter. The script RealTimeFourierFilter.m is a demonstration of a real-time Fourier filter. The script pre-computes a simulated signal starting in line 38, then it accesses the data point-by-point (lines 56, 57), and divides up the data stream into segments to compute each filtered section. The number of data points in each segment set by the variable “WindowWidth” (line 55). The larger this number, the fewer the number of segments that will be generated, but the higher the frequency resolution within each segment will be. On an average desktop PC (Intel Core i5 3 Ghz running Windows 10 home), with a window width of 1000 points, this script generates about 35 filtered segments per second with an average data rate (points per second) of about 34,000 Hz. Smaller segments (i.e., lower values of WindowWidth) generate proportionally lower average data rate (because the signal stream is interrupted more often to calculate and graph the filtered spectrum). The result of applying the filter to each segment is displayed in real-time during the data acquisition, and then at the end the script compares the entire raw signal input to the reassembled filtered output, as shown the figure above.



In this demonstration, the Fourier filter is set to bandpass mode, and is used to detect a 500 Hz sine wave (frequency set by 'f' in line 28) that occurs in the middle third of a very noisy signal (line 32), from about 0.7 sec to 1.3 sec. The 500 Hz sine wave is *so weak it cannot be seen* at all in the raw signal (upper panel of the figure on the left), but it really stands out in the filtered output (lower panel). The filter center frequency (CenterFrequency) and width (FilterWidth) are set in lines 46 and 47.

Real-time classical least squares. The classical least squares technique (page 188) can be applied in real time to 2D chromatography with array detectors that can acquire a complete spectrum multiple times per second over the entire chromatogram. This is explored in “Spectroscopy and chromatography combined: time-resolved Classical Least Squares” on page 363.

To apply any of these examples to real-time data from your sensor or instrument, you only need the main processing 'for' loop, replacing the first lines after the 'for' statement with a call to a function that acquires a single point of raw data and assigns it to y(n). If you do not need the data plotted out point-by-point in real-time, you can speed things up greatly by removing the “drawnow” statement at the end of the 'for' loop or by removing all the plotting code.

In the examples here, the *output* of the processing operation is used to plot or to print out the processed data point-by-point, but of course it could also be used as the input to another processing function or to a digital-to-analog converter or simply to trigger an alarm if certain specified results are obtained (e.g., if the signal exceeds a certain value for a specified length of time, or if a peak is detected at a specified

position or height, etc.).

Dealing with variable data arrays in spreadsheets

When applying spreadsheet templates of the type described in this book to your own data, you will need to modify the templates to accommodate different numbers of data points or components. This can be tedious to do, especially because you need to remember the syntax of each of the spreadsheet functions that you want to modify. This section describes ways to construct spreadsheets that *automatically* adapt to different data sets, without your taking the time and effort to modify the spreadsheet formulas for each case. This involves employing some less commonly used built-in functions in Excel or OpenOffice Calc, such as MATCH, INDIRECT, COUNT, IF, AND, and IF.

The MATCH function. In signal processing using spreadsheets, it is common to have x-y arrays of data of variable length, such as spectra (x=wavelength, y=absorbance or intensity) or chromatograms (x=time, y=detector response). For example, consider this small array of x and y values pictured in the spreadsheet fragment on the left, which shows an *independent variable x* in column A and a *dependent variable y* in column B. Spreadsheet formulas normally refer to cells by their row and column address, but for an x-y data set like this, *it is more natural to refer to a data point by its independent variable x, rather than by its row and column address*. For example, suppose you want to select the data point corresponding to x=2, no matter what cells they inhabit. You can do that with the MATCH function. For example, if you enter the desired x value into cell B2 (e.g., 2), then MATCH(B2,A5:A11)+ROW(A5) will return the row number of that point, which is 6 in this case. Later, if you were to move or expand this table, by dragging it or by inserting or deleting rows or columns, the *spreadsheet will automatically adjust* the MATCH function to compensate, returning the new row number of the requested point.

	A	B
1		
2		
3		
4	x	y
5	1	5
6	2	5.9
7	3	7
8	4	8
9	5	9.1
10	6	10
11	7	11

The INDIRECT function. The usual way to reference the value in a cell is to specify its row and column address. For example, in the array of x and y values pictured above, to refer to the contents of column B, row 6, you could write "=B6", which in this case will evaluate to 5.9. This is referred to as "**direct**" addressing. In contrast, to use "**indirect**" addressing you can write "=INDIRECT("B"&A1)", then put the number "6" in cell A1. The "&" character is simply "glue" that joins "B" to the contents of A1, so in that case "B"&A1 evaluates to "B6" and the result is the same as before: the contents of cell B6, which is 5.9. *However,* if you change cell A1 to 9, then "B"&A1 would evaluate to "B9", and the result would be the contents of cell B9, which is 9.1. In other words, the INDIRECT function allows the addresses of cells to be *calculated within the spreadsheet* rather than being typed in as a fixed number. This makes it possible for spreadsheets to adjust their own addresses based on a calculated result, for example to adjust their calculations to fit the number of data points in that data set.

These examples were done in what is called the "A1" reference style, where the columns are referred to by letters. It is also possible to use the "R1C1" reference style, where *both* the rows and the columns are referred to by *numbers*. For example, "=INDIRECT("R"&A2&"C"&A1,FALSE)", with the row number in A2 and the columns number in A1. (The "FALSE" just means that the "R1C1" reference style is used).

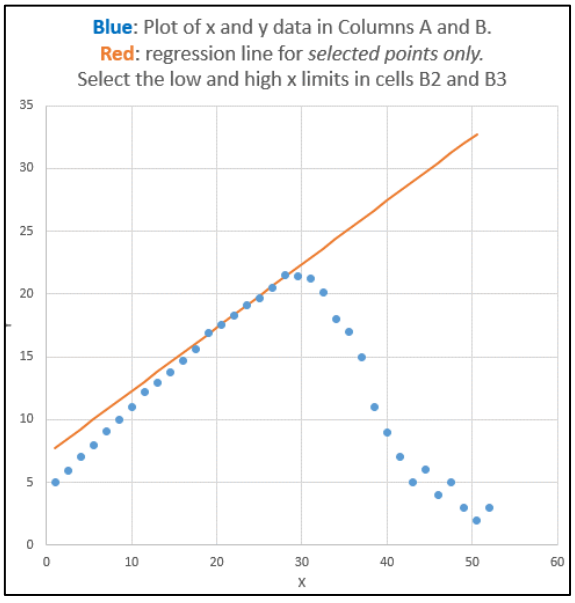
You can use the same technique to compute *ranges* of cell addresses. For example, suppose you

wanted to compute the sum of all the numbers in column B between a specified first row and specified last row. If you put the first row-number in A1 and the last row-number in row A2, the address of the *first* cell would be "B"&A1 and the address of the *last* cell would be "B"&A2. So, you would form the range of cell addresses by using "&" to glue together those two addresses, with a ":" character in-between ("B"&A1&":B"&A2). The sum would be SUM(INDIRECT("B"&A1&":B"&A2)), which is 56. Yes, it is longer, but the advantage over direct addressing is that you can adjust the range by changing just two cells rather that retyping the formula. It is the same for other functions that need a range of cells, such as AVERAGE, MAX, MIN, STDEV, etc. For examples of its use, see page 54.

For functions that require *two* ranges, separated by a comma, you can use the same technique. Suppose you want to compute the slope of the linear regression line between the x values in column A and the y values in column B in the spreadsheet excerpt on the previous page, using the built-in SLOPE function. SLOPE requires two ranges, first the dependent (y) values and them the independent x values. By *direct* addressing, the slope is SLOPE(B5:B11,A5:A11). By *indirect* addressing, you need two separate "indirect" functions, one for each range, separated by a comma. Here is what it looks like all together: SLOPE(INDIRECT("B"&A1&":B"&A2),INDIRECT("A"&A1&":A"&A2)), where the x values are in column A, the y values in column B, and the first and last row numbers are in cells A1 and A2 respectively. It works the same for the two related functions that calculate the INTERCEPT and RSQ (the R^2 value) of the regression line. I agree that it is confusing to read at first, but it works.

A working example. An example of the use of the MATCH and INDIRECT functions working together is demonstrated in the spreadsheet "SpecialFunctions.xlsx" (Graphic), which has a larger table of x-y data stored in columns A and B, starting in row 7. The idea here is that you can select a limited range of x values to work with by typing in the *lowest x* and the *highest x* value in cells B2 and B3, the two cells with a yellow background. The spreadsheet uses the MATCH functions in cells F2 and F3 to compute the corresponding row numbers, which are then used in the INDIRECT functions in the "Properties of selected data range" section to compute the maximum, average, and average of x and of y, and also the slope, intercept, and R^2 values of the y *vs* x linear regression line (page 161) over that selected x interval. The regression line, fitting *only* the data from x=20 to 29, is shown on red in the graph on the right, superimposed on the complete data set (blue dots). By simply changing the x-axis limits in cells B2 and B3, *the spreadsheet and the graph re-calculates, without you having to edit any of the cell formulas*. Try it yourself. (By the way, you can float your mouse pointer over any cell with a red mark in upper right corner to reveal its cell formula or an explanation). Columns J and K of this sheet also show how to use the "IF" and "AND" functions to

	A	B	C	D	E	F
1	**Select data range**			Use of MATCH function		
2	First x value	20.0	first selected row number =			19
3	Last x value	29.0	last selected row number =			25



copy data from columns A and B into columns J and K only those data points that fall between the two specific x limits.

If desired, you can add more data to the end of columns A and B, limited only by the range of the MATCH functions in cells F2 and F3 (which are initially set to 1000, but that could be as large as you need). The total number of numerical values in the data set is computed in cell I15, using the "COUNT" function (which, as the name suggests, counts the number of cells in a range that contains numbers).

Measuring peak location. A common signal processing operation is finding the x-axis value where the y-axis value is maximum. This can be broken down into three steps: (1) determine the maximum y value in the selected range with the MAX function, (2) determine the row number in which that number appears with the MATCH function, and (3) determine the value of x in that row with the INDIRECT function. These steps are illustrated in the same "SpecialFunctions.xlsx" spreadsheet in column H, rows 20-23. The result is that the maximum y (21.5) occurs at x=28. The three steps can even be combined into one long formula (cell H23), although this is harder to read than the formulas for the separate steps. The peak finder spreadsheet discussed on page 273 uses this technique.

The LINEST function. Indirect addressing is particularly useful when using *array functions* such as LINEST (page 178) or the matrix algebra functions (page 191). The demonstration spreadsheet "IndirectLINEST.xls" (Graphic link) shows how this works for the multiwavelength spectroscopy analysis of a mixture of three overlapping components by the CLS method (page 188). The measured mixture spectrum is in column C, rows 29-99 and the spectra of the three pure components are in columns D, E, and F. Cell C12 "=COUNT(C29:C1032)" counts the number of rows of data (i.e., number of *wavelengths*) in column C starting at row 29, and cell G3 counts the number of *components* (in this case 3). These are used to determine the first and last row and column for the indirect addresses in LINEST in cell C17. The measured peak heights calculated by LINEST for the three peaks are given in row 17, columns C, D, and E, and the predicted standard deviations are in the row below. In this spreadsheet the data are simulated (in columns O – U), so the true peaks heights are known and therefore the *absolute accuracy can be calculated* (row 26, C, D, and E) and compared to the predicted standard deviations. Press the **F9** key to recalculate with an independent noise sample, which is equivalent to taking another measurement of the same sample. Because of the use of INDIRECT addressing, you can add or subtract data points at the end of columns C – E and the calculations work with no other changes. For examples of its use in signal processing, see page 193.

Illuminating the invisible: Computer simulation of instruments

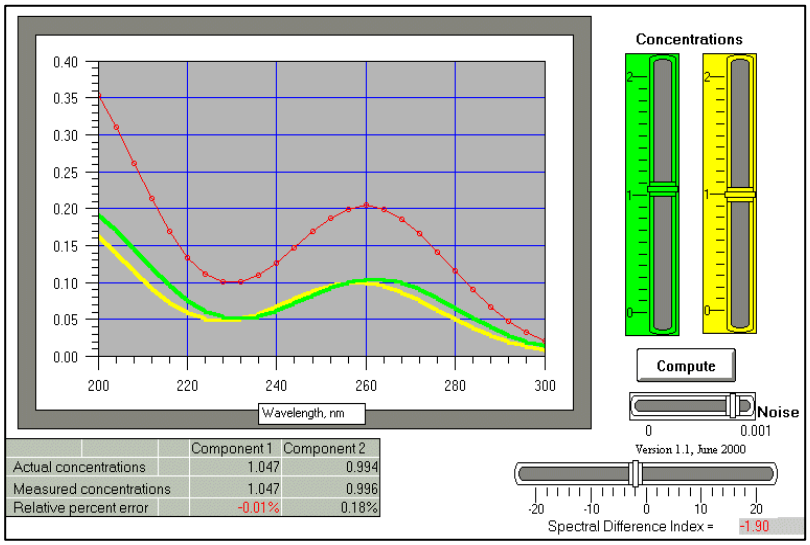
Throughout this book, I have often used computer simulations to test, demonstrate, and determine the range of applicability and the accuracy of various signal processing techniques. The aim is to generate realistic computer-simulated signals by adding together

(a) known *signal* component, such as one or more peaks, pulses, or sigmoidal steps,

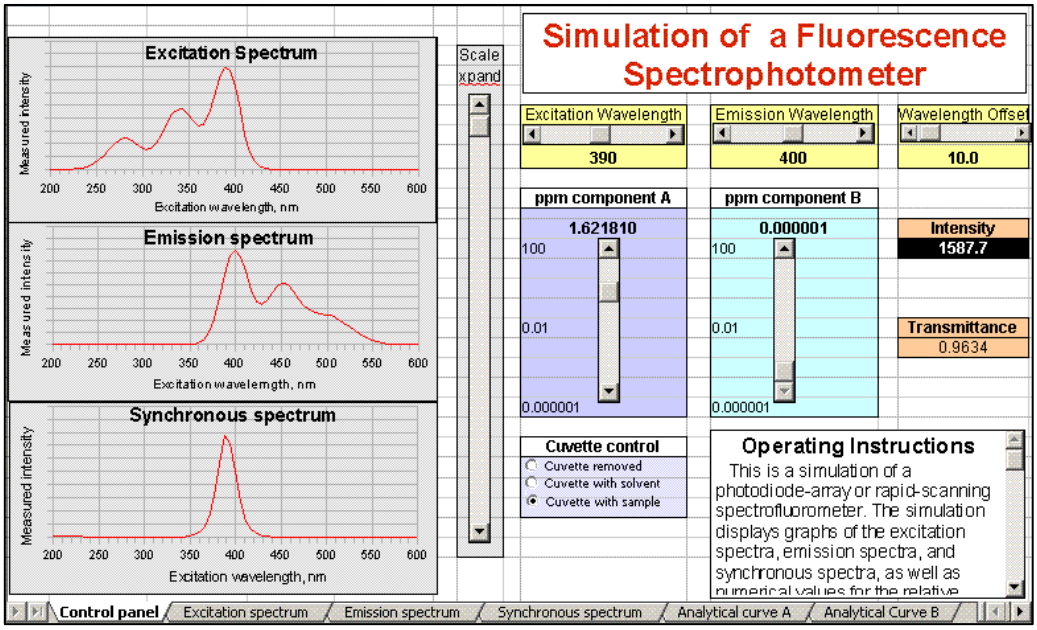
(b) a *baseline*, which may be flat, sloped, curved, or stepped, and

(c) random *noise*, (page 24), which may be various colors (page 27) and amplitude dependences (page 28).

This can be done either in Matlab/Octave, using the built-in and downloadable functions (page 479) for various peak shapes and types of random noise, or in spreadsheets, which can also be used to create attractive and intuitive user interfaces. Matlab "apps" and common spreadsheets have a built-in way to create a "GUI" (Graphic User Interface) with buttons, sliders, drop-down menus, etc. Some spreadsheet examples that I have created include SimulatedSignal6Gaussian.xlsx, PeakSharpeningDemo.xlsx, PeakDetectionDemo2.xls, TransmissionFittingDemoGaussian.xls, BeersLawCurveFit2.xls, and RegressionDemo.xls (below, on the left).



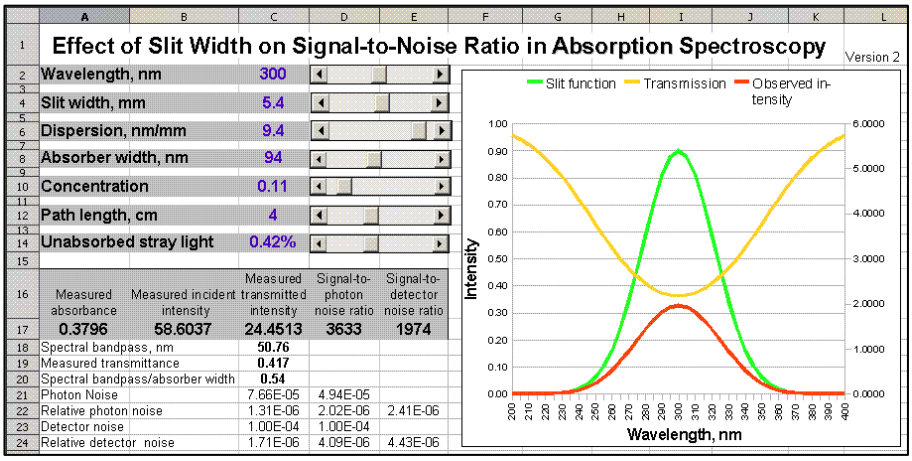
It is possible to make any aspect of a computer-generated signal randomly variable from measurement to measurement, with the aim of making the simulation as close as possible to the real signal behavior that you may have to measure. For example, in the section "The Battle Rounds: a comparison of methods" on page 298, the signal to be measured is a Gaussian peak located near the center of the recorded signal, with a fixed shape and width. The baseline, on the other hand, is highly variable, both in amplitude and in shape, and there is also added white noise. In another simulation, "Why measure peak area rather than peak height?", page 314, the signal peak itself is



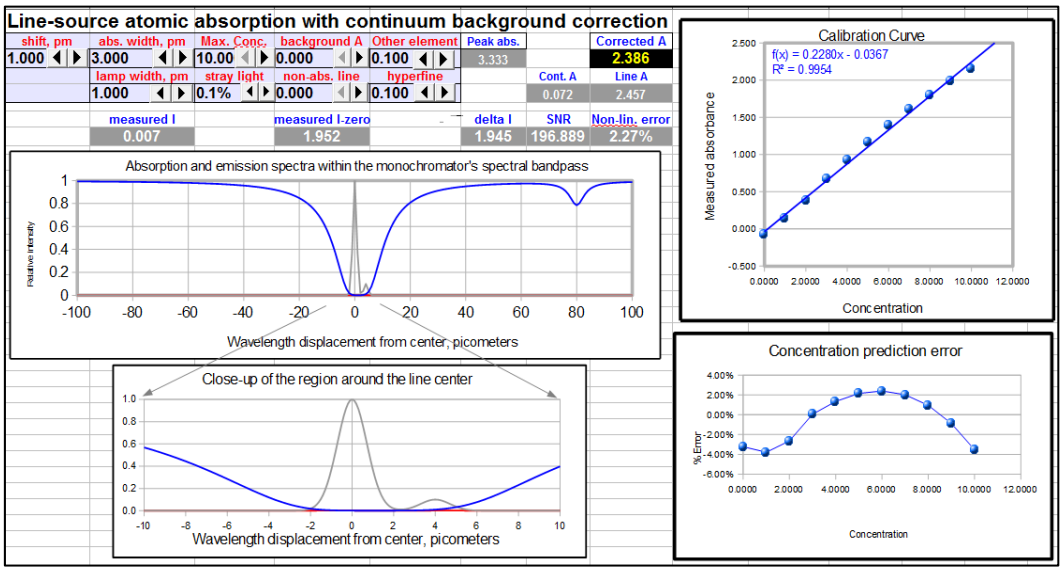
subject to a variable broadening process that causes the measured peak to be shorter and wider, but which has no effect of the total area. In the section "Measuring a buried peak", page 322, the signal is a small "child" peak that is buried under the tail of a much stronger "parent" peak. In all these cases, the *true underlying signal* is known to the software, so that, after the software measures the simulated *observed signal* with all its baseline and noise variability, it can calculate the error of measurement,

allowing you to compare different methods or to optimize the method's variables to obtain the best accuracy.

In some cases, it may be possible to simulate important aspects of an *entire measurement instrument* system. Several examples are shown here and in https://terpconnect.umd.edu/~toh/models/. This is most useful if both the signal magnitude and the noise can be predicted from first principles. For



example, in optical spectroscopy, the principles of physics and of geometrical optics can be used to predict the intensity of an incandescent light source, the transmission of a monochromator, and the signal generated by a photomultiplier, including the photon noise. When these are combined, it is possible to simulate the fundamental aspects of such instruments as a scanning fluorescence spectrometer (pervious page) or an atomic absorption instrument (below), to predict the analytical



calibration curves of absorption spectroscopy, to compare the theoretical signal-to-noise ratios of absorption and fluorescence measurement, and to predict the detection limits of atomic emission measurement of various elements, and the effect of slit width on signal-to-noise ratio in absorption spectroscopy (above). You can also simulate the operation of a lock-in amplifier (page 320), a wavelength modulation spectroscopy system, and even basic analog electronic and operational amplifier circuits. Note that *these are not simulations of particular commercial instruments* that might be used to train instrument operators. Rather, they are interactively manipulated mathematical models

that describe various parts of or aspects of each system, for the purpose of *illuminating hidden aspects* of their internal operation.

Who uses this book, its web site, documents, and software?

Since its inception, this book and the associated web site (http://terpconnect.umd.edu/~toh/spectrum/) and according to Google has been accessed from at least **206 countries** and several non-region-specific categories (e.g. satellite providers). These include many countries in the developing world, some very small countries (e.g. Liechtenstein, the Faroe Islands), relatively isolated countries (Cuba, North Korea, Myanmar/Burma), and even some war-torn regions (Afghanistan, Syria, Iraq, and Ukraine). Government control of Internet access is often an issue. For example, I've got fewer views from Cuba than from many other Spanish-speaking countries with *smaller populations*, such as Bolivia, Dominican Republic, Costa Rica, Puerto Rico, Panama, and Uruguay, even though Cuba has many active scientists, especially in the medical and pharmaceutical fields.

The first Web version went up in 1996, but I didn't start keeping track of page views until 2008. Since then there have been over *2 million page views*. The distribution of page view counts among countries is very long-tailed, with one-third of the views coming from the USA (except during major US holidays), half of the views coming from only 5 countries (USA, India, Germany, United Kingdom, and China) and 99% of the views coming from only 39 countries. Among the countries that have a relatively large number of page views *relative to their populations* are the USA, Germany, UK, Canada, Australia, Netherlands, Switzerland, Singapore, Israel, Belgium, Taiwan, South Korea, and Scandinavia. (Another web site of mine on a related subject, Interactive Computer Models for Analytical Chemistry Instruction, had got an additional 820,000 views at last count in 2017).

The Internet Service Providers with the largest number of views are Comcast, Verizon FIOS, Time Warner, Cloudflare, At&t U-verse, Deutsche Telekom (Germany), BSNL (India), and Cox Communication. Most views worldwide come from Windows machines, about 20% from Linux and Macintosh, and 10% from mobile devices. I have made efforts to make my pages more usable from mobile devices like smartphones.

About one-quarter of the views come *directly from educational institution ISPs* that have "School", "Ecole", "College", "Hochschule", "Univ...", "Academic", or "Institute of Technology" in their names. (The number of educational users is certainly larger than that because some users are no doubt accessing from other ISPs in homes or businesses). An analysis of 200,000 views in 2015 showed that the biggest educational users have been the University of California System (UCLA, Berkley, etc.), Indian Institute Of Technology system, the University of Texas system, Massachusetts Institute Of Technology, the University of Michigan, the University of Maryland (my home institution), Delft University of Technology (Netherlands), Stanford University, China Education And Research Network Center, the University Of Wisconsin System, and the University of Illinois.

Many of the large national laboratories are users, including Bell Canada, Oak Ridge, Pacific Northwest, Lawrence Livermore, Sandia, Brookhaven, National Renewable Energy Laboratory, SLAC, Fermilab, Lawrence Berkeley, NRC Canada, CERN, NIST, NASA, JPL, and NIH.

The most popular pages on the site at the time of this study were Peak Finding and Measurement,

Smoothing, Integration, Deconvolution, InteractivePeakFitter, and Signal Processing Tools. About 50% of the page views originate from search engines (80% of those using Google). The most common search keywords used are: "peak area", "convolution", "deconvolution", "peak detection", "signal processing pdf", "findpeaks matlab", "Fourier filter", and "smoothing". About 40% of the traffic comes from direct links (bookmarks or typed URLs) and about 10% comes from referring websites, usually from Wikipedia or from MathWorks. Unfortunately, page loads and search terms have become almost completely encrypted in recent years, so I can no longer tell which pages are being viewed and what is being downloaded. (Interestingly, that is not the case with Interactive Computer Models for Analytical Chemistry Instruction, which has only 75% encryption).

There were over 100,000 downloads of my software and documentation files, averaging several hundred file downloads per month, from both my web site and from my files on the Matlab File Exchange. The most commonly downloaded files are IntroToSignalProcessing.pdf, PeakFinder.zip, ipf12, CurveFitter....xlsx, iSignal, ipeak, PeakDetection.xlsx, and the complete site archive SPECTRUM.zip.

What factors influence the number of page views from different countries? The tools of data analysis, specifically regression (for example, using LINEST), can help answer this question. Obviously, one would expect that a country's population would be a factor, but it turns out that *the correlation between page view and population is very poor*, with a coefficient of determination (correlation coefficient or R^2 value) of only 0.36 (n=163 countries. Over 160,000 total page loads over the period from 2008 to 2017; graphic on next page). Note that because of the very large range of population sizes, I did a *log-log* correlation (page 467) to prevent the results from being totally dominated by the top few countries.

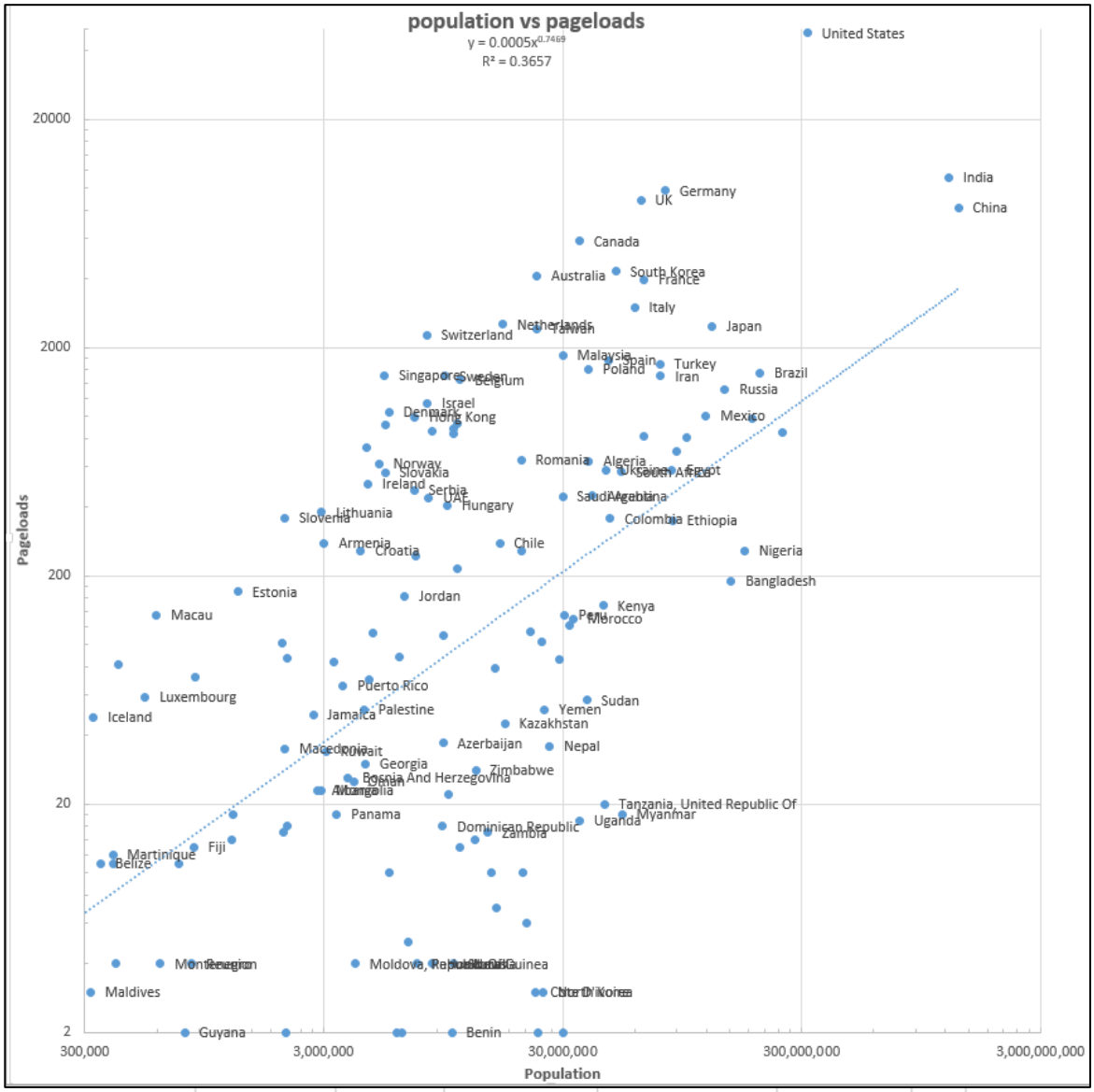
I also investigated the effect of other factors that might be more specific to the language and subject matter of my particular site, including

- the number of *English speakers* in each country,
- the number of *Internet users* in each country,
- the number of *universities* in each country, and
- the total *research and development budget* of each country.

All that information is freely available on the internet for *most* (but not all) of the countries (graphic link). By a good margin, *the most influential factor was the* ***research and development budget,*** for which the R^2 value was 0.76. This is perhaps not surprising given that my site concerns a very narrow and specialized topic: the technical aspects of computerized scientific data processing.

A log-log *multilinear regression* on all 5 of these factors together yielded an R^2 value of 0.84 (n=53 countries for which *all 5* factors were reported), which is a modest improvement over the research and development budget alone. (Since these calculations were made in 2017, page views from China have risen substantially and are now typically second to those from the USA).

For an Excel spreadsheet with all these data and calculations (between 2008 and 2015), see FinalCountriesSummary.xlsx



Satellite Provider	6	
Asia/pacific Regior	105	
Anonymous Proxy	64	
Europe	955	
Non-country hits	1130	1%
Country hits	159297	

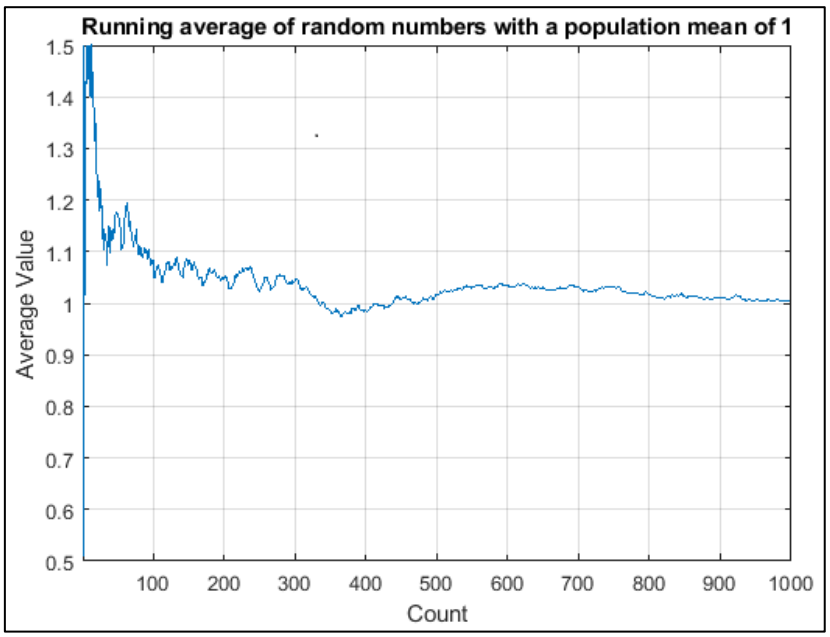
Log-log correlations to hits	n	R2
population	163	0.4
English speakers	92	0.579
Number of universities	146	0.6814
Internet users	145	0.6812
R&D spending	72	0.75
Patent applications, 2008-2012	133	0.591

What fields of study are represented? The users of my site include students, instructors, workers, and researchers in industry, environmental, medical, engineering, earth science, space, military, financial, agriculture, communications, and even music and linguistics. This conclusion is based on emails I have received, on the titles of journal articles that have cited my work, and on the ISPs of major web visitors. Judging from the ratio of downloads to emails, most people who have downloaded my software do not write to me about what they are doing, which of course is completely understandable. Also, of the people who do write to me, most do not tell me specifically what their

applications are, which is their prerogative. As a result, those sources give me incomplete information about the application areas where my programs are being applied. A much better indication of the width of applications can be got by looking at the titles of the over *750* published papers and patents that have cited my web pages and book (page 519).

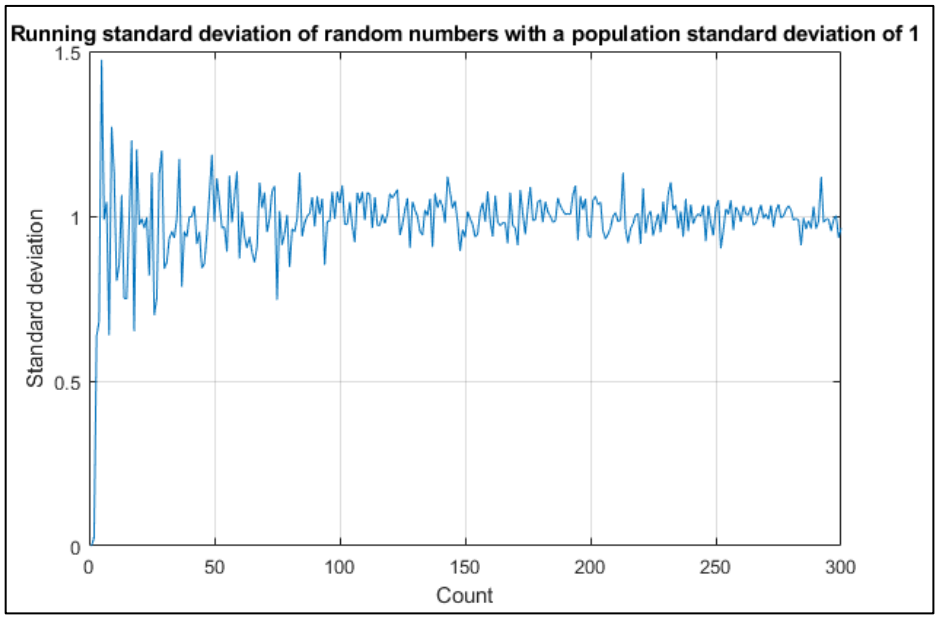
The Law of Large Numbers

The Law of Large Numbers is a theorem that describes large collections of numbers or observations that are subject to independent and identically distributed random variation, such as the result of performing the same experiment or measurement many times. The average of the results obtained from many trials should be close to the long-term value (called the "population average") and will tend to become closer as more trials are performed. It is an important idea because it guarantees stable long-term results for the averages of some random events. This is the reason gambling casinos can make so much money. Their games are designed to give the casino a significant advantage in the long run but highly variable results in the short term, guaranteeing plenty of (noisy) winners that tend to encourage the gamblers, as well as enough (quiet) losers so that the casino can make money. And that's why investors in the stock market make money in the long run, despite the unpredictable day-to-day variation, up one day and down the next. It's also why it is so hard to see *climate* change in the much wilder short-term hot and cold day-to-day and year-to-year swings in the *weather*.



But the idea that "the average will tend to become closer as more trials are performed" does *not* mean that the average becomes *steadily and irreversibly* closer. In fact, the average can wander around quite a bit as more data are included. Take the example on the left, which shows the running average of a set of normally-distributed independent random numbers whose population average is 1.000 and whose standard deviation is 1.000, as more and more numbers from that population are averaged, up to 1000. (This is generated by the simple Matlab script RunningAverage.m). Note that the average wanders around, reaching and crossing over the true population average (1.000) twice in this case before ending up near 1.0 after 1000 points are accumulated. But, full disclosure, I cheated a bit and gave you a relatively good example. In fact, if you ran this script again, the final average will most likely *not* be so close to 1.0. The predicted standard deviation of the average of *all* 1000 random numbers is reduced by a factor of 1/sqrt(1000), which is about 0.031, or 3% relative, meaning that most results will only fall within plus or minus 6% of the true average, that is, from 0.94 to 1.06.

The uncertainty of uncertainty. The situation is even more dire if you wish to estimate the *standard deviation* of a population from small samples. The graphic on the left shows the Matlab script RunningStandardDeviation.m. which calculates the standard deviation for the same population in the previous example. The sample standard deviation wanders around alarmingly for small samples and only settles down slowly. Even worse, the standard deviation for very small samples is biased down, often returning values far lower than usual. (For example, the calculated standard deviation of two data points drawn from a normal distribution is almost always lower that the true population standard deviation).

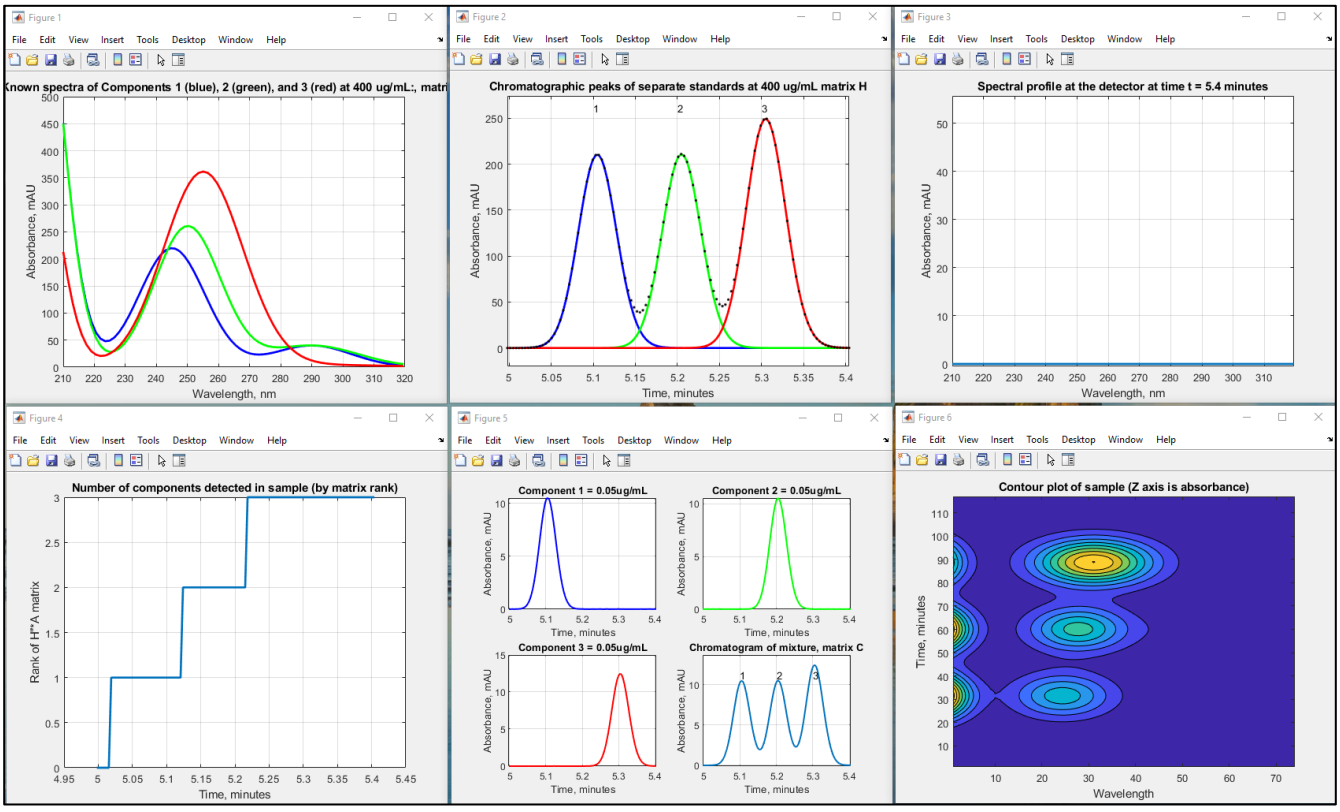


There is a well-documented tendency for people to *overestimate* the quality of small numbers of measurements, sometimes referred to as hasty generalization, or insensitivity to sample size, or the gambler's fallacy. This is related to the field of study of a famous pair of psychologists named Amos Tversky and Daniel Kahneman, who collaborated in a long-running study of human cognitive biases in the 1970s. They formulated a hypothesis that many people erroneously tend to believe in a false "Law of Small Numbers", which is the name they coined for the mistaken belief that a small sample drawn from a large population is representative of that large population. We would like to believe that scientists are immune to these foibles and that they always think logically and correctly. But scientists are only human, so it pays to be aware of this tendency, *particularly when a small sample of data supports your favorite hypothesis*. It is tempting to stop there, "while you're ahead". This is called "confirmation bias". Avoid it like the plague.

Of course, in many practical experimental measurements, you may really be limited to a small number of repeated measurements. There may be a fixed number of data points and no possibility of gathering more. Or the cost of gathering more data, in terms of money or time, may be excessive. For example, the process of calibrating an analytical instrument for quantitative measurement (page 337) may involve the preparation and measurement of several standard samples or solutions of known composition. If the calibration curve (the relationship between instrument reading and sample composition) is non-linear, it takes several different standards to define the curve, the more the better. But you have to consider not only cost of preparing many standards but also the cost of cleaning up and safely storing or disposing of the (potentially hazardous) chemicals afterwards. In other words, you may have to accept a smaller number of standards that would be ideal. The bottom line is, if you are limited to a small number of data points, do not over-estimate the quality of your results. To determine uncertainty ranges for a set of data using the 3-sigma rule (page 33), the distribution must be normal (Gaussian) *and* you need to know the standard deviation. For small sets of data, *both* are uncertain.

Spectroscopy and chromatography combined: time-resolved Classical Least-squares

The introduction of high-speed UV-Visible array detectors into high-performance liquid chromatography (HPLC) instruments has significantly increased the power of that method. The speed of such detectors allows them to acquire a complete UV-Visible spectrum multiple times per second over the entire chromatogram. In this section, I will use the data in a published technical report from *Shimadazu Scientific Instruments* which describes the separation of three positional isomers of methyl acetophenone: o-methyl (o-MAP), m-methyl (m-MAP), and p-methyl (p-MAP) by liquid chromatography with a diode-array UV detector. The ultraviolet absorption spectra of each of these three isomers at a concentration of 400 µg/mL is shown below on the left. The spectra are distinct but highly overlapping. The chromatographic separation, using the column and conditions specified in their report, is shown in the middle. In this case the chromatographic peaks are sufficiently resolved that the spectra of each component can be estimated at its peak without significant interference from neighboring peaks. The Shimadazu report describes their proprietary software, which uses a complex iterative approach to extract the spectra and chromatographic characteristics from the raw data.



Here I present a simpler non-iterative calculation technique based on the same chemical system, in which I consider each spectrum acquired by the detector as a separate sample mixture and apply the Classic Least-squares (CLS) method previously introduced (page 188), which requires that the *spectra of the components are known beforehand* and where adherence to the Beer-Lambert Law is expected. The spectra and chromatographic peaks are simulated digitally in the Matlab/Octave script TimeResolvedCLS.m, shown in the figure below, by modeling the spectrum of each component as the sum of three Gaussian peaks and the chromatographic peaks as exponentially modified Gaussians. This

is a "*data-based* simulation": the parameters were carefully adjusted to match the graphics in the technical report as closely as possible, to make this simulation as realistic as possible. The other parameters, such as the spectral resolution, sampling rate, and detector noise (stated to be 2 milli-absorbance units), were also based on that report. Note that the chromatographic peaks (middle figure) are not baseline resolved. Therefore, it is to be expected that quantitative calibration based on the measurement of peak areas in this chromatogram (for example by the perpendicular drop method, page 146, might be inaccurate, especially if the peak heights are very different. In fact, in this case, even though the concentrations of the three components are much lower (0.05 μg/mL for each), the peak areas measured by perpendicular drop are fairly accurate, differing only about 2% from the true values, mainly due to the slight asymmetry and nearly equal height of the three peaks. The spectra (left-hand figure) are even more highly overlapped than the chromatographic peaks, but they are *distinct in shape*, and that is the key.

Basically, you can treat this as a series of 3-component CLS calculations, one for each time-slice of the detector. The actual calculations can be done in two ways, depending on whether the spectra are processed one by one ("Alternative calculation #1", lines 113-146) or are collected for the entire chromatogram and then processed all at once ("Alternative calculation #2", lines 150-170). The first method looks like chromatography as the script executes. It computes the chromatographic peaks of the three components point by point as they evolve in time and plots them in the first three quadrants of figure window 3 (on the right). The second method calculates the entire chromatogram in one step at the end and makes the same final plots. (The second method is faster computationally, but that is not significant because it is the *chromatography* that takes all the time, not the *calculations*). Either way, the result is the same. The chromatographic peaks of the three components are *completely separated mathematically*, so their areas are easily measured, *no matter how much they overlap in time*. Note that, although the three *spectra* must be known, no knowledge of the chromatography peaks is required. They emerge separate and intact from the data, computationally.

Stress test. The advantage of a data-based simulation like this is that you can test how the method might work in more challenging applications by modifying the signal simulation parameters. To test the abilities and limitations of this method, I prepared a series of increasingly challenging scenarios, starting with the original experimental system pictured above and then making it *progressively more difficult* by changing the parameters of the simulation. Six scenarios are listed in the table below, along with the typical percent errors in peak area measurement by the CLS method and with links to the corresponding graphic and to the Matlab/Octave m-files. Each is a more challenging variation on the original methyl acetophenone analysis:

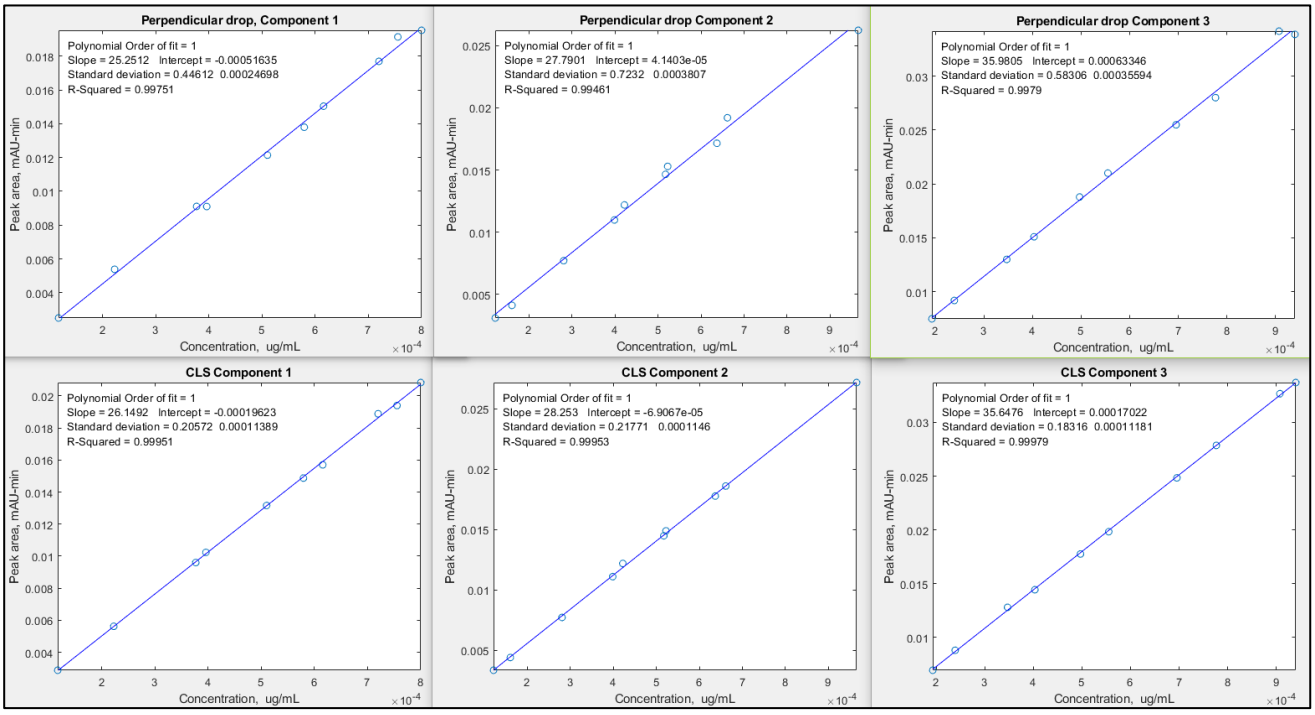
#2 has much more chromatographic peak overlap,
#3 has more asymmetrical peaks (much higher *tau*),
#4 has much more similar spectra, making them look identical,
#5, component 2 (the middle peak) has a concentration 100 times lower, and
#6 is the same as #5 except that the peaks are highly asymmetrical.

In all these cases, the normal perpendicular drop area measurement technique is either impossible (because there are no distinct peaks for each component) or is very much in error, but the CLS technique works well, giving very low errors, except when the middle peak concentration is 0.0001

ug/mL, which approaches the 2 mAU random noise limit of the detector. (Another variation, in the script TimeResolvedCLSbaseline.m, includes continuous correction for baseline drift by adding a 4th flat spectral component to account for the possibility of bubbles or turbidity in the light path of optical detectors).

Peak resolution	Spectral similarity	Peak asymmetry	Concentration ug/mL	Typical percent errors in area measurement	Links
1. Normal	Normal	Slight: tau=10	.05 .05 .05	.002% .002% .0016%	Graph mfile
2. Unresolved	Normal	Slight: tau=10	.01 .01 .01	-.06% -.053% -.041%	Graph mfile
3. Blended	Normal	Great: tau=40	.05 .05 .05	-.0004% -.013% -.066%	Graph mfile
4. Unresolved	Very close	Slight: tau=10	.01 .01 .01	.054% .049% .04%	Graph mfile
5. Unresolved	Very close	Slight: tau=10	.01 .0001 .01	0.026% 2.4% 0.019%	Graph mfile
6. Unresolved	Very close	Great: tau=40	.01 .0001 .01	-0.04% -3.8% -0.03%	Graph mfile

Even when the peaks are sufficiently resolved for the perpendicular drop method to work, it can suffer from an interaction between adjacent peak heights. A change in the peak height of one peak can affect the measurement of the area of adjacent overlapped peaks, because of shifts in the valley point between them. This is illustrated by TimeResolvedCLScalibration.m, which simulates the calibration curves (concentration vs peak area) for a three-component mixture similar to the above but modified so there is always a valley between the peaks, and then allows the three components to vary independently and randomly over a 2 x 10^{-4} to 9 x 10^{-4} µg/mL range. (Each time you run this you will get a different mix of concentrations). A typical set of calibration curves are shown below. In this case, the average absolute percentage error in area measurement is about 5% for the perpendicular drop method, with an R^2 of 0.995, but it is *less than 1% for the CLS measurement*, with an R^2 of 0.9995, a big improvement.



The CLS method is clearly very effective, but this proves only that the underlying *mathematics* works well. The method *still* requires that the spectra of all the components be known accurately. This requirement can be met in some applications, but in liquid chromatography there is a potential pitfall. If gradient elution and/or temperature programming are used *and* if the spectra of those chemical compounds are sensitive to the solvent and/or to temperature, possibly shifting their peaks slightly, then there will likely be additional errors in the CLS procedure. Obviously, this depends on the chemical system and will have to be evaluated on a case-by-case basis.

In other applications, some or all the components may simply be unknown, and you may want to obtain their spectra. This can be done *in situ* if the peak separation is at least as good as that depicted on page 363, because at each peak maximum there is virtually no contribution from adjacent peaks.

But this suggests another interesting use for this method. If you have a chromatographic method that achieves baseline separation, you could use that to obtain accurate spectra of each in situ. Then you could modify the conditions to achieve a *faster* chromatogram, for example by using a different column length or faster flow rate. Even if doing that results in incompletely resolved chromatograms, you could apply this CLS method to achieve accurate results in a much shorter time for multiple samples.

But what if the peaks are even more overlapped, so that pure component spectra are never achieved? In that case, more sophisticated methods must be used, such as the one described in the Shimadzu technical report. This involves making initial estimates of spectral and chromatographic peaks, followed by an iterative search for the best fit to the experimental data, subject to the imposition of some important known prior constraints, such as non-negativity of spectra and of the chromatography peaks (those peaks are always positive, except for random noise on the baseline), and the unimodality of the chromatography peaks (that is, each component gives one and only one chromatography peak). Methods of this type are much more sophisticated and will be left to a future expansion of this book.

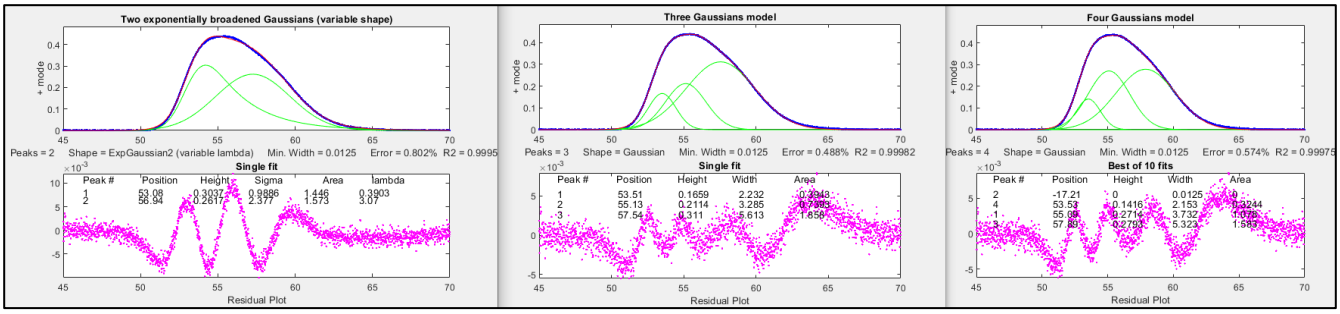
The mystery peak challenge

The objective of this exercise is to learn as much as you can about the underlying properties of a digitized signal using the signal processing tools in this book and, if possible, to obtain a mathematical description of the signal (script). At first glance, the signal (MysteryPeak.mat) appears to be a single, asymmetrical peak with a maximum at x=55.5. The signal-to-noise ratio seems to be very good - there's little visible noise unless you look very closely - and the signal begins and ends near zero, so baseline correction is likely not an issue. The bad news is that you do not know anything else. This might be a peak shape that is not describable mathematically. Or the asymmetry might be due to some asymmetrical process applied to an originally symmetrical peak shape, or it could be a group of closely spaced overlapping peaks, which is suggested by the faint bumps in the shape. Some quick preliminary curve fitting can be done using peakfit.m (page 397):

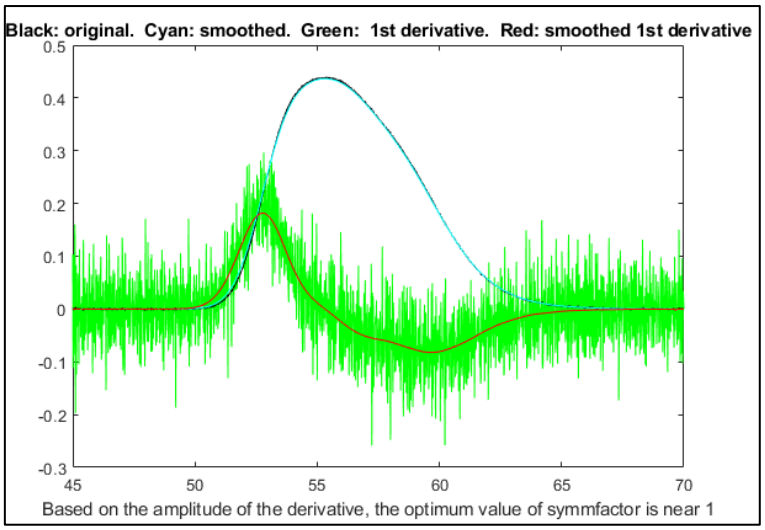
`[FitResults,GOF]=peakfit([x y],0,0,2,1)` for a two-Gaussian model (shape 1)

`[FitResults,GOF]=peakfit([x y],0,0,1,3)` for a single Logistic model (shape 3)

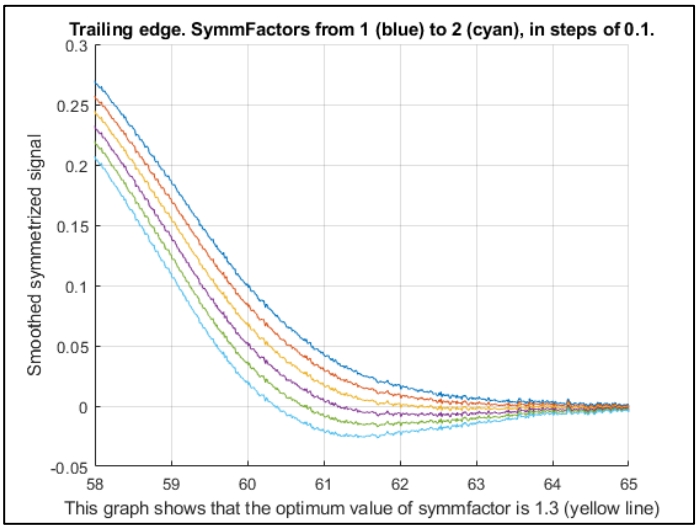
`[FitResults,GOF]=peakfit([x y],0,0,4,39,1)` for a 4 exponentially-broadened Gaussian model (shape 39)



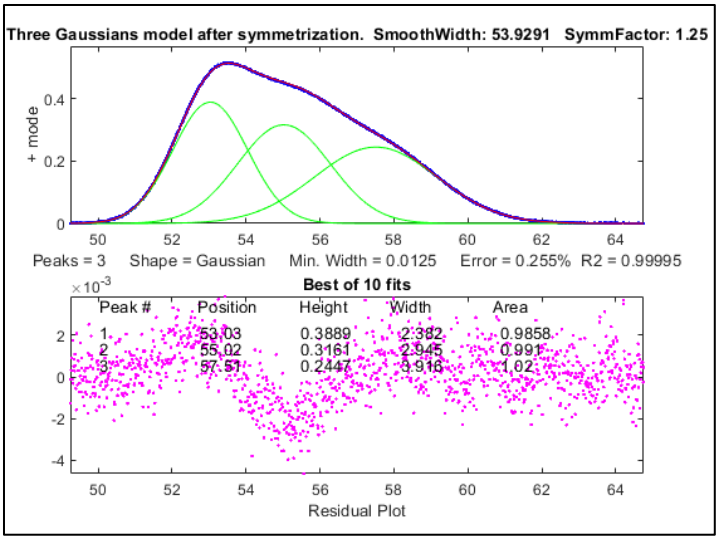
You could also use the interactive peak fitter ipf.m (page 417) for this purpose, so you can quickly change the model shape, number of peaks, starting guesses, data region to be fitted, etc., with single keystrokes and mouse clicks. Either way, the three initial fits in the figures above show that the signal contains a small amount of random noise, which appears to be white (so the signal has probably not been smoothed, which is fortunate) and which has a relative standard deviation of about 0.2%, based on 1/5th of the visual peak-to-peak value (page 24). But *unfortunately, these fits are not successful because their fitting errors (0.5 to 0.8%) are all significantly larger than the 0.2% random noise!* Trying different shapes and greater numbers of peaks does not help either, as it results in either higher fitting errors, unstable fits, or zero peak heights. There is just too much overlap for easy curve fitting (page 225).



Another approach to the problem of asymmetrical peaks is to use the technique of *first-derivative* symmetrization described on page 83. This applies specifically to *exponential* broadening, a common peak broadening mechanism. The idea is that if you compute the first derivative of an exponentially broadened peak, multiply it by a weighting factor equal to the time constant *tau* of the exponential, and add it to the original broadened signal, the result will be the original peak before broadening, which *makes the peak overlap less severe.* This works for any original peak shape. In this case you don't know *tau* beforehand, but you can try different values until the baseline after the peak is as low as possible but not negative, as shown in this GIF animation. This is easily done by using the symmetrize.m function, or interactively in iSignal, which has smoothing (**S** key), derivatives (**D**), symmetrization (**Shift-Y**), and curve fitting (**Shift-F**). The derivative of y with respect to x, by the "derivxy.m" function is quite noisy, as shown by the green line in the figure above. As usual, you must smooth the



derivatives of noisy signals to make them useful, but you must not over-smooth and distort the signals. As a rule of thumb, a smooth width equal to 1/10th of the number of data points in the halfwidth does not distort the signal visibly, as shown on page 60. Our peak has about 530 points, measured by `halfwidth(1:length(y),y)`, so a smooth width near 53 will not distort the signal peak, but it does eliminate most of the noise from the derivative (previous page, right). Also, you can see that the derivative, in y/x units, is comparable in numerical magnitude to the original signal, so the time constant *tau* (in x units) is probably somewhere near 1.0. Next, you add the product of the first derivative and *tau* to the original signal, looking at the trailing edge as you try six different *tau* values near 1.0. The graph above left shows that the optimum value is about 1.25. When this is applied to the entire signal, the result, shown on the right, has *more distinct bumps*. When that modified signal is used for curve fitting, you find that a 3-Gaussian model works quite well, with a fitting error of only 0.25%, close to the noise level. This is evidence that the signal consists of three closely spaced exponentially modified Gaussians (EMG). Normally there is no independent way to check the accuracy of the peak parameters so measured, but - full disclosure - *the signal in the case was not actually unknown* but rather was generated by the script MysteryPeaks.m. It does in fact consist of three EMGs, with peak maxima at x=53, 55, and 57.5, each with a time constant of 1.3, and all with peak areas of 1.0. The curve-fit results *after* symmetrization are within 0.1% for the peak positions and within 2% of the peak areas. In contrast, direct fitting of exponential Gaussians to the *original* data, using the *tau* you just determined, looks good (graphic) but gives *less accurate peak parameters* (and takes three times longer to compute).



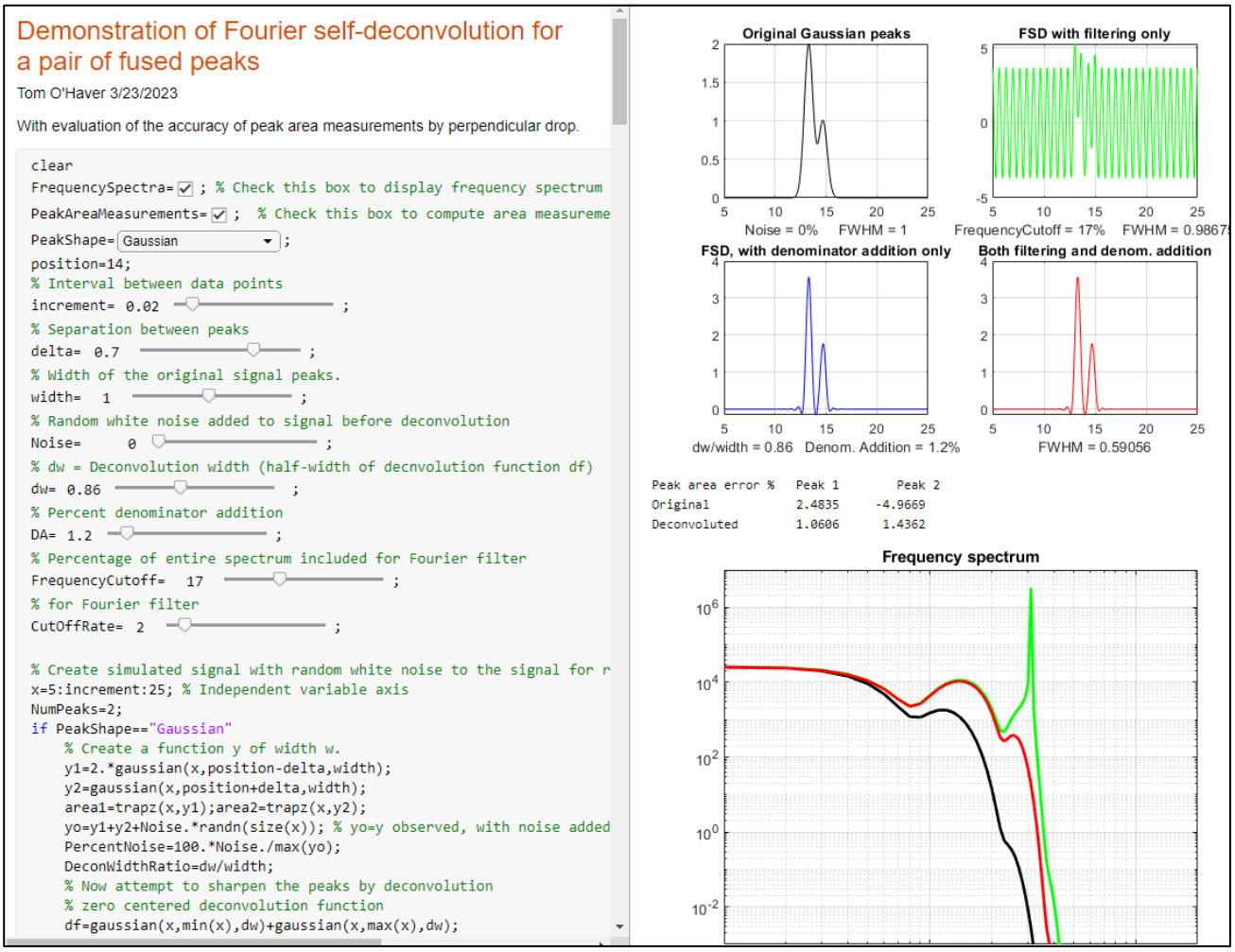
Developing Matlab Live Scripts and Apps

The usual way of developing programs in Matlab is to write scripts or functions (see page 36), developed and used in a screen environment similar that I have shown before. In these environments, you can change the values of parameters by typing them into the editor window or the command line. If there are many parameters and selections to explore, the cycle of editing and re-running can be slow and cumbersome, For example, the script "DenomAdditionDemo.m" demonstrates the Fourier deconvolution (page 114) of a pair of overlapping peaks, with the aim of increasing the resolution of the peaks. It has nine adjustable interacting variables. The sequence of computations is (1) create the simulated signal with two peaks and random noise, (2) create a zero-centered convolution function of the same shape and with variable width, (3) divide the Fourier transform of the simulated signal by the modified Fourier transform of a calculated convolution function of the same shape, (4) add a constant to the numerator of that division to reduce noise spikes (page 122), (5) compute the inverse transform that result, and (6) apply low-pass filtering to reduce noise. To explore the ability of this method, several factors can be adjusted: the shape, width, sampling interval, and separation of the two peaks,

the noise amplitude, the width of the deconvolution function, the denominator addition, and the filter cutoff frequency and rate – 9 variables. To do this, you would have to edit the script, save it, and re-run it, potentially many times. In this appendix, you will develop two different alternative ways to code this deconvolution demo with a more modern graphical user interface (GUI) that uses sliders, drop-down menus, check boxes, etc., to control the adjustable variables, rather than editing the script itself.

Matlab Live Scripts

A very easy alternative is to use Live Scripts (MATLAB R2016b and later). Live Scripts are documents that combine code, output, interactive controllers, and formatted text in a single environment. To convert the plain script DenomAdditionDemo.m into a Live Script, click **Editor**, **Save as**, **Save as type** > **Live Code files (.mlx)**. To add controllers for the assignment statements, (e.g. delta=0.64), select the numeric value (0.64), click **Control**, and select an appropriate control, for example a *numeric slider* or *spinner* for continuous variables. Then in the pop-up box, type in the min, max, and step values for that control. The editor adds a numeric slider: `delta= 0.64 ;` You can also use a *drop-down menu* for discrete selections, such as peak shape menu in the 4th line in the example below:



Shown above is a Live Script demonstration of self-deconvolution, computationally equivalent to the "DenomAdditionDemo.m" script. It has sliders for each of the 9 adjustable variables, which allows

those variables to be changed simply by sliding the pointers. Each slider has a numerical range that you can set to be in an "expected range" for that variable. In addition, the app has a pull-down menu to choose the peak shape (Gaussian, Lorentzian, etc.) and checkboxes for optional computation of frequency spectrum and of the peak area measurement accuracy. When any of these controls are changed, the app recalculates and updates the plots continuously as the slider pointer is moved. (Right-click on the right-hand panel and select "Disable synchronous scrolling"). If screen space is at a premium, you can hide the code from view by clicking on the **View** tab and clicking "**Hide Code**". For other details, see this page, or these examples. There is also a version for applying the method to experimental data stored on disk in .xlsx or .csv format, called DeconvoluteData.mlx (screen image). Just type in the name of your data file in the like that starts with "mydata=" near the top of the script. The script assumes that your x,y data are in the first two columns.

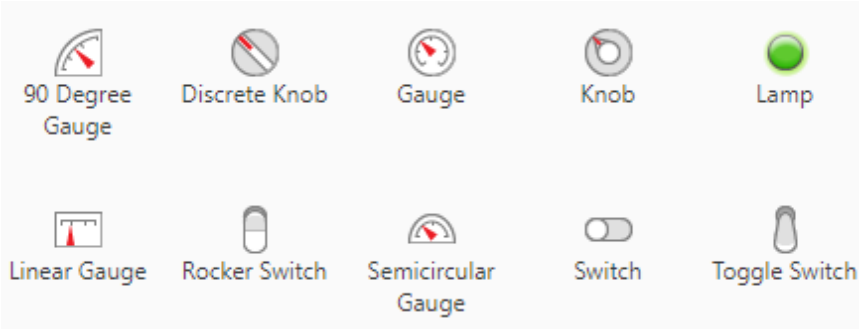
The table below compares several previously-covered keypress-operated interactive functions to Live Script versions that do essentially the same operations. Live scrips are easy to make and to use, but they are slightly slower in response to changes in settings. The keypress-operated functions are designed to respond virtually *instantly* in response to a keypress and they can be operated in full-screen mode (double-click the title bar), but they do require that you know the keys. In practice, though, you will quickly memorize the keys that you use the most and you will begin to appreciate the speed of the keypress-operated mode of operation.

Table of keypress-operated interactive functions and corresponding Live Scripts

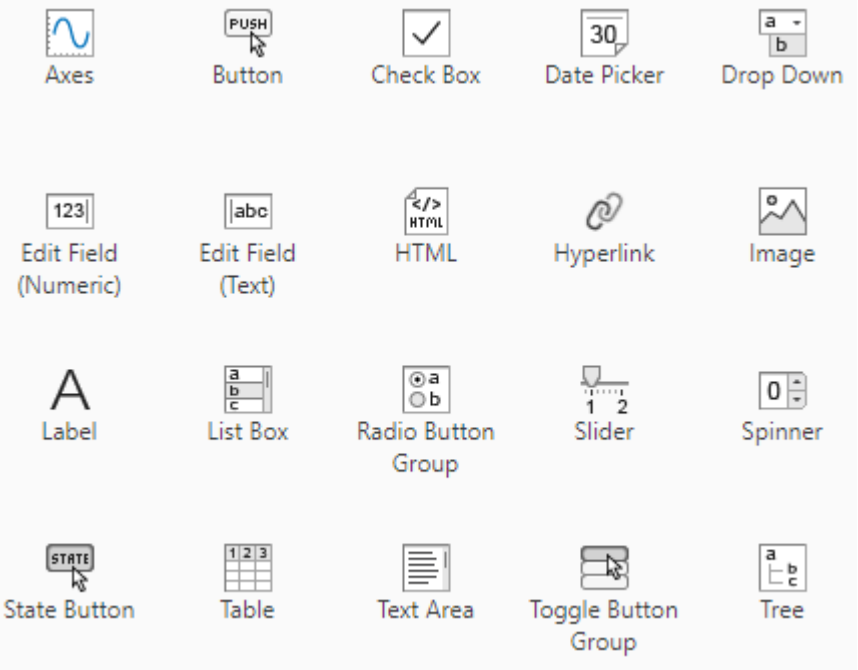
Operation	Keypress Interactive functions	Live Scripts
Data smoothing	iSignal: isignal.m, page 376	DataSmoothing.mlx, page 59
Differentiation	iSignal: isignal.m, page 376	DataDifferentiation.mlx, page 78
Self-deconvolution	iSignal: isignal.m, page 376	DeconvoluteData.mlx, page 126
Peak detection	iPeak: ipeak.m, page 255	PeakDetection.mlx, page 253
Peak fitting	Interactive peak fitter: ipf.m, page 417	PeakFittingTool.mlx, page 440
Fourier filtering	iFilter: ifilter.m, page 391	FourierFilterTool.mlx, page 396

A more sophisticated way: Matlab Apps

There is another development path that results in programs with a *more contemporary and polished graphical user interface* (GUI). This is ideal for deployment to users who *do not need to have access to the underlying code*, to prevent accidental editing or deletion of portions of the script. These are called Matlab "apps". There are examples of them in the toolboxes that may be included in your version of Matlab (or can be optionally purchased from Matlab). Type "ver" at the command line to see which ones are included in yours.

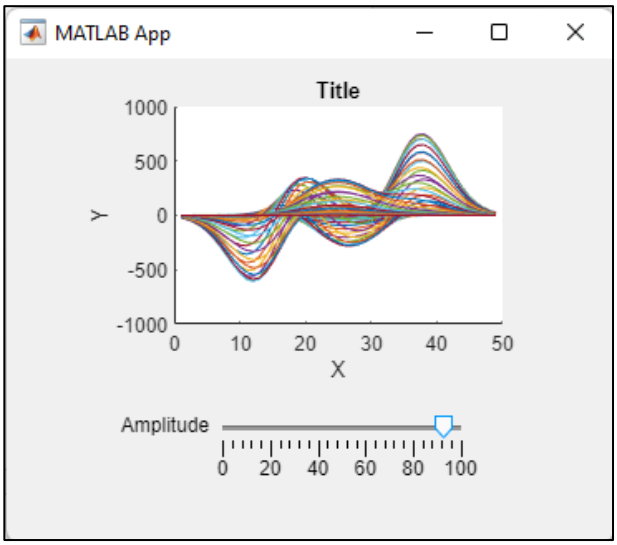


The process of *development* of such apps is more complex than coding the mathematically equivalent Live Script or function. But fortunately, Matlab has a built-in drag-and-drop development environment to build user interfaces. Just click on the APPS button at the top left. This brings up several app-related buttons as well as a list of apps already installed.

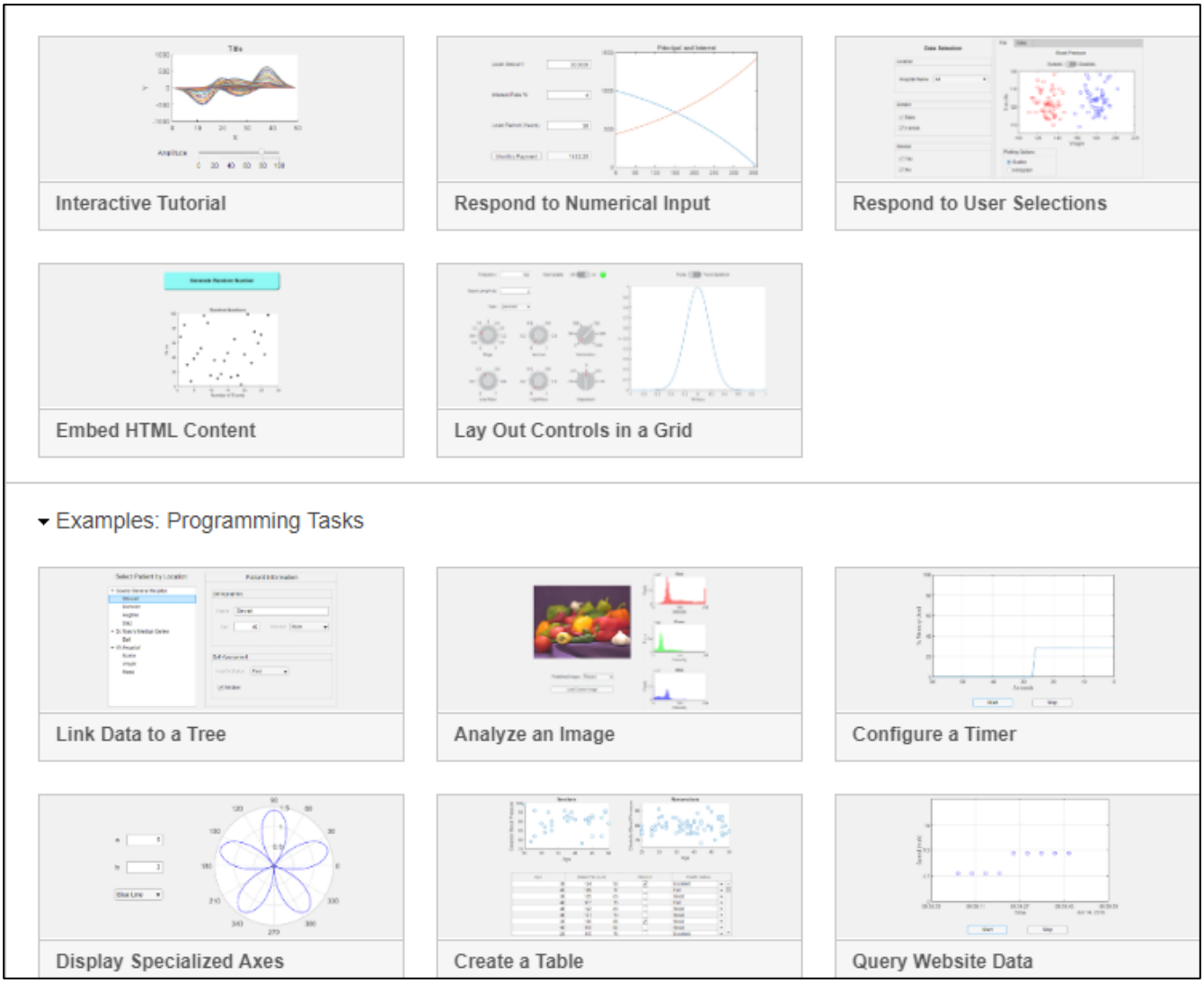


Clicking the **Design App** button, or typing "appdesigner" at the command prompt, brings up the App Designer screen, which has two main modes, selected by buttons on the right: the **Design View** and the **Code View**. In the Design View, you build your user interface - the "look" of your app - which might include menus, buttons, sliders, spinners, tables, graphs, etc., that you select from a large list of components (shown on the left), dragging and dropping the ones your need onto the blank layout on the right, and arranging them as you wish. What happens when you operate the control depends on the purpose of your app, and so naturally you must provide that code, which is called a "callback function". All this code is shown when you click the "Code View" button, both the code that is automatically generated when you add components to your design (shown with a *grey* background), which you cannot modify directly, and the code that you can type in to perform the desired calculations (shown with a *white* background).

The example shown on the right, described in "Create and Run a Simple App Using App Designer", displays a waveform whose amplitude you can interactively control with a slider. You can learn a lot by studying simple examples like this. There are many such examples that are built into *App Designer*. When you click "**New**" in the Designer mode, you'll get a display (shown below) of several examples that are already constructed.



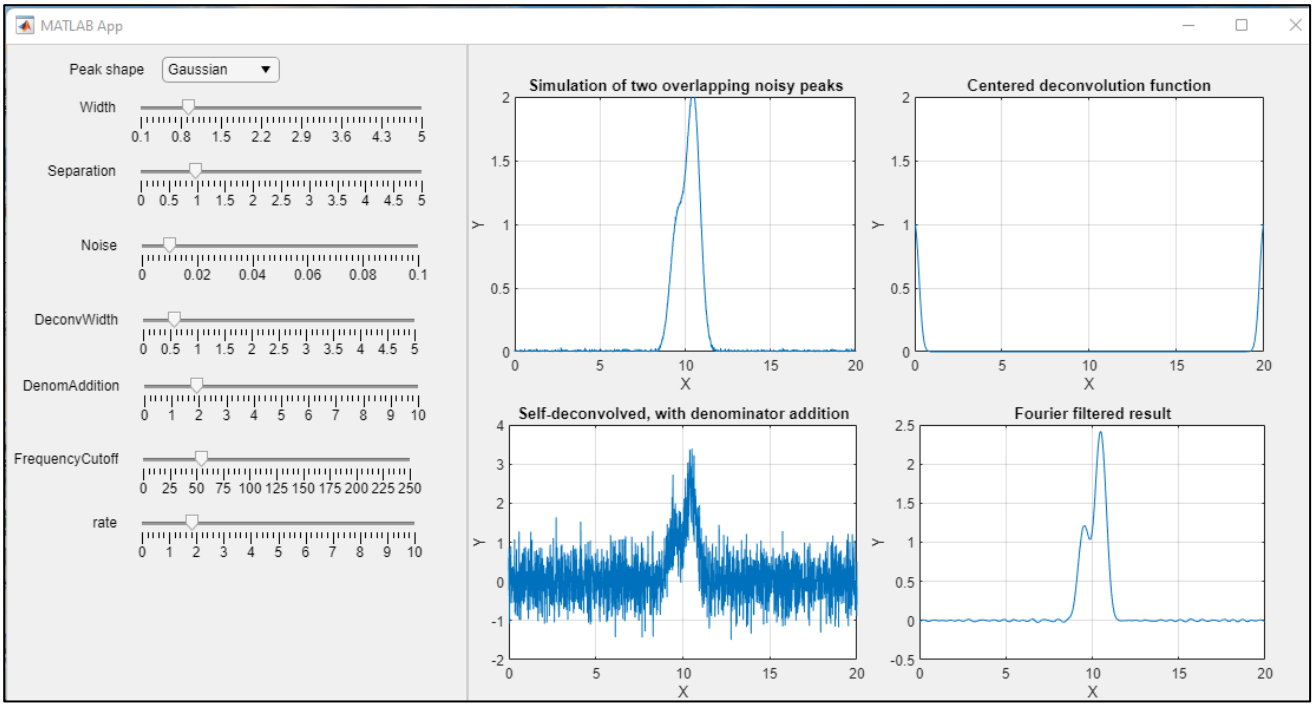
Rather than detailing all the required steps here, I will defer to the many excellent tutorials and YouTube videos already available. For example, there is a video tutorial titled "Getting Started and Hello World app" at https://www.youtube.com/watch?v=iga-YS6VbyE.



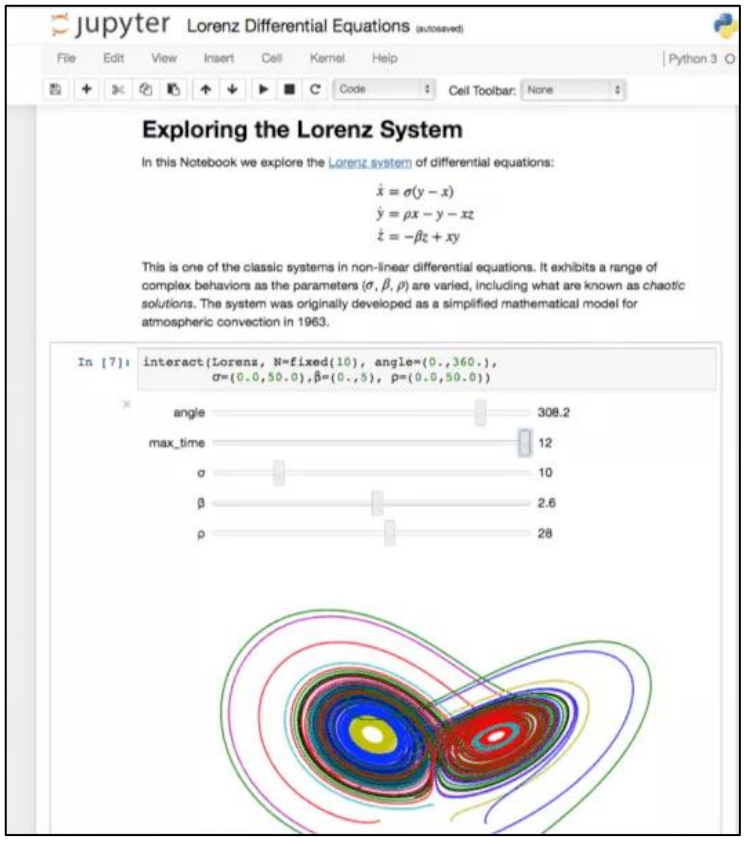
The *disadvantage* of apps is that they are more complex for the programmer. In fact, the amount of code and of coding time and effort that goes into the user interface design and interactivity usually far exceeds the code that is required for the mathematical computations. Creating an equivalent Live Script is much easier.

You must add the specific programming for the mathematical calculations, as in a regular script, which you type into the white space in the Code View, and *you can call any functions that you are previously written* and have saved in the Matlab path. You can also package any Matlab app you create into a single file, including any functions that you have called, so it can be easily shared with others.

Shown on the next page is a Matlab app demonstrating how a pair of noisy overlapping peaks can be sharpened and more clearly resolved by deconvolution, using Fourier filtering and denominator addition to control the noise amplification (page 117). You can download the installer from Self-decovolution demo.mlappinstall. (A Matlab Live Script version of this is shown on page 126, and a Python version in shown on page 455).



The coding required to get all this to work is considerably more complex than creating the equivalent Live Script shown above, but the app ends up looking like a finished end-user application, and it is easier and more fool-proof for an inexperienced person to operate.



Python has many libraries for developing such apps, such as PyQt5, Tkinter, and Jupyter Notebooks which are highly flexible, open-source, and which support multiple languages, making them ideal for a broad range of applications, particularly in data science and education. They are free and accessible to anyone with a Python installation. The simple example on the left has numeric sliders to control the variables in a non-linear system of equations. These can be complex to program, but fortunately *AI chatbots can write the code*, if you give it a detailed prompt (page 454).

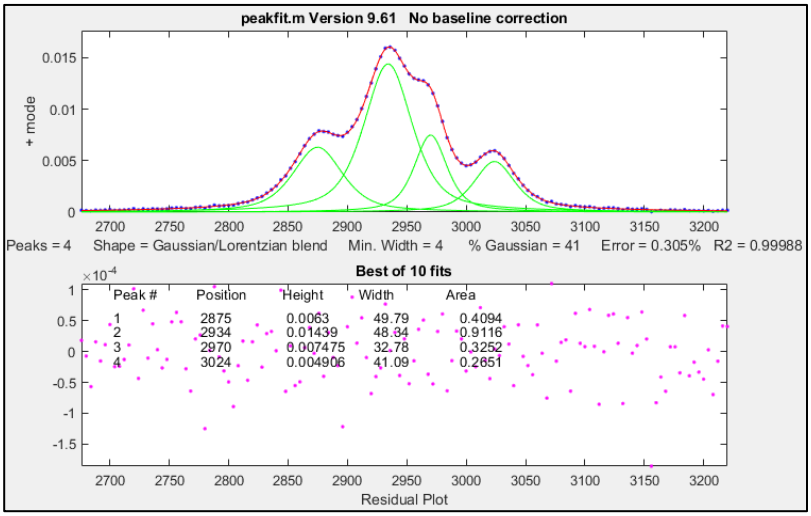
By means of comparison, Matlab Live Scripts are more tailored for professionals in engineering and scientific domains, but they require a MATLAB license, making them more specialized and potentially costly that Python.

Note: *Mathematica* (page 462) can also integrate code, graphics, and controllers into one document.

Using real-signal modeling to determine measurement accuracy.

It's common to use computer-generated signals whose characteristics are known exactly to establish the accuracy of a proposed signal processing method, analogous to the use of standard reference materials in analytical chemistry (page 337). But the problem with computer-generated signals is that they are often too simple or too ideal to be realistic, such as a series of peaks that are all equal in height and width, of some idealized shape such as a pure Gaussian, and with idealized added pure white noise. For the measurement of the areas of partly overlapping peaks (page 142), such idealized data sets will result in overly optimistic estimates of area measurement accuracy. One way to create more realistic known signals for a particular application is to use *iterative curve fitting* (page 199). If you can find a model that fits the experimental data with very low fitting error and with random residuals, then the peak parameters from that fit can be used to construct a realistic *synthetic* signal to estimate the precision and accuracy of the measurement. The big advantage is that the parameters of synthetic signals can be modified as you wish *to explore how a proposed measurement method would work under other experimental conditions* (e.g., if the peak shape were different, or if the peaks were more or less overlapped, or if the noise or the sampling frequency were higher or lower).

To demonstrate this idea for the measurement of the areas of overlapping peaks, I downloaded a



spectrum from the NIST IR database and found in that spectrum a set of four fused peaks near 3000 cm^{-1}, shown on the left. I used peakfit.m to fit those peaks with a model consisting of four Gaussian-Lorentzian sum (GLS) peaks of different widths (41% Gaussian was optimum), yielding a fitting error of only 0.3%, an R^2 of 0.99988, and unstructured random residuals, as shown in the lower panel. The best-fit peak parameters and the residual noise were then used in a self-contained script to create a synthetic model signal that is *essentially identical to the experimental spectrum*, except that the model has exactly known peak areas. Then, the script attempts to determine whether the simpler and faster perpendicular drop method could measure those areas accurately. To do this it uses second differentiation (page 63) to locate the original peak positions, measures the peak area by perpendicular drop (page 140), which by itself is not expected to work on such overlapped peaks, and finally repeats the area measurements after sharpening the peaks by Fourier self-deconvolution (page 118), using a low-pass Fourier filter to control the noise (page 128).

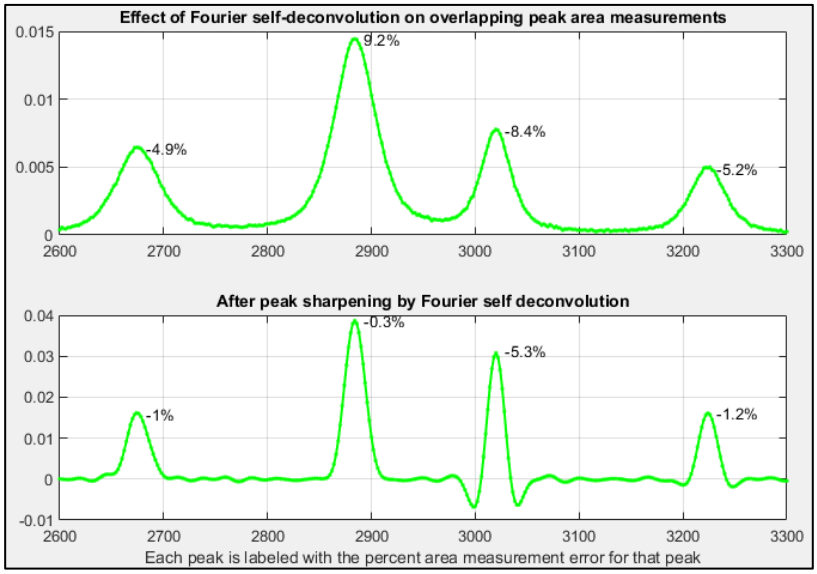
As shown by the **first figure** on the following page, the self-deconvolution sharpening can in fact improve the perpendicular drop area accuracy substantially, from an average error of 29% to only 3.1% after deconvolution. But because the peaks have different widths, *there is no single optimum deconvolution width*. Tests show that the best overall results are obtained when the deconvolution function *shape* is the same as in the original signal and when the deconvolution function *width* is 1.1

times the *average* peak width in the signal.




```
Sampling interval (cm-1)= 2
Change in peak separation (PeakSpread)= 0
Noise= 5e-05
GLS Shape (fraction Gaussian)= 0.41
Deconvolution Width= 23.7 points (1.1 times the mean signal peak width)
Frequency Cutoff= 20%
```

In the **second figure**, below, I did something that you obviously cannot do with real data: I *artificially spread out the peaks in the model signal*, with no other change, making their areas easier to measure. This shows more clearly that this choice of deconvolution function width causes the third peak to be "over sharpened", resulting in negative lobes for that peak (but recall that deconvolution is done in a way that *conserves total peak area*. See page 114). A more conservative approach, using the largest deconvolution width possible *without* the signal ever going negative (about 0.8 times the average peak width in this case) results in only a modest improvement in average area accuracy (from 27% to 12%; graphic). So "over sharpening" is not necessarily always bad.

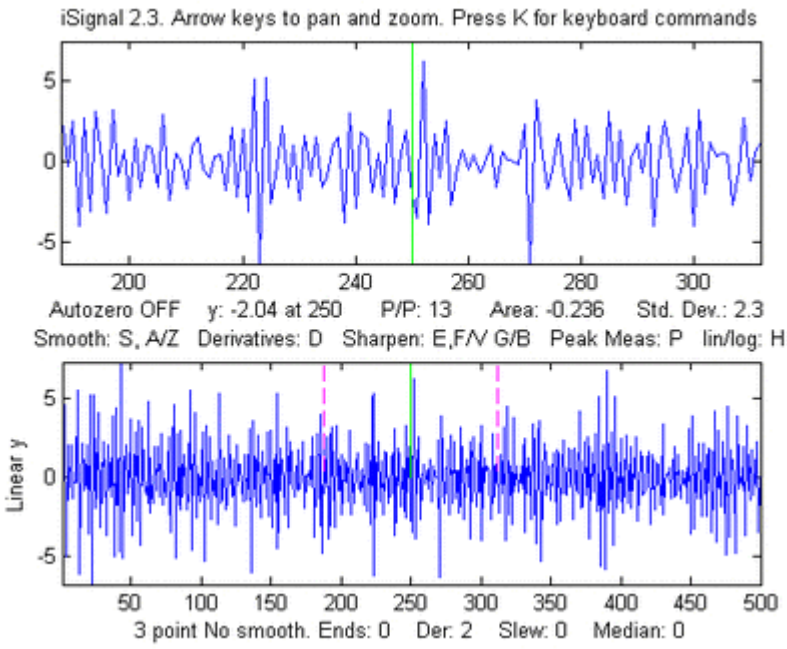



```
Change in peak separation (PeakSpread)= 100 (All other parameters
unchanged)
```

Signal processing software details.

Interactive smoothing, differentiation, and signal analysis (iSignal)

iSignal (or the Octave version isignaloctave.m) is a single self-contained m-file for performing smoothing, differentiation, peak sharpening, interpolation, baseline subtraction, Fourier frequency spectrum, least-squares peak fitting, deconvolution, and other useful functions on time-series data. Using simple keystrokes, you can adjust the signal processing parameters continuously while observing the effect on your signal dynamically. Click here to download the ZIP file "iSignal8.zip" that also includes some sample data for testing. You can also download iSignal from the Matlab File Exchange. You can also run iSignal in a web browser (just click on the figure window), even on Matlab Mobile. There is a separate version for Octave. The demo script "demoisignal.m" is a self-running demonstration of several features of the program and will test for proper installation; the title of each figure describes what is happening. Its basic operation of iSignal is similar to **iPeak** and **ipf.m.** The syntax is: `pY=isignal(Data);` or, to specify all the settings in advance:




```
[pY,Spectrum,maxy,miny,area,stdev] = isignal(Data, xcenter, xrange,
SmoothMode, SmoothWidth, ends, DerivativeMode, Sharpen, Sharp1, Sharp2,
Symize, Symfactor, SlewRate, MedianWidth, SpectrumMode);
```

"Data" may be a 2-column matrix with the independent variable (x-values) in the first column and dependent variable (y values) in the second column, or separate x and y vectors, or a single y-vector (in which case the data points are plotted against their index numbers on the x-axis). Only the first argument (Data) is required; all the others are optional. iSignal returns the processed dependent axis ('**pY**') vector (and, in the Spectrum Mode, the frequency spectrum matrix, '**Spectrum**') as the output arguments. It plots the data in the Matlab Figure window, the lower half of the window showing the entire signal, and the upper half showing a selected portion controlled by the pan and zoom keys (the four cursor arrow keys). The initial pan and zoom settings are optionally controlled by input arguments '**xcenter**' and '**xrange**', respectively. Double-click the figure window title bar to expand to full screen for a better view. Other keystrokes allow you to control the smooth type, width, and ends treatment, the derivative order (0th through 5th), and peak sharpening. (The initial values of all these parameters can be passed to the function via the optional input arguments **SmoothMode**, **SmoothWidth**, **ends**, **DerivativeMode**, **Sharpen**, **Sharp1**, **Sharp2**, **SlewRate**, and **MedianWidth**. See the examples below). Press **K** to see all the keyboard commands. ***Note:*** Make sure you do not click on the "Show

Plot Tools" button in the toolbar above the figure. That will disable normal program functioning. If you do, close the Figure window and start again.

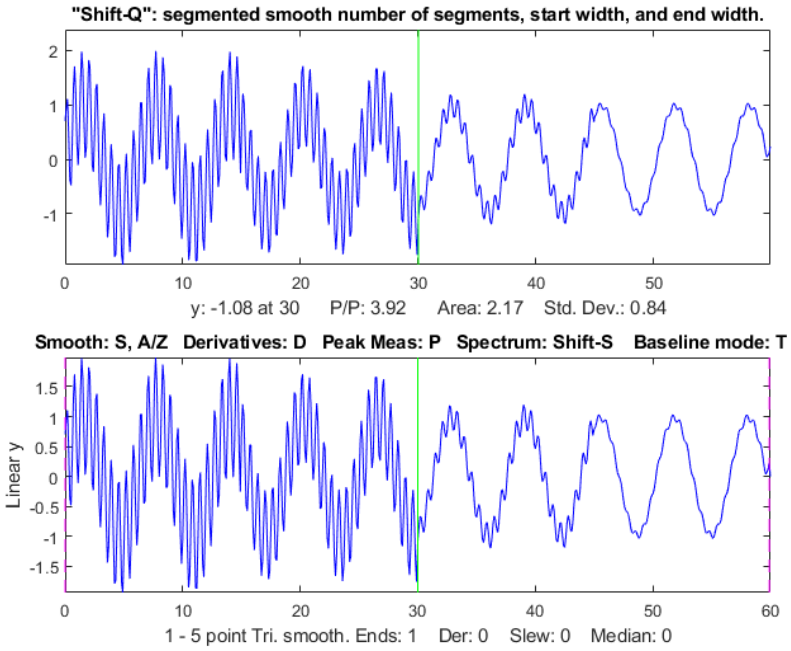
Smoothing

The **S** key (or input argument "**SmoothMode**") cycles through five smoothing modes:
If **SmoothMode**=0, the signal is not smoothed.
If **SmoothMode**=1, rectangular (sliding-average or boxcar)
If **SmoothMode**=2, triangular (2 passes of sliding-average)
If **SmoothMode**=3, p-spline (3 passes of sliding-average)
If **SmoothMode**=4, Savitzky-Golay smooth (thanks to Diederick).

The **A** and **Z** keys (or optional input argument **SmoothWidth**) control the **SmoothWidth**, *w*.
The **X** key toggles "**ends**" between 0 and 1. This determines how the "ends" of the signal (the first *w*/2 points and the last *w*/2 points) are handled when smoothing:

If **ends**=0, the ends are zero.
If **ends**=1, the ends are smoothed with progressively smaller smooths the closer to the end.
Generally, **ends**=1 is best, except in some cases using the derivative mode when **ends**=0 result in better vertical centering of the signal.

To specify a segmented smooth (more on page 333) press **Shift-Q.** You can specify the smooth width vector in *two ways*: at the prompt you can either (a) enter the number of segments (then you'll be prompted to enter the smooth widths in the first and last segments, and the computer will calculate integer values of smooth widths that are evenly divided between the specified first and last values), or (b) type in the smooth width vector *directly* including the square brackets, e.g. [1 3 3 9]. In either case, subsequently adjusting the smooth width with the **A** and **Z** keys will vary *all the segments by the same percentage factor*. (To return to an ordinary single segment smooth, enter 1 as the number of segments). The figure on the right shows a 4-segment smooth with smooth widths of 1, 2, 4, and 5. Note: when you are smoothing peaks, you can easily measure the effect of smoothing on peak height and width by turning on peak measure mode (press **P**) and then press **S** to cycle through the smooth modes.



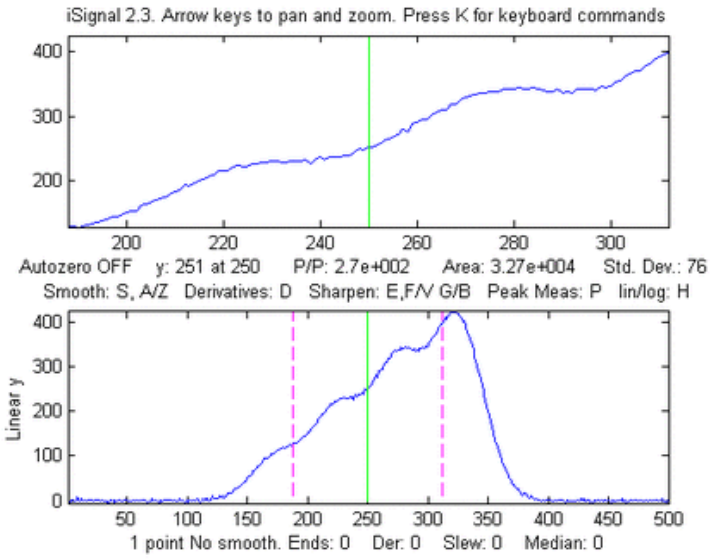
There are two special functions for removing or reducing sharp spikes in signals: the **M** key, which implements a median filter (it asks you to enter the spike width, e.g., 1,2, 3... points) and the ~ key, which limits the maximum rate of change.

Differentiation

The **D** / **Shift-D** keys (or optional input argument "**DerivativeMode**") increase/ decrease the derivative order. The default is 0. Careful optimization of the smoothing of derivatives is critical for an acceptable signal-to-noise ratio. An example is shown in the figure on the right. In SmoothModes 1 through 3, the derivatives are computed with respect to the independent variable (x-values), corrected for non-uniform x-axis intervals. In SmoothMode 4 (Savitzky-Golay) the derivatives are computed by the Savitzky-Golay algorithm. Click for GIF animation.

Peak sharpening

The **E** key (or optional input argument "**Sharpen**") turns off and on peak sharpening. The sharpening strength is controlled by the **F** and **V** keys (or optional input argument "**Sharp1**") and **B** and **G** keys (or optional argument "**Sharp2**"). The optimum values depend on the peak shape and width. For peaks of Gaussian shape, a reasonable value for **Sharp1** is PeakWidth^2/25 and for **Sharp2** is PeakWidth^4/800 (or PeakWidth^2/6 and PeakWidth^4/700 for Lorentzian peaks), where PeakWidth is the full width at half maximum of the peaks *expressed in the number of data points*. However, you do not need to "do the math" yourself, because *iSignal* can calculate sharpening and smoothing settings for Gaussian and for Lorentzian peak shapes using the **Y** and **U** keys respectively. Just isolate a single typical peak in the upper window using the pan and zoom keys, then press **Y** for Gaussian or **U** for Lorentzian peaks. Fine-tune the sharpening with the F/V and G/B keys and the smoothing with the A/Z keys. You can see this animation if you download the Microsoft Word 365 version, otherwise click this link. (The optimum settings depend on the width of the peak, so if your signal has peaks of widely different widths, one setting will not be optimum for all the peaks and you might consider using segmented smoothing as described above or on page 333). You can expect a decrease in peak width (and a corresponding increase in peak height) of about 20% - 50%, depending on the shape of the peak (the peak area is largely unaffected by sharpening). Excessive sharpening leads to baseline artifacts and increased noise. iSignal allows you to experimentally determine the values of these parameters that give the best trade-off between sharpening, noise, and baseline artifacts, for your purposes. You can easily measure the effect of sharpening quantitatively by turning on peak measure mode (press **P**) and then pressing **E** to toggle the sharpen mode off and on. Note: only the Savitzky-Golay smooth mode is used for peak sharpening.



Interactive convolution and deconvolution

In **iSignal 8.3** you can press **Shift-V** to display the menu of Fourier convolution and deconvolution operations (page 113) that allow you to convolute a Gaussian, Lorentzian, or exponential function with the signal, or to deconvolute a Gaussian, Lorentzian, or exponential function from the signal.

```
Fourier convolution/deconvolution menu
   1. Convolution
   2. Deconvolution
Select mode 1 or 2:  2

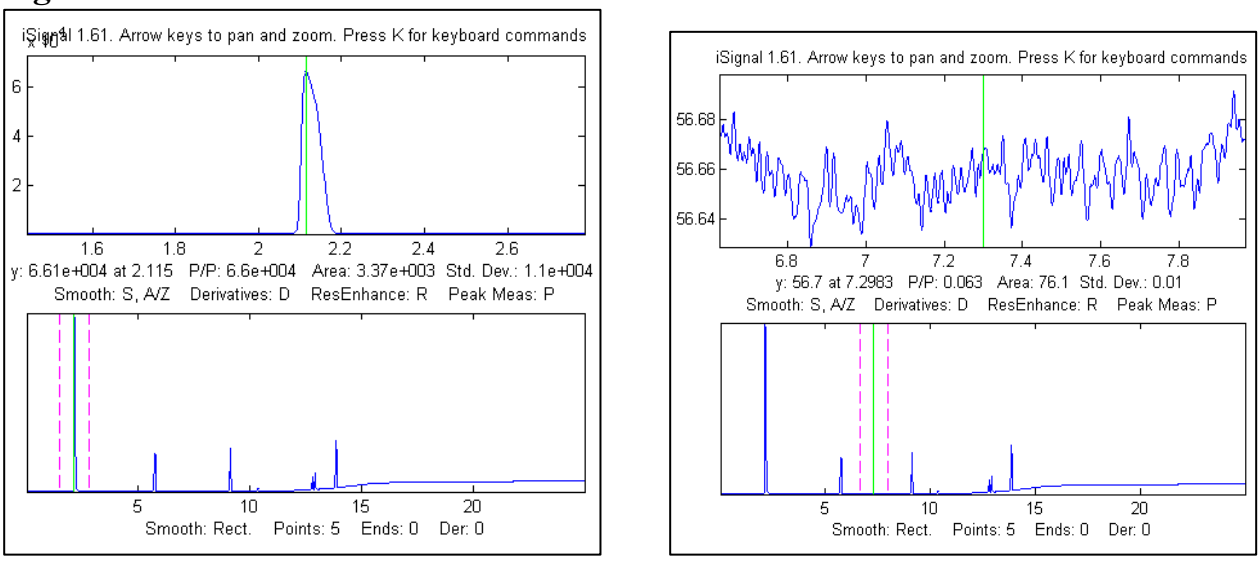
Shape of convolution/deconvolution function:
   1. Gaussian
   2. Lorentzian
   3. Exponential
Select shape 1, 2, or 3:  2

Enter the width in x units:
```

Then you enter the time constant (in x units), and press **Enter**. Then use the **3** and **4** keys to adjust the width of the deconvolution function by 10% (or **Shift-3** and **Shift-4** to adjust by 1%). You may need to adjust the smoothing also, if the signal is too noisy, but too much smoothing will broaden the peaks. For a real-signal application, see page 126. Note: in cases where the peak widths within a group of peaks are substantially different, it may be better to use segmented deconvolution.

Interactive Symmetrization (or "de-tailing") of exponentially broadened signals is performed by *weighted first-derivative addition* (page 83). Press the **Shift-Y** key, type in an estimated weighting factor (which is the time constant or tau of the exponential), and press **Enter**. To adjust, press the "**1**" and "**2**" keys to change the weighting factor by 10% and "**Shift-1**" and "**Shift-2**" keys to change by 1%. Increase the factor until the baseline after the peak goes negative, then decrease it slightly so that it is as low as possible but not negative. Run the script iSignalSymmTest.m (graphic) for an example signal with two overlapping exponentially broadened Gaussians.

Signal measurement



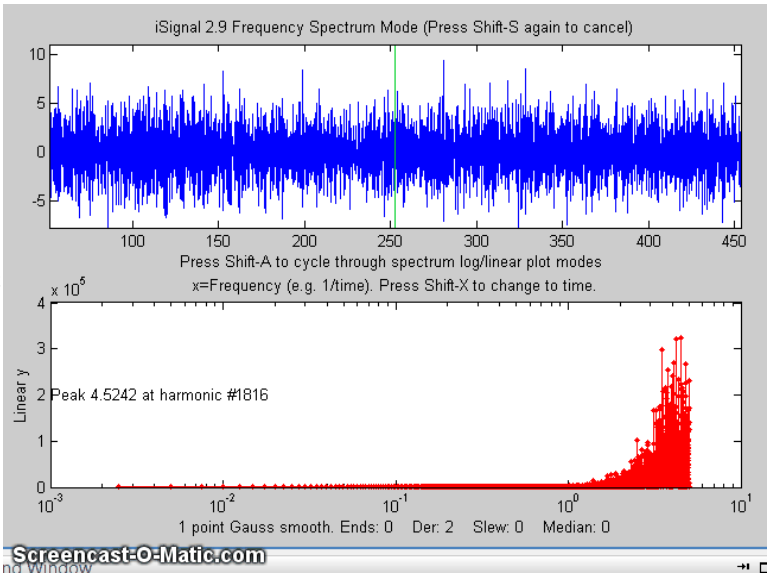
Signal-to-noise ratio (SNR) measurement of a signal with very high SNR. Left: The peak height of the largest signal peak is measured by placing the green center cursor on the largest peak. The peak-to-peak signal=66,000. Right: The noise is measured on a flat portion of the baseline: standard deviation of noise=0.01, therefore the SNR=66,000/.01 = 6,600,000

The cursor keys control the position of the green cursor (left and right arrow keys) and the distance between the dotted red cursors (up and down arrow keys) that mark the selected range displayed in the upper graph window. The label under the top graph window shows the value of the signal (y) at the

green cursor, the peak-to-peak (min and max) signal range, the area under the signal, and the standard deviation within the selected range (the dotted cursors). Pressing the **Q** key prints out a table of the signal information in the command window. If the optional output arguments maxy, miny, area, stdev are specified, iSignal returns the maximum value of y, the minimum value of y, the total area under the curve, and the standard deviation of y, in the selected range displayed in the upper panel.

Frequency Spectrum mode

The **Frequency Spectrum mode**, toggled on and off by the **Shift-S** key, computes the Fourier frequency spectrum (page 94) of the segment of the signal displayed in the upper window and displays it in the lower window (temporarily replacing the full-signal display). Use the pan and zoom keys to adjust the region of the signal to be viewed. Press **Shift-A** to cycle through four plot modes (linear, semilog X, semilog Y, or log-log) and press **Shift-X** to toggle between a *frequency* on the x-axis and *time* on the x-axis. Importantly, *all signal processing functions remain active in the frequency spectrum mode* (smooth, derivative, etc.) so you can immediately observe the effect of these functions on the frequency spectrum of the signal. (You can see this animation if you download the Microsoft Word 365 version, otherwise click the figure to open in a Web browser).



Press **Shift-T** to transfer the frequency spectrum to the signal in the upper panel, so you can pan and zoom and do other processing and measurements on the frequency spectrum. Press **Shift-S** again to return to the normal mode. Spectrum mode is a *visible mode*, indicated by the label at the top of the figure. To *start off* in the spectrum mode, set the 13th input argument, SpectrumMode, to 1. To *save* the spectrum as a new variable, call iSignal with the output arguments `[pY,Spectrum]`:

```
>> x=0:.1:60; y=sin(x)+sin(10.*x);

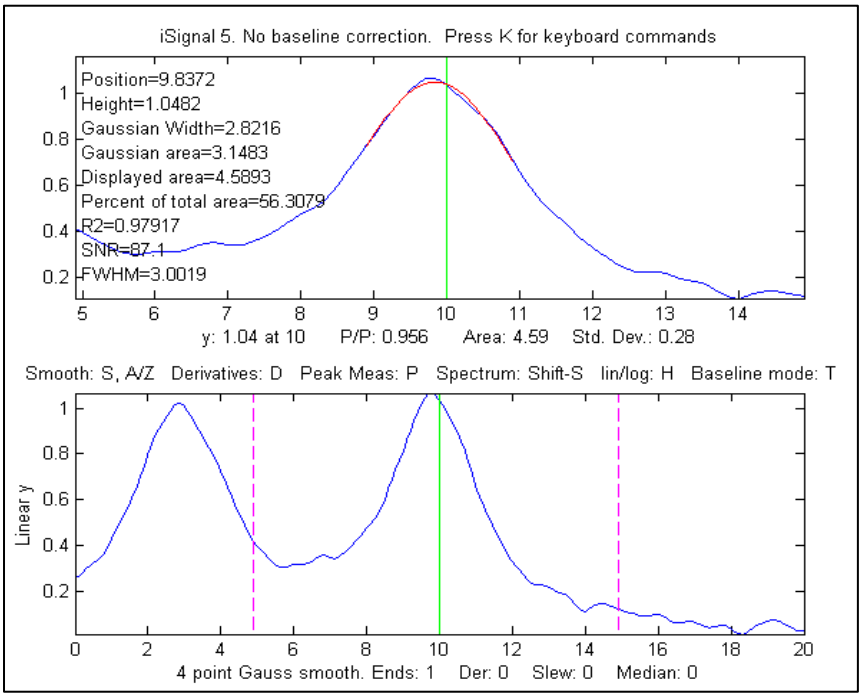
>> [pY,Spectrum]=isignal([x;y],30,30,4,3,1,0,0,1,0,0,0,1);

>> plot(Spectrum(:,1),Spectrum(:,2)) or plotit(Spectrum)
or isignal(Spectrum); or ipf(Spectrum); or ipeak(Spectrum)
```

Shift-Z toggles on and off the peak detection and labeling on the frequency/time spectrum. The peaks are labeled with their frequencies. You can adjust the peak detection parameters in lines 2192-2195 in version **5**. The **Shift-W** command displays the 3D waterfall spectrum, by dividing up the signal into segments and computing the power spectrum of each segment. This is mostly a novelty, but it may be useful for signals whose frequency spectrum varies over the duration of the signal. You are asked to choose the number of segments into which to divide the signal (that is, the number of spectra) and the type of 3D display (mesh, contour, surface, etc.)

Background subtraction

There are two ways to subtract the background from the signal: automatic and manual. To select an automatic baseline correction mode, press the **T** key repeatedly. It cycles through four modes (page 219): *No* baseline correction, *linear* baseline subtraction, *quadratic* baseline subtraction, *flat* baseline correction, then back to *no* baseline correction. When baseline mode is *linear*, a straight-line baseline connecting the two ends of the signal segment in the upper panel will be automatically subtracted. When the baseline mode is *quadratic*, a parabolic baseline connecting the two ends of the signal segment in the upper panel will be automatically subtracted. The baseline is calculated by computing a linear (or quadratic) least-squares fit to the first 1/10th of the points together with the last 1/10th of the points. Try to adjust the pan and zoom to include enough of the baseline at the beginning and end of the segment in the upper window to ensure that the automatic baseline subtract gets a good reading of the baseline. The *flat* baseline mode is used only for peak fitting (**Shift-F** key). The calculation of the signal amplitude, peak-to-peak signal, and peak area are all recalculated based on the baseline-subtracted signal in the upper window. If you are measuring peaks superimposed on a background, the use of the BaselineMode will have a big effect on the measured peak height, width, and area, but very little effect on the peak x-axis position.



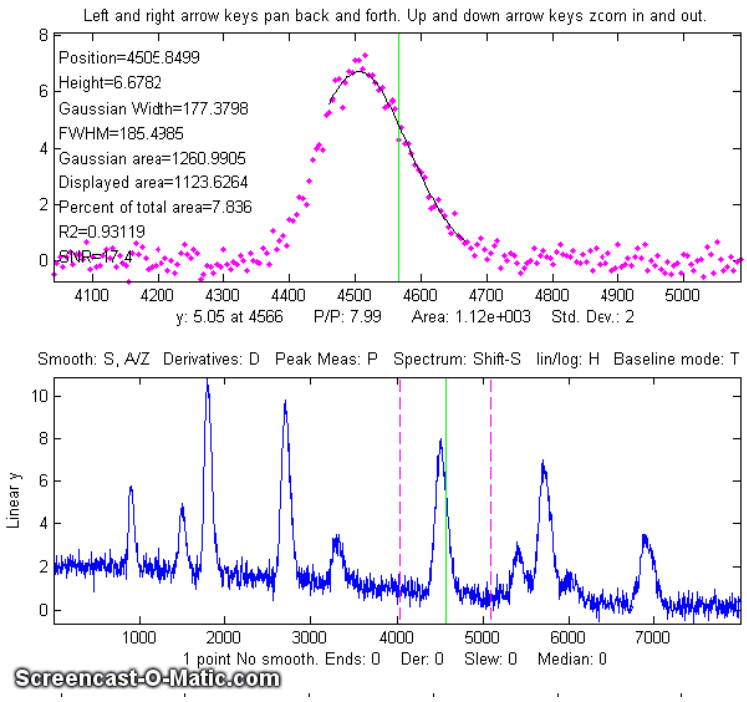
In addition to the four BaselineMode baseline subtraction modes for peak measurement, a *manually estimated* piecewise linear baseline can be subtracted from the *entire* signal in one operation. The **Backspace** key starts background correction operation. In the command window, type in the number of background points to click and press the **Enter** key. The cursor changes to crosshairs. Then click along the estimated background in the figure window, starting to the left of the x axis and placing the last click to the right of the x axis. When you click the last point, the linearly interpolated baseline between those points is subtracted from the signal. To restore the original background (i.e. to start over), press the '\' key (just below the backspace key).

Peak and valley measurement

The **P** key toggles off and on the "peak parabola" mode, which attempts to measure the one peak (or valley) that is centered in the upper window under the green cursor by superimposing a least-squares best-fit parabola (page 173) in red, on the center portion of the signal displayed in the upper window. (Zoom in so that the red overlays just the top of the peak or the bottom of the valley as closely as possible). The peak position, height, and width are measured by least-squares curve fitting of a

Gaussian (colored red in the upper panel) to the central part of the selected segment. (Change the pan and zoom to modify that region. The readings will change as the segment measured is changed). The "RSquared" value is the coefficient of determination. The closer to 1.000 the better. The peak parameters will be most accurate if the peaks are Gaussian. Other peak shapes, or very noisy peaks of any shape, will give only approximate results. However, the position and height, and area values are pretty good for any peak shape if the "RSquared" value is at least 0.99. The "SNR" is the signal-to-noise-ratio of the peak under the green cursor. It is the ratio of the peak height to the standard deviation of the residuals between the data and the best-fit line in red.

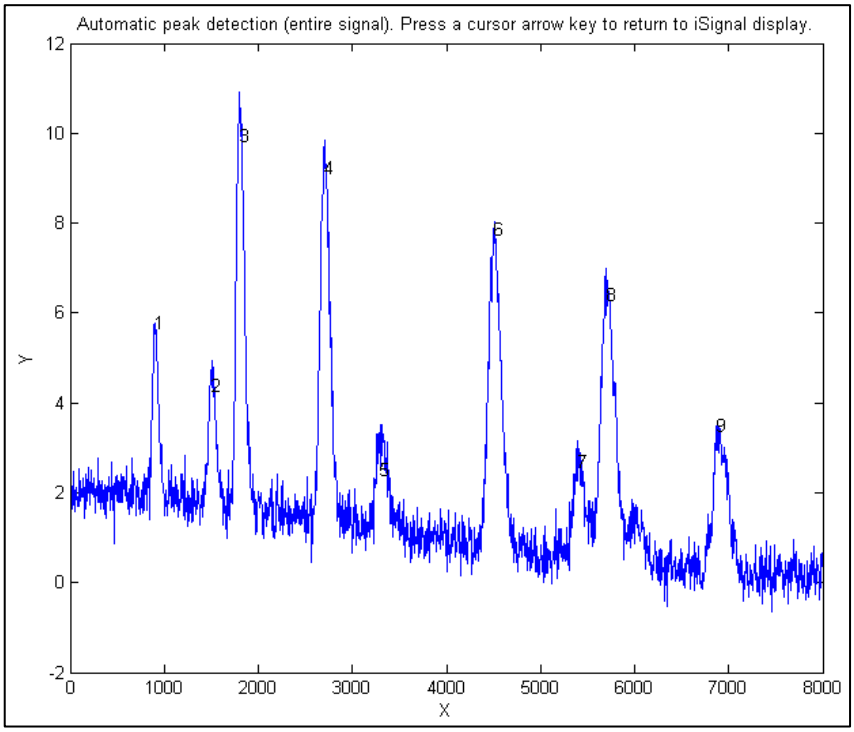
An example is shown in the figure on the right. If the peaks are superimposed on a non-zero background, subtract the background before measuring peaks, either by using the BaselineMode (**T** key) or the multi-point background subtraction (backspace key). Press the **R** key to print out the peak parameters in the command window.



Peak *width* is measured in *two ways*: the "Gaussian Width" is the width measured by Gaussian curve fitting (over the region colored in red in the upper panel) and is strictly accurate only for Gaussian peaks. The other method is direct measurement of the *full width at half maximum* ('FWHM') of the *central peak* in the upper panel (the peak marked by the green vertical line). This works for peaks of *any shape*, but it is computed only for the central peak and only if the half-maximum points fall within the zoom region displayed in the upper panel (otherwise it will return NaN). It will not be highly accurate for very noisy peaks. The Gaussian width will be more accurate for noisy or sparsely sampled peaks, but only if the peaks are at least approximately Gaussian. In the example on the left, the peaks are Lorentzian, with a true with of 3.0, plus added noise. In this case the measured FWHM (3.002) is more accurate than the Gaussian width (2.82), especially if a little smoothing is used to reduce the noise.

Peak *area* is also measured in *two ways*: the "Gaussian area" and the "Total area". The "Gaussian area" is the area under the Gaussian that is a best fit to the center portion of the signal displayed in the upper window, marked in red. The "Total area" is the area by the trapezoidal method over the entire selected segment displayed in the upper window. (The percentage of total area is also calculated). If the portion of the signal displayed in the upper window is a pure Gaussian with no noise and a zero baseline, then the two measures should agree almost exactly. If the peak is not Gaussian in shape, then the total area is likely to be more accurate, if the peak is well separated from other peaks. If the peaks are overlapped, but have a known shape, then peak fitting (**Shift-F**) will give more accurate peak areas. In the example

above, the Lorentzian peak at x=10 has a true area of 4.488, so in this case the total area (4.59) is more accurate than the Gaussian area (3.14), but it is too high because of overlap with the peak at x=3. Curve fitting both Lorentzians peaks together would yield the most accurate areas. If the signal is panned slightly left and right, using the left and right cursor keys or the "[" and "]" keys, the peak parameters displayed will change slightly due to the noise in the data - the more noise, the greater the change, as in the example on the left. If the peak is asymmetrical, as in this example, the peak widths displayed on one side will be greater than the other side.



There is an automatic peak finder that is based on the autopeaks.m function (activated by the **J** key). It asks for the peak density (roughly the number of peaks that fit into the signal record), then detects, measures, and displays the peak position, height, and area of all the peaks it detects in the processed signal currently displayed in the lower panel, plots and number the peaks as shown on the right and also plots each peak separately in Figure window 2 with peak, tangent, and valley points marked (click for graphic). (The requested peak density controls the peaks sensitivity - larger numbers cause the routine to detect larger numbers of narrower peaks, and smaller numbers ignore the fine structure and looks for broader peaks). It also prints out the peak detection parameters that it calculates for use by any of the findpeaks... functions (page 234). To return to the usual iSignal display, press any cursor arrow key. (**Shift-J** does the same thing for the segment displayed in the upper window).

Peak fitting

iSignal has an iterative curve fitting (page 199) method performed by peakfit.m. This is the most accurate method for the measurements of the areas of highly overlapped peaks. First, center the signal you wish to fit using the pan and zoom keys (cursor arrow keys), select the baseline mode by pressing the '**T**' key to cycle through the 4 baseline modes: none, linear, quadratic, and flat (see page 219). Press the **Shift-F** key, then type the desired peak shape by number from the menu displayed in the Command window (next page), enter the number of peaks, enter the number of repeat trial fits (usually 1-10), and finally *click the mouse pointer on the top graph where you think the peaks might be*. (For off-screen peaks, click *outside* the axis limits but *inside* the graph window). A graph of the fit is displayed in Figure window 2 and a table of results is printed out in the command window. iSignal can fit many different combinations of peak shapes and constraints:

Gaussians: y=exp(-((x-pos)./(0.6005615.*width)).^2)
Gaussians with independent positions and widths(default).....1
Exponentially-broadened Gaussian (equal time constants).....5
Exponentially-broadened equal-width Gaussian.....8
Fixed-width exponentially-broadened Gaussian.....36
Exponentially-broadened Gaussian (independent time constants).....31
Gaussians with the same widths.....6
Gaussians with preset fixed widths.....11
Fixed-position Gaussians.....16
Asymmetrical Gaussians with unequal half-widths on both sides.....14
Lorentzians: y=ones(size(x))./(1+((x-pos)./(0.5.*width)).^2)
Lorentzians with independent positions and widths.....2
Exponentially-broadened Lorentzian.....18
***Equal*-width Lorentzians.....7**
***Fixed*-width Lorentzian.....12**
Fixed-*position* Lorentzian.....17
Gaussian/Lorentzian blend (equal blends).....13
Fixed-width Gaussian/Lorentzian blend.....35
Gaussian/Lorentzian blend with independent blends).....33
Voigt profile with equal alphas).....20
Fixed-width Voigt profile with equal alphas.....34
Voigt profile with independent alphas.....30
Logistic: n=exp(-((x-pos)/(.477.*wid)).^2); y=(2.*n)./(1+n).....3
Pearson: y=ones(size(x))./(1+((x-pos)./((0.5.^(2/m)).*wid)).^2).^m....4
Fixed-width Pearson.....37
Pearson with independent shape factors, m.....32
Breit-Wigner-Fano.....15
Exponential pulse: y=(x-tau2)./tau1.*exp(1-(x-tau2)./tau1).....9
Alpha function: y=(x-spoint)./pos.*exp(1-(x-spoint)./pos);.....19
Up Sigmoid (logistic function): y=.5+.5*erf((x-tau1)/sqrt(2*tau2))...10
Down Sigmoid y=.5-.5*erf((x-tau1)/sqrt(2*tau2)).....23
Triangular.....21

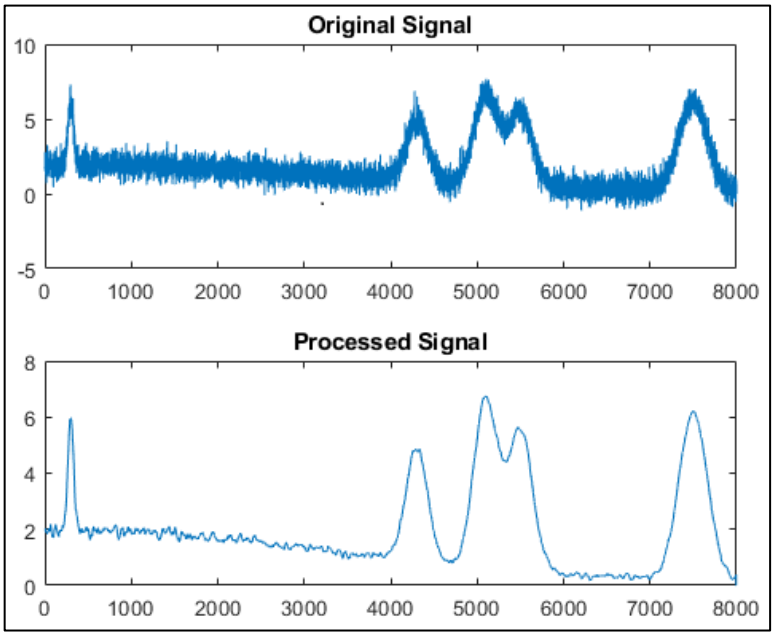
Note: if you have a peak that is an exponentially-broadened Gaussian or Lorentzian, you can measure both the "after-broadening" height, position, and width using the **P** key function, and the "before-broadening" height, position, and approximate width by fitting the peak to an exponentially-broadened Gaussian or Lorentzian model (shapes 5, 8,36, 31, or 18) using the **Shift-F** key function. The peak areas will be the same. Broadening does not affect the total peak area.

Polynomial fitting.

Shift-o fits a simple polynomial (linear, quadratic, cubic, etc.) to the upper panel segment and displays the coefficients (in descending powers) and the correlation coefficient R^2.

Saving the results

To save the processed signal to the disc as an x,y matrix in mat format, press the '**o**' key, then type in the desired file name into the "File name" field, then press **Enter** or click **Save.** To load into the workspace, type "load" followed by the file name you typed. The processed signal will be saved in a matrix called "Output". To plot the processed data, type "plot(Output(:,1),Output(:,2))".



Other keystroke controls

The **Shift-G** key toggles on and off a temporary grid on the upper and lower panel plots.

The **L** key toggles off and on the Overlay mode, which shows the original signal as a dotted line overlaid on the current processed signal, for the purposes of comparison.

Shift-B opens Figure window 2 and plots the original signal in the upper panel and the processed signal in the lower panel (as shown on the right).

The **Tab** key restores the original signal and cursor settings.

The **";"** key sets the selected region to zero (use to eliminate artifacts and spikes).

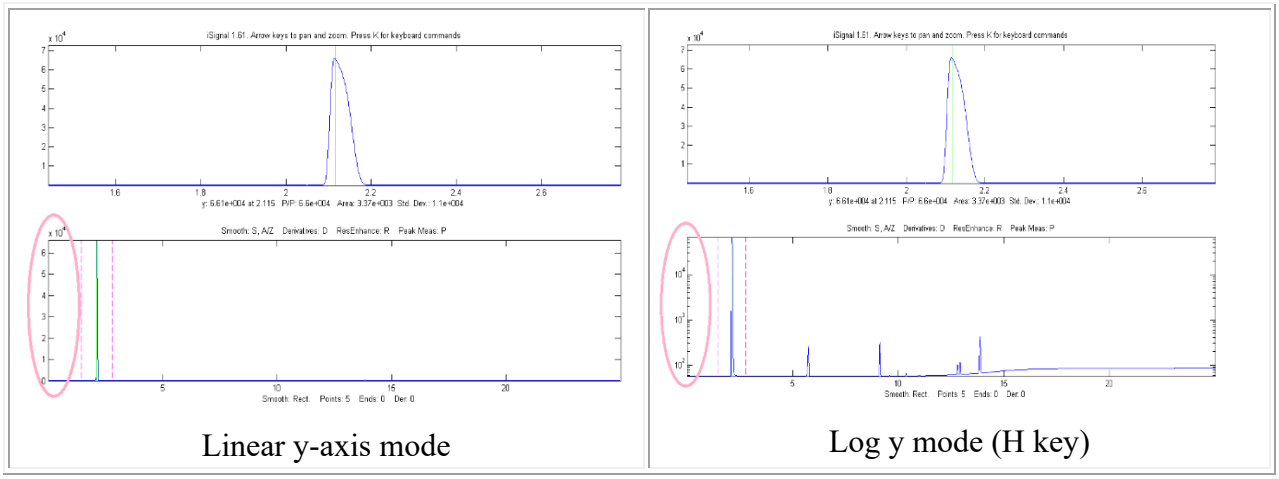
The "-" (minus sign) key is used to negate the signal (flip + for -).

Press **H** to toggle display of semilog y plot in the lower window, which is useful for signals with very wide dynamic range, as in the example in the figures below (zero and negative points are ignored in the log plot).

The '+' key takes the absolute value of the entire signal.

Shift-L replaces the signal with the processed version of itself, for the purpose of applying more passes of different widths of smoothing or higher orders of differentiation.

The ^ **(Shift-6)** key raises the signal to the specified power. To reverse this, simply raise to the reciprocal power. See Power transform method of peak sharpening on page 85.

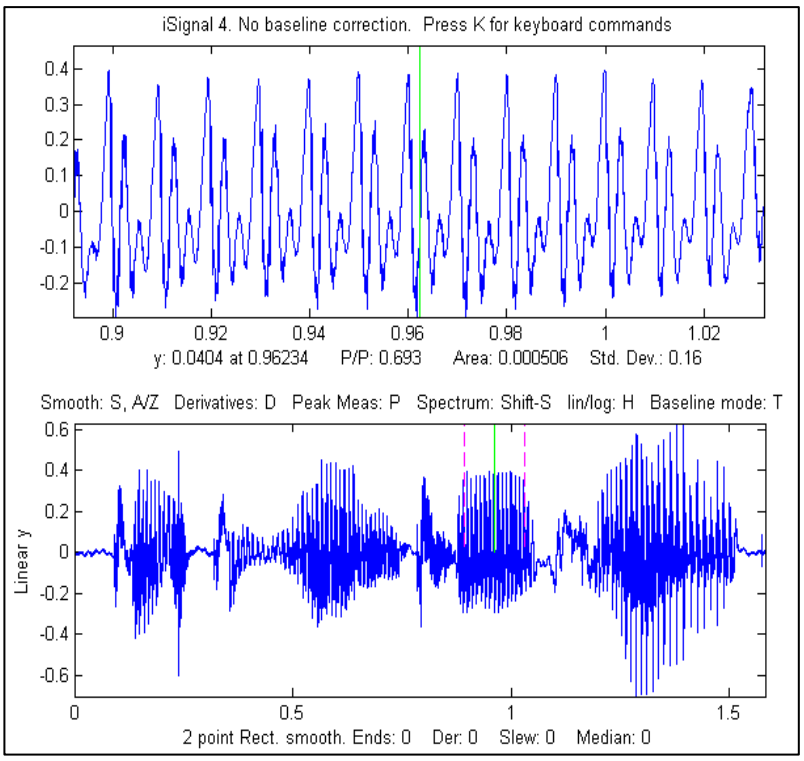


Linear y-axis mode

Log y mode (H key)

The **C** key condenses the signal by the specified factor ***n***, replacing each group of ***n*** points with their average (***n*** must be an integer, such as 2,3, 4, etc.). The **I** key replaces the signal with a linearly interpolated version containing ***m*** data points. This can be used either to increase or decrease the x-axis interval of the signal or to convert unevenly spaced values to evenly spaced values. After pressing **C** or **I**, you must type in the value of ***n*** or ***m*** respectively. You can press **Shift-C**, then click on the graph to print out the x,y coordinates of that point. This works on both the upper and lower panels, and on the frequency spectrum as well.

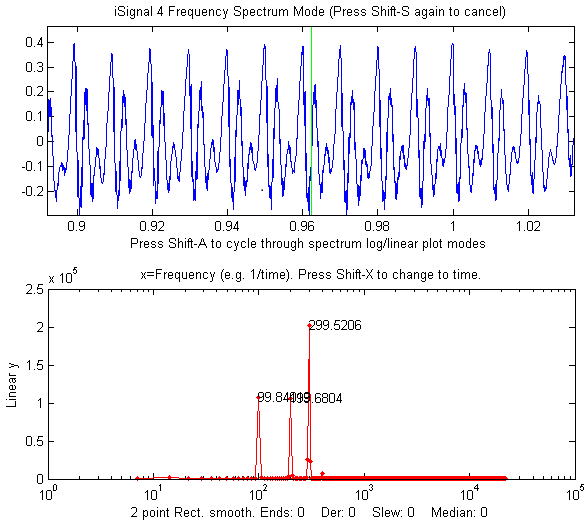
Playing data as audio.



Press **Spacebar** or **Shift-P** to play the segment of the signal displayed in the upper window as audio through the computer's sound output. Press **Shift-R** to set the sampling rate - the larger the number the shorter and higher-pitched will be the sound. The default rate is 44000Hz. Sounds or music files in WAV format can be loaded into Matlab using the built-in "wavread" function. The example on the right shows a 1.5825 sec duration audio recording of the spoken phrase "Testing, one, two, three" recorded at 44000 Hz, saved in WAV format (link), loaded into iSignal and zoomed in on the "oo" sound in the word "two". Press **Spacebar** to play the selected sound. Press **Shift-S** to display the frequency spectrum (page 94) of the selected region.

```
>> v=wavread('TestingOneTwoThree.wav');
>> t=0:1/44001:1.5825;
>> isignal(t,v(:,2));
```

Press **Shift-Z** to label the peaks in the frequency spectrum with their frequencies (right). Press **Shift-R** and type 44000 to set the sampling rate.



This recorded sound example allows you to experiment with the effect of smoothing, differentiation, and interpolation on the sound of recorded speech. Interestingly, different degrees of smoothing and differentiation will change the timbre of the voice but has *surprisingly little effect on the intelligibility.* This is because speech depends on the sequence of frequency components in the signal, which is not shifted in pitch or in time but merely changed in amplitude by smoothing and differentiation. Even computing the *absolute value* (+ key), which effectively doubles the fundamental frequency, does not make the sound unintelligible.

Shift-Ctrl-F transfers the current signal to Interactive Peak **F**itter (ipf.m, page 416) and **Shift-Ctrl-P** to transfer the current signal to Interactive **P**eak Detector (iPeak.m, page 255), if those functions are installed in your Matlab search path.

Press **K** to see *all* the keyboard commands.

EXAMPLE 1: Single input argument; data in two columns of a matrix [x;y] or in a single y vector

```
>> isignal(y);
>> isignal([x;y]);
```

EXAMPLE 2: Two input arguments. Data in separate x and y vectors.

```
>> isignal(x,y);
```

EXAMPLE 3: Three or four input arguments. The last two arguments specify the initial values of pan (xcenter) and zoom (xrange) in the last two input arguments. Using data in the ZIP file:

```
>> load data.mat
>> isignal(DataMatrix,180,40); or
>> isignal(x,y,180,40);
```

EXAMPLE 4: As above, but additionally specifies initial values of SmoothMode, SmoothWidth, ends, and DerivativeMode in the last four input arguments.

```
>> isignal(DataMatrix,180,40,2,9,0,1);
```

EXAMPLE 5: As above, but additionally specifies initial values of the peak sharpening parameters Sharpen, Sharp1, and Sharp2 in the last three input arguments. Press the **E** key to toggle sharpening on and off for comparison.

```
>> isignal(DataMatrix,180,40,4,19,0,0,1,51,6000);
```

EXAMPLE 6:
Using the built-in "humps" function:

```
>> x=[0:.005:2];y=humps(x);Data=[x;y];
```

4th derivative of the peak at x=0.9:

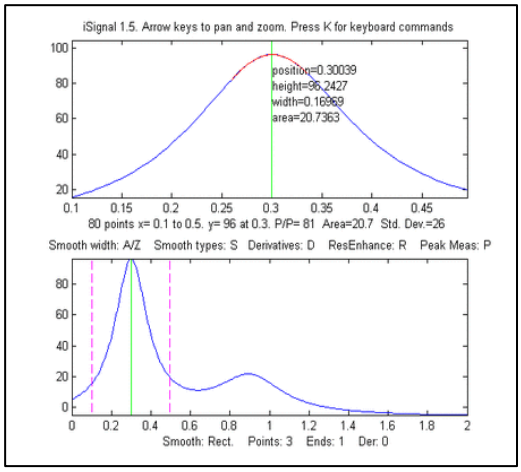
```
>> isignal(Data,0.9,0.5,1,3,1,4);
```

(You can see these animations run if you download the Microsoft Word 365 version, otherwise click on the figures).

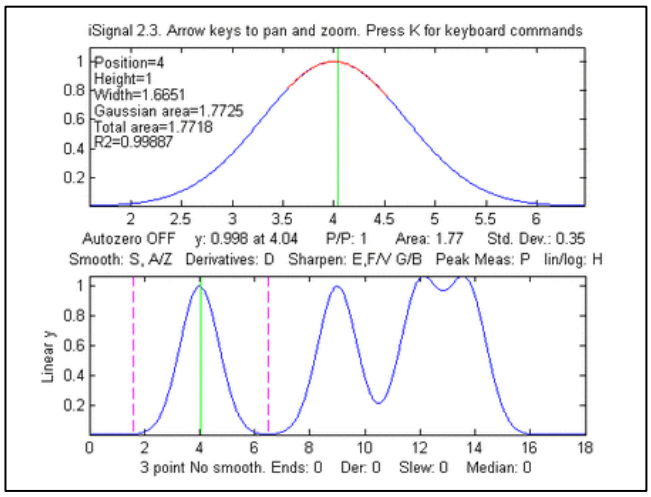
Peak sharpening applied to the peak at x=0.3:

```
isignal(Data,0.3,0.5,1,3,1,0,1,220,5400);
```

(Press **'E'** key to toggle sharpening ON/OFF.)

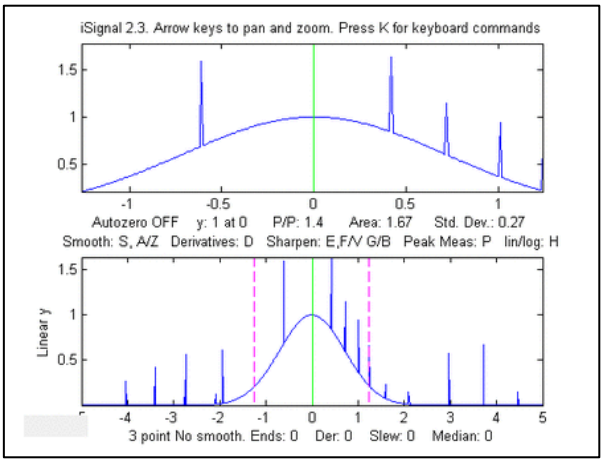


EXAMPLE 7: Measurement of peak area. This example generates four Gaussian peaks, all with the exact same peak height (1.00) and area (1.77). Click figure for animated GIF.




```
>> x=[0:.01:20];
>> y=exp(-(x-4).^2)+exp(-(x-9).^2)+exp(-(x-13).^2)+exp(-(x-15).^2);
>> isignal(x,y);
```

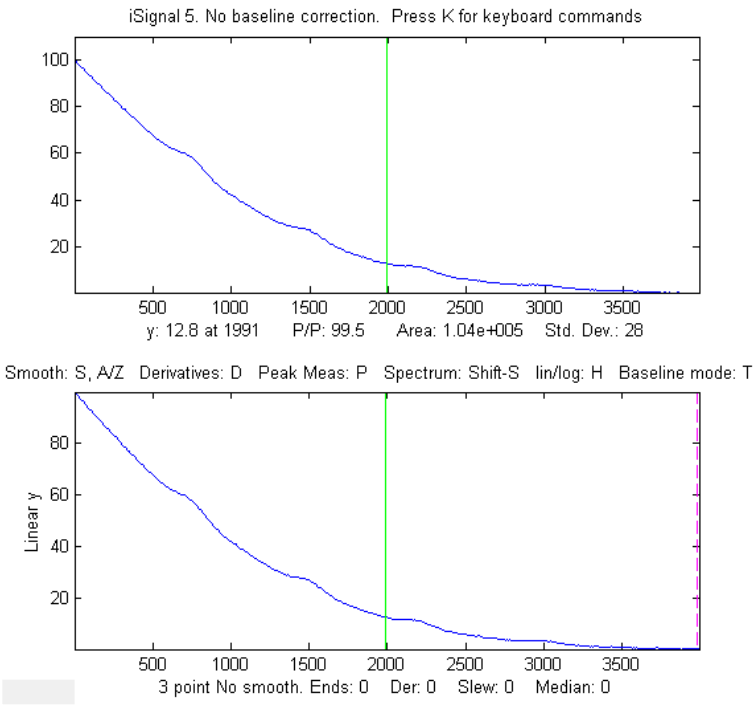
The first peak (at x=4) is isolated, the second peak (x=9) is slightly overlapped with the third one, and the last two peaks (at x= 13 and 15) are strongly overlapped. To measure the area under a peak using the perpendicular drop method (page 142), position the dotted red marker lines at the minimum between the overlapped peaks. Greater accuracy in peak area measurement using iSignal can be obtained by using the peak sharpening function to reduce the overlap between the peaks. This reduces the peak widths, increases the peak heights, but has no effect on the peak areas.



EXAMPLE 8: Single peak with random spikes (shown in the figure on the right). Compare smoothing vs spike filter (**M** key) and slew rate limit (~ key) to remove spikes.

```
x=-5:.01:5;
y=exp(-(x).^2);
for n=1:1000,
     if randn()>2,
          y(n)=rand()+y(n),
     end,
end;
isignal(x,y);
```

EXAMPLE 9: Weak peaks on a strong baseline.



The demo script isignaldemo2 (shown on the left) creates a test signal containing four peaks with heights 4, 3, 2, 1, with equal widths, superimposed on a very strong curved baseline, plus added random white noise. The objective is to extract a measure that is proportional to the peak height but independent of the baseline strength. Suggested approaches: (a) Use automatic or manual baseline subtraction to remove the baseline, measure peaks with the P-P measure in the upper panel. Or (b) use differentiation (with smoothing) to suppress the baseline. Or (c) use curve fitting (**Shift-F**), with baseline correction (**T**), to measure peak height. After running the script, you can press **Enter** to have the script perform an automatic 3rd derivative calibration, performed by lines 56 to 74. As indicated in the script, you can change several of the constants. Search for the word "change". (To use the derivative method, the *width* of the peaks must all be equal and stable, but the peak *positions* may vary within limits, set by the Xrange for each peak in lines 61-67). You must have isignal.m and plotit.m installed.

EXAMPLE 10: Direct entry into frequency spectrum mode, plotting returned frequency spectrum.

```
>> x=0:.1:60; y=sin(x)+sin(10.*x);
>> [pY, SpectrumOut]=isignal([x;y],30,30,4,3,1,0,0,1,0,0,0,1);
>> plot(SpectrumOut)
```

EXAMPLE 11: The demo script demoisignal.m is a self-running demo that requires iSignal 4.2 or later and the latest version of plotit.m to be installed.

EXAMPLE 12: Here's a simple example of a very noisy signal with lots of high-frequency (blue) noise obscuring a perfectly good peak in the center at x=150, height=1e-4; SNR=90. First, download the data file NoisySignal.mat into the Matlab search path, then execute these statements:

```
>> load NoisySignal
>> isignal(x,y);
```

Use the **A** and **Z** keys to increase and decrease the smooth width, and the **S** key to cycle through the available smooth type. Hint: use the P-spline smooth and keep increasing the smooth width. Zoom in on the peak in the center, press **P** to enter the peak mode, and it will display the characteristics of the peak in the upper left.

***iSignal* keyboard controls (Version 8.3):**

```
Pan signal left and right...Coarse pan: < and >
                            Fine pan: left and right cursor arrows
                            Nudge: [ and ]
 Zoom in and out.............Coarse zoom: / and "
                            Fine zoom: up and down cursor arrows
 Resets pan and zoom.........ESC
 Select entire signal........Ctrl-A
 Display grid................Shift-G  temporarily display grid on both panels
 Adjust smooth width.........A,Z (A=>more, Z=>less)
 Set smooth width vector.....Shift-Q  for segmented smooth
 Cycle smooth types..........S (No, Rect., Triangle, Gaussian, Savitzky-Golay)
 Toggle smooth ends..........X (0=ends zeroed  1=ends smoothed (slower)
 Symmetrize (de-tailing)     Shift-Y allows entry of symmetrize weighting factor
 Adjust Symmetrization.......1,2 keys: decrease, increase by 10%
                            Shift-1,Shift-2 decrease, increase by 1%
 ConV/deconVolution mode.....Shift-V presents menu of conv/deconv choices
 Adjust width................3,4: decrease,increase by 10%
                            Shift-3,Shift-4: decrease,increase by 1%
 Cycle derivative orders.....D/Shift-D Increase/Decrease derivative order
 Toggle peak sharpening......E (0=OFF 1=ON)
 Sharpening for Gaussian.....Y  Set sharpen settings for Gaussian
 Sharpening for Lorentzian...U  Set sharpen settings for Lorentzian
 Adjust sharp1...............F,V  F=>sharper, V=>less sharpening
 Adjust sharp2   ............G,B  G=>sharper, B=>less sharpening
 Slew rate limit (0=OFF).....~  Largest allowed y change between points
 Spike filter width (O=OFF)..m  spike filter eliminates sharp spikes
 Toggle peak parabola........P  fits parabola to center, labels vertex
 Fit polynomial to segment...Shift-o  Asks for polynomial order
 Fits peak in upper window...Shift-F (Asks for shape, number of peaks, etc.)
 Find peaks in lower panel...J (Asks for Peak Density)
 Find peaks in upper panel...Shift-J (Asks for Peak Density)
 Spectrum mode on/off........Shift-S (Shift-A and Shift-X to change axes)
 Peak labels on spectrum.....Shift-Z in spectrum mode
 Click graph to print x,y....Shift-C  Click graph to print coordinates
 Display Waterfall spectrum..Shift-W  Allows choice of mesh, surf, contour, etc.
 Transfer power spectrum.....Shift-T  Replaces signal with power spectrum
 Lock in current processing..Shift-L  Replace signal with processed version
 ConVolution/DeconVolution...Shift-V  Convolution/Deconvolution menu
 Power transform method..... ^ (Shift-6) Raises the signal to a specified power.
 Print peak report...........R  prints position, height, width, area
 Toggle log y mode...........H  semilog plot in lower window
 Cycles baseline mode........T  none, linear, quadratic, or flat baseline mode
 Restores original signal....Tab or Ctrl-Z key resets to original signal and modes
 Toggle overlay mode.........L  Overlays original signal as dotted line
 Display current signals.....Shift-B  Original (top) vs Processed (bottom)
 Baseline subtraction........Backspace, then click baseline at multiple points
 Restore background..........\  to cancel previous background subtraction
 Invert signal...............Shift-N  Invert (negate) the signal (flip + and -)
 Remove offset...............0  (zero) set minimum signal to zero
 Sets region to zero.........;  sets selected region to zero
 Absolute value..............+  Computes absolute value of entire signal
 Condense signal.............C  Condense oversampled signal by factor of N
 Interpolate signal..........i  Interpolate (resample) to N points
 Print report................Q  prints signal info and current settings
 Print keyboard commands.....K  prints this list of keyboard commands
 Print isignal arguments.....W  prints isignal function with all current arguments
```

```
Save output to disk.........O  Save .mat file with processed signal matrix
Play signal as sound........Spacebar or Shift-P  Play selection through speaker
Play signal as sound........Shift-R Change sampling rate for playing sound
Expand to full screen.......Double-click figure window title bar
Switch to ipf.m.............Shift-Ctrl-F  transfers current signal to
                             Interactive Peak Fitter, ipf.m
Switch to iPeak.............Shift-Ctrl-P  transfers current signal to
                             Interactive Peak Detector, ipeak.m
```

ProcessSignal, a Matlab/Octave command-line function that performs smoothing and differentiation on the time-series data set x,y (column or row vectors). Type "help ProcessSignal". It returns the processed signal as a vector that has the same shape as x, regardless of the shape of y. The syntax is

```
Processed=ProcessSignal(x,y,DerivativeMode,w,type,ends,Sharpen,factor1,
factor2,Symize,Symfactor,SlewRate,MedianWidth)
```

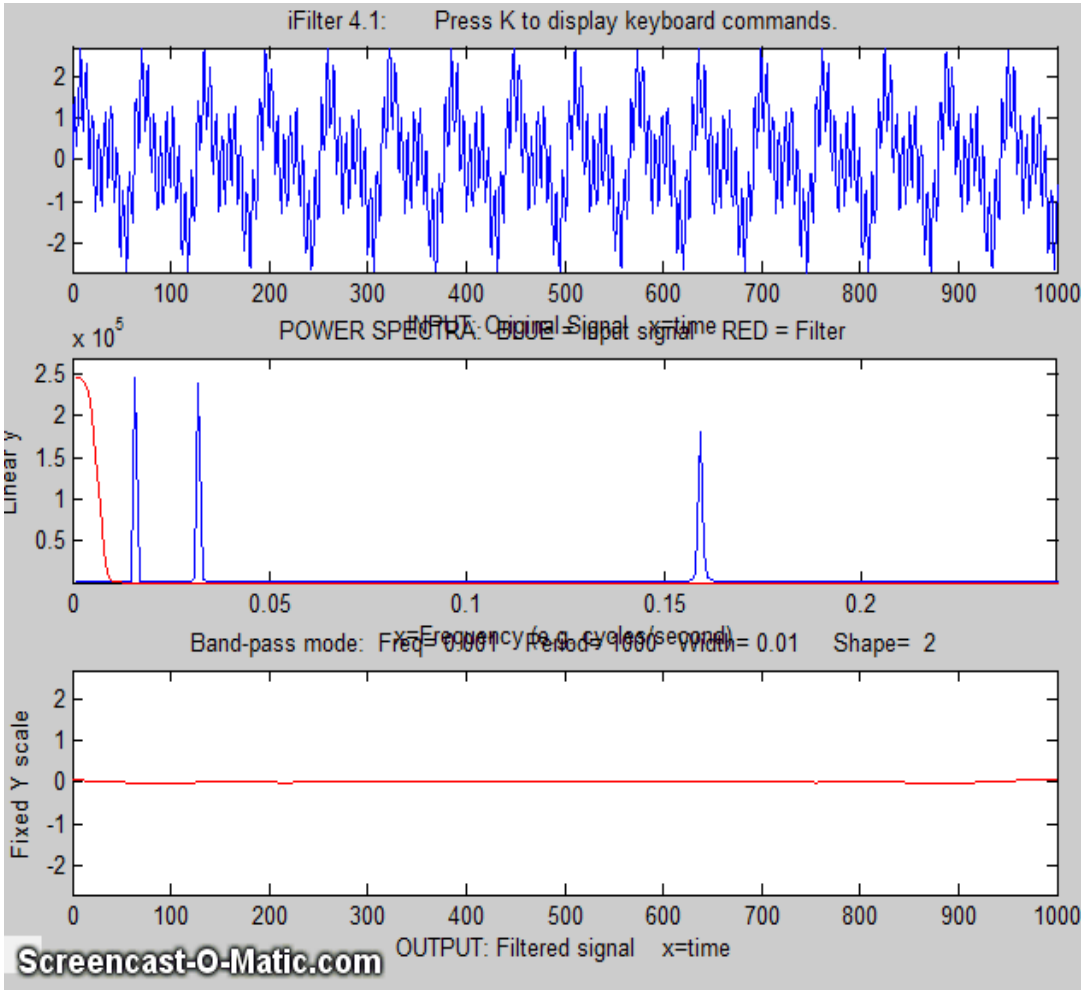
Keyboard-operated interactive Fourier filter

iFilter.m is a keyboard-operated interactive Fourier filter Matlab function, for time-series signal (x,y), with keyboard controls that allow you to adjust the filter parameters continuously while observing the effect on your signal dynamically. Optional input arguments set the initial values of center frequency, filter width, shape, plotmode (1=linear, 2=semilog frequency, 3=semilog amplitude, 4=log-log) and filter mode ('band-pass', 'low-pass', 'high-pass', 'band-reject (notch), 'comb pass', and 'comb notch'). In the comb modes, the filter has multiple bands located at frequencies 1, 2, 3, 4... multiplied by the center frequency, each with the same (controllable) width and shape. The interactive keypress operation works even if you run Matlab in a web browser, but not on Matlab Mobile. Octave users must use the alternative version ifilteroctave, which uses different keys for the filter center and width adjustment and works in the most recent version of Octave.

The filtered signal can be returned as the function value, saved as a ".mat" file on the disk, or played through the computer's sound system. Press **K** to list keyboard commands. This is a self-contained Matlab function that does not require any toolboxes or add-on functions. Click here to view or download and place it in the Matlab search path. At the Matlab command prompt, type:

```
FilteredSignal=ifilter(x,y)  or ifilter(y)  or ifilter(xymatrix) or
FilteredSignal=ifilter(x,y,center,width,shape,plotmode,filtermode)
```

Animation showing the operation of iFilter.m



Example 1 Periodic waveform with 2 frequency components at 60 and 440 Hz. (You can see this animation in the Microsoft Word 365 version, otherwise click the figure).

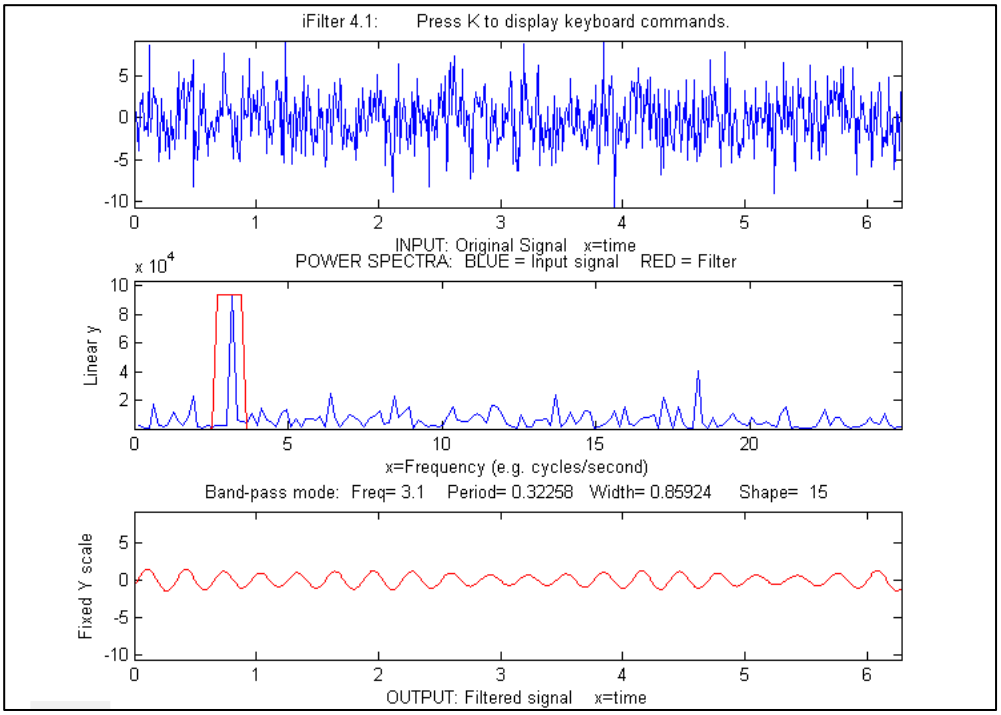
```
x=[0:.001:2*pi];
y=sin(60.*x.*2.*pi)+2.*sin(440.*x.*2.*pi);
ifilter(x,y);
```

Example 2: uses optional input arguments to set initial values:

```
x=0:(1/8000):.3;
y=(1+12/100.*sin(2*47*pi.*x)).*sin(880*pi.*x)+(1+12/100.*sin(2*20*pi.*x)).
*sin(2000*pi.*x);
ry=ifilter(x,y,440,31,18,3,'Band-pass');
```

Example 3: Picking one frequency out of a noisy sine wave (shown below).

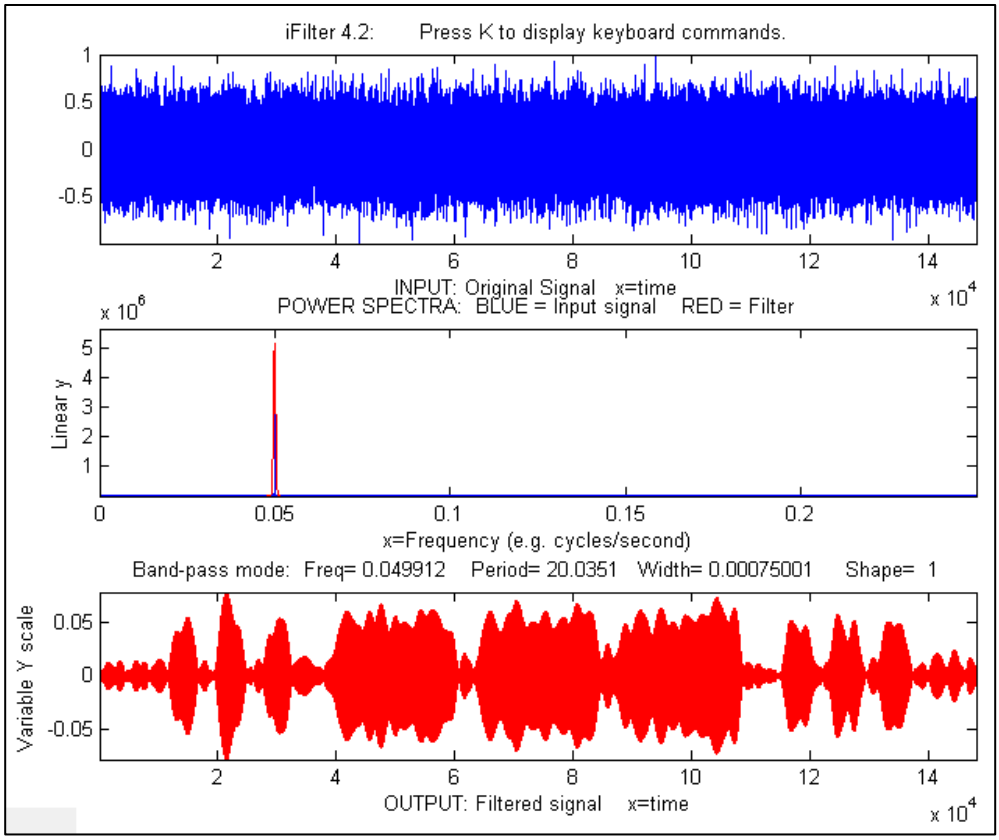
```
x=[0:.01:2*pi]';
y=sin(20*x)+3.*randn(size(x));
ifilter(x,y,3.1,0.85924,15,1,'Band-pass');
```



Example 4: Square wave with band-pass vs Comb pass filter

```
t = 0:.0001:.0625;
y=square(2*pi*64*t);
ifilter(t,y,64,32,12,1,'Band-pass');
ifilter(t,y,48,32,2,1,'Comb pass');
```

Example 5: MorseCode.m (graphic on next page) uses iFilter to demonstrate the abilities and limitations of Fourier filtering. (Graphic on next page.) It creates a pulsed fixed frequency sine wave that spells out "SOS" in Morse code (dit-dit-dit/dah-dah-dah/dit-dit-dit), then adds random white noise so that the SNR is very poor (about 0.1 in this example). The white noise has a frequency spectrum that is. The signal itself is concentrated mostly at a fixed frequency (0.05) but the modulation of the sine wave by the Morse Code pulses spreads out its spectrum over a narrow frequency range of about 0.0004. This suggests that a Fourier bandpass filter tuned to the signal frequency might be able to isolate the signal from the noise. As the bandwidth is reduced, the signal-to-noise ratio improves and the signal begins to emerges from the noise until it becomes clear, but if the bandwidth is too narrow, the *step response time* is too slow to give distinct "dits" and "dahs". The step response time is inversely proportional to the bandwidth. (Use the ? and " keys to adjust the bandwidth. Press '**P**' or the Spacebar to hear the sound). You can actually *hear* that sine wave component better than you can *see* it in the waveform plot (upper panel), because the ear works like a spectrum analyzer, with separate nerve endings assigned to specific frequency ranges, whereas the eye analyzes the graph *spatially*, looking at the overall amplitude and not at individual frequencies. Click for mp4 video of this script in operation, with sound. This video is also on YouTube at https://youtu.be/agjs1-mNkmY.

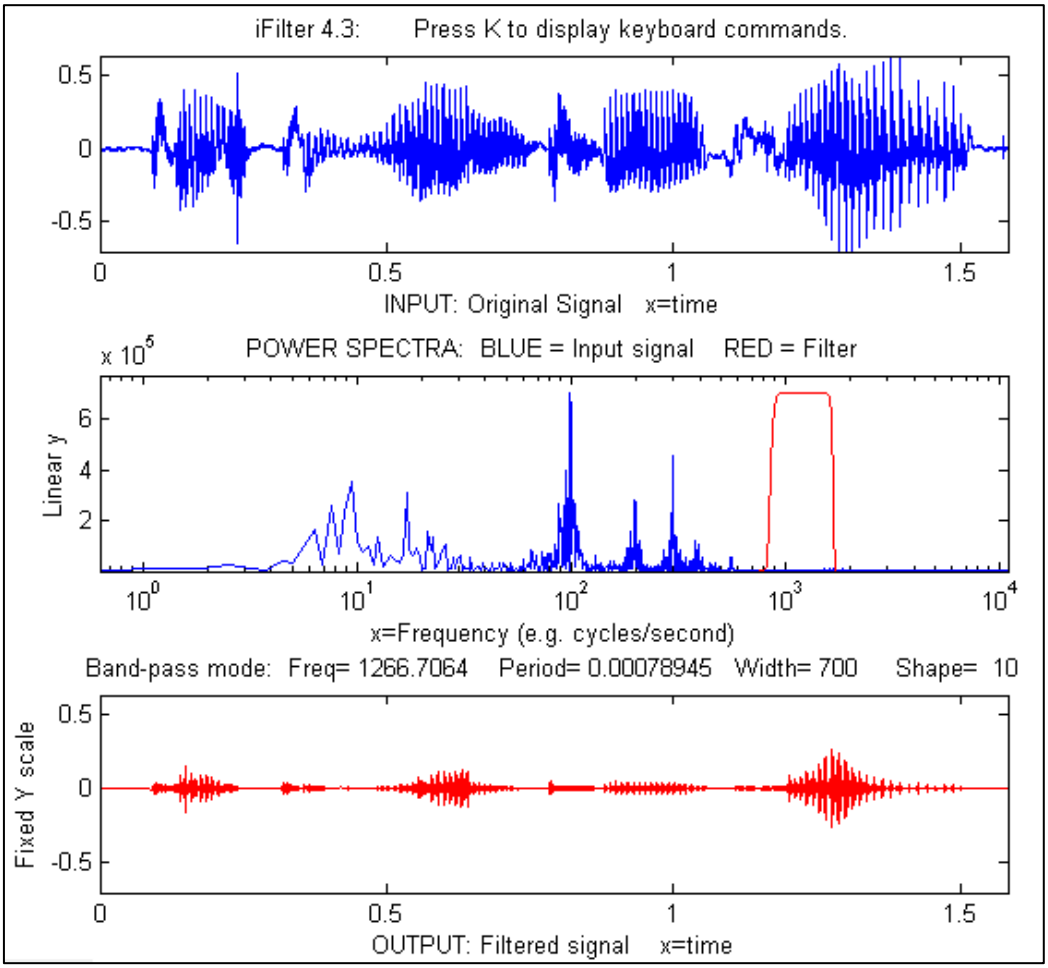


Example 6: This example (graphic on next page) shows a 1.6 sec duration audio recording of the spoken phrase "Testing, one, two, three", recorded at 44001 Hz and saved in both WAV format (download link) and in ".mat" format (download link). This latter file is loaded into iFilter like so:

```
load testing123
t=(1:length(testing123))/44000;
y=testing123(:,1);
ifilter(t,y);
```

iFilter is set to bandpass mode and tuned to a narrow segment above 1000 Hz that is *well above* the frequency range of most of the signal range. *That passband rejects most of the visible frequency components in the signal*, yet even in that case, the speech is still intelligible, demonstrating the remarkable ability of the ear-brain system to make do with a highly compromised signal, as long as the filter passes the frequencies above 500 Hz. Press **P** or **space** to hear the filter's output on your computer's sound system. Different filter settings will change the timbre of the sound, but you can still understand it.

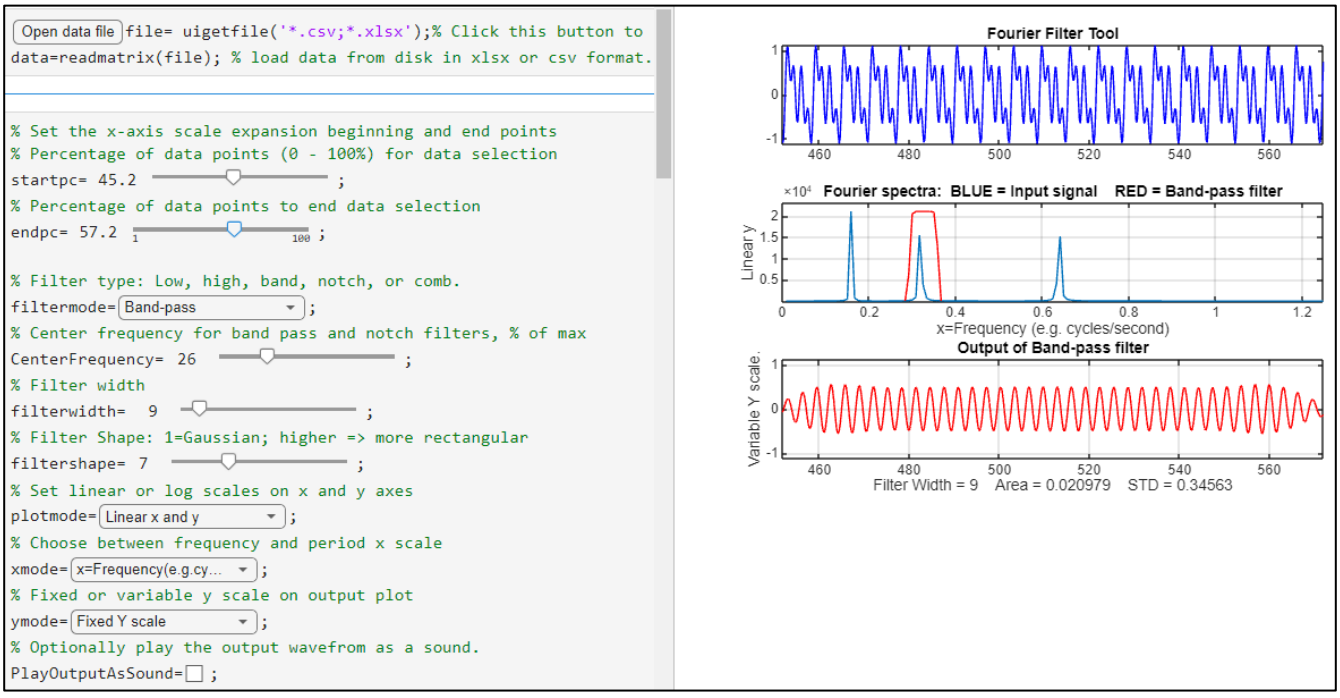
Note: to change any of the above iFilter demo scripts to Octave, simply change "ifilter" to "ifilteroctave" and make sure that ifilteroctave is in the path.



***iFilter* 4.3 Matlab version keyboard controls** (if the figure window is not topmost, click on it first): (The Octave version uses different keys for the filter center and width adjustment.)

```
Adjust filter frequency........Coarse (10% change): < and >
                               Fine (1% change): left and right cursor
                               arrows
Adjust filter width............Coarse (10% change): / and "
                               Fine (1% change): up and down cursor arrows
Filter shape...................A,Z (A more rectangular, Z more Gaussian)
Filter mode....................B=bandpass; N or R=notch (band reject);
                               H=High-pass;
                               L=Low-pass; C=Comb pass; V=Comb notch.
Select plot mode...............1=linear; 2=semilog frequency
                               3=semilog amplitude; 4=log-log
Print keyboard commands........K  Prints this list
Print filter parameters........Q or W Prints ifilter with input
                                arguments: center, width, shape,
                                plotmode, filtermode
Print current settings.........T  Prints list of current settings
Switch SPECTRUM X-axis scale...X switch between frequency and period
                                on the horizontal axis
Switch OUTPUT Y-axis scale.....Y switch output plot between fixed or
                                variable vertical axis.
Play output as sound...........P or Enter
Save output as .mat file.......S
```

Live Script Fourier Filter



FourierFilterTool.mlx is a Live Script version of the interactive Fourier filter iFilter.m. It can perform six different types of Fourier filtering on your own data stored in .csv or .xls format. The graphic display (above right) is essentially identical to the iFilter.m function, in this case showing a band-pass filter isolating one frequency from a complex periodic waveform.

To view the figures to the right of the control panel as shown above, right-click on the right-hand panel and select "Disable synchronous scrolling".

Clicking the "Open data file" button in line 1 opens a file browser, allowing you to navigate to your data file (in .csv or .xlsx format). The script assumes that your x,y data are in the first two columns. You can change that in lines 27 and 28. In the case shown here, the data file is 'ThreeSines.csv', shown as the 'file' variable in the Matlab workspace. The **startpc** and **endpc** sliders in lines 5 and 7 allow you to select which portion of the data range to process, from 0% to 100% of the total range of the data file.

There are drop-down menus and sliders to select the type of filter type, center frequency, filter width, filter shape, linear or log axes, time or frequency mode, and fixed or variable Y axis for the output.

Clicking the checkbox labeled “PlayOutputAsSound” will send the output waveform to the system’s speaker each time any control is changed. Un-check that box to cancel. (See iFilterTesting123.m for an interesting example of that function).

Matlab/Octave Peak Fitters

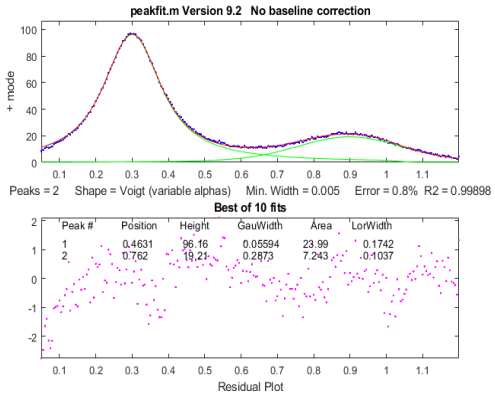
I have developed two Matlab peak fitting program for time-series signals, which uses an unconstrained non-linear optimization algorithm (page 199) to decompose a complex, overlapping-peak signal into its component parts. The objective is to determine whether your signal can be represented as the sum of fundamental underlying peaks shapes. Accepts signals of any length, including those with non-integer and non-uniform x-values. There are **two different versions**,

(1) a command line version (**peakfit.m**) for Matlab or Octave, Matlab File Exchange "Pick of the Week". The current version number is **9.61.**

(2) a keypress operated interactive version (**ipf.m or ipfoctave.m),** page 416. The current version number is 13.4. The corresponding demo script is Demoipfoctave.m

The difference between them is that peakfit.m is completely controlled by command-line input arguments and returns its information via command-line output arguments. Ipf.m allows interactive control via keypress commands. For automating the fitting of large numbers of signals, **peakfit.m** is better (see page 345), But **ipf.m** is best for exploring signals to determine the optimum fitting range, peak shapes, number of peaks, baseline correction mode, etc. Otherwise, they have similar curve-fitting capabilities. The basic built-in peak shape models available are illustrated on page 424. Custom peak shapes can be added (see page 437). See pages 433 and 438 for more information and useful suggestions.

Matlab/Octave command-line function: peakfit.m



Peakfit.m is a user-defined command window peak fitting function for Matlab or Octave, usable from a remote terminal. It is written as a self-contained function in a single m-file. (To view of download, click peakfit.m). It takes data in the form of a 2 by n matrix that has the independent variables (X-values) in row 1 and the dependent variables (Y-values) in row 2, or as a single dependent variable vector. The syntax is **`[FitResults, GOF, baseline, coeff, residuals, xi, yi, BootResults]=peakfit(signal, center, window, NumPeaks, peakshape, extra, NumTrials, start, BASELINEMODE, fixedparameters, plots, bipolar, minwidth, DELTA, clipheight)`**. Only the first input argument, the data matrix, is absolutely required. There are default values for all other inputs. All the input and output arguments are explained below.

The screen display is shown on the right. The upper panel shows the data as **blue dots**, the combined model as a **red line** (ideally overlapping the **blue dots**), and the model components as **green lines**. The dotted **magenta** lines are the first-guess peak positions for the last fit. The lower panel shows the residuals (difference between the data and the model).

You can download a ZIP file containing peakfit.m, DemoPeakFit.m, ipf.m, Demoipf.m, some sample data for testing. The test script testpeakfit.m or autotestpeakfit.m runs all the peakfit examples

sequentially to test for proper operation on your computer or version of Matlab/Octave. Takes 25 seconds.

For a discussion of the accuracy and precision of peak parameter measurement using peakfit.m, click here or see page 167.

The peakfit.m functionality can also be accessed by the keypress-operated interactive functions **ipf** (page 416), **iPeak** (page 416)**,** and **iSignal** (page 376) for *Matlab* or *Matlab Online* or the Octave versions whose names end in "octave".

Version 9.62: July 2021. Fixed a bug in the Voigt peak shape. Previous versions added peak shape **50** which implements the multilinear regression ("classical least-squares") method for cases in which the peak shapes, positions, and widths are *all known* and *only* the peak heights are to be determined. See **Example 40** below and the demonstration scripts peakfit9demo.m and peakfit9demoL.m for a demonstration and comparison of this to unconstrained iterative fitting.

Peakfit.m can be called with several optional additional command line arguments. *All input arguments (except the signal itself) can be replaced by zeros to use their default values.*

`peakfit(signal);`
Performs an iterative least-squares fit of a single unconstrained Gaussian peak to the entire data matrix "signal", which has x values in row 1 and Y values in row 2 (e.g. [x y]) or which may be a single signal vector (in which case the data points are plotted against their index numbers on the x axis).

`peakfit(signal, center, window);`
Fits a single unconstrained Gaussian peak to a portion of the matrix "signal". The portion is centered on the x-value "center" and has width "window" (in x units).

In this and in all following examples, ***set "center" and "window" both to 0 to fit the entire signal****.*

`peakfit(signal, center, window, NumPeaks);`
"NumPeaks" = number of peaks in the model (default is 1 if not specified).

`peakfit(signal, center, window, NumPeaks, peakshape);`
Number or vector that specifies the peak shape(s) of the model: **1**=unconstrained Gaussian, **2**=unconstrained Lorentzian, **3**=logistic *distribution*, **4**=Pearson, **5**=exponentially broadened Gaussian, **6**=equal-width Gaussians, **7**=equal-width Lorentzians, **8**=exponentially broadened equal-width Gaussians, **9**=exponential pulse, **10**= up-sigmoid (logistic *function*), **11**=fixed-width Gaussians, **12**=fixed-width Lorentzians, **13**=Gaussian/Lorentzian blend, **14**=bifurcated Gaussian, **15**=Breit-Wigner-Fano resonance, **16**=Fixed-position Gaussians, **17**=Fixed-position Lorentzians, **18**=exponentially broadened Lorentzian, **19**=alpha function, **20**=Voigt profile, **21**=triangular, **23**=down-sigmoid, **25**=lognormal distribution, **26**=linear baseline (see Example 28), **28**=polynomial (extra=polynomial order, Example 30), **29**=articulated linear segmented (see Example 29), **30**=independently-variable alpha Voigt, **31**=independently-variable time constant ExpGaussian, **32**=independently-variable Pearson, **33**=independently-variable Gaussian/Lorentzian blend, **34**=fixed-width Voigt, **35**=fixed-width Gaussian/Lorentzian blend, **36**=fixed-width exponentially-broadened

Gaussian, **37**=fixed-width Pearson;**38**= independently-variable time constant ExpLorentzian, **39=** alternative independently-variable time constant ExpGaussian (see Example 39 below), **40**=sine wave, **41**=rectangle, **42**=flattened Gaussian, **43**=Gompertz function (3 parameter logistic: `Bo*exp(-exp((Kh*exp(1)/Bo)*(L-t)+1))`), **44**=1-exp(-k*x), **45**: Four-parameter logistic y = maxy*(1+(miny-1)/(1+(x/ip)^slope)), **46**=quadratic baseline (see Example 38), **47**=blackbody emission, **48**=equal-width exponential pulse, **49**=Pearson IV, **50**=multilinear regression (known peak positions and widths). The function ShapeDemo demonstrates most of the basic peak shapes (graphic on page 424) showing the variable-shape peaks as multiple lines.

> **Note 1**: "unconstrained" simply means that the position, height, and width of each peak in the model can vary independently of the other peaks, as opposed to the equal-width, fixed-width, or fixed position variants. Shapes 4, 5, 13, 14, 15, 18, 20, and 34-37 are constrained to the same *shape constant*. Shapes 30-33 are completely unconstrained in position, width, *and* shape. Their shape variables are determined by iteration.
>
> **Note 2**: The value of the shape constant "extra" defaults to 1 if not specified in input arguments.
>
> **Note 3**: The peakshape argument can be a *vector of different shapes for each peak*, e.g. [1 2 1] for three peaks in a Gaussian, Lorentzian, Gaussian sequence. (The next input argument, 'extra', must be a vector of the same length as 'peakshape'. See **Examples 24**, **25**, **28** and **38**, below.

`peakfit(signal, center, window, NumPeaks, peakshape, extra)`

Specifies the value of 'extra', used in the Pearson, exponentially-broadened Gaussian, Gaussian/Lorentzian blend, bifurcated Gaussian, and Breit-Wigner-Fano shapes to fine-tune the peak shape. The value of "extra" defaults to 1 if not specified in input arguments. In version **5**, 'extra' can be a vector of different extra values for each peak).

`peakfit(signal, center, window, NumPeaks, peakshape, extra, NumTrials);`

Restarts the fitting process "NumTrials" times *with slightly different start values* and selects the best one (with lowest fitting error). NumTrials can be any positive integer (default is 1). In many cases, NumTrials=1 will be sufficient, but if that does not give consistent results, increase NumTrials until the result are stable.

`peakfit(signal, center, window, NumPeaks, peakshape, extra, NumTrials, start)`

Specifies the first guesses vector "start" for the peak positions and widths, e.g., start=[position1 width1 position2 width2 ...]. Only necessary for difficult cases, especially when there are a lot of adjustable variables. The start vector can be the approximate average values based on your experience, or it can be calculated from a previous simpler fit, as in this example. If you leave off “start”, or set it to zero, the program will generate its own start rough values (which is often good enough). See examples 14, 22, 28, 40, and 42 below for situations where specifying the start values is useful or necessary.

peakfit(signal, center, window, NumPeaks, peakshape, extra, NumTrials, start, BaselineMode)

As above, but "BaselineMode" sets the baseline correction mode in the last argument: **BaselineMode**=0 (default) does *not* subtract baseline from data segment. **BaselineMode**=1 interpolates a *linear* baseline from the edges of the data segment and subtracts it from the signal (assumes that the peak returns to the baseline at the edges of the signal). **BaselineMode=2** is like mode 1 except that it computes a *quadratic* curved baseline. **BaselineMode=3** compensates for a *flat* baseline without reference to the signal itself (does not require that the signal return to the baseline at the edges of the signal, as does modes 1 and 2). Coefficients of the polynomial baselines are returned in the third output argument "baseline".

peakfit(signal,0,0,0,0,0,0,0,2)

Use zeros as placeholders to use the default values of input arguments. In this case, **BaselineMode** is set to 2, but all others are the default values.

peakfit(signal, center, window, NumPeaks, peakshape, extra, NumTrials, start, BaselineMode, fixedparameters)

'fixedparameters' (10th input argument) specifies fixed widths or positions in shapes 11, 12, 16, 17, 34-37, one entry for each peak. When using peak shape 50 (multlinear regression), 'fixedparameters' must be a matrix listing the peak shape number (column 1), position (column 2), and width (column 3) of each peak, one row per peak.

peakfit(signal, center, window, NumPeaks, peakshape, extra, NumTrials, start, BaselineMode, fixedparameters, plots)

'plots' (11th input argument) controls graphic plotting: 0=no plot; 1=plots draw as usual (default)

peakfit(signal, center, window, NumPeaks, peakshape, extra, NumTrials, start, BaselineMode, fixedparameters, plots, bipolar)

(12th input argument) 'bipolar' = 0 constrain peaks heights to be positive; 'bipolar' = 1 allows positive and negative peak heights.

peakfit(signal, center, window, NumPeaks, peakshape, extra, NumTrials, start, BaselineMode, fixedparameters, plots, bipolar,minwidth)

'minwidth' (13th input argument) sets the minimum allowed peak width. The default if not specified is equal to the x-axis interval. Must be a vector of minimum widths, one value for each peak, if the multiple peak shape is chosen.

peakfit(signal,center, window, NumPeaks, peakshape, extra, NumTrials, start, BaselineMode, fixedparameters, plots, bipolar, minwidth,DELTA)

'DELTA' (14th input argument) controls the restart variance when NumTrials>1. Default value is 1.0. Larger values give more variance. Version 5.8 and later only.

[FitResults,FitError]= peakfit(signal, center, window...);

Returns the FitResults vector in the order peak number, peak position, peak height, peak width, and peak area), and the FitError (the percent RMS difference between the data and the model in the selected segment of the data) of the best fit.

Labeling the FitResults table: Using the "table" function, you can display FitResults in a neat table with column labels, using only a single line of code:

```
disp(table(FitResults(:,2), FitResults(:,3), FitResults(:,4),
FitResults(:,5), 'VariableNames', {'Position' 'Height' 'FWHM' 'Area'}))
```

```
    Position      Height       FWHM        Area
    ________    ________    _______    __________
    8.0763      3.8474      10.729     3.4038e-01
        20           1         3          3.1934
```

Additional columns of FitResults and VariableNames can be added for those peak shapes that display five or more results, such as the Voight shape:

```
disp(table(FitResults(:,2), FitResults(:,3), FitResults(:,4),
FitResults(:,5), FitResults(:,6), 'VariableNames', {'Position' 'Height'
'GauWidth' 'Area' 'LorWidth'}))
```

```
    Position    Height     GauWidth     Area      LorWidth
  _________   ________   _________   _______   _________
    0.80012     0.99987    0.30272     0.34744    0.39708
     1.2003     0.79806    0.40279     0.27601    0.30012
```

Calculating the precision of the peak parameters:

```
[FitResults,GOF,baseline,coeff,residuals,xi,yi,BootstrapErrors] =
peakfit([x;y],0,0,2,6,0,1,0,0,0);
```

Displays parameter error estimates by the *bootstrap method*. See page 170.

Optional output parameters:

1. **FitResults**: a table of model peak parameters, one row for each peak, listing peak number, peak position, height, width, and area (or, for shape 28, the polynomial coefficients, and for shape 29, the x-axis breakpoints).
2. **GOF ("Goodness of Fit")**, a 2-element vector containing the RMS fitting error of the best trial fit and the R-squared (coefficient of determination).
3. **baseline**: returns the polynomial coefficients of the interpolated baseline in linear and quadratic baseline modes (1 and 2) or the value of the constant baseline in flat baseline mode.
4. **coeff**: Coefficients for the polynomial fit (shape 28 only; for other shapes, coeff=0)
5. **residual**: vector of differences between the data and the best fit model. Can be used to measure the characteristics of the noise in the signal.
6. **xi**: vector containing 600 interpolated x-values for the model peaks.
7. **yi**: matrix containing the y values of model peaks at each xi. Type `plot(xi,yi(1,:))` to plot peak 1 or `plot(xi,yi)` to plot all the peaks.
8. **BootstrapErrors**: the presence of this triggers the bootstrap estimations of the standard deviations and interquartile ranges for each peak parameter of each peak in the fit (page 170).

Examples

Note: test script testpeakfit.m runs all the following examples automatically. (You can copy and paste,

or drag and drop, any of these single-line or multi-line code examples into the Matlab or Octave editor or into the command line and press **Enter** to execute it).

Example 1. Fits computed x vs y data with a single unconstrained Gaussian peak model.

```
> x=[0:.1:10];y=exp(-(x-5).^2); peakfit([x' y'])
ans =
       Peak number  Position   Height     Width      Peak area
            1           5        1        1.665       1.7725
```

Example 2. Fits small set of manually-entered y data to a single unconstrained Gaussian peak model.

```
> y=[0 1 2 4 6 7 6 4 2 1 0 ]; x=1:length(y);
> peakfit([x;y],length(y)/2,length(y),0,0,0,0,0,0)
       Peak number  Position   Height     Width      Peak area
           1        6.0001       6.9164     4.5213         32.98
```

Example 3. Measurement of very noisy peak with signal-to-noise ratio = 1. (Try several times).

```
> x=[0:.01:10];y=exp(-(x-5).^2) + randn(size(x)); peakfit([x;y])
       Peak number  Peak position   Height     Width      Peak area
           1        5.0951       1.0699       1.6668       1.8984
```

Example 4. Fits a noisy two-peak signal with a double unconstrained Gaussian model (NumPeaks=2).

```
> x=[0:.1:10]; y=exp(-(x-5).^2)+.5*exp(-(x-3).^2) + .1*randn(1,length(x));
> peakfit([x' y'],5,19,2,1,0,1)
       Peak number  Position   Height        Width      Peak area
            1       3.0001      0.49489        1.642      0.86504
            2       4.9927       1.0016       1.6597       1.7696
```

Example 5. Fits a portion of the humps function, 0.7 units wide and centered on x=0.3, with a single (NumPeaks=1) Pearson function (peakshape=4) with extra=3 (controls shape of Pearson function).

```
>> x=[0:.005:1];y=humps(x);
peakfit([x' y'],.3,.7,1,4,3);
```

Example 6. Creates a data matrix 'smatrix', fits a portion to a two-peak unconstrained Gaussian model, takes the best of 10 trials. Returns optional output arguments FitResults and FitError.

```
>> x=[0:.005:1]; y=(humps(x)+humps(x-.13)).^3; smatrix=[x' y'];
>> [FitResults,FitError]=peakfit(smatrix,.4,.7,2,1,0,10)

       Peak number  Position   Height        Width      Peak area
            1        0.4128  3.1114e+008      0.10448  3.4605e+007
            2        0.3161  2.8671e+008     0.098862  3.0174e+007
  FitError = 0.68048
```

Example 7. As above, but specifies the first-guess position and width of the two peaks, in the order [position1 width1 position2 width2]

```
> peakfit([x' y'],.4,.7,2,1,0,10,[.3 .1 .5 .1]);
```

Supplying a first guess position and width is also useful if you have one peak on top of another (like example 4, with both peaks at the same position x=5, but with different widths, in square brackets):

```
>> x=[2:.01:8];
>> y=exp(-((x-5)/.2).^2)+.5.*exp(-(x-5).^2) + .1*randn(1,length(x));
>> peakfit([x' y'],0,0,2,1,0,1,[5 2 5 1])
          Peak number   Position    Height         Width       Peak area
             1         4.9977       0.51229        1.639       0.89377
             2         4.9948        1.0017       0.32878      0.35059
```

Example 8. As above, returns the vector xi containing 600 interpolated x-values for the model peaks and the matrix yi containing the y values of each model peak at each xi. Type `plot(xi,yi(1,:))` to plot peak 1 or `plot(xi,yi,xi,sum(yi))` to plot all the model components and the total model (sum of components).

```
> [FitResults, GOF, baseline, coeff, residuals, xi, yi]= …
peakfit(smatrix,.4,.7,2,1,0,10);
> figure(2); clf; plot(xi,yi,xi,sum(yi))
```

Example 9. Fitting a single unconstrained Gaussian on a linear background, using the linear BaselineMode (9th input argument = 1)

```
>> x=[0:.1:10]';y=10-x+exp(-(x-5).^2);
>> peakfit([x y],5,8,0,0,0,0,0,1)
```

Example 10. Fits a group of three peaks near x=2400 in `DataMatrix3` with three equal-width exponentially broadened Gaussians.

```
>> load DataMatrix3
>> [FitResults,FitError]= peakfit(DataMatrix3,2400,440,3,8,31,1)
          Peak number   Position    Height         Width       Peak area
             1         2300.5       0.82546       60.535       53.188
             2         2400.4       0.48312       60.535       31.131
             3         2500.6       0.84799       60.535       54.635
FitError = 0.19975
```

Note: if your peaks are trailing off to the *left*, rather than to the right as in the above example, simply use a *negative* value for the time constant (in ipf.m, press **Shift-X** and type a negative value).

Example 11. Example of an unstable fit to a signal consisting of two unconstrained Gaussian peaks of equal height (1.0). The peaks are too highly overlapped for a stable fit, even though the fit error is small, and the residuals are unstructured. Each time you re-generate this signal, it gives a different fit, with the peak's heights varying about 15% from signal to signal.

```
>> x=[0:.1:10]';
>> y=exp(-(x-5.5).^2) + exp(-(x-4.5).^2) + .01*randn(size(x));
>> [FitResults,FitError]= peakfit([x y],5,19,2,1)
          Peak number   Position    Height         Width       Peak area
             1         4.4059       0.80119       1.6347       1.3941
             2         5.3931        1.1606       1.7697       2.1864
FitError = 0.598
```

Much more stable results can be obtained using the equal-width Gaussian model (`peakfit([x y],5,19,2,6)`), but that is justified only if the experiment is legitimately expected to yield peaks of equal width. See page 210 - 219.

Example 12. Baseline correction. Demonstrations of the four "BaselineModes", for a single Gaussian on large baseline, with position=10, height=1, and width=1.66. The BaselineMode is specified by the 9th input argument (which can be 0,1,2, or 3).

BaselineMode=0 means to ignore the baseline (default mode if not specified). In this case, this leads to large errors.

```
>> x=8:.05:12;y=1+exp(-(x-10).^2);
>> [FitResults,FitError,baseline]=peakfit([x;y],0,0,1,1,0,1,0,0)
          Peak#     Position       Height        Width       Area
             1          10       1.8561        3.612       5.7641
 FitError =5.387
```

BaselineMode=1 subtracts linear baseline from edge to edge. Does not work well in this case because the signal does not return completely to the baseline at the edges.

```
>> [FitResults,FitError,baseline]=peakfit([x;y],0,0,1,1,0,1,0,1)
         Peak#     Position       Height        Width       Area
            1       9.9984      0.96161       1.5586       1.5914
FitError = 1.9801
baseline = 0.0012608       1.0376
```

BaselineMode=2 subtracts quadratic baseline from edge to edge. Does not work well in this case because the signal does not return completely to the baseline at the edges.

```
>> [FitResults,FitError,baseline]=peakfit([x;y],0,0,1,1,0,1,0,2)
         Peak#     Position       Height        Width       Area
            1       9.9996      0.81762       1.4379       1.2501
FitError = 1.8205
baseline = -0.046619       0.9327       -3.469
```

BaselineMode=3 subtracts a flat baseline automatically, *without* requiring that the signal returns to baseline at the edges. This mode works best for this signal.

```
>> [FitResults,FitError,baseline]=peakfit([x;y],0,0,1,1,0,1,0,3)
         Peak#     Position       Height        Width       Area
            1           10       1.0001       1.6653       1.7645
FitError = 0.0037056
baseline = 0.99985
```

In some cases, you can regard the baseline as an additional "peak". In the following example, the baseline is strongly sloped, but straight. In that case the most accurate result is obtained by using a *two-shape* fit, specifying the peak shape as a *vector*, which fits the peak as a *Gaussian* (shape 1) and the baseline as a *variable-slope straight li*ne (**shape 26**).

```
>> x=8:.05:12;y=x + exp(-(x-10).^2);
>> [FitResults,FitError]=peakfit([x;y],0,0,2,[1 26],[1 1],1,0)

         Peak#     Position       Height        Width       Area
            1           10            1       1.6651       1.7642
            2        4.485      0.22297         0.05       40.045
FitError =0.093
```

In the following example, the baseline is *curved*, so you may be able to get good results with BaselineMode=2:

```
>> x=[0:.1:10]';y=1./(1+x.^2)+exp(-(x-5).^2);
>> [FitResults,FitError,baseline]=peakfit([x y],5,5.5,0,0,0,0,0,2)
          Peak#     Position       Height        Width       Area
             1       5.0091      0.97108        1.603       1.6569
FitError = 0.97661
baseline = 0.0014928    -0.038196      0.22735
```

Example 13. Same as example 4, but with *fixed-width* Gaussian (shape 11), width=1.666. The 10th input argument is a vector of fixed peak widths (in square brackets), one entry for each peak, which may be the same or different for each peak.

```
>> x=[0:.1:10];y=exp(-(x-5).^2)+.5*exp(-(x-3).^2)+.1*randn(size(x));
>> [FitResults,FitError]=peakfit([x' y'],0,0,2,11,0,0,0,0,[1.666 1.666])
         Peak number  Position   Height        Width      Peak area
            1        3.9943      0.49537        1.666      0.87849
            2        5.9924      0.98612        1.666       1.7488
```

Example 14. Peak area measurements. Four Gaussians with a height of 1 and a width of 1.6651. All four peaks have the same theoretical peak area (1.772). The four peaks can be fit together in one fitting operation using a 4-peak Gaussian model, with only rough estimates of the first-guess positions and widths (in square brackets). The peak areas thus measured are much more accurate than the perpendicular drop method (page 142):

```
>> x=[0:.01:18];
>> y=exp(-(x-4).^2)+exp(-(x-9).^2)+exp(-(x-12).^2)+exp(-(x-13.7).^2);
>> peakfit([x;y],0,0,4,1,0,1,[4 2 9 2 12 2 14 2],0,0)
         Peak number  Position   Height        Width      Peak area
            1             4            1       1.6651       1.7725
            2             9            1       1.6651       1.7725 …
            3            12            1       1.6651       1.7724
            4          13.7            1       1.6651       1.7725
```

This works well even in the presence of substantial amounts of random noise:

```
>> x=[0:.01:18]; y=exp(-(x-4).^2)+exp(-(x-9).^2)+exp(-(x-12).^2)+exp(-(x-
13.7).^2)+.1.*randn(size(x));
>> peakfit([x;y],0,0,4,1,0,1,[4 2 9 2 12 2 14 2],0,0)
         Peak number  Position   Height        Width      Peak area
            1        4.0086      0.98555       1.6693       1.7513
            2        9.0223       1.0007        1.669       1.7779
            3        11.997       1.0035       1.6556       1.7685
            4        13.701       1.0002       1.6505       1.7573
```

Sometimes experimental peaks are affected by exponential broadening, which does not by itself change the true peak areas, but does shift peak positions and increase peak width, overlap, and asymmetry, as shown when you try to fit the peaks with Gaussians. Using the same noise signal from above:

```
>> y1=ExpBroaden(y',-50);
>> peakfit([x;y1'],0,0,4,1,50,1,0,0,0)
```

```
  Peak number   Position    Height          Width        Peak area
        1        4.4538      0.83851        1.9744        1.7623
        2        9.4291       0.8511        1.9084        1.7289
        3        12.089      0.59632          1.542       0.97883
        4        13.787       1.0181        2.4016        2.6026
```

Peakfit.m (and ipf.m) have an exponentially-broadened Gaussian peak shape (shape #5) that works better in those cases, recovering the *original* peak positions, heights, widths, and areas. (Adding a first-guess vector as the 8th argument improves the reliability of the fit in some cases).

```
>> y1=ExpBroaden(y',-50);
>> peakfit([x;y1'],0,0,4,5,50,1,[4 2 9 2 12 2 14 2],0,0)
          Peak#    Position       Height        Width          Area
            1             4            1       1.6651       1.7725
            2             9            1       1.6651       1.7725
            3            12            1       1.6651       1.7725
            4          13.7      0.99999       1.6651       1.7716
```

An easy way to obtain a good first-guess vector is to perform a simple Gaussian fit initially and have the script use the FitResults from that fit as elements of the first-guess vector, as in this example.

Example 15. Displays a table of parameter error estimates. See DemoPeakfitBootstrap for a self-contained demo of this function.

```
>> x=0:.05:9; y=exp(-(x-5).^2)+.5*exp(-(x-3).^2)+.01*randn(1,length(x));
>> [FitResults,LowestError,baseline,residuals,xi,yi,BootstrapErrors]=
peakfit([x;y],0,0,2,6,0,1,0,0,0);

Peak #1        Position     Height        Width        Area
Mean:         2.9987       0.49717      1.6657       0.88151
STD:          0.0039508    0.0018756    0.0026267    0.0032657
STD (IQR):    0.0054789    0.0027461    0.0032485    0.0044656
% RSD:        0.13175      0.37726      0.15769      0.37047
% RSD (IQR): 0.13271      0.35234      0.16502      0.35658

Peak #2        Position     Height        Width        Area
Mean:         4.9997       0.99466      1.6657       1.7636
STD:          0.001561     0.0014858    0.00262      0.0025372
STD (IQR):    0.002143     0.0023511    0.00324      0.0035296
% RSD:        0.031241     0.14938      0.15769      0.14387
% STD (IQR): 0.032875     0.13637      0.16502      0.15014
```

Example 16. Fits both peaks of the Humps function with a Gaussian/Lorentzian blend (shape 13) that is 15% Gaussian (Extra=15). The 'Extra' argument sets the percentage of Gaussian shape.

```
>> x=[0:.005:1];y=humps(x);
[FitResults,FitError]= peakfit([x' y'], 0.54,0.93,2,13,15,10,0,0,0)

          Peak#      Position       Height        Width       Area
             1       0.30078        190.41       0.19131       23.064
             2       0.89788        39.552       0.33448       6.1999
 FitError = 0.34502
```

Example 17. Fit a slightly asymmetrical peak with a bifurcated Gaussian (shape 14). The 'Extra' argument (=45) controls the peak asymmetry (50 is symmetrical).

```
>> x=[0:.1:10];y=exp(-(x-4).^2)+.5*exp(-(x-5).^2)+.01*randn(size(x));
>> [FitResults,FitError]=peakfit([x' y'],0,0,1,14,45,10,0,0,0)
          Peak#      Position       Height        Width       Area
             1        4.2028        1.2315         4.077        2.6723
 FitError =0.84461
```

Example 18. Returns output arguments only, without plotting or command window printing (11th input argument = 0, default is 1)

```
>> x=[0:.1:10]';y=exp(-(x-5).^2);
>> FitResults=peakfit([x y],0,0,1,1,0,0,0,0,0,0)
```

Example 19. Same as example 4, but with *fixed-position* Gaussian (shape 16), positions=[3 5].

```
>> x=[0:.1:10];y=exp(-(x-5).^2)+.5*exp(-(x-3).^2)+.1*randn(size(x));
>> [FitResults,FitError]=peakfit([x' y'],0,0,2,16,0,0,0,0,[3 5])
        Peak number   Position    Height         Width       Peak area
            1             3       0.49153        1.6492       0.86285
            2             5        1.0114        1.6589         1.786
FitError =8.2693
```

Example 20. Exponentially modified Lorentzian (shape 18) with added noise. As for peak shape 5, peakfit.m recovers the original peak position (9), height (1), and width (1).

```
>> x=[0:.01:20];
>> L=lorentzian(x,9,1)+0.02.*randn(size(x));
>> L1=ExpBroaden(L',-100);
>> peakfit([x;L1'],0,0,1,18,100,1,0,0)
```

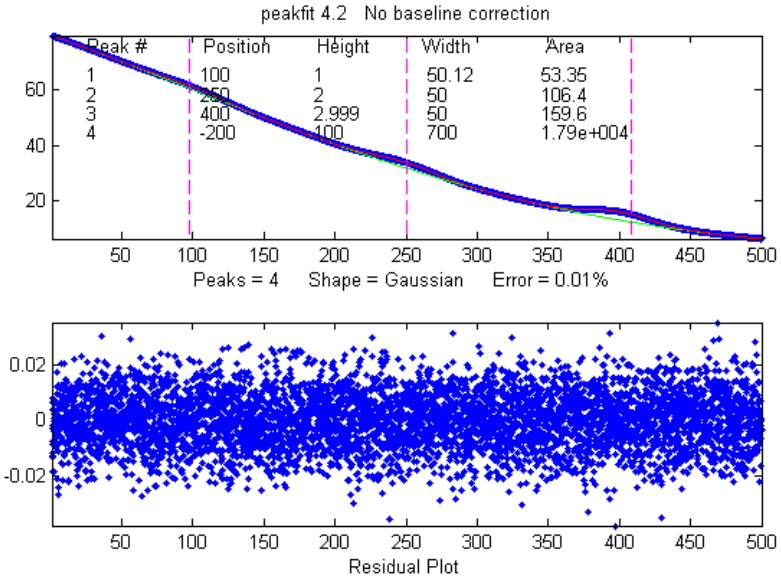
Example 21. Fitting humps function with two unconstrained Voigt profiles (version 9.5)

```
>>disp('Peak      Position        Height      Width      Area     Alpha')
>> [FitResults,FitError]=peakfit(humps(0:.01:2),60,120,2,30,1.7,5,0)

Peak     Position      Height        Width         Area         Alpha
1        31.629        95.175        19.469       2404.5       2.1355
2        90.736        19.826        33.185       764.32       1.3188

GOF =  0.7618    0.9991
```

Example 22. Measurement of three weak Gaussian peaks at x=100, 250, 400, superimposed in a very strong curved baseline plus noise. peakfitdemob.m, illustrated below. The peakfit function fits *four* peaks, treating the baseline as a 4th peak whose peak position is negative. (The true peaks heights are 1, 2, and 3, respectively). Because this results in so many adjustable variables (4 peaks x 2 variable/peak = 8 variables), you need to specify a "start" vector, like Example 7. You can test the reliability of this method by changing the peak parameters in lines 11, 12, and 13 and see if the peakfit function will successfully track the changes and give accurate results for the three peaks without having to change the start vector. See Example 9 on iSignal.html for other ways to handle this signal.



Example 23. 12th input argument (+/- mode) set to 1 (bipolar) to allow negative as well as positive peak heights. (Default is 0)

```
>> x=[0:.1:10];y=exp(-(x-5).^2)-.5*exp(-(x-3).^2)+.1*randn(size(x));
>> peakfit([x' y'],0,0,2,1,0,1,0,0,0,1,1)
      Peak#     Position       Height        Width       Area
          1       3.1636      -0.5433         1.62      -0.9369
          2       4.9487      0.96859       1.8456       1.9029
FitError =8.2757
```

Example 24. Version **5** or later. Fits humps function to a model consisting of one Lorentzian and one Gaussian peak.

```
>> x=[0:.005:1.2];y=humps(x);
>> [FitResults,FitError]=peakfit([x' y'],0,0,2,[2 1],[0 0])
        Peak#     Position       Height        Width       Area
           1      0.30178       97.402      0.18862       25.116
           2      0.89615       18.787      0.33676       6.6213
FitError = 1.0744
```

Example 25. Five peaks, five different shapes, all heights = 1, all widths = 3, "extra" vector included for peaks 4 and 5.

```
>>  x=0:.1:60;
>> y=modelpeaks2(x,[1 2 3 4 5],[1 1 1 1 1],[10 20 30 40 50],[3 3 3 3 3],[0
0 0 2 -20])+.01*randn(size(x));
>> peakfit([x' y'],0,0,5,[1 2 3 4 5],[0 0 0 2 -20])
```

You can also use this technique to create models with all the *same* shapes but with different values of 'extra' using a vector of 'extra' values, or (in version 5.7) with different minimum width restrictions by using a vector of 'minwidth' values as input argument 13.

Example 26. Minimum width constraint (13th input argument)

```
>> x=1:30;y=gaussian(x,15,8)+.05*randn(size(x));
```

No constraint (minwidth=0):

```
peakfit([x;y],0,0,5,1,0,10,0,0,0,1,0,0);
```

Widths constrained to values 7 or above:

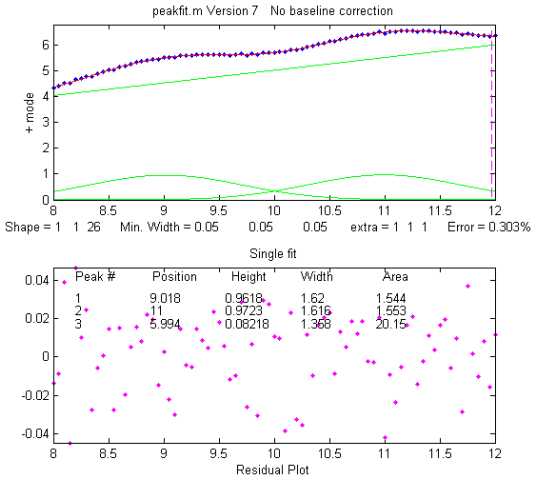
```
peakfit([x;y],0,0,5,1,0,10,0,0,0,1,0,7);
```

Example 27. Noise test with very noisy peak signal: peak height and RMS noise both equal to 1.

```
>> x=[-10:.05:10];y=exp(-(x).^2)+randn(size(x));
>> P=peakfit([x;y],0,0,1,1,0,10);
```

Example 28: Weak Gaussian peak on sloped straight-line baseline, 2-peak fit with one Gaussian and one variable-slope straight line ('slope', shape 26, peakfit version 6 and later only).

```
>> x=8:.05:12; y=x + exp(-(x-10).^2);
[FitResults,FitError]= peakfit([x;y],0,0,2,[1 26],[1 1],1,0)
           Peak#     Position       Height        Width       Area
              1           10            1       1.6651     1.7642
              2        4.485      0.22297         0.05     40.045
  FitError =0.093
```



To make a more difficult example, this one has *two* weak Gaussian peaks on sloped straight-line baseline. In this case you can use a 3-peak model with peakshape=[1 1 26], which is helped by adding rough first guesses ('start') using the 'polyfit' function to generate automatic first guesses for the sloping baseline. The third component (peak 3) is the baseline.

```
x=8:.05:12
y=x/2+exp(-(x-9).^2)+exp(-(x-
11).^2)+.02.*randn(size(x));
start=[8 1 10 1 polyfit(x,y,1)];
peakfit([x;y],0,0,3,[1 1 26],[1 1 1],1,start)
```

See example 38 (page 411) for a similar example with a *curved* baseline.

Example 29: **Segmented linear fit (Shape 29)**. You specify the number of segments in the 4th input argument ('NumPoints') and the program attempts to find the optimum *x-axis positions* of the breakpoints that minimize the fitting error. The vertical dashed magenta lines mark the x-axis breakpoints. Another example with a single Gaussian band .

```
>> x=[0.9:.005:1.7];y=humps(x);
>> peakfit([x' y'],0,0,9,29,0,10,0,0,0,1,1)
```

Example 30: **Polynomial fit (Shape 28)**. Specify the order of the polynomial (any positive integer) in the 6th input argument ('extra'). (The 12th input argument, 'bipolar', is set to 1 to plot the entire y-axis range when y goes negative).

```
>> x=[0.3:.005:1.7];y=humps(x);y=y + cumsum(y);
>> peakfit([x' y'],0,0,1,28,6,10,0,0,0,1,1)
```

Example 31: The Matlab/Octave script NumPeaksTest.m uses peakfit.m to demonstrate one way to determine the minimum number of model peaks needed to fit a set of data, plotting the fitting error vs the number of model peaks, and looking for the point at which the fitting error reaches a minimum. This script creates a noisy computer-generated signal containing a user-selected 3, 4, 5 or 6 underlying peaks, fits to a series of models containing 1 to 10 model peaks, plots the fitting errors vs the number of model peaks and then determines the vertex of the best-fit parabola. The nearest integer is usually the correct number of peaks underlying peaks. Also requires that the plotit.m function be installed.

Example 32: Examples of *unconstrained* variable shapes 30-33 and shape 39, all of which have *three* iterated variables (position, width, and shape):

a. **Voigt** (shape 30). Returns Alphas (ratios of Lorentzian width to Gaussian width) as 6th column.

```
x=1:.1:30; y=modelpeaks2(x,[13 13],[1 1],[10 20],[3 3],[20 80]);
disp('Peak#     Position     Height    Width     Area   Alpha')
[FitResults,FitError] = peakfit([x;y],0,0,2,30,2,10).
```

b. **Exponentially broadened Gaussian** (shape 31):

```
load DataMatrix3;
peakfit(DataMatrix3, 1860.5,364,2,31,3,5,[1810 60 30 1910 60 30])
```

Version **8.4** also includes an alternative exponentially broadened Gaussian, shape 39, which is parameterized differently (see Example 39 on the next page).

c. **Pearson** (shape 32)

```
x=1:.1:30;
y=modelpeaks2(x,[4 4],[1 1],[10 20],[5 5],[1 10]);
[FitResults,FitError] = peakfit([x;y],0,0,2,32,0,5)
```

d. **Gaussian/Lorentzian blend** (shape 33):

```
x=1:.1:30; 0
y=modelpeaks2(x,[13 13],[1 1],[10 20],[3 3],[20 80]);
[FitResults,FitError]=peakfit([x;y],0,0,2,33,0,5)
```

Example 34: Using the built-in "sortrows" function to sort the FitResults table by peak position (column 2) or peak height (column 3).

```
 >> x=[0:.005:1.2]; y=humps(x);
 >> FitResults,FitError]=peakfit([x' y'],0,0,3,1)
 >> sortrows(FitResults,2)
ans =
            2      0.29898       56.463      0.14242       8.5601
            1      0.30935       39.216      0.36407       14.853
            3      0.88381       21.104      0.37227       8.1728
>> sortrows(FitResults,3)
ans =
            3      0.88381       21.104      0.37227       8.1728
            1      0.30935       39.216      0.36407       14.853
            2      0.29898       56.463      0.14242       8.5601
```

Example 35: Version 7.6 or later. Using the fixed-width Gaussian/Lorentzian blend (shape 35).

```
>> x=0:.1:10; y=GL(x,4,3,50)+.5*GL(x,6,3,50) + .1*randn(size(x));
>> [FitResults,FitError]=peakfit([x;y],0,0,2,35,50,1,0,0,[3 3])
    peak      position       height       width        area
     1        3.9527        1.0048           3        3.5138
     2        6.1007        0.5008           3        1.7502
   GoodnessOfFit = 6.4783       0.95141
```

Compared to variable-width fit (shape 13), the fitting error is larger but nevertheless results are more accurate (when true peak width is known, width = [3 3]).

```
>> [FitResults,GoodnessOfFit]= peakfit([x;y],0,0,2,13,50,1)
         1        4.0632        1.0545        3.2182        3.9242
         2        6.2736       0.41234        2.8114        1.3585
  GoodnessOfFit = 6.4311       0.95211
```

Note: to display the FitResults table with column labels, call peakfit.m with output arguments [FitResults...] and type :

```
disp('      Peak number  Position      Height      Width       Peak
area');disp(FitResults)
```

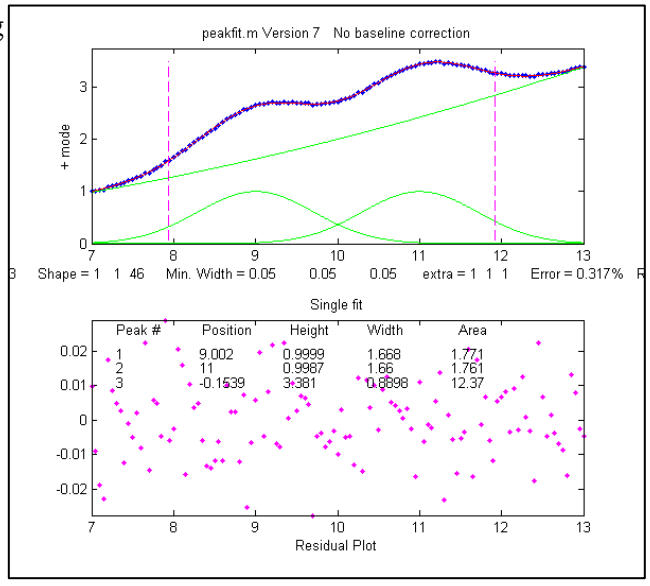
Example 36: Variable exponent broadened Lorentzian function, shape 38. (Version 7.7 and above only). FitResults has an added 6th column for the measured time constant.

```
>> x=[1:100]';
>> y=explorentzian(x',40,5,-10)+.01*randn(size(x));
>> peakfit([x y],0,0,1,38,0,10)
```

Example 37: 3-parameter logistic (Gompertz), shape 43. (Version 7.9 and above only). Parameters labeled Bo, Kh, and L. FitResults extended to 6 columns.

```
>> t=0:.1:10;
>> Bo=6;Kh=3;L=4;
>> y=Bo*exp(-exp((Kh*exp(1)/Bo)*(L-t)+1))+.1.*randn(size(t));
>> [FitResults,GOF]=peakfit([t;y],0,0,1,43)
```

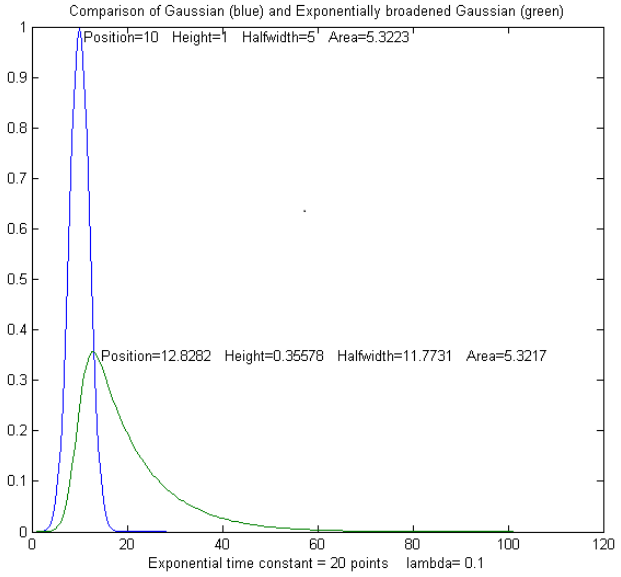
Example 38: Shape 46, 'quadslope'. Two overlapping Gaussians (position=9,11, heights=1, widths=1.666) on a curved baseline, using a 3-peak fit with peakshape=[1 1 46], default NumTrials and start.




```
>> x=7:.05:13;
>> y=x.^2/50+exp(-(x-9).^2)+exp(-(x-
11).^2)+.01.*randn(size(x));
>> [FitResults,FitError]=
peakfit([x;y],0,0,3,[1 1 46],[1 1 1])
```

Screen image on the right. Note: if the baseline is much higher in amplitude than the peak amplitude, it will help to supply an approximate 'start' value and to use NumTrials > 1.

Example 39: Comparison of alternative unconstrained exponentially broadened Gaussians, shapes 31 and 39. Shape 31 (expgaussian.m) creates the shape by performing a Fourier convolution of a specified Gaussian by an exponential decay of specified time constant, whereas shape 39 (expgaussian2.m) uses a mathematical expression for the final shape so produced. Both result in the *same peak shape* but are parameterized differently. Shape 31 reports the peak height and position as that of the *original* Gaussian before broadening, whereas shape 39 reports the peak height of the broadened *result*. Shape 31 reports the width as the FWHM (full width at half maximum) and shape 39 reports the standard deviation (sigma) of the Gaussian. Shape 31 reports the exponential factor and the *number of data points* and shape 39 reports the *reciprocal of time constant* in time units. See the script GaussVsExpGauss.m (on the left). See Matlab Figure windows **2** and **3**. For multiple-peak fits, both shapes usually require a reasonable first guess ("start") vector for best results.



Method	Position	Height	Halfwidth	Area	Exponential factor
Shape 31	10	1	5	5.3223	20.0001
Shape 39	12.8282	0.35578	11.7731	5.3217	0.1

See the script DemoExpgaussian.m for a more detailed explanation.

Example 40: Use of the "start" vector in 4-Gaussian fit to the "humps" function

```
x=[-.1:.005:1.2];y=humps(x);
```

First attempt with default start values gives poor fit that varies from trial to trial:

```
[FitResults,GOF]=peakfit([x;y],0,0,4,1,0,10)
```

Second attempt specifying approximate "start" values in the 8th input argument gives much better fit:

```
start=[0.3 0.13 0.3 0.34 0.63 0.15 0.89 0.35];
[FitResults,GOF]=peakfit([x;y],0,0,4,1,0,10,start)
```

Example 41: Peakfit 9 and above. Use of peak shape 50 ("multilinear regression") when the peak *positions and widths* are known, and only the peak *heights* are unknown. The peak shapes, positions, and widths are specified in the 10th input argument "fixedparameters", which must in this case be a *matrix* listing the peak shape number (column 1), position (column 2), and width (column 3) of each peak, one row per peak. See the demonstration scripts peakfit9demo.m and peakfit9demoL.m.

Example 42: RandPeaks.m is a script that demonstrates the accuracy of iterative peak fitting when no customized "start" values are provided, that is, knowing only the peak shape and number of peaks. It generates any number of overlapping Gaussian peaks (NumPeaks in line 9) of random position, height, and width and calls the peakfit function. Calculates the average percent errors in position, height, and width. As you increase the number of peaks, accuracy degrades, even if R^2 remains close to 1.00.

How do you find the correct input arguments for peakfit?

If you have no idea where to start, you can use the Interactive Peak Fitter (ipf.m) to quickly try different fitting regions, peak shapes, numbers of peaks, baseline correction modes, number of trials, etc. When you get a good fit, you can press the "**W**" key to print out the command line statement for peakfit.m that will perform that fit in a single line of code, with or without graphics.

Working with the fitting results matrix "FitResults".

Suppose you have performed a multi-peak curve fit to a set of data, but you are interested only in one or a few specific peaks. It is not always reliable to simply go by peak index number (the first column in the FitResults table). Peaks sometimes change their position in the FitResults table arbitrarily, because the *fitting error is independent of the peak order* (the sum of peaks 1+2+3 is exactly the same as 2+1+3 or 3+2+1, etc.). But you can sort this out by using the Matlab/Octave "sortrows" command to reorder the table in order of peak position or height. Also useful in such cases is my function val2ind(v, val), which returns the index and the value of the element of vector 'v' that is closest to 'val' (download this function and place in the Matlab search path). For example, suppose you want to extract the peak height (column **3** of FitResults) of the peak whose position (column **2** of FitResults) is closest to a particular value, call it "TargetPosition". There are three steps:

```
VectorOfPositions=FitResults(:,2);
IndexOfClosestPeak=val2ind(VectorOfPositions, TargetPosition);
HeightOfClosestPeak=Fitresults(IndexOfClosestPeak,3);
```

For an example of this use in a practical application, see RandomWalkBaseline.m.

Another way to use the FitResults matrix is to compute start values for further fits.

Demonstration script for peakfit.m

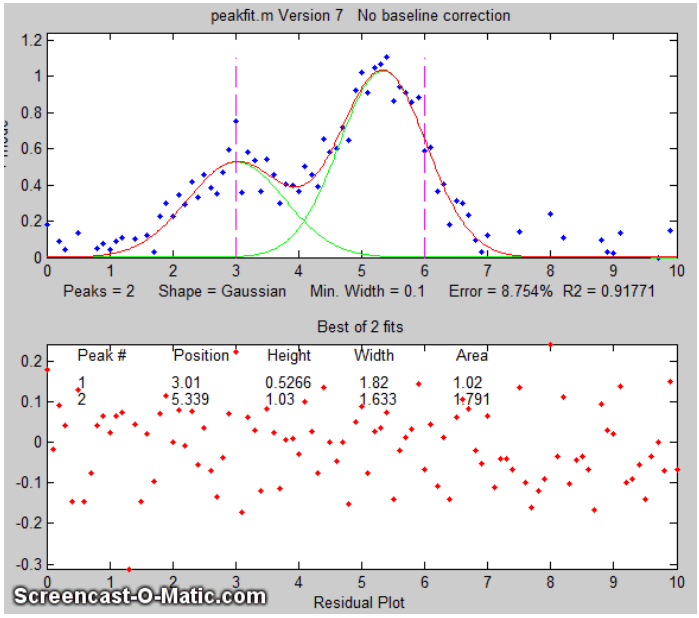
DemoPeakFit.m is a demonstration script for peakfit.m. It generates an overlapping Gaussian peak signal, adds normally-distributed white noise, fits it with the peakfit.m function (in line 78), repeats this many times ("NumRepeats" in line 20), then compares the peak parameters (position, height, width, and area) of the measurements to their actual values and computes accuracy (percent error) and precision (percent relative standard deviation). You can change any of the initial values in lines 13-30. Here is a typical result for a two-peak signal with Gaussian peaks:

Percent errors of measured parameters:

```
Position       Height        Width        Area
 0.048404      0.07906      0.12684      0.19799
 0.023986      0.38235     -0.36158     -0.067655

|Average Percent Parameter Error for all peaks:
 0.036195      0.2307       0.24421      0.13282
```

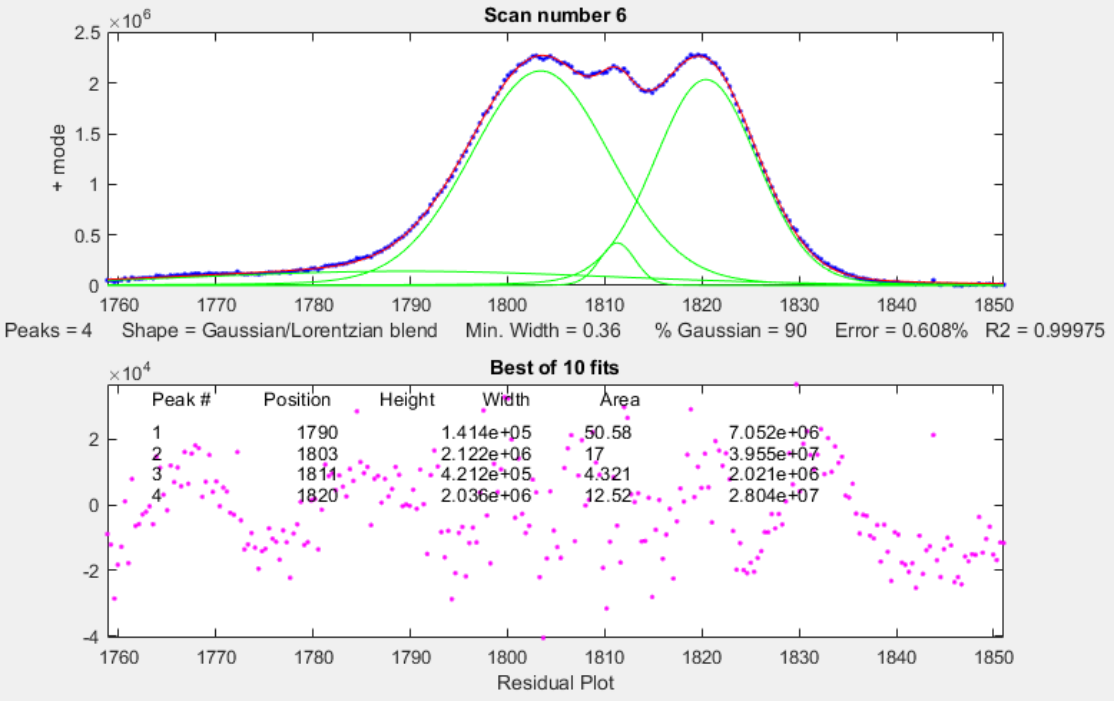
In these results, you can see that the accuracy and precision (%RSD) of the peak *position* measurements are always the best, followed by peak *height*, and then the peak *width* and peak *area*, which are usually the worst.



DemoPeakFitTime.m is a simulation that demonstrates how to apply multiple curve fits to a signal that is changing with time. The signal contains two noisy Gaussian peaks (like the illustration on the right) in which the peak position of the *second* peak increases with time and the other parameters remain constant (except for the noise). The script creates a set of 100 noisy signals (on line 5) containing two Gaussian peaks where the position of the second peak changes with time (from x=6 to 8) and the first peak remains the same. Then it fits a 2-Gaussian model to each of those signals (on line 8), stores the FitResults in a 100 × 2 × 5 matrix, displays the signals and the fits graphically with time (click to play animation), then plots the measured peak position of the two peaks vs time on line 12. Here is a real-data example with exponential pulse that varies over time. For an example of automating the processing of multiple stored data files, see page 345.

Fitting peaks in multi-column data

The script FittingPeaksInMultiColumnData.m shows how to read a multi-column data set from an Excel spreadsheet and to use peakfit.m to fit a selected portion of the signal in each column with a model consisting of 4 overlapping peaks. The data, a group of subsets extracted from the NIST InfraRed spectrum of benzene. The results for each subset are printed out and stored in the 4x5x6 matrix PP.



Dealing with complex signals with lots of peaks

When a signal consists of lots of peaks on a highly variable background, the best approach is often to use peakfit is "center" and "window" arguments to break up the signal into segments containing smaller groups of overlapping peaks with their segments of background, isolating the peaks that do not overlap with other peaks. The reasons for this are several:

(a) peakfit.m works better if the number of variables for each fit is reduced;
(b) it is easier to compensate for the local background over those smaller segments;
(c) with smaller fits, you may not need to supply starting guesses for the peak position and widths;
(d) you can easily skip over peaks or data regions that you are not interested in;
(e) It is actually faster for the computer to execute a series of smaller peakfit() commands than a single one encompassing the entire data range in one go.

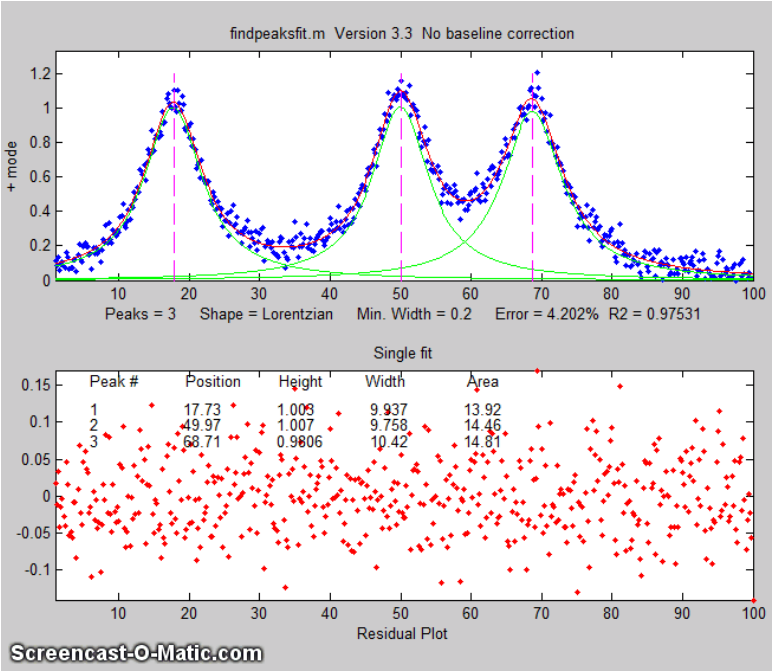
An easy way to do this is to use my interactive peak fitter **ipf.m** (page 424) to explore various segments of the signal by panning and zooming and to try some trial fits and baseline correction settings, then press the "**w**" key to print out the peakfit syntax for that segment, with all its input arguments. Copy, paste, and edit the syntax for each segment as desired, then paste them into you code:

```
[FitResults1, GOF1] = peakfit(datamatrix, center1, window1...
[FitResults2, GOF2] = peakfit(datamatrix, center2, window2...
```

Assign your data matrix to “datamatrix”. Each line will use the same data but with different “center” and “window” values. The other input arguments (peak shape, number of peaks, “extra”, number of trials, start values, baseline correction, etc.) may also be different if you changed then in ipf.m.

Automatically Finding and Fitting Peaks

findpeaksfit.m is essentially a combination of findpeaks.m and peakfit.m. It uses the number of peaks found and the peak positions and widths determined by findpeaks as input for the peakfit.m function, which then fits the entire signal with the specified peak model. This combination function is more convenient than using findpeaks and peakfit separately. It yields better values than findpeaks, because peakfit fits the entire peak, not just the top part, and because it deals with non-Gaussian and overlapped peaks. However, it fits only those peaks that are found by findpeaks, so you will have to make sure that every peak that contributes to your signal is located by findpeaks. The syntax is:

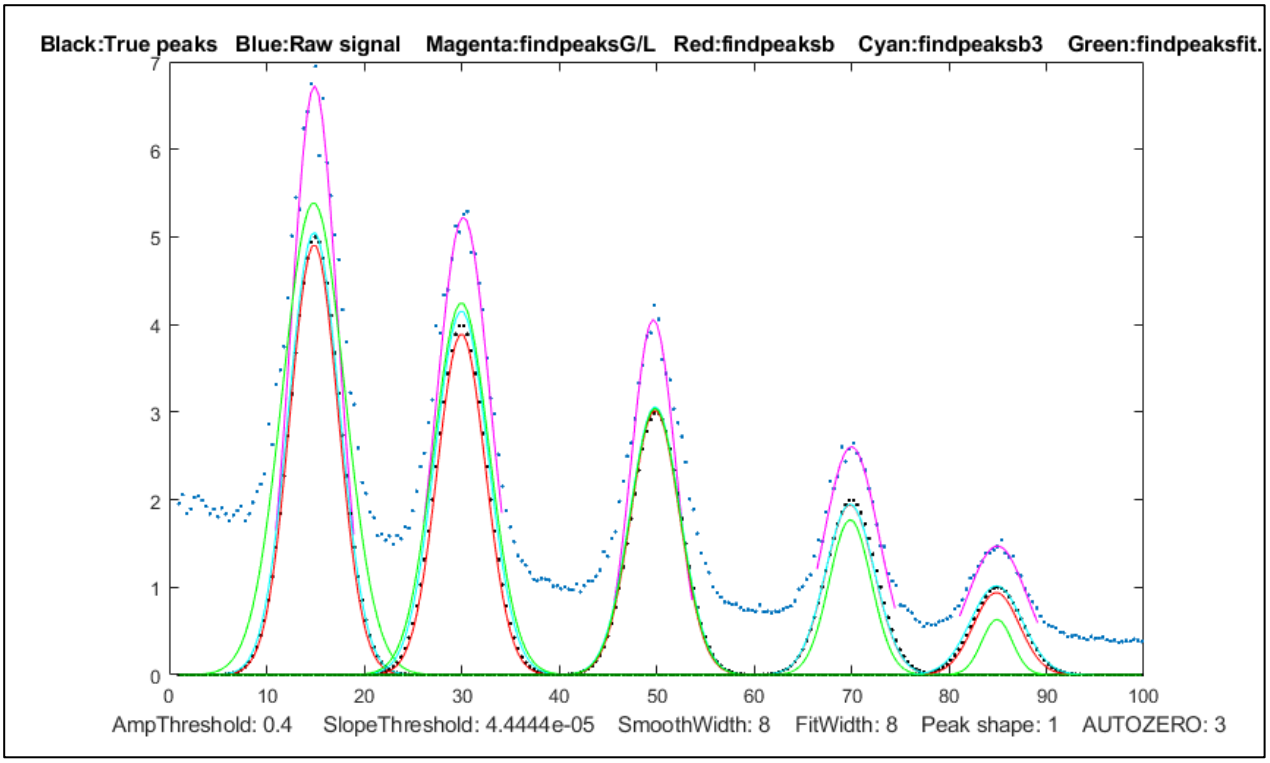



```
function [P,FitResults,LowestError,residuals,xi,yi] = findpeaksfit(x, y,
SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, smoothtype,
peakshape, extra, NumTrials, BaselineMode, fixedparameters, plots)
```

The first seven input arguments are exactly the same as for the findpeaks.... functions (page 236). If you have been using findpeaks or *iPeak* to find and measure peaks in your signals, you can use those same input argument values for findpeaksfit.m. The remaining six input arguments of findpeaksfit.m are for the peakfit function (page 397). If you have been using peakfit.m or ipf.m to fit peaks in you signals, you can use those same input argument values for findpeaksfit.m. Type "help findpeaksfit" for more information. This function is included in the ipfl3.zip distribution.

The animation on the right was created by the demo script findpeaksfitdemo.m. It shows findpeaksfit finding and fitting the peaks in 150 signals in real-time. Each signal has from 1 to 3 noisy Lorentzian peaks in variable locations.

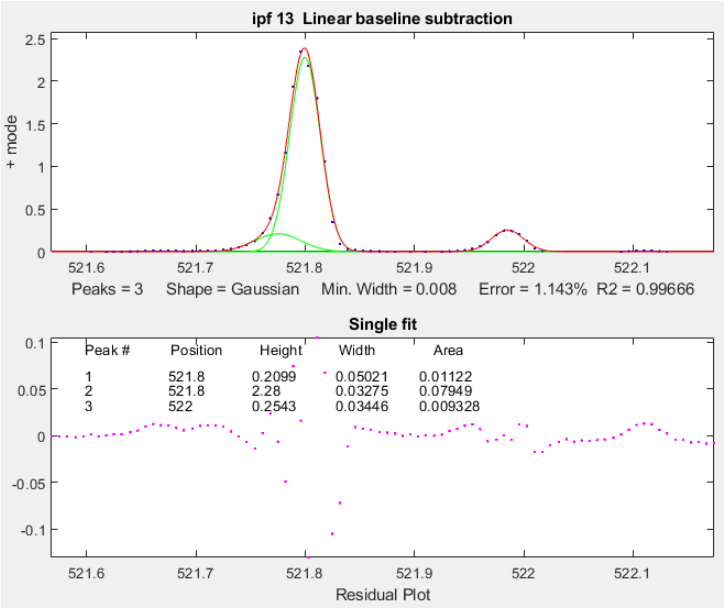
The script FindpeaksComparison.m compares the peak parameter accuracy of four peak detection functions: findpeaksG/L, findpeaksb, findpeaksb3, and findpeaksfit applied to a computer-generated signal with multiple peaks plus variable types and amounts of baseline and random noise. The last three of these functions include iterative peak fitting equivalent to peakfit.m, in which the number of peaks and the "first guess" starting values are determined by findpeaksG/L. Typical result shown below.



Average absolute percent errors of all peaks

	Position error	Height error	Width error	Elapsed time, sec
findpeaksG	0.365331	35.5778	11.6649	0.005768
findpeaksb	0.28246	2.7755	3.4747	0.069061
findpeaksb3	0.28693	2.2531	2.9951	0.49538
findpeaksfit	0.341892	12.7095	18.3436	0.273

Keypress operated Interactive Peak Fitter (ipf.m)



ipf.m for Matlab (Version 13.3, September 2019), or ipfoctave.m for Octave, is a peak fitter for x,y data that uses keyboard commands and the mouse cursor. It is a self-contained function, in a single m-file. (The interactive keypress operation also works if you run Matlab Online in a web browser, but it does not work in Matlab Mobile on iPads and iPhones). The flexible input syntax allows you to specify the data as separate x,y vectors or as a 2 x n matrix, and to optionally define the initial focus by adding "center" and "window" values as additional input arguments, where 'center' is the desired x-value in the center of the upper window and "window" is the desired width of that window. Double-click the figure window title bar to expand to full screen for a better view. Examples:

1 input argument:

```
ipf(y) or ipf([x;y]) or ipf([x;y]');
```

2 input arguments:

```
ipf(x,y) or ipf([x;y],center) or ipf([x;y]',center);
```

3 input arguments:

```
ipf(x,y,center) or ipf(y,center,window) or
ipf([x;y],center,window) or ipf([x;y]',center,window);
```

4 input arguments:

```
ipf(x,y,center,window);
```

Like *iPeak* and iSignal, ipf.m starts out by showing the entire signal in the lower panel and the selected region that will be fitted in the upper panel (adjusted by the same cursor controls keys as *iPeak* and iSignal). After performing a fit, the upper panel shows the data as **blue dots**, the total model as a **red line** (ideally overlapping the blue dots), and the model components as **green lines**. The dotted **magenta lines** are the first-guess peak positions for the last fit. The lower panel shows the residuals (difference between the data and the model). **Important note**: Make sure you do not click on the "Show Plot Tools" button in the toolbar above the figure. That will disable normal program functioning. If you do, close the Matlab Figure window and start again. Animated instructions on its use are available online at https://terpconnect.umd.edu/~toh/spectrum/ifpinstructions.html.

Version history. Version 13.3: Adds baseline corrections modes 4 and 5. Version 13.2 displays "Working..." while the fit is in progress; modified "d" key to save model data to disc as SavedModel.mat. Version 13 added new peak shapes, a total of 24 shapes is now selectable by single keystroke and 49 total selectable from the "-" menu. Version 12.4: Changed IQR in the bootstrap method to IQR/1.34896, which estimates the RSD without outliers for a normal distribution. Version 11.1 adds minimum peak width constraint (**Shift-W**); adds saturation maximum (**Shift-M**) to ignore points above saturation maximum (useful for fitting peaks whose peaks are flat because they have reached saturation). Version 11 adds polynomial fitting (**Shift-o** key). Version 10.7 corrects bugs in equal-width Lorentzians (shape 7) and in the bipolar (+ and -) mode. Version 10.5, adds **Shift-Ctrl-S** and **Shift-Ctrl-P** keys to transfer the current signal to iSignal and *iPeak*, respectively, if those functions are installed in your Matlab search path; Version 10.4. Moves fitting result text to bottom panel of graph. Log mode: (**M** key) toggles log mode on/off, fits log(model) to log(y). Replaces bifurcated Lorentzian with the Breit-Wigner-Fano resonance peak (**Shift-B** key); see http://en.wikipedia.org/wiki/Fano_resonance. **Ctrl-A** selects all; **Shift-N** negates signal. Version 10 adds *multiple-shape models*; Version 9.9 adds '+=' key to flip between + (positive peaks only) and bipolar (+/- peaks) modes; Version 9.8 adds **Shift-C** to specify custom first guess ('start'). Version 9.7 adds Voigt profile shape. Version 9.6 added an additional BaselineMode that subtracts a flat baseline without requiring that the signal return to the baseline at both ends of the signal segment. Version 9.5 added exponentially broadened Lorentzian (peak shape 18) and alpha function (peak shape 19);Version 9.4: added bug fix for height of Gaussian/ Lorentzian blend shape; Version 9.3 added **Shift-S** to save the Matlab Figure window as a PNG graphic to the current directory. Version 9.2: bug fixes; Version 9.1 added fixed-position Gaussians (shape 16) and fixed-position Lorentzians (shape 17) and a peak shape selection menu (activated by the '_-' key).

Demoipf.m is a demonstration script for ipf.m, with a built-in simulated signal generator. To download these m-files, right-click on these links, select **Save Link As...**, and click **Save**. You can also download a ZIP file containing ipf.m, Demoipf.m, and peakfit.m.

Example 1: Test with pure Gaussian function, default settings of all input arguments.

```
>> x=[0:.1:10];y=exp(-(x-5).^2);ipf(x,y)
```

In this example, the fit is essentially perfect, no matter what are the pan and zoom settings or the initial first-guess (start) values. (However, the peak area, the last fit result reported, includes *only the area within the upper window,* so it does vary). If there were noise in the data or if the model were imperfect, then *all* the fit results will depend on the exact the pan and zoom settings.

Example 2: Test with "center" and "window" specified.

```
>> x=[0:.005:1];y=humps(x).^3;
```

`>> ipf(x,y,0.335,0.39)` focuses on first peak

`>> ipf(x,y,0.91,0.18)` focuses on second peak

Example 3: Isolates a narrow segment toward the end of the signal.

```
>> x=1:.1:1000;y=sin(x).^2;ipf(x,y,843.45,5)
```

Example 4: Very noisy peak (SNR=1).

```
x=[0:.01:10];y=exp(-(x-5).^2)+randn(size(x));ipf(x,y,5,10)
```

Press the **F** key to fit a Gaussian model to the data.

Press the **N** key several times to see how much uncertainty in peak parameters is caused by the noise.

Example 5: 1-4 Gaussian peaks, no noise, zero baseline, all peak parameters are integer numbers. Illustrates the use of the **X** key (best of 10 fits) as the number of peaks increases.

```
Height=[1 2 2 3 3 3 4 4 4 4];
Position=[10 30 35 50 55 60 80 85 90 95]; Width=[2 3 3 4 4 4 5 5 5 5];
x=[0:.01:100];y=modelpeaks(x,10,1,Height,Position,Width,0);
ipf(x,y);
```

ipf keyboard controls (Version 13.4): Obtained by pressing the **K** key

```
Pan signal left and right...Coarse: < and >
                            Fine: left and right cursor arrow
                            Nudge: [ ]
Zoom in and out.............Coarse zoom: ?/ and "'
                            Fine zoom: up and down arrow keys
Select entire signal........Crtl-A (Zoom all the way out)
Resets pan and zoom.........ESC
Select # of peaks...........Number keys 1-9, or press 0 key to
                            enter number manually
Peak shape from menu........- (minus or hyphen), then type
                              number or shape vector and Enter
Select peak shape by key....g unconstrained Gaussian
                            h equal-width Gaussians
                            Shift-G fixed-width Gaussians
                            Shift-P fixed-position Gaussians
                            Shift-H bifurcated Gaussians
                               (equal shape, a,z adjust)
```

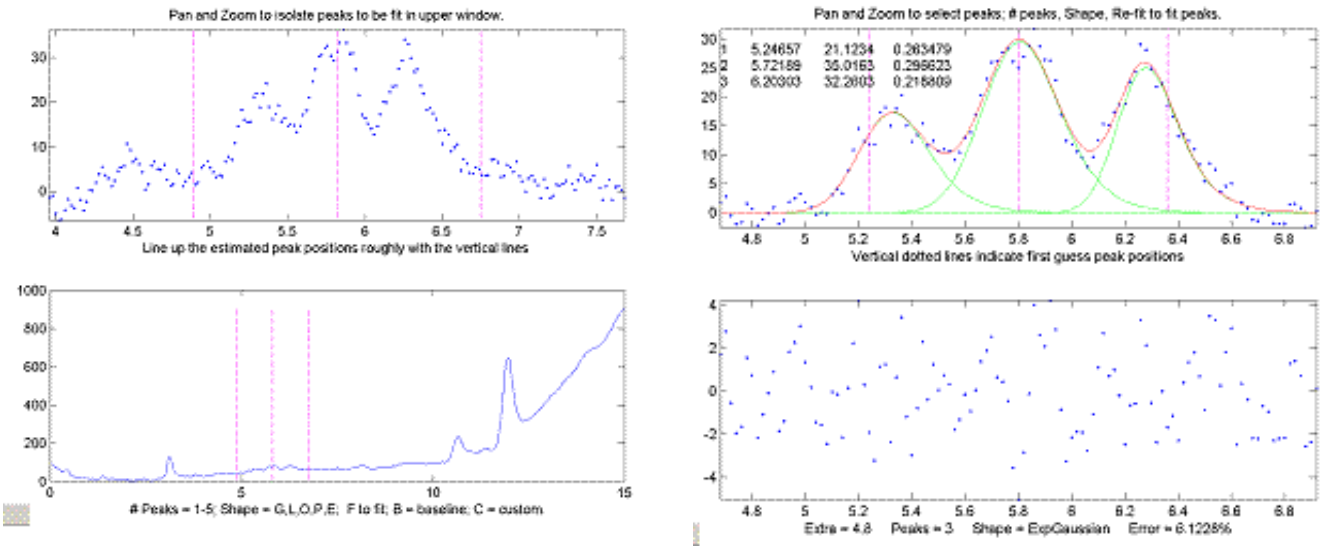
```
                                e Exponential-broadened Gaussian
                                 (equal shape, a,z adjust)
                                Shift-R  ExpGaussian (var. tau)
                                j exponential-broadened equal-width Gaussians
                                  (equal shape, a,z adjust)
                                l unconstrained Lorentzian
                                :; equal-width Lorentzians
                                Shift-[ fixed-position Lorentzians
                                Shift-E Exponential-broadened Lorentzians
                                    (equal shape, a,z adjust)
                                Shift-L Fixed-width Lorentzians (a,z adjust)
                                o LOgistic distribution (Use
                                  Sigmoid for logistic function)
                                p Pearson (a,z keys adjust shape)
                                Shift-L Pearson IV variable asymmetry
                                u exponential pUlse
                                  y=exp(-tau1.*x).*(1-exp(-tau2.*x))
                                Shift-U Alpha: y=(x-tau2)./
                                       tau1.*exp(1-(x-tau2)./tau1)
                                s Up Sigmoid (logistic function):
                                  y=.5+.5*erf((x-tau1)/sqrt(2*tau2))
                                Shift-D Down Sigmoid
                                 y=.5-.5*erf((x-tau1)/sqrt(2*tau2))
                                ~` Gauss/Lorentz blend (equal shape,
                                Shift-V Voigt profile (a/z adjusts
                                        a,z adjust shape)
                                Shift-B Breit-Wigner-Fano (equal
                                       shape a,z adjust)
Fit.........................f
Select BaselineMode ........t selects none, tilted, quadratic,
                              flat, tilted mode(y), flat mode(y)
+ or +/- peak mode..........+=  Flips between + peaks only and
                                +/- peak mode
Invert (negate) signal......Shift-N
Toggle log y mode OFF/ON....m  Log mode plots and fits
                               log(model) to log(y).
2-point Baseline............b, then click left and right baseline
Set manual baseline.........Backspace, then click baseline at
                                      multiple points
Restore original baseline...|\  to cancel previous background subtraction
Cursor start positions......c, then click on each peak position
Type in start vector........Shift-C Type or Paste start vector
                                    [p1 w1 p2 w2 ...]
Print current start vector..Shift-Q
Enter value of 'Extra'......Shift-x, type value (or vector of values
                                     in brackets), press Enter.
Adjust 'Extra' up/down......a,z: 5% change; upper case A,Z:0.5% change
Print parameters & results..q
Print fit results only......r
Compute bootstrap stats.....v  Estimates standard deViations of parameters
Fit single bootstrap........n  Extracts and Fits siNgle
                               bootstrap sub-sample.
Plot signal in figure 2.....y
save model to Disk..........d  Save model to Disk as SavedModel.mat.
```

```
Refine fit..................x Takes best of 10 trial fits
                              (change number in line 227 of ipf.m)
Print peakfit function......w  Print peakfit function with all
                               input arguments
Save Figure as png file.....Shift-S  Saves Figure window as Figure1.png,
                                     Figure2.png, etc.
Display current settings....Shift-?  displays list of current settings
Fit polynomial to segment...Shift-o  Asks for polynomial order
Enter minimum width.........Shift-W  Constrains peak widths to
                                     a specified minimum.
Enter saturation maximum....Shift-M  Ignores points above a
                                     specified saturation maximum.
Switch to iPeak.............Shift-Ctrl-P transfers current
                                         signal to iPeak.m
Switch to iSignal...........Shift-Ctrl-S transfers current
                                         signal to iSignal.m
```

(The function ShapeDemo demonstrates the basic peak shapes graphically, showing the variable-shape peaks as multiple lines; graphic on page 424)

Practical examples with real experimental data:

1. Fitting weak and noisy chromatographic peaks with exponentially modified Gaussians.

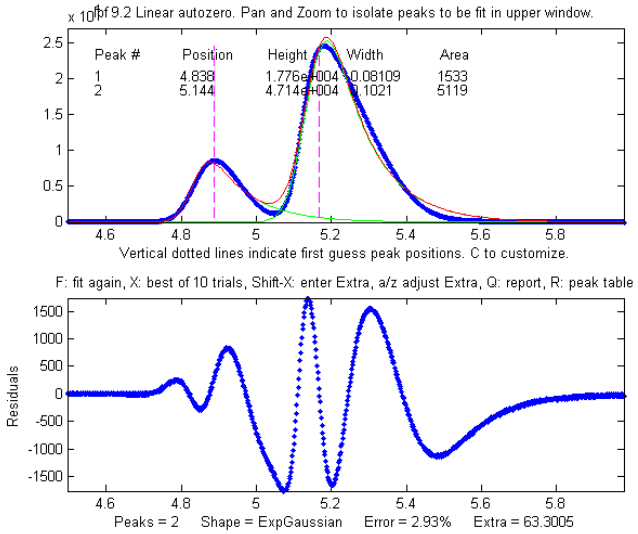


a. *In this real-data example, pan and zoom controls are used to isolate a segment of a chromatogram from 4 to 7.5 minutes that contains three very weak peaks near 5.8 minutes, on an offset and slightly tilted baseline. The lower plot shows the whole chromatogram, and the upper plot shows the selected segment. Only the peaks in that segment are subject to the fitting operation. Pan and zoom are adjusted so that the signal returns to the local baseline at the ends of the segment.*

b. *Pressing **T** selects Baseline Mode 1, causing the program to subtract a linear baseline interpolated from these data points. Pressing **3**, **E** selects a 3-peak exponentially broadened Gaussian model (a common peak shape in chromatography). Pressing **F** initiates the fit. The **A** and **Z** keys are then used to adjust the time constant ("Extra") to obtain the best fit. The residuals (bottom panel) are random and exhibit no obvious structure, indicating that the fit is as good as is*

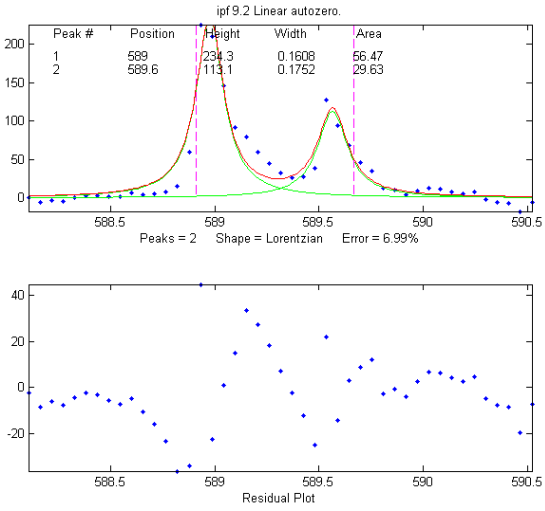
possible at this noise level. A bootstrap error analysis (page 166) indicates that the relative standard deviation of the measured peak heights is predicted to be less than 3%.

2. Measuring the areas of peaks. In the next example, a sample of room air is analyzed by gas chromatography (data source). The resulting chromatogram shows two slightly overlapping asymmetrical peaks, the first for oxygen and the second for nitrogen. The area under each peak is presumed to be proportional to the gas composition. The perpendicular drop method (page 142) of measuring the areas gives peak areas in a ratio of 25% and 75%, compared to the actual 21% and 78% composition, which is not very accurate, possibly because the peaks are so asymmetric. However, an exponentially broadened Gaussian (a commonly encountered peak shape in chromatography) gives a good fit to the data, using ipf.m and adjusting the exponential term with the A and Z keys to get the best fit. The results for a two-peak fit, shown in the ipf.m screen on the right and in the R-key report below, show that the peak areas are in a ratio of 23% and 77%, which is a bit better. With a *3-peak fit,* modeling the nitrogen peak as the sum of *two* closely overlapping peaks whose areas are added together, the curve fit is much better (less than 1% fitting error), and indeed the result in that case is 21.1% and 78.9% - which is considerably closer the actual composition.




```
Percent Fitting Error =2.9318%  Elapsed time = 11 sec.
   Peak#  Position   Height    Width     Area
    1     4.8385     17762    0.081094  1533.2
    2     5.1439     47142    0.10205   5119.2
```

3. The accuracy of peak position measurement can be good even if the fitting error is rather poor. In this example, an experimental high-resolution atomic emission spectrum is examined in the region of the well-known spectral lines of the element sodium. Two lines are found there (figure on the right), and when an unconstrained Lorentzian *or* Gaussian model is fit to the data, the peak wavelengths are determined to be 588.98 nm and 589.57 nm:




```
Percent Fitting Error 6.9922%
Peak#   Position   Height    Width     Area
 1       588.98    234.34   0.16079   56.47
 2       589.57    113.18   0.17509   29.63
```

Compare this to the ASTM recommended wavelengths for this element (588.995 and 589.59 nm) and you can see that the error is no greater than 0.02 nm, which is considerably *less than the interval*

between the data points (0.05 nm). And this is despite the fact that the fit is not particularly good, because the peaks shapes are rather oddly shaped (possibly because of self-absorption, because these particular atomic lines are strongly *absorbing* as well as strongly emitting). This high degree of absolute accuracy compared to a reliable exterior standard is a testament to the excellent wavelength calibration of the instrument on which these experimental data were obtained, but it also shows that peak position is by far the most precisely measurable parameter in peak fitting, even when the data are noisy and the curve fit is not particularly good. The bootstrap standard deviation estimates (page 170) calculated by ipf.m for both wavelengths is 0.015 nm (see #17 in the next section), and using the 2 x standard deviation rule-of-thumb would have predicted a probable error within 0.03 nm. (An even lower *fitting error* can be achieved by fitting to 4 peaks, but the *position accuracy* of the larger peaks remains virtually unchanged).

4. How many peaks to model? In the second and third examples above, the number of peaks in the model was suggested by the data and by the expectations of each of those experiments (There are two major gases in air. Sodium has a prominent and well-known doublet at that wavelength). In the first example, no *a priori* expectation of the number of peaks was available, but the data suggested three obvious peaks, and the residuals were more random and unstructured with a 3-peak model, suggesting that no additional model peaks were needed. In many cases, however, the number of model peaks is not so clearly indicated. In a previously described example on page 214, the fitting error keeps getting lower as more peaks are added to the model, yet the residuals remain "wavy" and never become random. Without further knowledge of the experiment, it is impossible to know which the “real peaks” are and what is just "fitting the noise". But in some cases, the data may suggest something other than your pre-conceived notions. Sometimes data is nature’s way of tapping you on the shoulder.

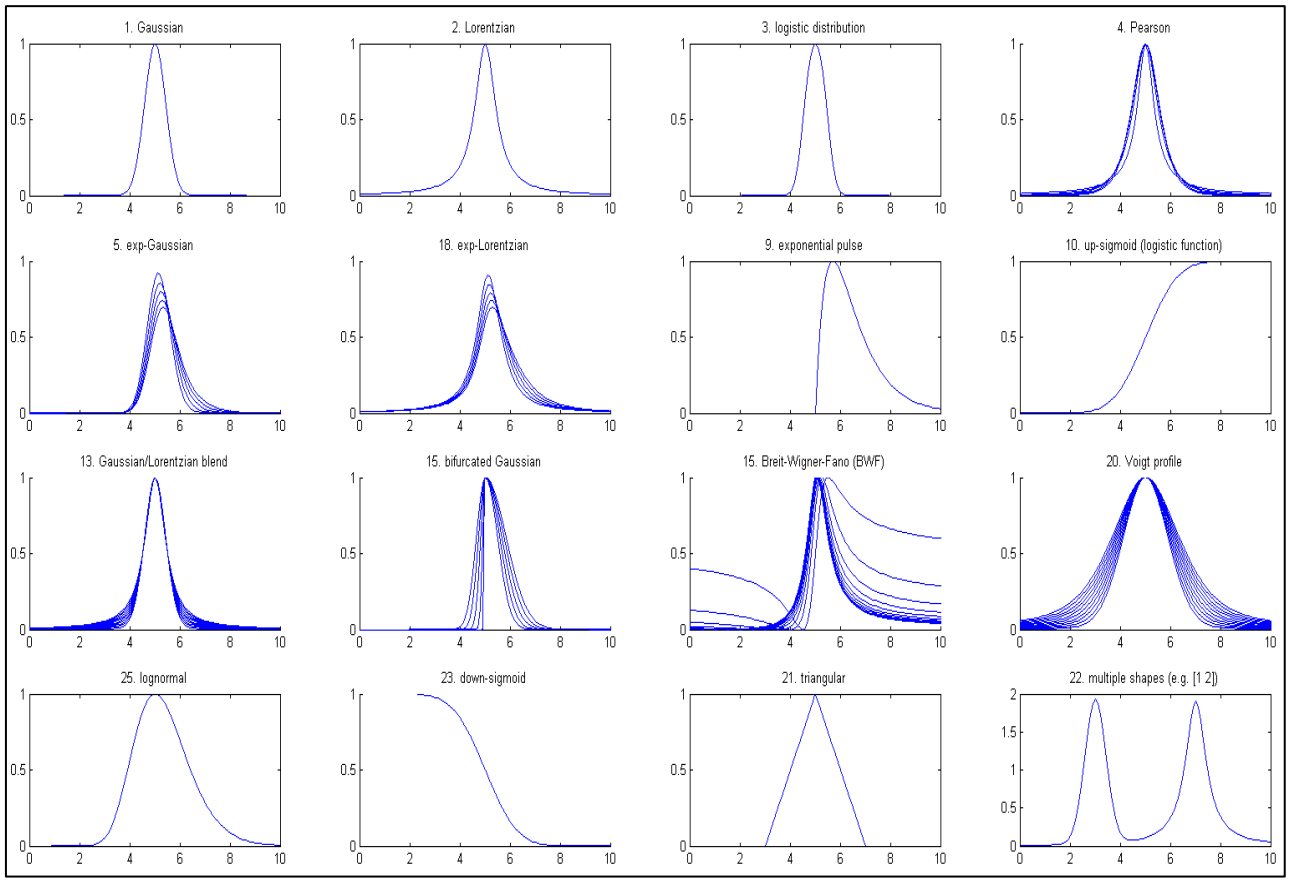
Operating instructions for ipf.m (version 13.4).

[Animated instructions available at https://terpconnect.umd.edu/~toh/spectrum/ifpinstructions.html](https://terpconnect.umd.edu/~toh/spectrum/ifpinstructions.html)

1. At the command line, type **`ipf(x,y)`**, (x = independent variable, y = dependent variable) or **`ipf(datamatrix)`** where "datamatrix" is a matrix that has x values in row or column 1 and y values in row or column 2. Or if you have only one signal vector y, type **`ipf(y)`**. You may optionally add to additional numerical arguments: **`ipf(x,y,center,window);`** where 'center' is the desired x-value in the center of the upper window and "window" is the width of that window.

2. Use the four **cursor arrow keys** on the keyboard to pan and zoom the signal to isolate the peak or group of peaks that you want to fit in the upper window. (Use the **<** and **>** and **?** and " keys for coarse pan and zoom and the square bracket keys **[** and **]** to nudge one point left and right). *The curve fitting operation applies only to the segment of the signal shown in the top plot*. The bottom plot shows the entire signal. Try not to get any undesired peaks in the upper window or the program may try to fit them. To select the *entire* signal, press **Ctrl-A**.

3. Press the number keys (**1**– **9**) to choose the number of model peaks, that is, the minimum number of



peaks that you think will suffice to fit this segment of the signal. For more than 9 peaks, press **0**, type the number, and press **Enter**.

4. Select the desired model **peak shape**. In ipf.m version 13, there are 24 different peak shapes available by keystroke, e.g., **G**=Gaussian, **L**=Lorentzian, **U**=exponential pulse, **S**=sigmoid (logistic

function), etc. Press **K** to see a list of all commands. You can also select the shape by number from an even larger menu of 49 shapes by pressing the - (minus) key and selecting the shape by number. If the peak widths of each group of peaks is expected to be the same or nearly so, select the "equal-width" shapes. If the peak widths or peak positions are known from previous experiments, select the "fixed-width" or "fixed position" shapes. These more constrained fits are faster, easier, and much more stable than regular all-variable fits, especially if the number of model peaks is greater than 3 (because there are fewer variable parameters for the program to adjust - rather than an independent value for each peak).

5. A set of vertical dashed lines are shown on the plot, one for each model peak. Try to fine-tune the **Pan** and **Zoom** keys so that the signal goes to the baseline at both ends of the upper plot and so that the peaks (or bumps) in the signal *roughly* line up with the vertical dashed lines. This does not have to be exact.

6. If you want to allow negative peaks as well as positive peaks, press the + key to flip to the +/- mode (indicated by the +/- sign in the y-axis label of the upper panel. Press it again to return to the + mode (positive peaks only). You can switch between these modes at any time. To negate the entire signal, press **Shift-N**.

7. Press **F** to initiate the curve-fitting calculation. In version 13.2, the center of the graph displays "Working..." while the fit is in progress. Each time you press **F**, another fit of the selected model to the data is performed with slightly different starting positions, so that you can judge the stability of the fit with respect to changes in starting first guesses. Keep your eye on the residuals plot and on the "Error %" display. Do this several times, trying for the lowest error and the most unstructured random residuals plot. At any time, you can adjust the signal region to be fit (step 2), the baseline position (step 9 and 10), change the number of peaks (step 3), or peak shape (step 4), then press the **F** key again to compute another fit. If the fit seems unstable, try pressing the **X** key a few times (see #14, below).

8. The model parameters of the last fit are displayed in the upper window. For example, for a 3-peak fit:

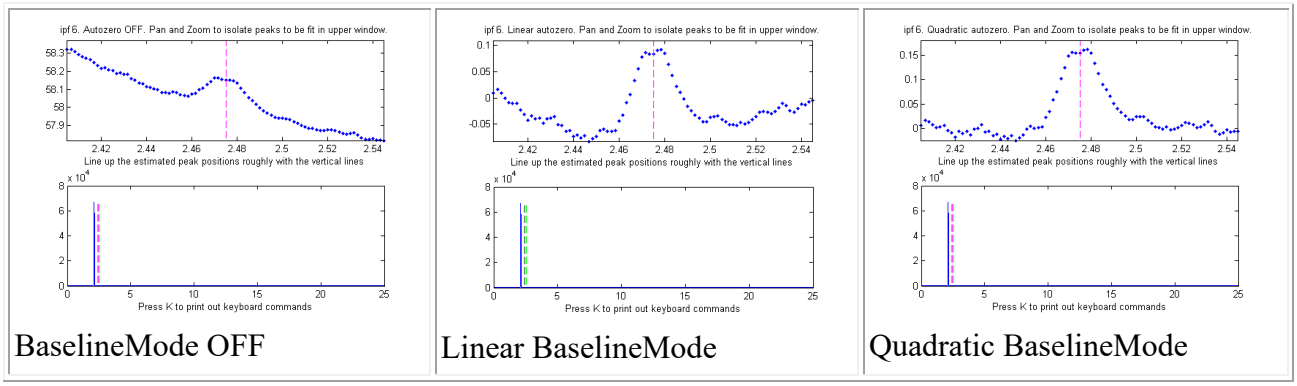
```
Peak# Position   Height    Width     Area
 1    5.33329    14.8274   0.262253  4.13361
 2    5.80253    26.825    0.326065  9.31117
 3    6.27707    22.1461   0.249248  5.87425
```

The column are, left to right: the peak number, peak position, peak height, peak width, and the peak area. (Note: for exponential pulse (**U**) and sigmoid (**S**) shapes, Position and Width are replaced by Tau1 and Tau2). Press **R** to print this table out in the command window. Peaks are numbered from left to right. (The area of each component peak within the upper window is computed using the trapezoidal method and displayed after the width). Pressing **Q** prints out a report of settings and results in the command window, like so:

```
Peak Shape = Gaussian
Number of peaks = 3
```

```
Fitted range = 5 - 6.64
Percent Error = 7.4514    Elapsed time = 0.19 Sec.
Peak#   Position      Height       Width        Area
1       5.33329      14.8274     0.262253     4.13361
2       5.80253       26.825     0.326065     9.31117
3       6.27707      22.1461     0.249248     5.87425
```

9. To select the baseline correction mode, press the **T** key repeatedly. It cycles through *6 baseline correction modes*: *none*, *linear tilted*, *quadratic*, *flat, tilted mode(y), flat mode(y)*. When baseline subtraction is linear, a straight-line baseline connecting the two ends of the signal segment in the upper panel will be automatically subtracted. When baseline subtraction is quadratic, a parabolic baseline connecting the two ends of the signal segment in the upper panel will be automatically subtracted. The *mode(y)* method subtracts the *most common y value* from all the points in the selected region. For peak-type signals where the peaks usually return to the baseline between peaks, this is usually the baseline even if the signal does not return to the baseline at the ends like modes 2 and 3 (graphic example). Use the quadratic baseline correction if the baseline is curved, as in these examples:



BaselineMode OFF | Linear BaselineMode | Quadratic BaselineMode

10. If you prefer to set the baseline manually, press the **B** key, then click on the baseline to the LEFT the peak(s), then click on the baseline to the RIGHT the peak(s). The new baseline will be subtracted and the fit re-calculated. (The new baseline remains in effect until you use the pan or zoom controls). Alternatively, you may use multipoint background correction for the entire signal: press the **Backspace** key, type in the desired number of background points and press the **Enter** key, then click on the baseline starting at the left of the lowest x-value and ending to the right of the highest x-value. Press the \ key to restore the previous background to start over.

11. In some cases, it will help to manually specify the first-guess peak positions: press **C**, then click on your estimates of the peak positions in the upper graph, once for each peak. A fit is automatically performed after the last click. Peaks are numbered in the order clicked. For the most difficult fits, you can type **Shift-C** and then type in or paste in (**Shift-V**) the *entire start vector,* complete with square brackets, e.g., "[pos1 wid1 pos2 wid2 ...]" where "pos1" is the *position* of peak 1, "wid1" is the *width* of peak 1, and so on for each peak. The custom start values remain in effect until you change the number of peaks or use the pan or zoom controls. Hint: if you Copy the start vector and keep it in the paste buffer, you can use the **Shift-C** key and paste it back in after changing the pan

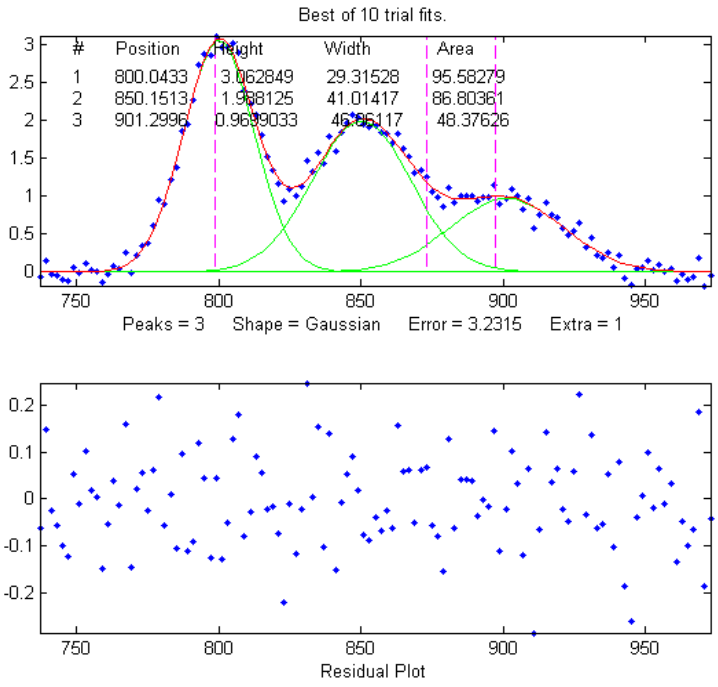
or zoom controls. Note: It is possible to click beyond the x-axis range, to try to fit a peak whose maximum is outside the x-axis range displayed. This is useful when you want to fit a curved baseline by treating it as an additional peak whose peak position if off-scale (real data graphic example).

12. The **A** and **Z** keys control the "shape" parameter ('extra') that is used only if you are using the "equal shape" models such as the Voigt profile, Pearson, exponentially broadened Gaussian (ExpGaussian), exponentially-broadened Lorentzian (ExpLorentzian), bifurcated Gaussian, Breit-Wigner-Fano, or Gaussian/Lorentzian blend. For these models, the shapes are variable with the **A**and **Z** keys but are the same for all peaks in the model. For the Voigt profile, the "shape" parameter controls *alpha,* the ratio of the Lorentzian width to the Doppler width. For the Pearson shape, a value of 1.0 gives a Lorentzian shape, a value of 2.0 gives a shape roughly half-way between a Lorentzian and a Gaussian, and larger values give a nearly Gaussian shape. For the exponentially broadened Gaussian shapes, the "shape" parameter controls the exponential "time constant" (expressed as the number of points). For the Gaussian/Lorentzian blend and the bifurcated Gaussian shape, the "shape" parameter controls the peak asymmetry (a values of 50 gives a symmetrical peak). For the Breit-Wigner-Fano, it controls the Fano factor. You can enter an initial value of the "shape" parameter by pressing **Shift-X** , typing in a value, and pressing the **Enter** key. For multi-shape models, enter a vector of "extra" values, one for every peak, enclosed in square brackets. For single-shape models, you can adjust this value using the **A** and **Z** keys (hold down the **Shift** key to fine tune). Seek to minimize the Error % or set it to a previously determined value. Note: if fitting multiple overlapping variable-shape peaks, it is easier to fit a single peak first, to get a rough value for the "shape" parameter, then just fine-tune that parameter for the multipeak fit if necessary.

13. For situations where the shapes may be different for each peak and you want the computer to determine the best-fit shape for each peak separately, use the shapes with three unconstrained iterated variables: 30=variable alpha Voigt, 31=variable time constant ExpGaussian (**Shift-R**), 32=variable shape Pearson, 33=variable percent Gaussian/Lorentzian blend. These models are more

time-consuming and difficult, especially for multiple overlapping peaks.

14. For difficult fits, it may help to press **X,** which restarts the iterative fit 10 times *with slightly different first guesses* and takes the one with the lowest fitting error. In version 13.2, the center of the graph displays "Working..." while the fits are in progress. This will take a little longer, obviously. (You can change the number of trials, "NumTrials", in or near line 227 - the default value is 10). *The peak positions and widths resulting from this best-of-10 fit then become the starting points for subsequent fits*, so the fitting error should gradually get smaller and smaller each time you press **X**, until it settles down to a minimum. If none of the 10 trials gives a lower fitting error than the previous one, nothing is changed. Those starting values remain in effect until you change the number of peaks or use the pan or zoom controls. (Remember: equal-width fits, fixed-width fits, and fixed position shapes are both faster, easier, and much more stable than regular variable fits, so use equal-width fits whenever the peak widths are expected to be equal or nearly so, or fixed-width (or fixed position) fits when the peak widths or positions are known from previous experiments).



15. Press **Y** to display the entire signal full screen without cursors, with the last fit displayed in green. The residual is displayed in red, on the same y-axis scale as the entire signal.

16. Press **M** to switch back and forth between log and linear modes. In log mode, the y-axis of the upper plot switches to semilog-y, and log(model) is fit to log(y), which may be useful if the peaks vary greatly in amplitude.

17. Press the **D** key to save the fitting data to disc as SavedModel.mat, containing two matrices: DataSegment (the raw data segment that is fit) and ModelMatrix (a matrix containing each component of the model interpolated to 600 points in that segment). To place these into the workspace, type load SavedModel. To plot the saved DataSegment, type plot(DataSegment(:,1), DataSegment(:,2)). To plot SavedModel, type plot(ModelX,ModelMatrix). Each component in the model will be plotted in a different color.

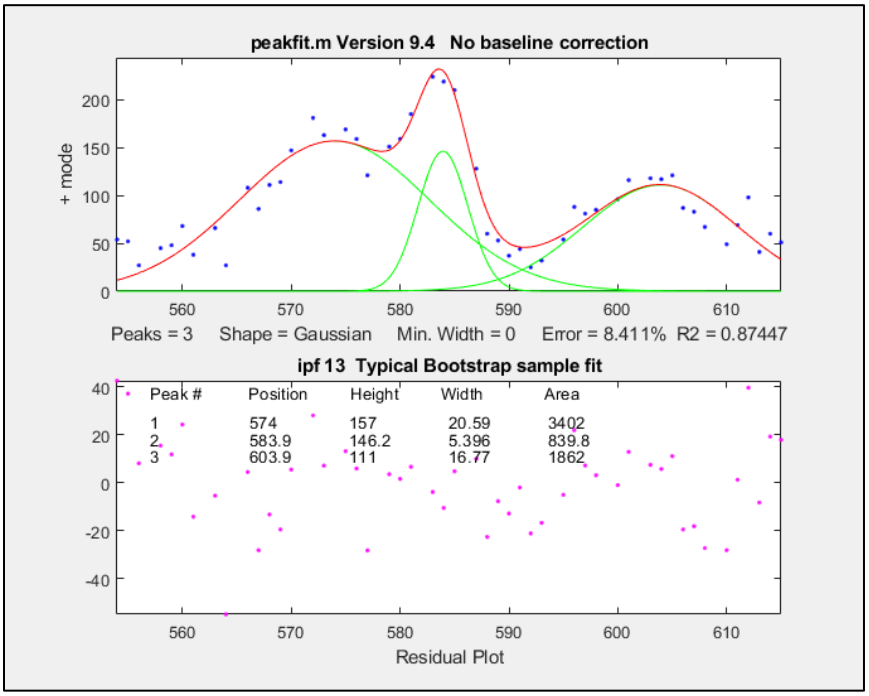
18. Press **W** to print out the peakfit.m function with all input arguments, including the last best-fit values of the first guess vector. You can copy and paste the peakfit.m function into your own code or into the command window, then replace "datamatrix" with your own x-y matrix variable.

19. Both ipf.m, and peakfit.m are able to estimate the expected variability of the peak position, height, width, and area from the signal, by using the bootstrap sampling method (page 170). This involves extracting 100 bootstrap samples from the signal, fitting each of those samples with the model, then computing the uncertainty of each peak: the standard deviation (RSD) and the relative percent standard deviation (%RSD). Basically, this method calculates weighted fits of a single data set, using a different set of different weights for each sample. This process is computationally intensive and can take several minutes to complete, especially if the number of peaks in the model and/or the number of points in the signal are high.

To activate this process in ipf.m, press the **V** key. It first asks you to type in the number of "best-of-x" trial fits per bootstrap sample (the default is 1, but you may need higher number here if the fits are occasionally unstable. Try 5 or 10 here if the initial results give NaNs or wildly improbable numbers). (To activate this process in peakfit.m, you must use version 3.1 or later and include all six output arguments, e.g. `[FitResults, LowestError, residuals, xi, yi, BootstrapErrors]=peakfit...)`. In version 13.2, the center of the graph displays "Working..." while the fits are in progress. The program displays the results as a table in the command window. For example, here is a three-peak Gaussian fit to some noisy experimental data, followed by a bootstrap statistics calculation:

```
Shape= Gaussian    % Fitting Error= 8.1876%    R2= 0.89861
         Peak#      Position      Height       Width       Area
            1        573.97       156.96       20.591      3401.6
            2        583.94       146.22       5.3956      839.78
            3        603.94       110.96       16.77       1861.7

Number of fit trials per bootstrap sample (0 to cancel): 1
Computing bootstrap sampling statistics....May take several minutes.
```




```
Peak #1
     Parameter       Mean        STD        STDIQR      PercentRSD     PercentRSDIQR
    ____________    _____      _______     _______     __________     _____________
    {'Position'}    573.89     0.33752     0.28748      0.058813       0.050093
    {'Height'  }    158.53      3.6802      3.0271        2.3215         1.9095
    {'width'   }    19.038       1.408      1.2717        7.3954         6.6795
    {'Area'    }    3187.6      194.89      178.79        6.1141         5.6088

Peak #2
     Parameter       Mean        STD        STDIQR      PercentRSD     PercentRSDIQR
    ____________    _____      _______     _______     __________     _____________
    {'Position'}    583.84     0.098986     0.1007      0.016954       0.017247
    {'Height'  }    156.03       10.837      11.916        6.9454         7.6372
    {'width'   }    5.5557      0.26153     0.24418        4.7074         4.3952
    {'Area'    }    924.28       95.719      76.147        10.356         8.2385

Peak #3
     Parameter       Mean        STD        STDIQR      PercentRSD     PercentRSDIQR
    ____________    _____      _______     _______     __________     _____________
    {'Position'}     603.4     0.36056     0.33927      0.059754       0.056227
    {'Height'  }    113.24      3.9028      4.3535        3.4465         3.8445
    {'width'   }    16.128      1.2027      1.2781         7.457         7.9247
    {'Area'    }    1843.3      79.733      87.735        4.3256         4.7598
Elapsed time is 10.655257 seconds.
```

Note that, despite the noise in the data and the 8.4% fitting error, the bootstrap relative standard deviations are not so bad, especially for peak positions and heights. Notice that the RSD of the peak position is *much lower* than that of the height and width and area. This is a typical pattern. Also, be aware that the reliability of the computed variability depends on the assumption that the noise in the signal is representative of the average noise in repeated measurements. If the number of data points in the signal is small, these estimates can be very approximate.

If the RSD and the RSDIQR are roughly the same (as in the example above), then the distribution of bootstrap fitting results is close to normal, and the fit is stable. If the RSD is substantially greater than

RSD IQR, then the RSD is biased high by "outliers" (obliviously erroneous fits that fall far from the norm), and in that case you should use the RSD IQR rather than the RSD, because the IQR is much less influenced by outliers. (Alternatively, you could use another model or a different data set to see if that gives more stable fits).

A likely pitfall with the bootstrap method, when applied to iterative fits, is the possibility that one (or more) of the bootstrap fits will go astray - that is, will result in peak parameters that are wildly different from the norm, causing the estimated variability of the parameters to be too high. For that reason, in ipf version 12.3, *two measures* of uncertainty are calculated: (a) the regular *standard deviation* (STD) and (b) the standard deviation estimated by dividing the interquartile range (IQR) by 1.34896. The IQR is more robust to outliers. For a *normal* distribution, the interquartile range is on average equal to 1.34896 times the standard deviation. If one or more of the bootstrap fits fails, resulting in a distribution of peak parameters with large outliers, the regular STD will be much greater than the IQR. In that case, a more realistic estimate of variability is IRQ/1.34896. It is best to try to increase the fit stability by choosing a better peak shape model, or adjusting the fitted range (pan and zoom keys), or the background subtraction (**T** or **B** keys), or the start positions (**C** key), and/or selecting a higher number of fit trials per bootstrap (which will increase the computation time). As a quick preliminary test of bootstrap fit stability, pressing the **N** key will perform a *single* iterative fit to a random bootstrap sub-sample and plot the result. Do that several times to see whether the bootstrap fits are stable enough to be worth computing a 100-sample bootstrap (**V** key). **Note**: it is normal for the stability of the bootstrap sample fits (**N** key: click here for animation) to be poorer than the full-sample fits (**F** key; click here for animation), because the latter includes only the variability caused by changing the starting positions for one set of data and noise, whereas the **N** and **V** keys aim to include the variability caused by the random noise in the sample by fitting bootstrap sub-samples. Moreover, the best estimates of the measured peak parameters are those obtained by the normal fits of the full signal (**F** and **X** keys), *not* the means reported for the bootstrap samples (**V** and **N** keys), because there are more independent data points in the full fits and because the bootstrap averages are influenced by the outliers that occur more commonly in the bootstrap fits. The bootstrap results are useful only for estimating the variability of the peak parameters, not for estimating their mean values. The **N** and **V keys** are also very useful ways to determine if you are using too many peaks in your model. *Superfluous peaks will be very unstable when **N** is press repeatedly* and will have in much higher standard deviation of its peak height when the **V key** is used.

Peakfit and the bootstrap method can work well for estimating the precision of peak parameter measurements, even when the signal-to-noise ratio is quite poor. For an example based on the IR spectrum of benzene, from the NIST Quantitative Infrared Database, see FittingWeakPeaks.pdf.

Shift-o fits a simple polynomial (linear, quadratic, cubic, etc.) to the segment of the signal dis-played in the upper panel and displays the polynomial coefficients (in descending powers) and the R^2.

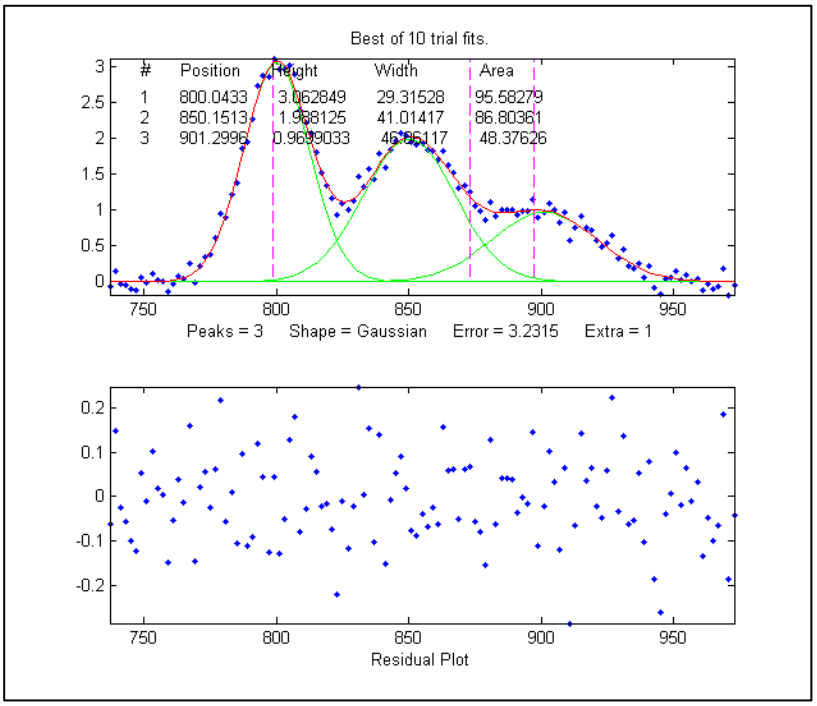
20. If some peaks are saturated and have a flat top (clipped at maximum height), you can make the program ignore the saturated points by pressing **Shift-M** and entering the maximum Y values to keep. Y values above this limit will simply be ignored. Peaks below this limit will be fit as usual.

To limit how narrow the model peaks can be, press **Shift-W,** and enter the minimum peak width

allowed.

Demoipf.m

Demoipf.m is a demonstration script for ipf.m, with a built-in simulated signal generator. The true values of the simulated peak positions, heights, and widths are displayed in the Matlab command window, for comparison to the FitResults obtained by peak fitting. The default simulated signal contains six independent groups of peaks that you can use for practice: a triplet near x = 150, a singlet at 400, a doublet near 600, a triplet near 850, and two broad single peaks at 1200 and 1700. Run this demo and see how close to the actual true peak parameters you can get. The useful thing about a simulation like this is that you can get a feel for the accuracy of peak parameter measurements, that is, the difference between the true and measured values of peak parameters. To download these m-files, right-click on the links, select **Save Link As...**, and click **Save**. To run it, place both ipf.m and Demoipf in the Matlab search path, then type **Demoipf** at the Matlab command prompt.



An example of the use of this script is shown in the figure. Here we focus on the 3 fused peaks located near x=850. The true peak parameters (before the addition of the random noise) are:

Position	Height	Width	Area
800	3	30	95.808
850	2	40	85.163
900	1	50	53.227

When these peaks are isolated in the upper window and fitted with three Gaussians, the results are:

Position	Height	Width	Area
800.04	3.0628	29.315	95.583
850.15	1.9881	41.014	86.804
901.3	0.9699	46.861	48.376

So you can see that the accuracy of the measurements is excellent for peak position, good for peak height, and least good for peak width and area. It is no surprise that the least accurate measurements are for the smallest peak with the poorest signal-to-noise ratio. Note: the predicted standard deviation of these peak parameters can be determined by the bootstrap sampling method, as described in the previous section. You would expect that the measured values of the peak parameters (comparing the true to the measured values) would be within about 2 standard deviations of the true values listed above).
Demoipf2.m is identical, except that the peaks are superimposed on a strongly curved baseline, so you can test the accuracy of the baseline correction methods (# 9 and 10, above).

Execution time of peak fitting and other signal processing tasks

By execution time, I mean the time it takes for one operation to be performed, exclusive of plotting or printing the results, when Matlab is running on a standard Windows PC. For iterative peak fitting, the biggest factors that determine the execution time are (a) the speed of the computer, (b) the number of peaks, and (c) the peak shape:

a) The execution time can vary over a factor of 4 or 5 or more between different computers. For example a small laptop, with 1.6 GHz, dual-core Athlon CPU with 4 Gbytes RAM, will be a lot slower than a desktop computer with a 3.4 GHz i7 CPU with 16 Gbytes RAM. Run the Matlab "bench.m" benchmark test to see how your computer stacks up.

b) The execution time increases with the product of the number of peaks in the model times the number of iterated variables per peak. (See PeakfitTimeTest.m).

c) The execution time varies greatly (sometimes by a factor of 100 or more) with the peak shape, with the exponentially-broadened shapes being the slowest and the fixed-width shapes being the fastest. See PeakfitTimeTest2.m and PeakfitTimeTest2a.m. Equal-width and fixed-width shape variations are always faster.

d) The execution time increases directly with NumTrials in peakfit.m. The "Best of 10 trials" function (**X** key in ipf.m) takes about 10 times longer than a single fit.

Other factors that are less important are the number of data points in the fitted region (but only if the number of data points is very large. For example see PeakfitTimeTest3.m) and the starting values (good starting values can reduce execution time slightly. PeakfitTimeTest2.m and PeakfitTimeTest2a.m have examples of that). Note: some of these scripts need DataMatrix2 and DataMatrix3, which you can download from http://tinyurl.com/cey8rwh.

TimeTrial.txt is a text file comparing the speed of *18 different signal processing tasks* running on 5 different systems: (1) Windows 10, 64-bit, 3.6 GHz core i7, with 16 GBytes RAM, using Matlab 9.9 (R2020b) Update 3 academic, (2) Matlab 2017b Home, (3) Matlab Online R2018b in the Chrome browser, running on older desktop or laptop PCs, (4) Matlab Mobile on an iPad, and (5) Octave 6.2.0 running on a desktop computer. The Matlab/Octave code that generated this is TimeTrial.m, which runs all the tasks one after the other and prints out the elapsed times for your particular machine, in addition to the times previously recorded for each task on each of the five software systems. TimeTrial.xlsx

summarizes the comparison of Matlab to Octave. Also see page 440 for a speed comparison between *Matlab* and *Python* running a few different tasks.

Iterative Curve Fitting Hints and Tips

1. If the fit fails completely, returning all zeros, the data may be formatted incorrectly. The independent variable ("x") and the dependent variable ("y") must be separate vectors or columns of a 2-x n matrix, with x in the first row or column. Or it may be that first guesses ("start") need to be provided for that fit.
2. It is best *not* to smooth your data prior to curve fitting. Smoothing can distort the signal shape and the noise distribution, making it harder to evaluate the fit by visual inspection of the residuals plot. Smoothing your data beforehand makes it impossible to achieve the goal of obtaining a random unstructured residuals plot and it increases the chance that you will "fit the noise" rather than the actual signal. The bootstrap error estimates are invalid if the data are smoothed.
3. The most important factor in non-linear iterative curve fitting is selecting the *underlying model peak function*, for example, Gaussian, Equal-width Gaussians, Lorentzian, etc. (see page 210). It is worth spending some time finding and verifying a suitable function for your data. If the peak widths of each group of peaks are expected to be the same or nearly so, select the "equal-width" shapes (available for the Gaussian and Lorentzian shapes) are faster, easier, and much more stable than regular variable-width fits. But it is important to understand that a good fit is *not by itself proof* that the shape function you have chosen is the correct one. In some cases, the wrong function can give a fit that looks perfect. For example, consider a 5-peak Gaussian model that has a low percent fitting error and for which the residuals look random - usually an indicator of a good fit (Click for a graphic if you are reading online). But in fact, in this case, the model is *wrong*. Those data came from an experimental domain where the underlying shape is fundamentally non-Gaussian but, in some cases, can look very like a Gaussian. It is important to get the model right for the data and not depend solely on the goodness of fit.
4. You should always use the *minimum* number of peaks that adequately fit your data. (page 210). Using too many peaks may result in an unstable fit - the green lines in the upper plot, representing the individual component peaks, will bounce around wildly for each repeated fit, without significantly reducing the Error. A very useful way to determine if you are using too many peaks in your model is to use the **N key** (see #10, below) to perform a single fit to a bootstrap sub-sample of points. *Superfluous peaks will be very unstable when **N** is press repeatedly*. (You can get better statistics for this test, at the expense of time, by using the **V key** to compute the standard deviation of 100 bootstrap sub-samples).
5. If the peaks are superimposed on a background or baseline, then that must be accounted for *before* fitting, otherwise, the peak parameters (especially height, width, and area) will be inaccurate. Either subtract the baseline from the *entire* signal using the **Backspace** key (#10 in Operating Instructions, above) or use the **T** key to select one of the automatic baseline correction modes (# 9 in Operating Instructions, above).
6. This program uses an iterative non-linear search function (*"modified Simplex"*) to determine the peak positions and widths that best match the data. This requires first guesses for the peak positions and widths. (The peak *heights* do not require first guesses, because they are linear parameters. The program determines them by linear regression). The default first guesses for the peak positions are

made by the computer based on the pan and zoom settings and are indicated by the magenta vertical dashed lines. The first guesses for the peak widths are computed from the zoom setting, so the best results will be obtained if you zoom in so that the group of peaks is isolated and spread out as suggested by the peak position markers (vertical dashed lines).

7. If the peak components are very unevenly spaced, you might be better off entering the first-guess peak positions yourself by pressing the **C** key and then clicking on the top graph where you think the peaks might be. None of this must be exact - they're just first guesses, but if they are too far off it can throw the search algorithm off. You can also type in the first guesses for position *and* width manually by pressing **Shift-C**).
8. Each time you perform another iterative fit (e.g., pressing the **F** key), the program adds small random deviations to the first guesses, to determine whether an improved fit might be obtained with slightly different first guesses. This is useful for determining the robustness or stability of the fit *with respect to starting values*. If the error and the values of the peak parameters vary slightly in a tight region, this means that you have a robust fit (for that number of peaks). If the error and the values of the peak parameters bounce around wildly, it means the fit is not robust (try changing the number of peaks, peak shape, and the pan and zoom settings), or it may simply be that the data are not good enough to be fit with that model. Try pressing the **X** key, which takes the best of 10 iterative fits and uses those best-fit values as the *starting first guess* for subsequent fits. So, each time you press **X**, if any of those fits yield a fitting error less than the previous best, that one is taken as the start for the next fit. As a result, the fits tend to get better and better gradually as the **X** key is pressed repeatedly. Often, even if the first fit is terrible, subsequent **X**-key fits will improve considerably.
9. The variability in the peak parameters from fit to fit using the **X** or **F** keys is only an estimate of the uncertainty caused by the curve fitting procedure (but *not* of the uncertainty caused by the noise in the data, because this is only for one sample of the data and noise;or that you need the **N** key fits).
10. To examine the robustness or stability of the fit *with respect to random noise in the data*, press the **N** key. Each time you press **N**, it will perform an iterative fit on a different subset of data points in the selected region (called a "bootstrap sample"; see page 170). Click for animation. If that gives reasonable-looking fits, then you can go on to compute the peak error statistics by pressing the **V** key. If on the other hand the **N** key gives wildly different fits, with highly variable fitting errors and peak parameters, then the fit is not stable, and you might try the **X** key to take the best of 10 fits and reset the starting guesses, then press **N** again. In difficult cases, it may be necessary to increase the number of trials when asked (but that will increase the time it takes to complete), or if that does not help, use another model or a better data set. (The **N** and **V** keys are a good way to evaluate a multi-peak model for the possibility of superfluous peaks, see # 4 above).
11. If you do not find the peak shape you need in this program, look at the next section to learn how to add your own new ones, or write me at toh@umd.edu and I'll see what I can do.
12. If you try to fit a very small independent variable (x-axis) segment of a very large signal, say, a region that is only 1000th or less of the entire x-axis range, you might encounter a problem with unstable fits. If that happens, try subtracting a constant from x, then perform the fit, then add in the subtracted amount to the measured x positions.

13. If there are very few data points on the peak, it might be necessary to reduce the minimum width (set by minwidth in peakfit.m or Shift-W in ipf.m) to zero or to something smaller than the default minimum (which defaults to the x-axis spacing between adjacent points).
14. Difference between the **F**, **X**, **N**, and **V** keys in ipf.m:
    - **F key**: Slightly varies the starting values and performs a single iterative fit using all the data points in the selected region.
    - **X key**: Performs 10 iterative trial fits using all the data points in the selected region, slightly varying the starting values before each trial, then takes the one with the lowest fitting error. Press it again to repeat and refine the fit. Takes about 10 times longer than the **F** key.
    - **N key**: Slightly varies the starting values and performs an iterative single fit using a random subset of the data points in the selected region. Use to visualize the stability of the fit with respect to random noise. Takes the same time as the **F** key.
    - **V key**: Asks for several trial fits, then performs 100 iterative fits each on a separate random subset of the data points in the selected region, each fit using the specified number of trials and taking the best one, then calculates the mean and standard deviation of the peak parameters of all 100 best-fit results. Use to quantify the stability of peak parameters with respect to random noise. Takes about 100 times longer than the **X** key.

Extracting the equations for the best-fit models

The equations for the peak shapes are in the peak shape menu in ipf.m, isignal.m, and ipeak.m. Here are the expressions for the shapes that are expressed mathematically rather than algorithmically:

```
Gaussian:           y =exp(-((x-pos)/(0.60056120439323*width)) ^2)
Lorentzian:         y =1/(1+((x-pos)/(0.5*width))^2)
Logistic:           y =exp(-((x-pos)/(.477*wid))^2); y=(2*n)/(1+n)
Lognormal:          y = exp(-(log(x/pos)/(0.01*wid)) ^2)
Pearson:            y =1/(1+((x-pos)/((0.5^(2/m))*wid))^2)^m
Breit-Wigner-Fano:  y =((m*wid/2+x-pos)^2)/(((wid/2)^2)+(x-pos)^2)
Alpha function:     y =(x-spoint)/pos*exp(1-(x-spoint)/pos)
Up Sigmoid:         y =.5+.5*erf((x-tau1)/sqrt(2*tau2))
Down Sigmoid        y =.5-.5*erf((x-tau1)/sqrt(2*tau2))
Gompertz:           y =Bo*exp(-exp((Kh*exp(1)/Bo)*(L-t) +1))
FourPL:             y = 1+(miny-1)/(1+(x/ip)^slope)
OneMinusExp:        y = 1-exp(-wid*(x-pos))
EMG (shape 39)      y = s*lambda*sqrt(pi/2)*exp(0.5*(s*lambda)^2-lambda*(t-
mu))*erfc((1/sqrt(2))*(s*lambda-((t-mu)/s)))
```

The peak parameters (height, position, width, tau, lambda, etc.) that are returned by the FitResults displayed on the graph and in the command window. For example, if you fit a set of data to a single Gaussian and get…

```
Peak#    Position     Height    Width       Area
  1       0.30284       87.67     0.23732      22.035
```

...then the equation would be:

```
y = 87.67*exp(-((x-0.30284)/(0.6005612*0.23732)).^2)
```

If you specify a model of more than one peak, then the equation is the *sum* of each peak in the model. For example, fitting the built-in Matlab "humps" function using a model of 2 Lorentzians

```
>> x=[0:.005:2];y=humps(x);[FitResults,GOF]=peakfit([x' y'],0,0,2,2)

    Peak#    Position   Height    Width     Area
      1      0.3012   96.9405    0.1843   24.9270
      2      0.8931   21.1237    0.2488    7.5968
```

Using the expression for a Lorentzian on the previous page, the equation for two peaks would be:

```
y = 96.9405*(1/(1+((x-0.3012)/(0.5*0.1843))^2)) + 21.1237*(1//(1+((x-
0.8931)/(0.5*0.2488)).^2))
```

It is also possible to use multiple shapes in one fit, by specifying the peak shape parameter as a vector. For example, you could fit the first peak of the "humps" function with a Lorentzian and the second peak with a Gaussian by using [2 1] as the shape argument.

```
>> x=[0:.005:2];y=humps(x);peakfit([x' y'],0,0,2,[2 1])

    Peak#    Position   Height    Width      Area
      1      0.3018   97.5771    0.1876   25.4902
      2      0.8953   18.8877    0.3341    6.7180
```

In that case the expression would be y = height1 *Lorentzian + height2*Gaussian:

```
y = 97.5771*(1/(1+((x-0.3018)/(0.5*0.1876))^2)) + 18.8877*exp(-((x-
0.8953)/(0.60056120439323*0.3341))^2)
```

Note: To obtain the *digitally sampled* model data, use peakfit is 6th and 7th *output* parameters, xi and yi, which return a 600-point interpolated model as a vector of x values and a matrix of y values with one row for each component. Type `plot(xi,yi)` to plot each model peak separately in a different color or `plot(xi,yi(1,:))` to plot just peak 1.

“

How to add a new peak shape to peakfit.m, ipf.m, iPeak, or iSignal

It is easier than you think to add your own custom peak shape to peakfit.m (or to those interactive functions that use peakfit.m internally, such as ipf.m, iSignal, or iPeak), if you have a mathematical expression for your shape. The easiest way is to *modify an existing peak shape* that you do not plan to use, replacing it with your new function. Take care to pick a shape to sacrifice that has the *same number of variables and constraints* are your new shape. For example, if your shape has *two* iterated parameters (e.g., variable position and width), you could modify the Gaussian, Lorentzian, or triangular shape (number 1, 2 or 21, respectively). If your shape has *three* iterated variables, use shapes like 31, 32, 33, or 34. If your shape has *four* iterated variables, use shape 49 (“double gaussian”, Shift-K). If your shape has an 'extra' parameter, like the equal-shape Voigt, Pearson, BWF, or blended Gaussian/ Lorentzian, use one of those. If you need an exponentially modified shape, use the exponentially modified Gaussian (5 or 31) or Lorentzian (18). If you need equal widths or fixed widths, etc., use one

of those shapes. ***This is important***. You *must* have the *same number of variables and constraints*, because the structure of the code is different for each class of shapes.

There are just two required steps to the process:

1. Let us say that your shape has *two* iterated parameters, and you are going to sacrifice the triangular shape, number 21. Open peakfit.m or ipf.m in the Matlab editor and re-write the *old* shape function ("triangular", located near line 3672 in ipf.m - there is one of those functions for each shape - by changing both the *name* of the function and the *mathematics* of the assignment statement (e.g., `g = 1-(1./wid).*abs(x-pos);`). You can use the same variables ('x' for the independent variable, 'pos' for peak position, 'wid' for peak width, etc.) Scale your function to have a peak height of 1.0 (e.g., after computing y as a function of x, divide by `max(y)`).

2. Use the search function in Matlab to find all instances of the name of the old function and replace it with the new name, *checking "Wrap around" but leaving "Match case" and "Whole word" unchecked* in the Search box. If you do it right, for example, all instances of "triangular" and all instances of "fittriangular' will be modified with your new name replacing "triangular". **Save** the result (or **Save as...** with a modified file name).

Done! Your new shape will now be shape 21 (or whatever was the shape number of the old shape you sacrificed). In ipf.m, it will be activated by the same keystroke used by the old shape (e.g., Shift-T for the triangular, key number 84), and in iSignal and in iPeak, the menu of peak shapes will have been modified by the search and replace in step 2.

If you wish, you can change the keystroke assignment in ipf.m. First, find a key or Shift-key that is not yet assigned: pressing an unassigned key gives an "UnassignedKey" error statement when you press that key with ipf.m running. Then change the old key number 84 to that unassigned one in the big "`switch double(key),`" case statement near the beginning of the code.

Which to use? *peakfit*, ipf, *findpeaks…*, iPeak, or iSignal?

I designed iPeak (page 416), iSignal (page 376), peakfit (page 397), and ipf (page 416), or their Octave versions, each with a different emphasis, although there is some overlap in their functions. Briefly, iSignal combines several basic functions, including smoothing, differentiation, spectrum analysis, etc. iPeak and the various findpeaks functions focus on finding multiple peaks in large signals. Peakfit and ipf focus on iterative peak fitting. But there is some overlap: iPeak and iSignal can also perform least-squares iterative peak fitting, and iSignal can perform peak finding. In addition, iSignal, iPeak, and ipf are *interactive*, and work in *Matlab* or in *Matlab Online* in a web browser, whereas their *command-line* versions ProcessSignal.m, the many variations of findpeaks.m, and peakfit.m are functions. The main point is that the interactive functions are better for exploration and trying out different techniques, whereas the command-line functions are better for automatic hand-off processing of masses of data.

> **Common features**. The interactive programs iSignal, iPeak, and ipf all have several features in common.

(a) The **K** key displays the keyboard controls for each program.

(b) The Matlab versions use pan and zoom keys are the four cursor keys – left and right for pan, up and down for zoom. *The Octave versions use the < and > keys (with and without shift).*

(c) **Ctrl-Shift-A** selects the entire signal (that is, zooms out all the way). In the Octave versions, the ; key does this.

(d) The **W** key. To facilitate transfer of settings from one of these functions to another or to a command-line version, all these functions use the **W** key to print out the syntax of other related functions, with the pan and zoom settings and other numerical input arguments specified, ready for you to Copy and Paste into your own scripts or back into the command window. For example, you can convert a curve fit from ***ipf*** into the command-line ***peakfit*** function. Or you can convert a peak finding operation from ***ipeak*** into the command line ***findpeaksG*** or ***findpeaksb*** or ***findpeaksb3*** functions. This provides a way to deal with signals that require different signal processing in different regions of their x-axis ranges, by allowing you to create a series of command-line functions for each local region that, when executed in sequence, quickly process each segment of the signal appropriately and can be repeated easily for any number of other examples of that same type of signal.

(e) All these programs use the **Shift-Ctrl-S**, **Shift-Ctrl-F**, and **Shift-Ctrl-P** keys to transfer the current signal, as a global variables X and Y, to i**S**ignal.m, ip**f**.m, and i**P**eak.m, respectively.

First time here? Check out these *animated Web demos* of ipeak.m and ipf.m. Or download these Matlab demo functions that compare ipeak.m with peakfit.m for signals with a few peaks and signals with many peaks and that shows how to adjust ipeak to detect broad or narrow peaks. These self-contained demos include all required Matlab functions. Just place them in your path and click **Run** or type their name at the command prompt. Or you can download all these demos together in idemos.zip. peakfitVSfindpeaks.m performs a direct comparison of the peak parameter accuracy of findpeaks vs peakfit.

Note 1: Click on the Matlab Figure window to activate the interactive keyboard tools. Make sure you do not click on the "Show Plot Tools" button in the toolbar above the figure. That will disable normal program functioning. If you do, just close the Figure window and start again

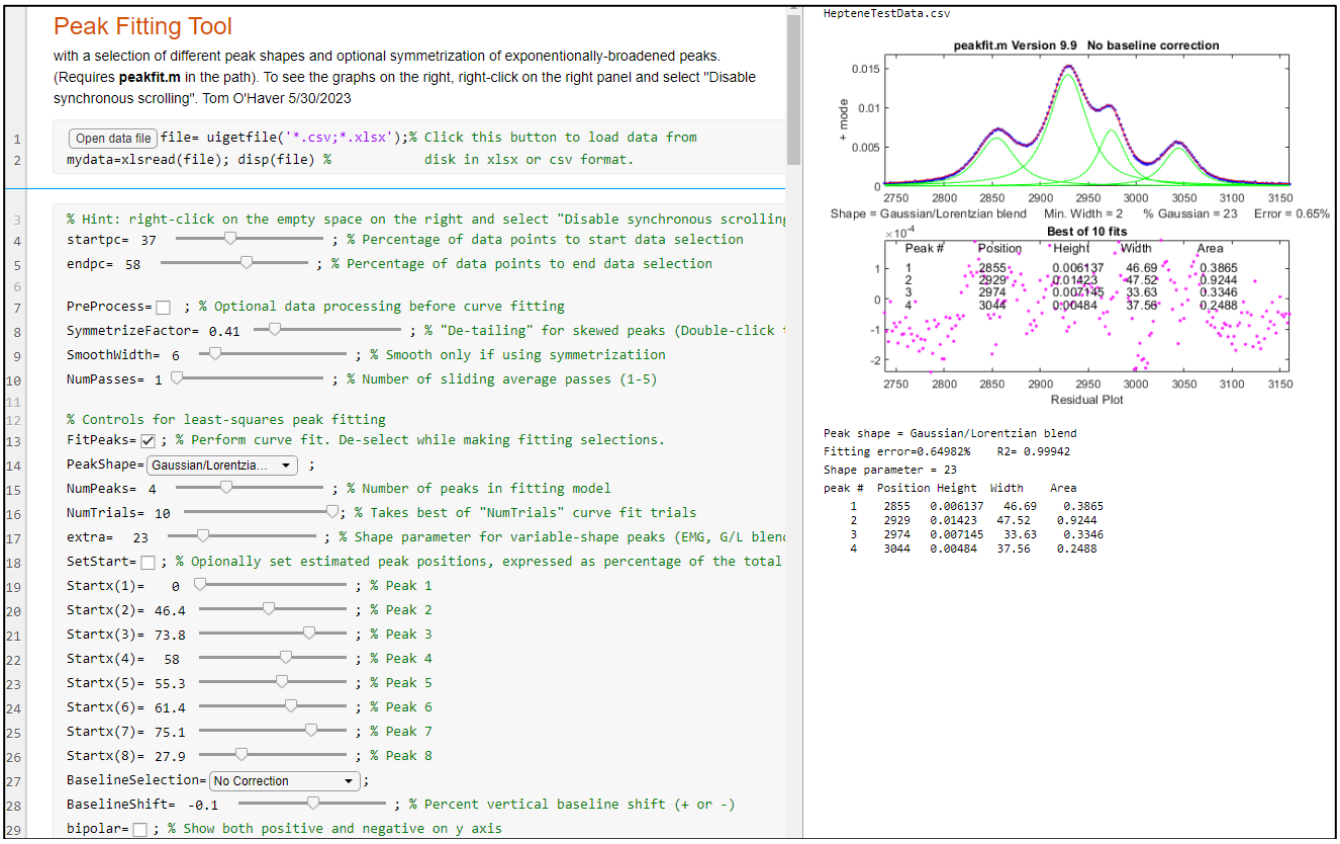
Note 2: The interactive keypress-operated iPeak, iSignal, iFilter, and ipf functions also work when you run Matlab in a web browser (just click on the figure window first), but they do not currently work on Matlab Mobile. Separate versions are required if you are using Octave (with "octave" added to the filename, e.g. ipfoctave.m instead of ipf.m, etc.)

Note 3: Scripts and functions must be located on your computer in a location specified by the "Matlab search path", which is the set folders in the file system that Matlab uses to locate files. See "What Is the MATLAB Search Path?"

See page 370 for a table listing of Live Script tools and their corresponding keypress-driven functions.

Live Script Peak Fitter tool

PeakFittingTool.mlx performs iterative least-squares peak fitting on experimental data stored on disk. Clicking the "**Open data file**" button in line 1 opens a file browser, allowing you to navigate to your data file (in .csv or .xlsx format. The script assumes that your x,y data are in the first two columns). But before opening a file, it's a good idea to temporarily de-select the "**FitPeaks**" checkbox in line 14, then wait until you have set the other controls. That way you will avoid waiting for unnecessary curve fit operations until the appropriate settings are complete. (Sometimes curve fitting operations can be slow and can take several seconds in difficult cases). With FitPeaks switched off, the program simply displays a plot of the selected data file. Note: to view the graphic plots to the right of the code, as shown here, right-click on the empty space on the right and select "Disable synchronous scrolling".



Adjust the **startpc** and **endpc** sliders in lines 4 and 5 to isolate groups of closely-spaced peaks that can be fit together. Try to spread them out as evenly as possible, as shown in the figure above. (If all the peaks are well separated and do not overlap, and if the number of peaks varies unpredictable from signal to signal, you may be between off using the Peak Detector Tool (Peakdetector.mlx), which also has a peak fitting function).

The "**PreProcess**" check box (line 7) allows for some optional preliminary pre-processing. The **SymmetrizeFactor** slider preforms "symmetrization" or "de-tailing" for peaks that are skewed by exponential broadening, by means of the first-derivative addition. Increase the value of SymmetrizeFactor until the peak is as narrow as possible without the trailing edge falling below the

baseline. The SmoothWidth and NumPasses sliders (lines 9 and 10) permits sliding average smoothing on the signal, which is useful for cases where high-frequency noise obscures the peaks. The "**VerticalShift**" slider (line 11) allows for positive and negative shift in the baseline position, to compensate for baseline offset.

The **PeakShape** drop-down menu allows you to select the peak shape of the fitting model. **NumPeaks** sets the number of peaks in the model. NumTrials, restarts the fitting process "NumTrials" times with slightly different start values and selects the best one (with lowest fitting error). **NumTrials** can be any positive integer. In many cases, NumTrials=1 will be sufficient, but if that does not give consistent results, increase it until the results are stable. The "extra" slider is used to fine-tune the certain peak shapes, e.g., the Pearson, exponentially-broadened Gaussian, and Gaussian/Lorentzian blend. Adjust this to minimize the fitting error.

After all these settings have been made, then you can click the **FitPeaks** checkbox (line 13). A fit will be performed, and the resulting peak table displayed in the right-hand panel, as in the graphic above. Thereafter, any changes in the setting will cause an immediate recalculation of the curve fit.

In difficult cases, better results can be obtained if you specify the estimated positions of the peaks, especially if the peaks are very irregularly spaced or if some peaks appear only as shoulders or bulges rather than as distinct peaks. Select the **SetStart** check box and adjust the sliders to the predicted relative peak positions, for each peak in the model in lines 19 to 26. The length of these sliders represents the x-axis range displayed in the figure.

If the baseline for the group of peaks is offset from zero, you can correct that by using the **BaselineShift** slider in line 28. If the baseline for the group of peaks is tilted or curved, you can use the **BaselineSelection** menu in line 27 to choose a baseline correction that attempts to estimate the baseline from the edges of the signal range. The **Bipolar** check box (line 29) controls whether to display both positive and negative signal values in the graphic or only positive values.

Additional shapes may easily be added to the **PeakShape** menu by selecting other shapes form the list of predefined shapes and their corresponding number at https://terpconnect.umd.edu/~toh/spectrum/InteractivePeakFitter.htm, adding that name and number to the others in the switch/case statement in lines 52-73, then adding that new shape to the drop-down menu on line 15. Just follow that pattern of the shapes already there.

Another example of Peak Fitting Tool shows it fitting a group of weak peaks in the middle of a much larger signal (chrom.csv), in this case using the exponentially broadened Gaussian shape and the "Tilted mode" baseline correction (line 27).

In fitting asymmetrical peaks what have an exponential skew, you can either try to remove the asymmetry by using the **SymmetrizeFactor** slider (example) followed by fitting a symmetrical peak shape or by selecting an exponentially broadened peak shape (example). Both approaches can yield similar results as in these examples, but the former method is often faster.

Note: you can double-click sliders to change their ranges if the initial range is insufficient.

Artificial Intelligence and Signal Processing

In 2022, the artificial intelligence research and deployment company OpenAI, introduced a conversational "large-language" model called ChatGPT, which they had trained on billions of pages of text, almost every book ever published, all of Wikipedia, and selected websites. (You can try it for no cost at https://chat.openai.com/.) The current version, as of late 2025, is ChatGPT-5. There are now many other competing chatbots, such as Google's Gemini, Microsoft's CoPilot, Perplexity, Anthropic's Claude, use.ai's Grok, and DeepSeek.

The strength of chatbots is in language interpretation and writing. For example, chatbots are quite good at tasks like suggesting possible titles for papers, talks, or proposals that you are working on. Just feed it all or a portion of what you have written. They can also paraphrase and outline, which can be useful for writing condensed summaries or abstracts. Chatbots can offer opinions and suggestions as to whether your writing is suitable for a particular audience and will offer to re-write the whole thing, for example to make it more formal or less formal. Their knowledge base is extraordinarily wide and they can often answer very specific questions about technical topics. They are often better than search engines at answering very specific questions about hardware or software programs that would otherwise force you to plow through pages and pages of documentation. They can even write a first draft based on a general description, to help you get started. (For example, the table on page 462 was written by ChatGPT-5).

AI as a programmer's assistant.

But AIs can go well beyond human language tasks. They can also write code in several computer languages commonly used by scientists, such Matlab, Python, Mathematica, C/C++, R, Pascal, Fortran, HTML, JavaScript, Excel macros, etc. But the critical question is: *Can AIs write code easier, quicker, and more accurately that a human, specifically a human who is a busy scientist, not a trained computer programmer?*

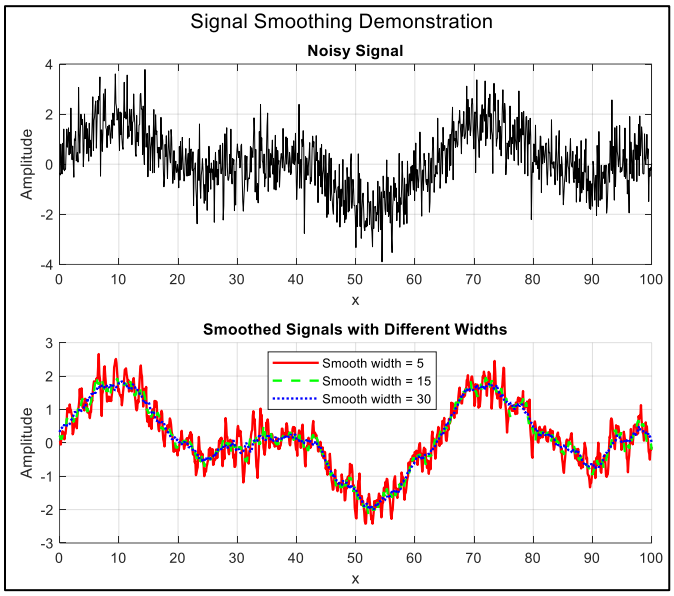
Based on my testing, chatbots can generate working code in Matlab or Python for quite a range of signal processing applications, *if you describe the task adequately*. I found that they could write code that works for many basic signal processing tasks, if the description is sufficiently complete. But for more complex tasks, its code might not work at all or might not do what you expect. Most often, the cause of failure is that *I have written the prompt unclearly or incompletely.* It's somewhat misleading that, even in cases where a chatbot's code does not work, *it is presented in good style*, divided into clearly identified sections, usually with explanatory comments for each line, appropriate indentation, examples of use, and even warning that the code may fail under certain circumstances (e.g., division by zero, etc.). There are even AI services that support code development, which currently include GitHub Copilot, Amazon CodeWhisperer, Codex, Tabnine, and even Cursor, Replit, Bolt, and Lovable. Here I will focus on general-purpose chatbots such as ChatGPT, which has a feature called "Canvas" that integrates chat and code editing for Python and other text-based formats (page 456).

It's a good idea to start simple and work up. A simple example where chatbots work perfectly is *Caruana's Algorithm* (page 174), which is a fast way to estimate the parameters of a signal peak that is locally Gaussian near its maximum. I asked the chatbots to create a function that

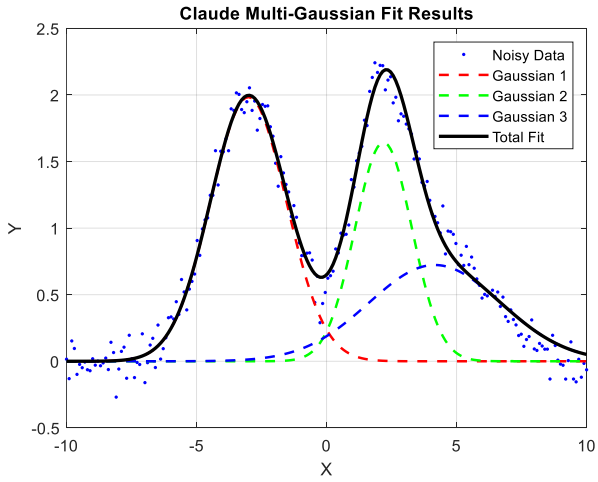
"*accepts two vectors x and y approximating a digitally sampled peak, takes the natural log of y, fits a parabola to the resulting data, then computes the position, FWHM, and height of the Gaussian from the coefficients of the parabolic fit*".

In this case the chatbot does all the required algebra and creates code that is functionally identical to my hand-coded version. Chatbots can be very handy for creating code and graphic illustrations for instructional materials. For example, I asked both ChatGPT and Microsoft's CoPilot to:

"*Write a Matlab script that creates a plot with two horizontal subplots, the top one showing a signal with several peaks and some random noise, and the bottom one showing the results of applying a smooth of three different widths to that signal, each shown in a different line style and color, with a legend identifying each plot and its smooth width*".



This results in a script that generates the graphic shown here, just as requested, although CoPilot included only one of the smooth widths in its legend. (Note that the chatbots are forced to make choices for several things that I did not specify exactly, including the shape of the peaks, the sampling rate and signal length, and the three smooth widths. Moreover, it adds titles to both subplots, even though I did not specify that detail). It is particularly handy that, if you want to generate a Python equivalent for example, you can simply say "*How can I do that in Python*" and it creates a working python script that imports the required libraries and generates an almost identical graphic. Although in this case it chose a much lower sampling rate. (The same may be true of the other languages that it knows, but I have not tested that). You can also feed it a program written in one of its languages and ask it to convert it into another of its languages. Sometimes the code generated by chatbots will result in an error, notated by a red warning message in the console of both Matlab and Python/Spyder. You can just copy that error message and paste it directly into the chatbot and it will attempt to fix the problem, often successfully.



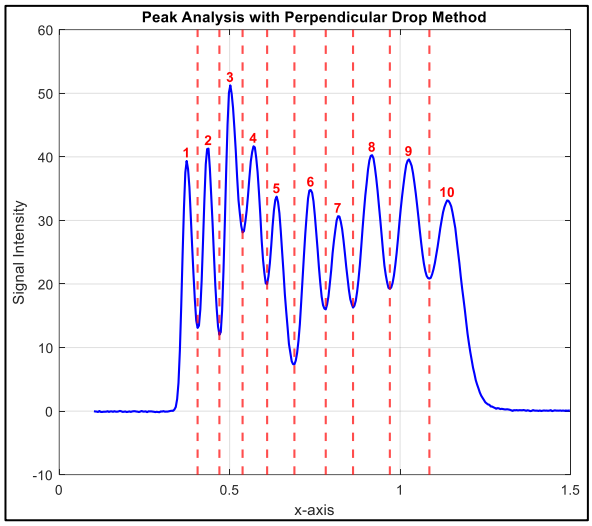
Another query that created well-structured, easily-modified, functional code is

"*Create a function that fits the sum of n Gaussian functions to the data in x and y, given the value of n and initial estimates of the positions and widths of each Gaussian, then repeats the fit 10 times with slightly different values of the initial estimates, takes the one with the lowest fitting error and returns its values for the position, width, and height of all the best-fit Gaussians*".

ChatGPT and Claude produce code that works and, without me even asking, creates a script that demonstrates it with a nice graph. Other cases where chatbot code worked well:

> "*Create a function that accepts a signal (x,y), then applies the perpendicular drop method to measure the peak widths. Plot the test signal and draw the vertical lines at each minimum*."

Both ChatGPT and Claude managed this neatly, as shown in the graphic below.



Other examples include:

- creating a Kalman filter,
- demonstrating the bootstrap method,
- detecting peaks and estimating their areas by perpendicular drop (shown on the right),
- demonstrating wavelet denoising (for which Claude's version worked better that ChatGPT's),
- and creating a function that symmetrizes exponentially broadened peaks by first-derivative addition, where Claude worked better than ChatGPT, Gemini, or Perplexity (graphic).

Clearly asking a chatbot to perform routine tasks such as these is quick and convenient, especially if you are creating the equivalent code in more than one language. It spits out the code faster than I can type. For larger more complex projects, you could break up the code into smaller chunks or functions that a chatbot can handle separately *and that you can test* separately, then combine them later as needed.

A chatbot always presents its results nicely formatted, with correct spelling and grammar, which many people interpret as a "confident attitude". This inspires trust in the results, but just as for people, *being confident does not always mean being correct.* There are several important caveats:

First, the code that a chatbot generates is not necessarily unique. If you ask it to repeat the task, you'll sometimes get different code (unless the task is so simple that there is only one possible way to do it correctly). This is not necessarily a flaw. There is often more than one way to code a function for a particular purpose. If the code requires variables to be defined, the names of those variables will be chosen by a chatbot and won't always be the same from trial to trial. Moreover, unless you specify the *name* of the function itself, a chatbot will choose that name as well, based on what the function does).

Second, and more importantly, there may be more than one way to *interpret your request*, unless you have very carefully worded it to be unambiguous. Take the example of data smoothing (page 42). On the face of it, this is a simple process. Suppose you ask for a function that performs a sliding average smooth of width *n* and applies it *m* times. How will that request be interpreted? If you simply say

> "*...apply an n-point sliding average smooth to the y data and repeats it m times*",

you will get this code, which does what you asked but probably not what you want. The point of applying repeat smooths is to apply the smooth again *to the previously smoothed data*, not to the *original* data, as this code does. The general rule is that *n* passes of a *m*-width smooth results in a center-weighted smooth of width *n***m*-*n*+1. You will get what you want if you ask ChatGPT for a

function that:

> "*…applies an n-point sliding average smooth to y and then repeats the smooth on the previously smoothed data m times*".

This is a small but critical difference, resulting in this code, whereas the previous code returns a singly-smoothed result no matter the value of *m*.

Third, there may be unspecified details or side effects that may require addressing, such as the expectation that the number of data points in the signal *after* processing should be the same as *before* processing. In the case of smoothing, for example, there is also the question of what to do about the first *n* and last *n* data points, for which there are not enough data points to compute a complete smooth (see page 45). There is also the requirement that the smooth operation should be constructed so that it does not shift the x-axis positions of signal features, which is critical in many scientific applications. Human-coded smooth algorithms, such as fastsmooth, consider all these details.

Another example of unspecified details is the measurement of the full width at half-maximum (FWHM) of smooth peaks of any shape. The function I wrote for that task is "halfwidth.m". I used its description as the ChatGPT query:

> *"…a function that accepts a time series x, y, containing one or more peaks, and computes the full width at half maximum of the of the peak in y nearest xo. If xo is omitted, it computes the halfwidth from the maximum y value*".

The AI came up with "computeFWHM.m", which works well if the sampling rate is high enough. However, the AI's version fails to interpolate between data points when the half-intensity points fall *between* data points and thus is inaccurate when the sampling rate is low, *because I did not specify that it should do so*. "CompareFWHM.m" compares both functions on some synthetic data with adjustable sampling rate.

Also, chatbots cannot evaluate the graphical results of their calculations. For example, if you ask it to apply some operation, and the graphical result is clearly incorrect, it does not have a clue. But you can complain about that to the chatbot and it will attempt a correction, sometimes successfully.

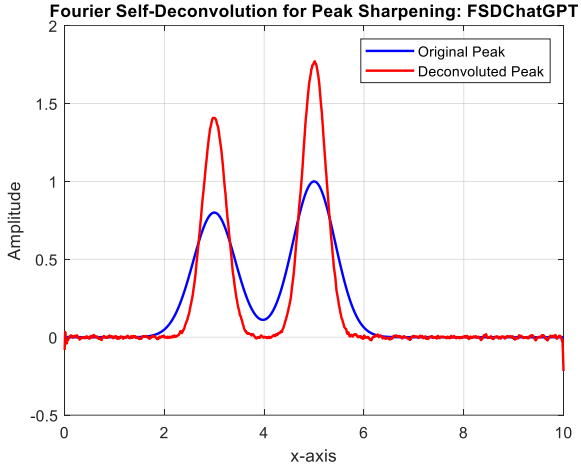
Another technical kink relates to the common Matlab practice of saving functions that you have written as a separate file in the path and then later calling that saved function from a script that you are writing, relying on Matlab to find it in the path. If you ask ChatGPT to convert that new script to another language, you must embed your external functions directly into the code (Matlab R2016b or later).

A more challenging signal processing task is iterative fitting of noisy overlapping peaks (page 204). I asked ChatGPT to…

> "*fit the sum of n Gaussian functions to the data in x and y, given initial estimates of the positions and widths of each Gaussian, returning the position, width, and height of all the best-fit Gaussians*".

ChatGPT came up with the attractively coded "iterativefitGaussians.m". The closest hand-coded equivalent in my tool chest was fitshape2.m (page 204). Both codes work, are about the same length, and require similar input and output arguments. There is seldom a uniquely correct answer to iterative fitting problems, but the difference in performance between these two codes is instructive. The self-contained script "DemoiterativefitGaussians2.m" compares these two functions for a simulated signal

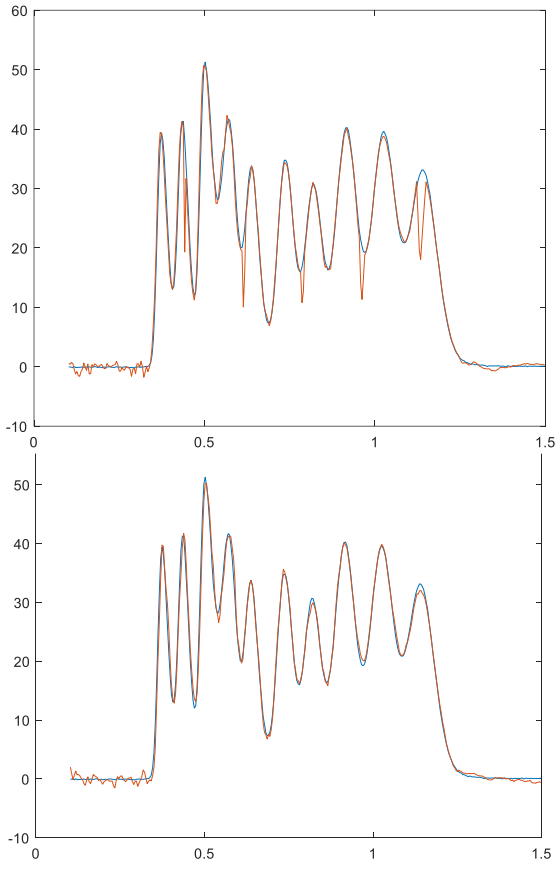
with three noisy peaks whose true parameters are set in lines 13-15. For an "easy" test case, with little peak overlap, both codes work well. But if the peaks overlap significantly, ChatGPT's code fails to yield a good fit but my fitshape2.m code does work. The difference is probably due to the different minimization functions employed (lsqcurvefit.m vs fminsearch.m).



One task that gave the chatbots some trouble was Gaussian Fourier self-deconvolution (page 114). After several attempts at writing prompts, ChatGPT did the job with the prompt:

"*Matlab function that performs Fourier self-deconvolution as a means for sharpening the width of signal peaks in x,y, by computing the Fourier transform of the original signal, dividing that by the Fourier transform of a zero-centered Gaussian function of width deconvwidth, performing an inverse Fourier transform of the result, and applies a smoothing function of width smoothwidth to the deconvoluted result*". (Matlab code. Python code)

Segmented processing functions, treated on page 333, are very useful in cases where the signal varies in character along its length so that a single processing function setting is not optimum. Basically, the idea is that you break up the signal into segments, process each segment with different settings, then reassemble the processed segments. A simple example is segmented smoothing (page 55). In the signal shown in blue, below left, the peaks increase in width as x increases, a smaller smooth width should be applied to the early peaks to prevent peak distortion. The following prompt seemed reasonable:



"Write a function that accepts a signal vector y and a vector s of length n, makes n copies of y, smooths each one with a smooth width equal to the nth element of the vector s, then reassembles the smoothed signal from the nth segment of the nth copy."

But when the resulting function is applied to the signal in blue on the left, the smoothed signal, in red, exhibits serious discontinuities. The problem is caused by the "end effects" of smoothing, described on page 45. I then tried writing a more explicit and lengthy description:

"Write a function that accepts a signal vector y, a number of segments n, an initial smooth width Wstart and a final smooth width of Wend, creates vector S of length n, whose first element is Wstart, last element is Wend, and the other elements are evenly distributed between Wstart and Wend. The function makes n copies of y, smooths each one with a smooth width equal to the nth element of the vector S, then reassembles the smoothed signal from the nth segment of the nth copy". The result is shown on the left.

For more complex functions, it rapidly gets to the point where it's impossible to completely describe the function in enough detail. This is especially true of open-source code that has been developed incrementally over time, with lots of added options and features suggested by many users, such as my 5500-line interactive peak fitter, ipf.m (page 417), or my 4000-line peakfit.m function (page 397) which takes up to 12 input arguments, provides 8 output values, and fits 50 different peak shapes. Even *describing* such functions completely for a chatbot would be tedious at best. It's unrealistic to expect any chatbot to completely replace such efforts.

ChatGPT cannot write Matlab Live Scripts or Apps because they are not text files. (In contrast, chatbots can write GUI applications in Python. See page 454). However, you can easily convert a chatbot-generated Matlab script into a Live Script, as described on page 368, or you can have chatbots write keyboard-controlled interactive functions, like those described on page 397. I tested this idea with a simple pan-zoom-smooth function, using the prompt:

> "*Write a Matlab function that accepts and plots a signal as an x,y pair and has pan and zoom functions that are interactively controlled by the keyboard cursor keys, using the left and right cursor keys to pan back and forth and the up and down keys to zoom in and out. Add a smoothing function that uses the A and Z keys to increase and decrease the smooth width*."

Oddly, ChatGPT and Claude failed with the zoom part of this function, for reasons that are not clear, but Perplexity succeeded. This code could serve as the bones of a customized interactive function with your own additional keypress functions added. This approach, while primitive by GUI standards, is easy to program and can be quite versatile, considering the large number of keys available on a standard keyboard (with and without the Shift key pressed). It also has the advantage that screen space is not occupied by GUI widgets. See page 506 for examples of such keystroke-driven applications.

A lot of thought and experience goes into hand-coded programs. An experienced human coder knows the typical range of applications and anticipates typical problems and limitations, especially as those apply to the specific field of applications, such as signals generated by different types of scientific instrumentation. Of course, AIs do know a great deal about specific computer languages and their capabilities and inbuilt functions, which can be very useful, especially if you are re-coding an algorithm in a new or less familiar computer language. But AIs are no replacement for human experience. It goes without saying that you *must* test any code that a chatbot gives you, just as you must test your own code. And AI is no replacement for *understanding* the basis of these signal processing concepts.

I recommend that you be bold and optimistic when exploring the coding abilities of AIs. *Ask them to do more than you think they should be able to do. You may be surprised*. There is no (or little) cost involved, and AIs are completely non-judgmental and totally tireless. The results you get may not work exactly as you wish, but every failure is a learning opportunity, and you can simply explain the nature of the failure to the AI and it will attempt to correct it. Even if it takes several trials to get it right, the whole process will probably be faster than writing all the code yourself. In addition, you can learn quite a bit by inspecting the code that it generates and the functions and imported modules that it employs. For example, you can use AIs to develop Graphic User Interface (GUI) applications in Python (page 454) or in Mathematica (page 462).

Python: free, open-source and powerful

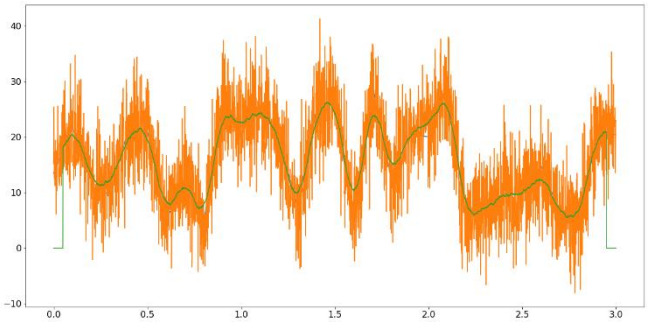
A popular alternative to Matlab for scientific programming is Python, which is a free and open-source language, whereas Matlab is closed, proprietary, and can be costly. The two languages are different in many ways, and an experienced Matlab programmer might have some difficulty converting to Python, and *vice versa*, but chatbots can help you, both in setting up a suitable Python environment and in writing code (page 442). Anything that I have done in this book using Matlab, you can do in Python. I do not intend to provide a complete introduction to Python, any more than I did for Matlab or for spreadsheets, because there are many good online sources. Rather, I will review a few of the crucial computational methods, showing how to do the calculations in Python and compare the code side-by-side to the equivalent calculation in Matlab. In addition, I will compare the execution times of both, running on the same desktop computer system, to see if there is any advantage in execution speed or code size one way or the other. For these tests, I was running Python 3.8.8, Anaconda Individual Edition 2020.11, using the included Spyder 5.0.5 desktop (screen graphic), and Matlab 2021: 9.10.0.1602886 (R2021a), both running on a Dell XPS i7 3.5Ghz tower. Both Matlab and Python/Spyder have an integrated editor, code analysis, debugging, ability to edit variables, etc.

Sliding average signal smoothing

(Based on Nita Ghosh's Scientific data analysis with Python: Part 1)

A "sliding average" smooth is simple to implement using the inbuilt "mean" function in both languages.

Python




```
import numpy as np
import matplotlib.pyplot as plt
from pytictoc import TicToc
t = TicToc()
plt.rcParams["font.size"] = 16
plt.rcParams['figure.figsize'] = (20, 10)
sigRate = 1000 #Hz
time = np.arange(0,3, 1/sigRate)
n = len(time)
p = 30 #poles for random interpolation
ampl = np.interp(np.linspace(0,p,n),np.arange(0,p),np.random.rand(p)*30)
noiseamp = 5
noise = noiseamp*np.random.randn(n)
signal = ampl + noise
t.tic() # start clock
plt.plot(time, ampl)
plt.plot(time, signal)
filtSig = np.zeros(n)
k = 50
for i in range(k,n-k-1):
    filtSig[i] = np.mean(signal[i-k:i+k])
plt.plot(time, filtSig)
t.toc()  # stop clock print time
# Save signal to file
np.savetxt('SlidingAverageSignal.out',(time, signal), delimiter=',')
```

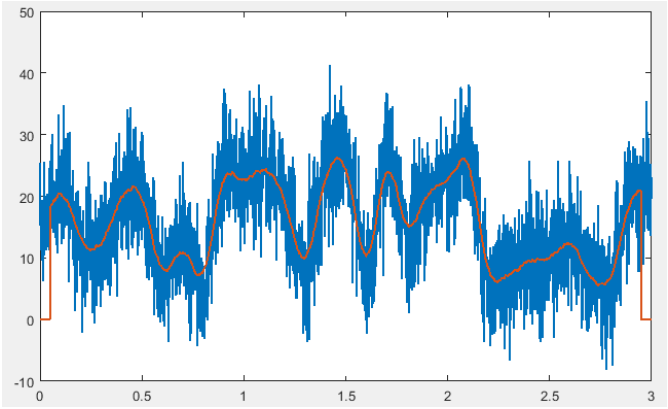
In the first several lines, Python must import some packages that are required for numerical computing. Matlab has much of that built-in, but there are optional add-on "toolboxes" (e.g. page 132). The

Python code creates a simulated noisy "signal" in the first block of code and then uses the inbuilt "savetext" function to save it to disc for Matlab to read, ensuring that *the exact same signal is used for both*.

Matlab

```
clf
load SlidingAverageSignal.out
x=SlidingAverageSignal(1,:);
y=SlidingAverageSignal(2,:);

tic % start clock
lx=length(x);
filtsig=zeros(1,lx);
k=50; % smooth width
for i=k:lx-k-1
    filtsig(i)=mean(y(i-k+1:i +k));
end
plot(x,y,'g',x,filtsig,'linewidth',1.5)
toc % stop clock and print elapsed time
```



The fundamental thing here is the use of the "mean" function in both languages to compute the mean of *k* adjacent points for the smoothed signal. In Python this is done by the lines:

```
k = 50
for i in range(k,n-k-1):
    filtSig[i] = np.mean(signal[i-k:i+k])
```

and in Matlab by:

```
k=50;
for i=k:lx-k-1
    filtsig(i)=mean(y(i-k+1:i +k));
end
```

You can see how similar they are. (I've used the same variable names to make the comparison easier). A key difference is how a block of code is specified, such as a *loop* or a *function definition*. In Python, this is done using *indentation*, which is *rigorously enforced*. In Matlab, it is done using "end" statements. Indentation is optional in Matlab but, as a matter of good coding style, is easily done by the "Smart Indent" in the right-click menu. (There is an example of function definition in the iterative least-squares example on page 451). Another important difference: Python uses square brackets to enclose the index to arrays rather that parentheses as does Matlab. Python arrays are indexed from *zero*; Matlab's are indexed from 1. So, for example, the first two elements of an array A in Python are A[0] and A[1] but in in Matlab they are A(1) and A(2). Little things mean a lot.

To compare the execution times, I have used the "tic" and "toc" statements (inbuilt in Matlab but added to Python by the TicToc package) to start and stop a timer. I have **bold-faced** those lines in the code, to make it clear that I am counting and timing only the "inner" portion of the code that will have to be repeated if you have multiple data sets to process. I don't count the time required by the initial set-up, loading of packages, etc. – things that need be done only once.

Python: 8 lines; 0.04 – 0.08 sec
Matlab: 7 lines: 0.007 – 0.008 sec

Both programs are about the same length, but Matlab is clearly faster in this case.

Fourier transform and (de)convolution

The Fourier transform (FT) is fundamental for computing frequency spectra, convolutions, and deconvolution. The codes here simply create a vector "a" of random numbers, compute the FT, multiply it by itself, and then inverse FT the result, both using the fft and ifft functions. This is basically a Fourier convolution. Deconvolution would be the same except that the two Fourier transforms would be *divided*. As before, **tic** and **toc** mark the timed blocks.

Python:

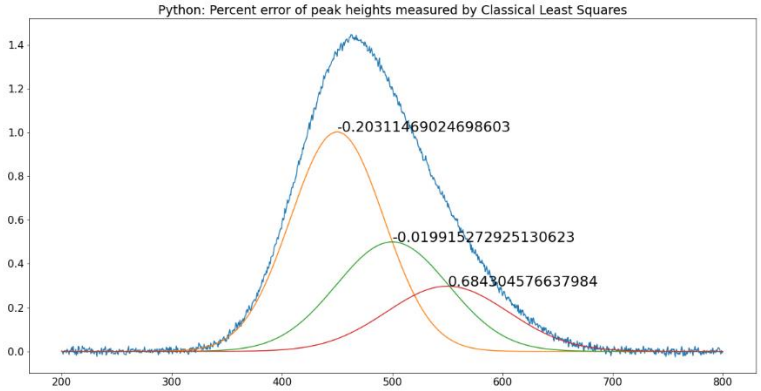
```
import numpy as np
from scipy import fft
from pytictoc import TicToc
t = TicToc()
t.tic() # start clock
min_len = 93059  # prime length is the worst case for speed
a = np.random.randn(min_len)
b = fft.fft(a)
c=b*b
d=fft.ifft(c)
t.toc()  # stop clock and print elapsed time
```

Matlab:

```
tic
MinLen = 93059; % prime length is the worst case for speed
a=randn(1,MinLen);
b = fft.fft(a)
c=b.*b;
d=ifft(c);
toc
```

Python: 5 lines; 0.01 – 0.04 sec (The execution time apparently varies with different random number sequences).
Matlab: 5 lines: 0.008 – 0.009 sec

Classical Least Squares



The Classical Least Squares (CLS) technique has long been used in spectroscopic analysis of mixtures, where the spectra of the individual components are known but which overlap additively in mixtures (page 188). A comparison of Python and Matlab coding for this method is given on page 198. (I wrote the Matlab code first, copied and pasted it into the Spyder editor, and converted it line by line). After the required importation of required Python packages, the codes (NormalEquationDemo.py and NormalEquationDemo.m) are remarkably similar, differing mainly in the use of indentation, the way functions are defined, the way arrays are indexed, the way vectors are concatenated into matrixes, and the coding of matrix transpose, exponentiation and dot products.

Python: 50 lines; 0.017 sec. Matlab: 41 lines. 0.018 sec.

Iterative least-squares fitting

(Based on Chris Ostrouchov's code on https://chrisostrouchov.com/post/peak_fit_xrd_python/)

Python:

```
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize
from pytictoc import TicToc
t = TicToc()
def g(x, A, μ, σ):
    return A / (σ * math.sqrt(2 * math.pi)) * np.exp(-(x-μ)**2 / (2*σ**2))
def f(x):
    return np.exp(-(x-2)**2) + np.exp(-(x-6)**2/10) + 1/(x**2 + 1)
A = 100.0 # intensity
μ = 4.0  # mean
σ = 4.0 # peak width
n = 500 # Number of data points in signal
x = np.linspace(-10, 10, n)
y = g(x, A, μ, σ) + np.random.randn(n)

t.tic() # start clock
def cost(parameters):
    g_0 = parameters[:3]
    g_1 = parameters[3:6]
    return np.sum(np.power(g(x, *g_0) + g(x, *g_1) - y, 2)) / len(x)
initial_guess = [5, 10, 4, -5, 10, 4]
result = optimize.minimize(cost, initial_guess)
g_0 = [250.0, 4.0, 5.0]
g_1 = [20.0, -5.0, 1.0]
x = np.linspace(-10, 10, n)
y = g(x, *g_0) + g(x, *g_1) + np.random.randn(n)
fig, ax = plt.subplots()
ax.scatter(x, y, s=1)
ax.plot(x, g(x, *g_0))
ax.plot(x, g(x, *g_1))
ax.plot(x, g(x,*g_0) + g(x,*g_1))
ax.plot(x, y)
t.toc()  # stop clock and print elapsed time

np.savetxt('SavedFromPython.out', (x,y), delimiter=',') # Save signal to file
```

Iterative curve fitting is more complex than the previous examples. Both Python and Matlab have optimization functions ("optimize.minimize" in the Python code, above. The inbuilt "fminsearch" in the Matlab code, below). The optimize function in Python is more flexible and can use several different optimization methods. Matlab uses the Nelder-Mead method, which is used below (see page 204), but there is also an optional optimization toolbox that provides alternative methods if needed.

```
clear
clf
load SavedFromPython.out
x=SavedFromPython(1,:);
y=SavedFromPython(2,:);
start=[-5 3 6 3];
format compact
global PEAKHEIGHTS
tic
FitResults=fminsearch(@(lambda)(fitfunction(lambda,x,y)),start);
```

```
NumPeaks=round(length(start)./2);
for m=1:NumPeaks
  A(m,:)=shapefunction(x,FitResults(2*m-1),FitResults(2*m));
end
model=PEAKHEIGHTS'*A;
plot(x,y,'- r',x,model)
hold on;
for m=1:NumPeaks
    plot(x,A(m,:)*PEAKHEIGHTS(m));
end
hold off
toc % stop clock and print elapsed time
function err = fitfunction(lambda,t,y)
global PEAKHEIGHTS
A = zeros(length(t),round(length(lambda)/2));
  for j = 1:length(lambda)/2
     A(:,j) = shapefunction(t,lambda(2*j-1),lambda(2*j))';
  end
PEAKHEIGHTS = A\y';
z = A*PEAKHEIGHTS;
err = norm(z-y');
end
function g = shapefunction(x,a,b)
g = exp(-((x-a)./(0.6005612.*b)) .^2); % Expression for peak shape
end
```

Python: 16 lines; 0.02-0.08 sec
Matlab: 25 lines; 0.01-0.03 sec

The Matlab algorithm here is the same as used by my peakfit.m (page 396) and ipf.m (page 417) functions. The elapsed times of both Matlab and Python vary with the random noise sample, but in this example, Matlab has an advantage in computational speed, which might be significant some applications.

An excellent step-by-step introduction to iterative curve fitting in Python is by Emily Grace Ripka.

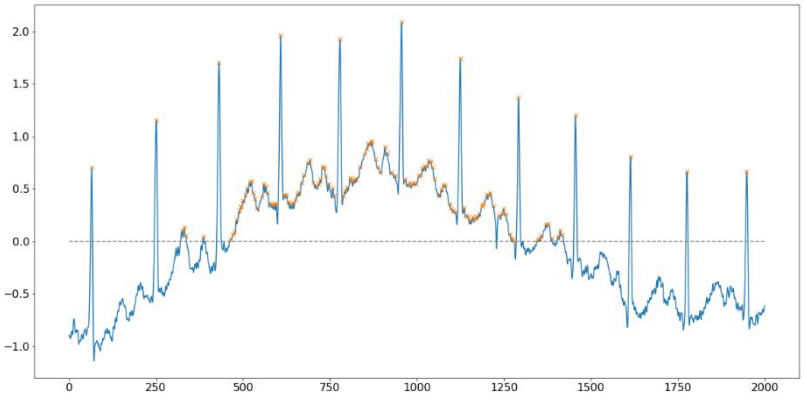
Peak Detection

The automatic detection of peaks is a common requirement. The coding required is a little more complex. The codes are based on the Python example in the SciPy documentation: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.find_peaks.html. Both languages have a "find peaks" function, but Matlab's function is rather slow, so I prefer to code my own algorithm in Matlab using nested loops (each with an "end" statement but also indented for clarity). The signal used here is a portion of an electrocardiogram that is included as a sample signal in Python. The task is to detect the peaks that exceed a specified amplitude threshold and mark them with a red "x" on the plot.

Python:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.misc import electrocardiogram
from scipy.signal import find_peaks
from pytictoc import TicToc
t = TicToc()
x = electrocardiogram()[2000:4000]
t.tic() # start clock
```

```
peaks, _ = find_peaks(x, height=0)
plt.plot(x)
plt.plot(peaks, x[peaks], "x")
plt.plot(np.zeros_like(x), "--", color="gray")
plt.show()
t.toc()  # stop clock and print elapsed time
np.savetxt('FindPeaks.out',(x), delimiter=',')
```



Matlab:

```
load electrocardiogram.out;
y=electrocardiogram;
tic % start clock
  plot(1:length(y),y,'-k')
  height=0;
  x=1:length(y);
  peak=0;
  for k = 2:length(x)-1
      if y(k) > y(k-1)
          if y(k) > y(k+1)
              if y(k)>height
                  peak = peak + 1;
                  P(peak,:)=[x(k) y(k)];
              end
          end
      end
  end
  hold on
  for k=1:length(P)
      text(P(k,1)-12,P(k,2),'x','color',[1 0 0])
  end
  grid
  hold off
toc % stop clock and print elapsed time
```

Python: 5 lines; 0.01 sec
Matlab: 19 lines: 0.007 sec `as written` (1.5 sec using the Matlab inbuilt findpeaks.m function)
Here, the advantage goes to Python, because its inbuilt "findpeaks" function is superior to Matlab's.

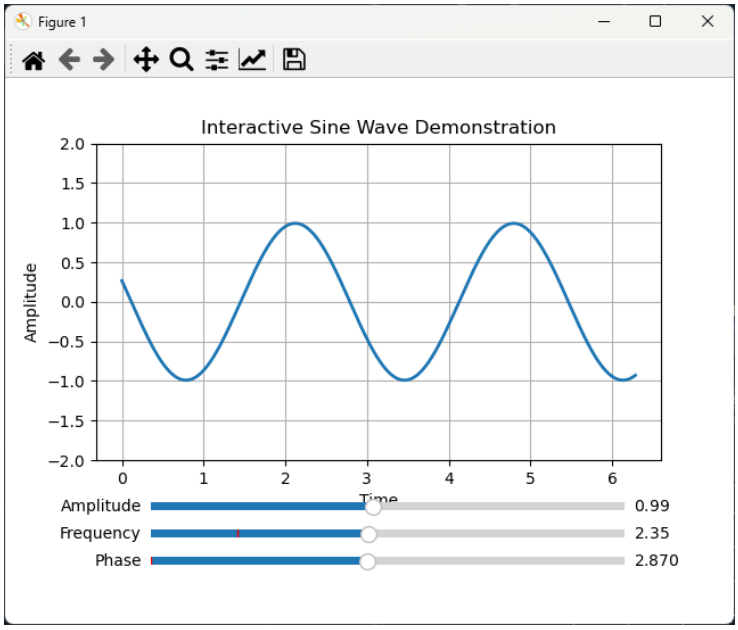
The bottom line

On average, Matlab is slightly faster for most of these signal processing applications, but that advantage could easily be outweighed by the fact that Python is free and open-source, which is a strong argument in its favor. (And Python is certainly faster than the free Matlab clone Octave). I would recommend this: if you have a background in computer science or have taken some courses, then you will probably prefer Python. Others might find Matlab easier to work with. I like both languages, but as a scientist I have mainly used Matlab, because it has so many well-developed toolboxes and because it acts and feels more finished and polished than either Anaconda/Python/Spyder or the Matlab clone Octave, in my opinion. After all, *Mathworks Inc* has remained in business selling the very costly Matlab, while Python is free. There must be a reason. (As of 2023, over 2,200 universities worldwide have a Campus-Wide License for MATLAB and Simulink, as my campus thankfully does).

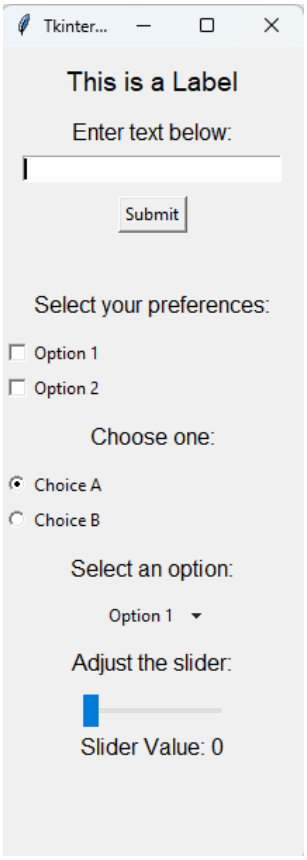
Using AI to develop Graphic User Interface applications in Python.

While Matlab has a GUI development environment for Matlab Apps (page 370), the process is complex and, because it is not text-based, cannot be coded by a chatbot. Python has several libraries for creating GUI apps for various purposes, but again the details can be complex and it is arguably easier to *describe* the desired functionality to a chatbot and to have it write the code and handle all the messy details. The fully functional example on the right was created by this remarkably short prompt:

> *"Write a python script that creates an interactive demonstration of the properties of a sine wave, with three sliders to demonstrate the amplitude, frequency, and phase".*



The Python code is 58 lines long. In this case I allowed ChatGPT to choose the GUI library and to design the layout. It chose "matplotlib", a popular Python library for creating static, animated, and interactive visualizations that is widely used in data analysis and scientific computing.



But matplotlib is not without problems. Its "widgets" (sliders, radio buttons, etc.) can sometimes be finicky in certain environments, especially when running inside integrated development environments like Spyder. Fortunately, Python has more robust GUI libraries like PyQt5 and Tkinter, which are a better solution for creating interactive applications.

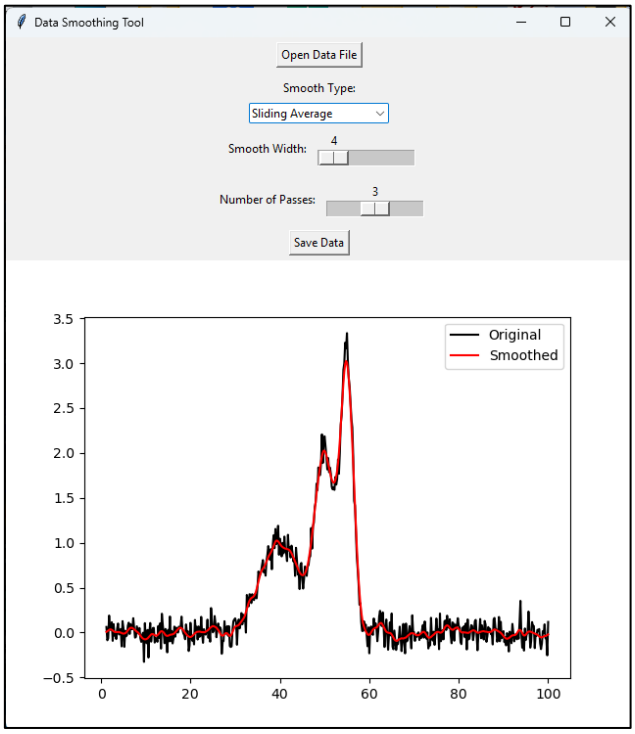
I asked ChatGPT to "create a simple demonstration of the main Tkinter widgets. The result is shown on the left, including:

1. **Label**: Displays static text.
2. **Entry**: A single-line text box.
3. **Button**: Performs an action when clicked.
4. **Checkbox**: Allows multiple selections.
5. **Radio Button**: Allows a single selection from multiple options.
6. **Dropdown Menu**: Lets the user choose from a predefined list.
7. **Slider**: Adjusts the value of a variable by sliding.
8. **Dynamic Updates**: Text and slider values are updated interactively.

This, plus its ability to handle mouse motions and buttons, seems to cover most of the GUI widgets one is likely to need for signal processing applications.

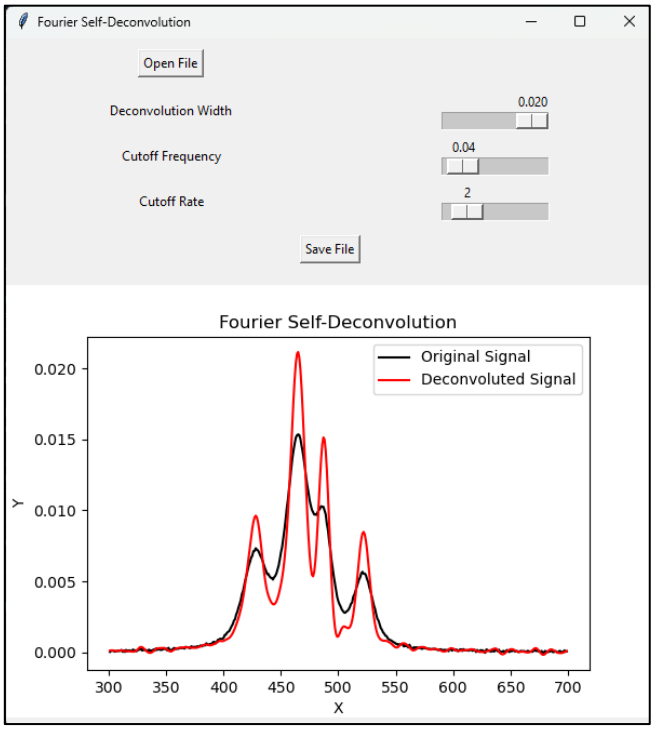
For a more practical signal processing application, I asked ChatGPT to create a Python GUI data smoothing tool that:

"… has an Open File button that opens a file browser, allowing you to navigate to your data file (in .csv or .xlsx format. Assume that the x,y data are in the first two columns) and a Save File button for saving the smoothed result. Plot the original signal in black and the smoothed result in red. Add a SmoothType drop-down menu that selects the smoothing algorithm. The first choice in that menu is a multi-pass sliding average smooth. Add two sliders, the first that controls the smooth width (from 3 points to 10% of the number of points in the data) and the second that controls the number of passes of the sliding average (1-5 passes). The second item in the SmoothType menu should be the Savitsky-Golay smooth, with two sliders to control the frame length and polynomial order of the smooth. Allow the user to zoom in and out to inspect details the signals using the mouse wheel to expand or contract the x-axis range. Use the mouse left-button drag to pan back and forth. Update the plot for each change in any of the controls."



It took only a minute for the AI to generate this 198-line solution, quicker that most people can *type*. The Python code uses the Tkinter, matplotlib, pandas, and scipy libraries. It runs without trouble in the Spyder IDE or directly from the Anaconda PowerShell prompt window. Even if it does not look or behave like you wanted, you can just ask to add or change those aspects and the AI will do it.

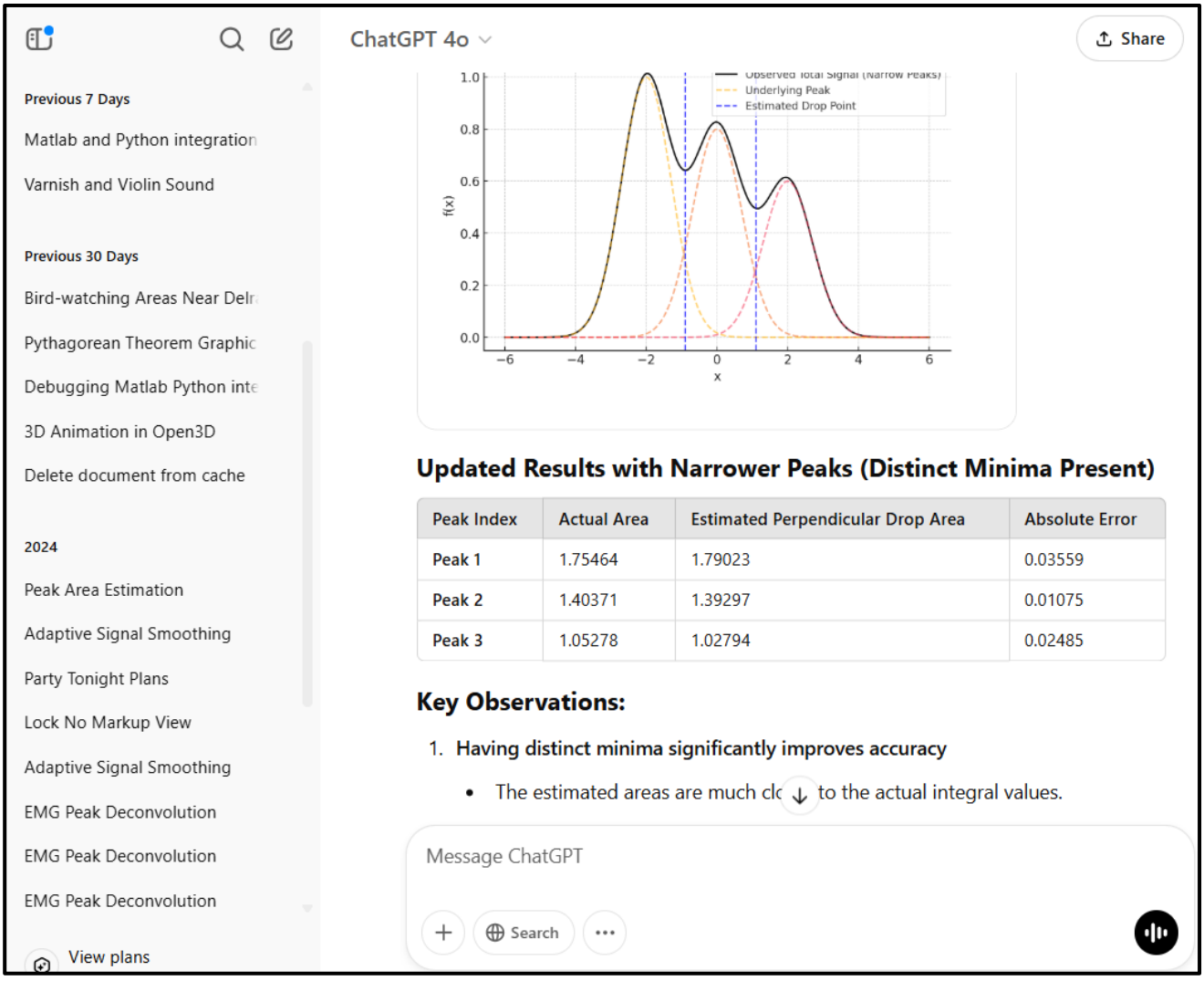
Clearly this could be expanded as needed for many signal-processing purposes, along the lines of the Matlab Live Scripts described on page 369.
The example on the right below is an AI-written tool for Gaussian Fourier self-deconvolution (page 117), with sliders to control the width of the Gaussian deconvolution function and the cutoff frequency and rate of the Fourier filter. 98-line Python script. Data file. Python's GUI apps like this are more versatile and are better at isolating the user interface from the code. (Matlab Live Scripts embed the GUI controllers directly in the code, which is very convenient for rapid coding and prototyping, but it also makes the code vulnerable to accidental modification if the scripts are used by inexperienced users).



This only scratches the surface of what AI coding can do. My advice to you is to be ambitious and ask AI to do more than you think it can do. You might be surprised.

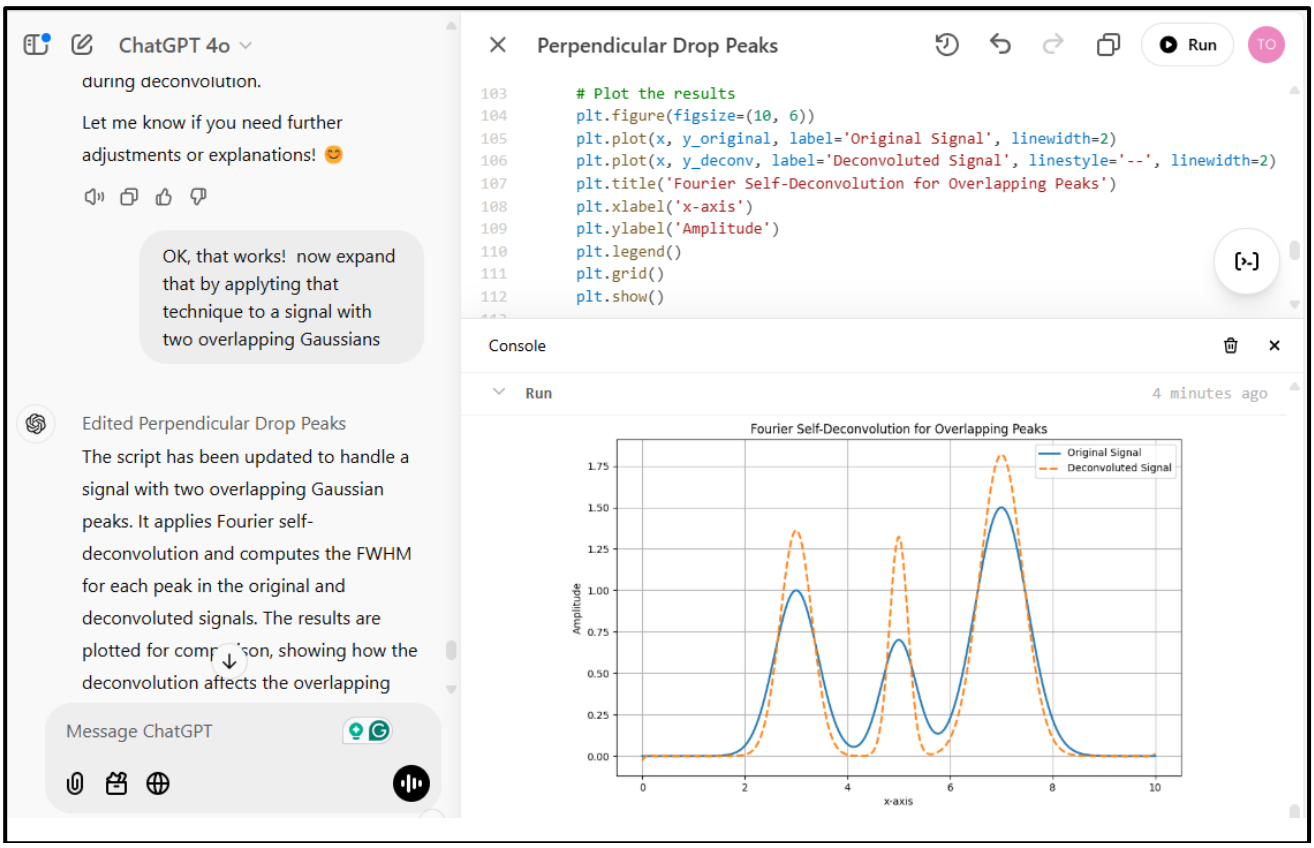
ChatGPT + Python

When you are working with coding, specifically Python coding, ChatGPT has two modes of operation: **Chat mode** and **Canvas mode**, which is optimized for code generation and testing. Chat Mode is the default. In that mode everything stays in a single conversation flow in a large browser window, with code, discussions, tables, and graphic visualizations. You can expand or download the graphics or change the colors of the lines interactively. (As always, a scrolling list of other chats is on the left). This mode is good for iteratively testing small changes and for back-and-forth problem-solving. If requested, it will display the code it uses and you can copy and paste that into Spyder or into a Note and save it as a ".py" file and run it from the command window. (ChatGPT can write code in Matlab as well but cannot execute it. You must copy and paste the code into Matlab's editor window).



In the example shown above, I asked ChatGPT to "prepare a graphic example of the perpendicular drop method of estimating the areas of overlapping peaks". (Interestingly, it "cheated" in its first attempt and used its prior knowledge of the individual peak parameters. Once I pointed that out, it used a properly data-driven approach. Artificial intelligence only goes so far. *Always be skeptical*). You can ask for other changes, such as different peak shapes, heights, and widths, greater or less peak separation, noise added to the original signal, or you can ask it to display the peak area errors as percentages in the table.

In Canvas Mode the code is kept in a dedicated editor window where you collaboratively edit code or text files. It provides versioning options, such as "Show changes" and "Previous version", so you can track edits and revert to earlier versions of code if needed. You can make changes, ask questions, and request modifications, all while seeing the current document content in real time. You can return the Chat Mode simply by refreshing the browser.



The Canvas mode is shown here by the example of Fourier self-deconvolution. The left-hand panel shows a portion of the chat conversation. When you click the Opens Canvas button in the upper right of the browser window, it displays the code that it creates in that panel. You can ask for any type of calculation, simulation, with tabular or graphical output. The text and graphic outputs are displayed in the lower right Console. You can ask ChatGPT for additions or modifications and the code will be corrected. Buttons allow you to revert to the previous version, fix bugs, copy the code for pasting elsewhere, port to C++, Java, JavaScript, TypeScript, etc. If an error pops up, click on it to have the AI attempt a correction. (Unfortunately, within the Canvas environment, you cannot directly execute code that requires external libraries like Tkinter to install or set up, but you can copy and paste the code into Python's editor).

Formats supported by Canvas include Python, C/C, Java , JavaScript, HTML/CSS, JSON/XML, and Markdown. See https://openai.com/index/introducing-canvas/

Chatbots are developing rapidly as I write this in late 2025. By the time you read this, they will certainly have added new features and capabilities.

Connecting Matlab and Python

Up to now, I have considered Matlab and Python as separate islands, but here you will learn how to connect them, enabling the integration of Matlab's powerful numerical capabilities with Python's vast libraries and flexibility. This can be useful in both directions. For example, from Matlab you can call those Python GUI applications that were so easily developed using ChatGPT to write the code (page 454). And from Python, you can call any of the Matlab functions in the Matlab File Exchange or even those that I have developed (page 479), such as the peak fitting functions peakfit.m (page 397) and its interactive cousin ipf.m (page 417). The following step-by-step guide, tailored for MATLAB users, shows you how to set up Python, install the MATLAB Engine for Python, and use Python to interact with MATLAB. I assume that you have a recent version of Matlab already installed. Note: if you have worked with Python before, it might be advisable to create a virtual environment for experimentation.

First, for this purpose, Matlab works only with Python 3.11. Visit https://www.python.org/downloads/ and download Python 3.11. During installation, check the box **Add Python to PATH**, select **Customize Installation** and *ensure that all features are selected*. To verify Installation, open a command prompt or terminal (Type CMD in the search at the bottom) and type:

```
python --version
```

Confirm that the output displays Python 3.11.x.

Installing the MATLAB Engine for Python

Open a Command Prompt or terminal. Change to the MATLAB Engine installation directory:

```
cd "C:\Program Files\MATLAB\R2024b\extern\engines\python"
```

Run the following command to install the engine:

```
python -m pip install .
```

*Caution: You **must** include the **space** and the **period** at the end: they are important.*

Open the command window and type :

```
python
```

The prompt will change to >>>, indicating that you are in a Python session. Type in these commands into the Notepad and save them as a ".py" file and run it from the command window.

```
import matlab.engine
eng = matlab.engine.start_matlab()
print("MATLAB is running!")
eng.quit()
```

If MATLAB starts successfully and displays "`MATLAB is running!`", the installation is complete.

Examples of calling MATLAB from Python

Here's a simple example that calls Matlab's built-in square root function (sqrt) from Python. In the command window, type `python` and execute the following code:

```
eng = matlab.engine.start_matlab()
result = eng.sqrt(16.0)
print("Square root:", result)
eng.quit()
```

The result is printed out in the command window.

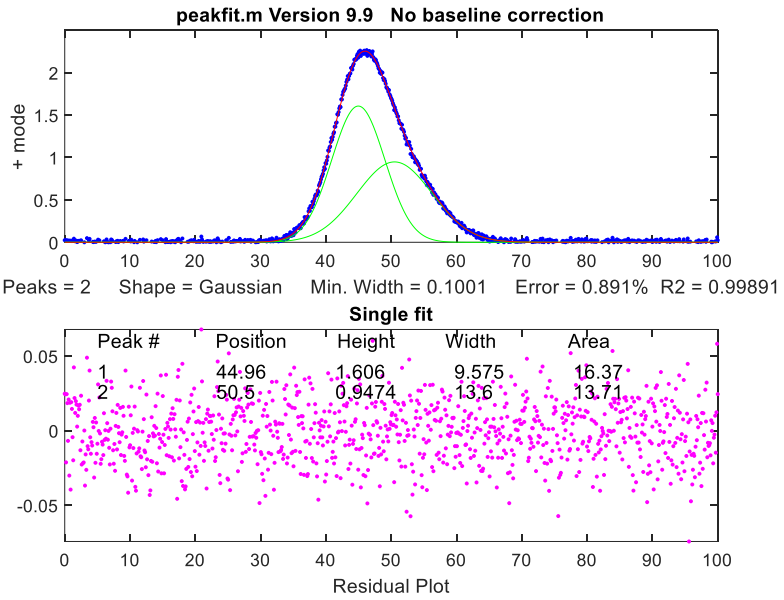
```
Square root: 4.0
```

(The slight delay before printing out the result is normal). You can also execute previously-developed MATLAB functions saved as ".m" files directly from Python. As a simple example, suppose you have defined a Matlab function called "magnitude" and saved it in the current path:

```
function result=magnitude(a,b)
   result=sqrt(a^2+b^2);
end
```

Now you just replace `result=eng.sqrt(16.0)` in the previous example with `result=eng.magnitude(3, 5)`. Run the modified script in Python and the results will be printed out as before: 5.8310.

Of course, you can call functions that are much more useful than those trivial examples. Here are two examples. The first calls my 4000-line Matlab peakfit.m function (page 397). The Python script is CallpeakfitDemo.py, located in the path: **C:\Users\Name\Dropbox\SPECTRUM**



where "Name" is your username. The script creates a test signal consisting of two over-lapping Gaussian bands, adds some noise, calls my peakfit.m function, passes the x,y signal vectors, the number of peaks and the peak shape to that function, which performs an iterative least-squares fit, prints out the best-fit results in the command window, and displays the Matlab figure in a separate window. (The figure is a standard Matlab figure window, which means that it can be resized, the labels can be edited, and it can be saved in multiple formats).

There are two ways to call this function from the command window. You can change the current path:

```
C:\Users\Name\Dropbox\SPECTRUM>python CallpeakfitDemo.py
```

or specify the path when you call the function:

```
C:\Users\Name>python C:\Users\Name\FunctionsDirectory\CallpeakfitDemo.py
```

In addition to the graph shown above, the script also prints out the fit results and "goodness of fit" measures in tabular form in the command window:

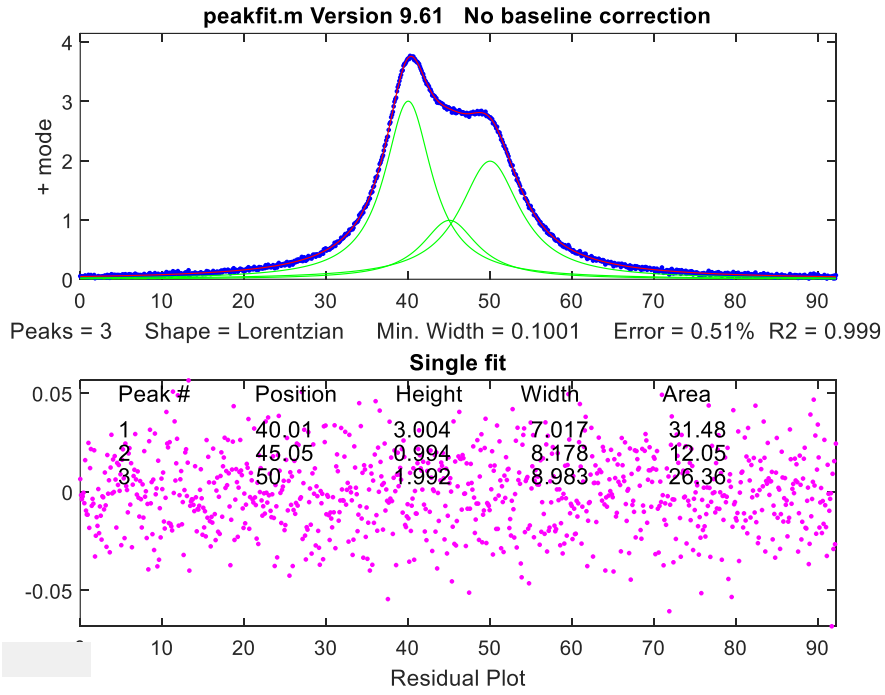
```
Peak #   Position     Height       Width        Area
-------------------------------------------------------
1        44.96        1.606        9.575        16.37
2        50.5         0.9474       13.6         13.71

Goodness of Fit:
% Fitting Error   R-squared
-----------------------------
0.8915            0.9989
```

(Note: there will be a delay of about 2 seconds before the graph and printout is displayed).

The second example is more interactive: CallipfDemo.py, which is also in the same directory. This script creates a test signal consisting of three overlapping Lorentzian bands, adds some noise, and passes the data to ipf.m, which is my keypress-operated interactive curve fitting explorer. You call it in the same way as above. (`C:\Users\Name\Wherever` is the path to CallipfDemo.py on your computer).

```
C:\Users\Name\Wherever>python CallifpDemo.py
```



Again, this figure is a standard Matlab figure window, with all its properties. Click on the figure window and use the keypresses to pan and zoom (using the cursor keys), select the desired peak shape (G=Gaussian, L=Lorentzian, etc.), the number of peaks in the model (the number keys), the first-guess start positions (press C, then click on the graph for each peak start position), etc. Other keypress commands are documented on page 419.

The same technique can be used for my other Matlab keypress operated functions. Just replace ipf.m with iSignal, isignal.m, page 376, for interactive signal exploration, iPeak, ipeak.m, page 255 for interactive peak detection or iFilter, ifilter.m, page 391 for interactive Fourier filtering. (Note: You can download the entire SPECTRUM folder, which contains all those Matlab functions and more, with their documentation, from this link).

Calling Python from Matlab is done with the "system" command. From the Matlab command line:

```
system('C:\Users\YourUserName\YourScript.py')
```

For example, you could use this to call any of the Python GUI modules developed on page 454 such as sine.py, DataSmoothingGUI2.py, or GausSelfDconv.py.

Connecting GNU Octave and Python

Matlab is very expensive to purchase, and if your campus or workplace does not have a site license for it, you might be interested in a less expensive route. GNU Octave (page 22) is a possible alternative. It is almost completely compatible with Matlab and it is a free open-source product. The Python library **oct2py** provides a high-level Python interface to Octave, allowing you to run Octave commands from Python, pass data back and forth, and call Octave functions. In the Command Prompt window:

```
pip install oct2py
```

Then type :

```
   python -Xfrozen_modules=off
```

The prompt will change to **>>>**, indicating that you are in a Python session.

```
>>> import oct2py
>>> oc = oct2py.Oct2Py()
```

Next, try displaying the sum of 2+2:

```
>>> print(oc.eval('disp(2+2)'))
```

Python responds with:

```
4
```

Next, change the path to the location of your Octave functions:

```
>>> oc.eval('addpath(\"C:/Users/UserName/directory\");disp(\"\")')
```

Where "directory" is the name of the folder where the Octave functions that you have written are stored. In the following example, the custom function "magnitude" simply computes the square root of the sum of the squares of two numbers:

```
>>> print(oc.magnitude(3,5))

5.830951894845301
```

So, all you must do is to add the "oc." in front of the name of any Matlab or Octave function that is in that path.

Calling Python from Octave

This works exactly the same as in Matlab, using the "system" function. From the Octave command line:

```
system('python myscript.py')
```

For example, you could call the GUI functions sine.py, DataSmoothingGUI2.py, or GausSelfDconv.py.

To capture the output of a function:

```
[status, cmdout] = system('python myscript.py'); disp(cmdout);
```

Mathematica in Signal Processing – A Comparison with MATLAB and Python

Mathematica is a symbolic-numeric computing platform with a tightly integrated notebook interface and its own high-level language. While Matlab and Python are widely used in engineering and scientific computing, Mathematica is less widely used in those domains but it brings unique capabilities, particularly in symbolic computation, rule-based programming, and dynamic interactivity. It was developed by visionary British computer scientist Stephen Wolfram.

Comparison Table

Feature	Mathematica	MATLAB	Python
Language	Wolfram Language (symbolic first)	MATLAB (imperative, array-oriented)	Python with NumPy/SciPy
Symbolic Computation	✅ Native & deep symbolic capabilities	❌ Limited (Symbolic Math Toolbox)	⚠️ SymPy (less integrated)
Numerical Computation	✅ Built-in	✅ Built-in	✅ NumPy/SciPy
Plotting & Visualization	✅ High-level, publication-ready	✅ Strong 2D/3D plotting	✅ Matplotlib, Plotly, etc.
Interactivity	✅ Built-in (Manipulate[])	⚠️ Via GUI tools	⚠️ Via Tkinter/PyQt/etc.
Notebook Interface	✅ Native .nb	✅ Live Scripts (.mlx)	✅ Jupyter Notebooks
Cost / Licensing	Commercial (academic available)	Commercial (academic available)	Open-source (mostly free)
Signal Processing Tools	⚠️ Smaller, symbolic-oriented	✅ Extensive toolboxes	✅ SciPy, PyWavelets
Learning Curve	Steep for programming, shallow for math	Moderate (especially for engineers)	Moderate–steep (depends on libraries)
LaTeX / Docs Integration	✅ Native typesetting	✅ Good formatting	⚠️ Requires plugins

Like Matlab and Python, AI chatbots can be used to create Wolfram code, as illustrated in example 2 on the following page.

Unique Strengths of Mathematica for Signal Processing

Signals may be captured live from sensors, instruments and data feeds, stored in files and databases and generated from simulated models and processes.

1. Notebook format

Mathematica integrates code and graphics into one document (simple example: code, graphic). Here's a more elaborate example, demonstrating second-derivative peak sharpening (page 79).

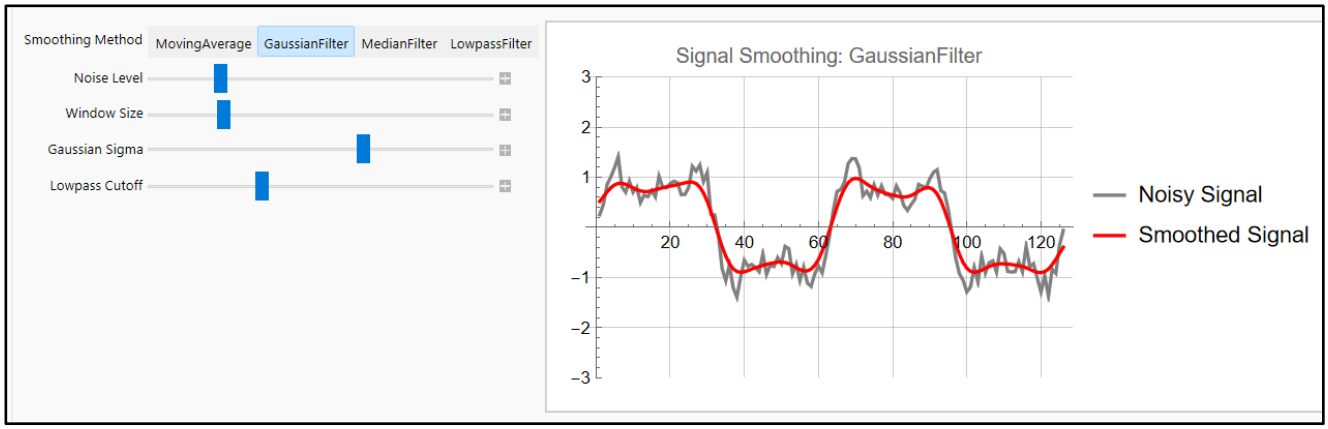
2. Symbolic Signal Models

Mathematica excels at creating exact symbolic expressions for transforms and filters. For example:

```
FourierTransform[Exp[-a t] UnitStep[t], t, ω]
```

3. Simple dynamic Interactivity:

Mathematica's clever Manipulate[] function allows you to create real-time demos with sliders, checkboxes, and controls. Here's an example: a demonstration of different kinds of data smoothing filters. The Wolfram code, the graph, and the interactive controls are all contained within one Mathematica document, similar to a *Matlab Live Script* (page 369). (Click the "Enable Dynamics" button on the upper right of the Mathematica notebook).



4. Specify symbolic forms in plain text format and display in traditional form:

```
In[9]:=
TraditionalForm[
 Defer[F[ω] == Integrate[f[t] Exp[I ω t], {t, -∞, ∞}]]]
```

Out[9]//TraditionalForm=

$$F(\omega) = \int_{-\infty}^{\infty} f(t) \exp(i\, t\, \omega)\, dt$$

Which is the symbolic form of the Fourier transform.

5. Symbolic + Numeric Fusion:

You can derive a formula analytically and then apply it numerically. For example, you can derive a symbolic derivative like so:

```
In[27]:= D[Sin[a x]/x, {x, 1}]
```

Out[27]=

$$\frac{a \cos[a x]}{x} - \frac{\sin[a x]}{x^2}$$

In this code, **D** is the symbolic derivative: [D[expr, {x, n}] returns the nth derivative of the expression expr with respect to x). Then you can evaluate that derivative at specific numerical values:

```
In[28]:= N[D[Sin[a x]/x, x] /. {a -> 2.3, x -> 1.2} ]

In[29]:= -2.03742
```

N converts exact/symbolic results to numeric ones.

6. Unified Language for Math, Graphics, and GUIs:

Mathematica uses one coherent language for symbolic computation, numerical analysis, plotting, and GUI development.

Limitations in Signal Processing Use

• There are fewer built-in signal processing functions compared to MATLAB's Signal Processing Toolbox or Python's SciPy.
• There is less widespread use in engineering practice, so fewer textbooks and examples.
• It has a steeper learning curve for procedural programming tasks.
• I have found that data importing in Mathematica is less robust than Matlab's Import Wizard or Python's Pandas data analysis package.

Suggested Use Cases in Signal Processing

• Symbolic manipulation of transfer functions
• Interactive educational tools and demonstrations
• Visualizing filter behavior and transforming domains
• Hybrid analytical/numerical derivations
• Related examples: Signal processing, Filter design, Signal analysis, Live playground.

Mathematica may not be the first tool of choice for signal processing implementation, but it excels in theory, visualization, and interactivity. For readers already familiar with MATLAB or Python, Mathematica offers an opportunity to approach signal processing from a more symbolic and exploratory perspective, especially when clarity, derivation, or publication-quality graphics are the goals (true vector graphics, consistent styling via options, precise typography & math, color-blind-safe palettes, and high-quality export).

Note: this section was written with the help of ChatGPT5.

Worksheets for Analytical Calibration Curves

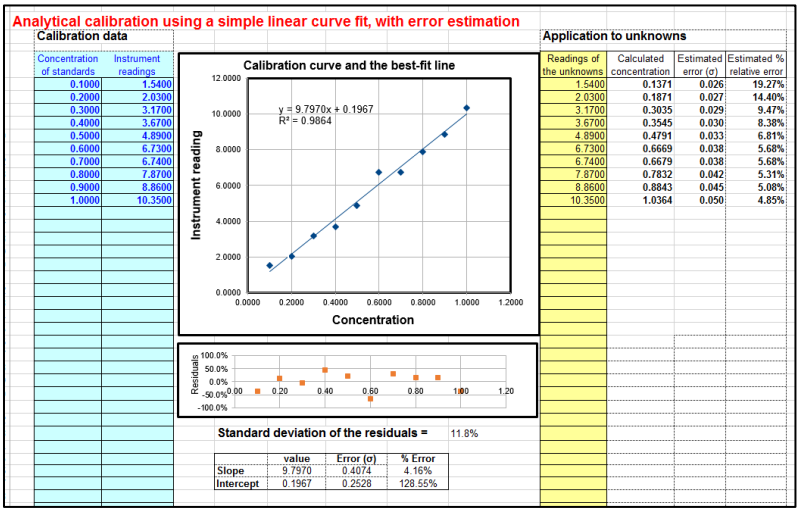
Background

In analytical chemistry, the accurate quantitative measurement of the composition of samples, for example by various types of spectroscopies, usually requires that the method be calibrated using standard samples of known composition. This is most commonly, but not necessarily, done with solution samples and standards dissolved in a suitable solvent, because of the ease of preparing and diluting accurate and homogeneous mixtures of samples and standards in solution form. In the calibration curve method (page 337), a series of external standard solutions with different concentrations is prepared and measured. A least-squares calculation (page 162) fits a line or curve to the data and the resulting equation is used to convert readings of the unknown samples into concentration. The advantages of this method are that (a) the random errors in preparing and reading the standard solutions are averaged over several standards, and (b) non-linearity in the calibration curve can be detected and can be avoided (by diluting into the linear range) or compensated (by using non-linear curve fitting methods).

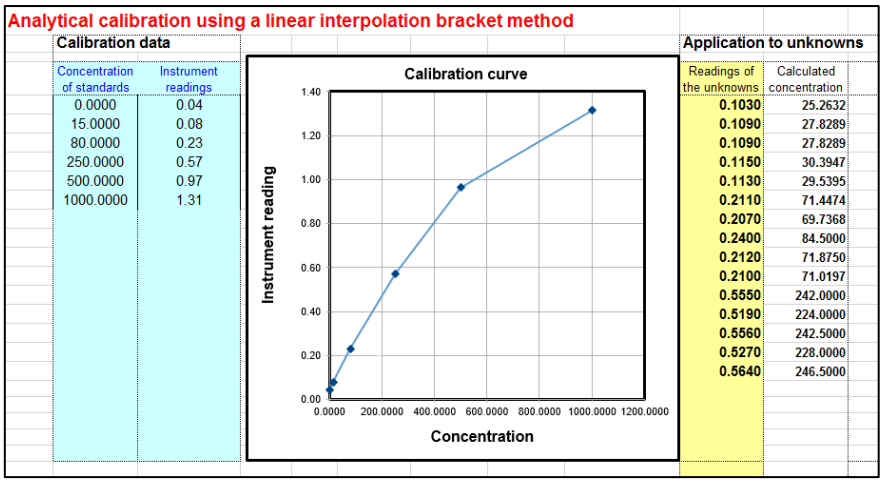
Below are seven different fill-in-the-blanks spreadsheet templates for performing the calibration curve fitting and concentration calculations for analytical methods using the calibration curve. All you need to do is to type in (or paste in) the concentrations of the standard solutions and their instrument readings (e.g., absorbances, intensities, or whatever method you are using) and the instrument readings of the samples to be measured ("unknowns"). The spreadsheet automatically plots and fits the data (page 161), then uses the equation of that curve to convert the readings of the unknown samples into concentration. You can add and delete calibration points at will, to correct errors or to remove outliers. The sheet re-plots and recalculates automatically.

Fill-in-the-blanks worksheets for several different calibration methods

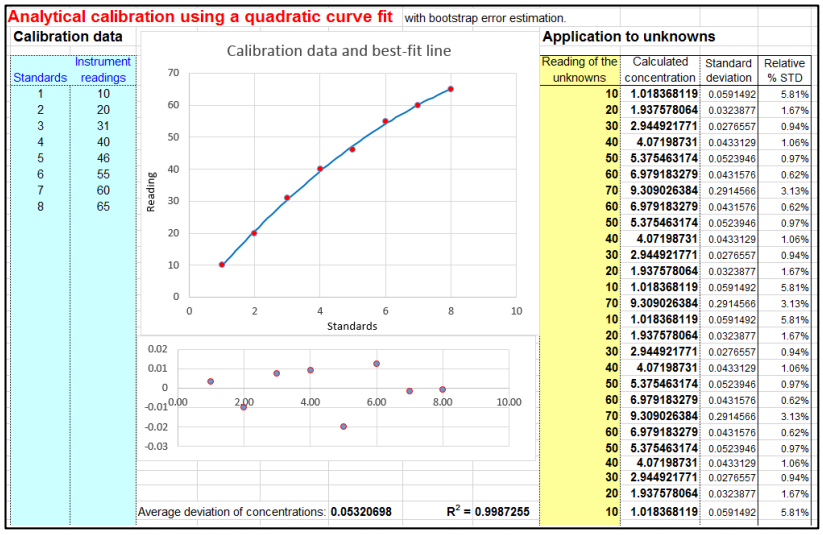
A first-order (straight line) fit of measured signal **A** (y-axis) vs concentration **C** (x-axis). The model equation is **A** = *slope* * **C** + *intercept*. This is the most common and straightforward method, and it is the one to use if you *know* that your instrument response is linear. This fit uses the equations described and listed on page 177. You need a minimum of *two* points on the calibration curve. The concentration of unknown samples is given by *(***A** - *intercept) / slope* where **A** is the measured signal and *slope* and *intercept* from the first-order fit. If you would like to use this method of calibration for your own data, download in Excel or OpenOffice Calc format. View equations for linear least-squares.



Linear interpolation calibration. In the linear interpolation method, sometimes called the bracket method, the spreadsheet performs a linear interpolation between the two standards that are just above and just below each unknown sample, rather than doing a least-squares fit over then entire calibration set. The concentration of the sample Cx is calculated by C1s+(C2s-C1s)*(Sx-S1s)/ (S2s-S1s), where S1x and S2s are the signal readings given by the two standards that are just above and just below the unknown sample, C1s and C2s are the concentrations of those two standard solutions, and Sx is the signal given by the sample solution. This method may be useful if none of the least-squares methods can fit the entire calibration range adequately (for instance, if it contains two linear segments with different slopes). It works well enough if the standards are spaced closely enough so that the actual signal response does not deviate significantly from linearity between the standards. However, this method does not deal well with random scatter in the calibration data due to random noise, because it does not compute a "best-fit" through multiple calibration points as the least-squares methods do. Download a template in Excel (.xls) format.

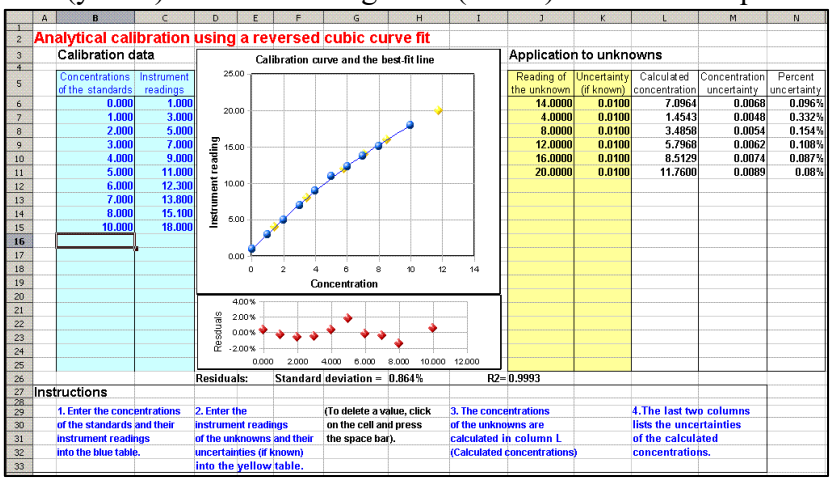


A quadratic fit of measured signal **A** (y-axis) vs concentration **C** (x-axis). The model equation is $\mathbf{A} = aC^2 + b\mathbf{C} + c$. This method can compensate for non-linearity in the instrument response to concentration. This fit uses the equations described and listed on page 177. You need a minimum of *three* points on the calibration curve. The concentration of unknown samples is calculated by solving this equation for **C** using the classical "quadratic formula", namely **C** = (-*b*+SQRT(b^2-4**a**(*c*-**A**)))/(2**a*), where **A** = measured signal, and *a*, *b*, and *c* are the three coefficients from the quadratic fit. If you would like to use this method of calibration for your own data, download in Excel or OpenOffice Calc format. View equations for quadratic least-squares. The alternative version CalibrationQuadraticB.xlsx computes the concentration standard deviation (column **L**) and percent relative standard deviation (column **M**) using the bootstrap method. You need at least 5 standards for the error calculation to work. If you get a "#NUM!" or

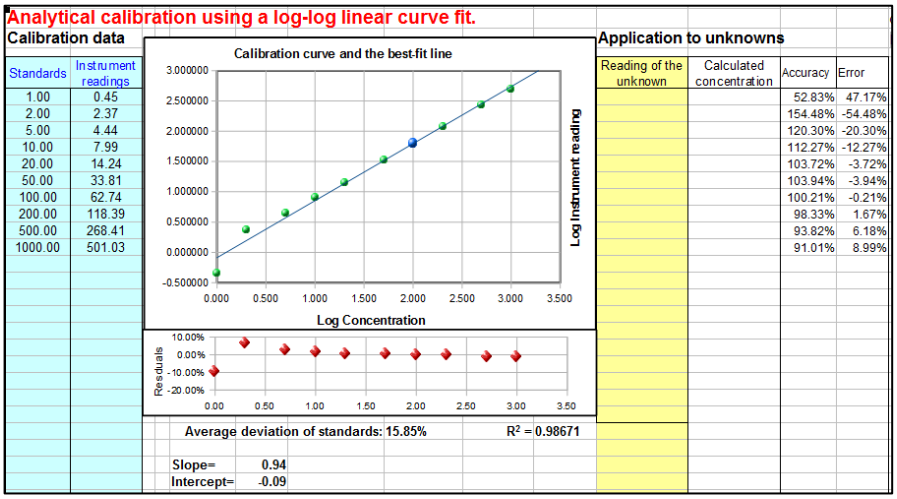


#DIV/0" in columns **L** or **M**, just press the **F9** key to re-calculate the spreadsheet. There is also a *reversed* quadratic template and example, which is analogous to the reversed cubic (#5 below).

A reversed cubic fit of concentration **C** (y-axis) vs measured signal **A** (x-axis). The model equation is $\mathbf{C} = a\mathbf{A}^3 + b\mathbf{A}^2 + c\mathbf{A} + d$. This method is sometimes used to compensate for more complex non-linearity than the quadratic fit. A "reversed fit" flips the usual order of axes, by fitting concentration as a function of measured signal. The aim is to avoid the need to solve a cubic equation when the calibration equation is solved for **C** and used to convert the measured signals of the unknowns into concentration. This coordinate transformation is a short-cut, commonly done in least-squares curve fitting, at least by non-statisticians, to avoid mathematical messiness when the fitting equation is solved for concentration and used to convert the instrument readings into concentration values. However, this reversed method is theoretically not optimum, as demonstrated for the quadratic case Monte-Carlo simulation in the spreadsheet NormalVsReversedQuadFit2.ods (Screenshot), and should be used only if the experimental calibration curve is so non-linear that it cannot be fit by other simpler means. The reversed cubic fit is performed using the LINEST function on Sheet3. You need a minimum of *four* points on the calibration curve. The concentration of unknown samples is calculated directly by $a\mathbf{A}^3+b\mathbf{A}^2+c*\mathbf{A}+d$, where **A** is the measured signal, and *a*, *b*, *c*, and *d* are the four coefficients from the cubic fit. The math is shown and explained better in the template CalibrationCubic5Points.xls (screen image), which is set up for a 5-point calibration, with sample data already entered. To expand this template to a greater number of calibration points, follow these steps exactly: select **row 9** (click on the "9" row label), right-click and select **Insert**, and repeat for each additional calibration point required. Then select **row 8** columns **D** through **K** and drag-copy them down to fill in the newly created rows. That will create all the required equations and will modify the LINEST function in O16-R20. There is also another template, CalibrationCubic.xls, which uses some spreadsheet "tricks" to *sense the number of calibration points automatically* that you enter and adjust the calculations accordingly. You can download in Excel or OpenOffice Calc format.



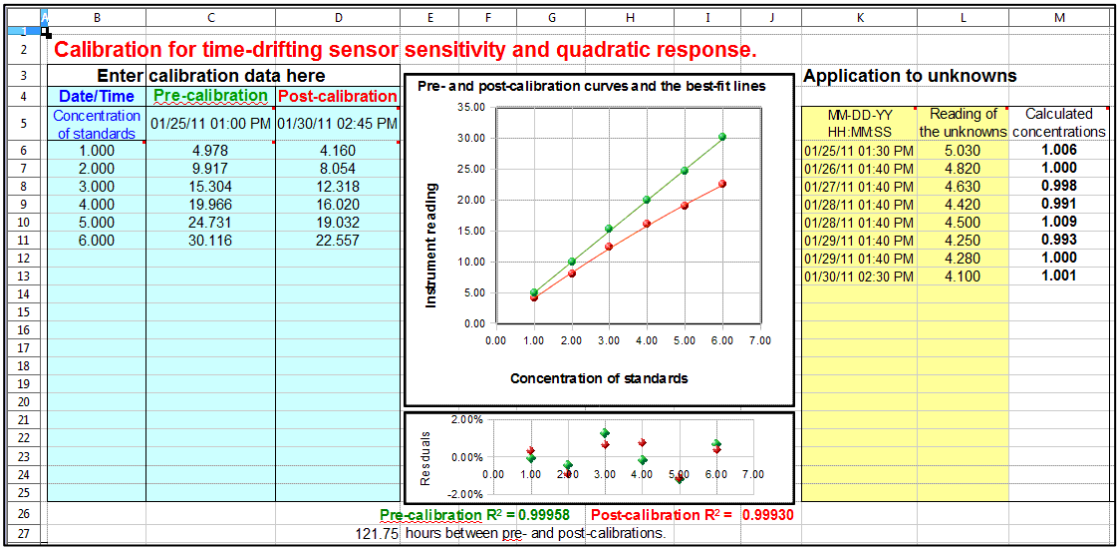
Log-log Calibration. In log-log calibration, the logarithm of the measured signal **A** (y-axis) is plotted against the logarithm of concentration **C** (x-axis) and the calibration data are fit to a linear or quadratic model, as in #1 and #2 above. The concentration of unknown samples is obtained by taking the logarithm of the instrument readings, computing the corresponding logarithms of the concentrations from the calibration equation, then taking the anti-log to obtain the concentration. (These additional steps do not introduce any additional math error, because the log and anti-log conversions can be made quickly and without significant error by the computer). Log-log calibration is sometimes used for *data with a very large range of values* because it distributes the relative fitting error more evenly among the calibration points, preventing the higher calibration points from dominating and causing excessive errors in the low points. In some cases (e.g., Power Law relationships) a nonlinear relationship between signal and concentration can be completely linearized by a log-log transformation. (Some official government-regulated laboratories operate under rules that do not allow the use of nonlinear least-squares fits to calibration data, under the assumption that a nonlinear response is a symptom of equipment failure that should be corrected. But they do not proscribe computing logarithms, so the use of log-log transformation might help in such cases). *However*, because of the use of logarithms, the data set must not *contain any zero or negative values*.



To use this method of calibration for your own data, download the templates for log-log linear (Excel or Calc) or log-log quadratic (Excel or Calc).

Drift-corrected calibration. All the above methods assume that the calibration of the instrument is *stable with time* and that the calibration (usually performed before the samples are measured) remains valid while the unknown samples are measured. In some cases, however, instruments and sensors can *drift*, that is, the *slope* and/or *intercept* of their calibration curves, and even their *linearity*, can gradually change with time after the initial calibration. You can test for this drift by measuring the standards again *after* the samples are run, to determine how different the second calibration curve is from the first. If the difference is not too large, it is reasonable to assume that the drift is approximately linear with time, that is, that the calibration curve parameters (intercept, slope, and curvature) have changed linearly as a function of time between the two calibration runs. It is then possible to correct for the drift if you record the *time* when each calibration is run and when each unknown sample is measured. The drift-correction spreadsheet (CalibrationDriftingQuadratic) does the calculations: it computes a quadratic fit for the pre- and post-calibration curves, then it uses linear interpolation to estimate the calibration curve parameters for each separate sample based on the time it was measured. The method works perfectly only if the drift is linear with time (a reasonable assumption if the amount

of drift is not too large), but in any case, it is *better than simply assuming that there is no drift at all*. If you would like to



use this method of calibration for your own data, download in Excel or OpenOffice Calc format. (See instructions, page 471)

Error calculations. In many cases, it is important to calculate the likely error in the computed concentration values (column **K**) caused by imperfect calibration. This is discussed on page 167, "Reliability of curve fitting results". The linear calibration spreadsheet (download in Excel or OpenOffice Calc format) performs a classical algebraic error-propagation calculation (page 167) on the equation that calculates the concentration from the unknown signal and the slope and intercept of the calibration curve. The quadratic calibration spreadsheet (Download in Excel or OpenOffice Calc format) performs a bootstrap calculation (page 170). You must have at least 5 calibration points for these error calculations to be even minimally reliable: the more the better. That is because these methods need a representative sample of deviations from the ideal calibration line. If the calibration line fits the points exactly, then the computed error will be zero.

Calibration data

Concentration of standards	Instrument readings
1	1.83
2	2.59
5	4.21
10	8.17
20	14.56
50	33.83
100	63.14
200	118.43
500	269.2
1000	500.57

Comparison of calibration methods

To compare these various methods of calibration, I will take one set of real data and subject it to five different calibration curve-fitting methods. The data set, shown on the left, has 10 data points covering a wide (1000-fold) range of concentrations. Over that range, the instrument readings are not linearly proportional to concentration. These data are used to construct a calibration curve, which is then fit using three different models, using the spreadsheet templates described above, and then the equations of the fits, solved for concentration, are used to calculate the concentration of each standard according to that calibration equation. For each method, I compute the relative percent difference between the actual concentration of each

standard and the concentration calculated from the calibration curves. Then I calculated the average of those errors for each method. The objective of this exercise is to determine which method gives the lowest average error for all 10 standards *in this data set*.

The three methods used and their relative percent errors are: (1) linear, 196% error, (2) quadratic, 50% error, and (3) log-log linear, 20% error. ComparisonOfCalibrations.xlsx summarizes the results. For this data set, the best method is the log-log linear, but that does not mean that this method will always be the best in every situation. These calibration data are non-linear, *and* they cover a very wide range of x-values (concentrations), which is a challenge for most calibration methods.

Instructions for using the calibration templates

1. Download and open the desired calibration worksheet from among those listed above (page 465).

2. Enter the concentrations of the standards and their instrument readings (e.g., absorbance) into the blue table on the left. Leave the rest of the table blank. You must have at least two points on the calibration curve (three points for the quadratic method or four points for the cubic method), including the blank (zero concentration standard). If you have multiple instrument readings for one standard, it is better to enter each as a separate standard with the same concentration, rather than entering the average. The spreadsheet automatically gives more weight to standards that have more than one reading.

3. Enter the instrument readings (e.g., absorbance) of the unknowns into the yellow table on the right. You can have any number of unknowns up to 20. (If you have multiple instrument readings for one unknown, it is better to enter each as a separate unknown, rather than averaging them, so you can see how much variation in calculated concentration is produced by the variation in instrument reading).

4. The concentrations of the unknowns are automatically calculated and displayed column K. If you edit the calibration curve, by deleting, changing, or adding more calibration standards, the concentrations are automatically recalculated.

For the linear fit (CalibrationLinear.xls), if you have three or more calibration points, the estimated standard deviation of the slope and intercept will be calculated and displayed in cells **G36** and **G37**, and the resulting standard deviation (SD) of each concentration will be displayed in rows **L** (absolute SD) and **M** (percent relative SD). These standard deviation calculations are estimates of the variability of slopes and intercepts you are likely to get if you repeated the calibration over and over multiple times under the same conditions, assuming that the deviations from the straight line are due to *random variability* and not a systematic error caused by non-linearity. If the deviations are random, they will be slightly different from time to time, causing the slope and intercept to vary from measurement to measurement. However, if the deviations are caused by systematic non-linearity, they will be the same from measurement to measurement, in which case these predictions of standard deviation will not be relevant, and you would be better off using a polynomial fit such as a quadratic or cubic. The reliability of these standard deviation estimates also depends on the number of data points in the curve fit. They improve with the square root of the number of points.

5. You can remove any point from the curve fit by deleting the corresponding X and Y values in the table. To delete a value, right-click on the cell and click "Delete Contents" or "Clear Contents". The

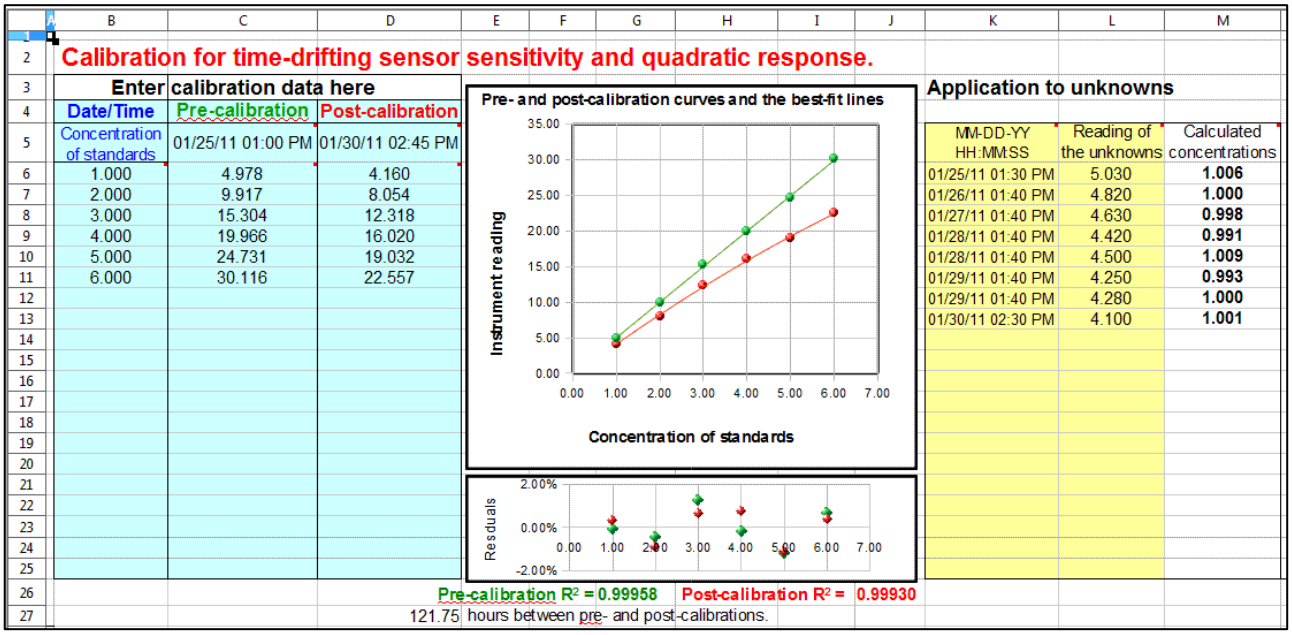
spreadsheet automatically re-calculates and the graph re-draws. If it does not, press F9 to recalculate. (Note: the cubic calibration spreadsheet must have contiguous calibration points with no blank or empty cells in the calibration range).

6. The linear calibration spreadsheet also calculates the coefficient of determination, R^2, which is an indicator of the "goodness of fit", in cell **C37**. R^2 is 1.0000 when the fit is perfect but less than that when the fit is imperfect. The closer to 1.0000 the better.

7. A "residuals plot" is displayed just below the calibration graph (except for the interpolation method). This shows the difference between the best-fit calibration curve and the actual readings of the standards. The smaller these errors, the more closely the curve fits the calibration standards. (The standard deviation of those errors is also calculated and displayed below the residuals plot. The lower this standard deviation, the better).

You can tell a lot by looking at the shape of the residual plot: if the points are scattered randomly above and below zero, it means that the curve fit is *as good as it can be,* given the random noise in the data. But if the residual plot has a smooth shape, say, a U-shaped curve, then it means that there is a mismatch between the curve fit and the actual shape of the calibration curve, suggesting that another curve fitting technique might be tried (say, a quadratic or cubic fit rather than a linear one) or that the experimental conditions be modified to produce a less complex experimental calibration curve shape.

8. **Drift-corrected calibration.** If you are using the spreadsheet for *drift-corrected calibration*, you must measure *two* calibration curves, one *before* and one *after* you run the samples, and you must record the date and time each calibration curve is measured. Enter the concentrations of the standards into column **B**. Enter the instrument readings for the first (pre-) calibration into column **C** and the date/time of that calibration into cell **C5**. Enter the instrument readings for the post-calibration into column **D** and the date/time of that calibration into cell **D5**. The format for the date/time entry is **Month-Day-Year Hours:Minutes:Seconds**, for example, 6-2-2011 13:30:00 for June 2, 2011, 1:30 PM (13:30 on the 24-hour clock). Note: if you run both calibrations on the same day, you can leave off the date and just enter the time. In the graph, the pre-calibration curve is in **green,** and the post-calibration curve is in **red.** Then, for each unknown sample measured, enter the date/time (in the same format) into column **K** and the instrument reading for that unknown into column **L**. The spreadsheet computes the drift-corrected sample concentrations in column **M**. Note: Version 2.1 of this spreadsheet (July 2011) allows different sets of concentrations for the pre- and post-calibrations. Just list all the concentrations used in the "Concentration of standards" column (**B**) and put the corresponding instrument readings in columns **C** *or* **D**, or *both*. If you do not use a particular concentration for one of the calibrations, just leave that instrument reading blank.



This figure shows an application of the drift-corrected quadratic calibration spreadsheet to a remote sensing experiment. In this demonstration, the calibrations and measurements were made over a period of several days. The pre-calibration (column **C**) was performed with six standards (column **B**) on 01/25/2011 at 1:00 PM. Eight unknown samples were measured over the following five days (columns **L** and **M**), and the post-calibration (column **D**) was performed after then last measurement on 01/30/2011 at 2:45 PM. The graph in the center shows the pre-calibration curve in green and the post-calibration curve in red. As you can see, the sensor (or the instrument) had drifted over that time period, the sensitivity (slope of the calibration curve) becoming 28% smaller and the curvature becoming noticeably more non-linear (concave down). This may have been caused in this case by the accumulation of dirt and algal growth on the sensor over time. Whatever the cause, both the pre- and post-calibration curves fit the quadratic calibration equations very well, as indicated by the residuals plot and the "3 nines" coefficients of determination (R^2) listed below the graphs. The eight "unknown" samples that were measured for this test (yellow table) were *the same sample measured repeatedly* - a standard of concentration 1.00 units - but you can see that the sample gave lower instrument readings (column **L**) each time it was measured (column **K**), due to the drift. Finally, the drift-corrected concentrations calculated by the spreadsheet (column **M** on the right) are all very close to 1.00, with a standard deviation of 0.6%, showing that the drift correction works well, within the limits of the random noise in the instrument readings and subject to the assumption that the drift in the calibration curve parameters is linear with time between the pre- and post-calibrations.

Frequently Asked Questions (taken from emails and search engine queries)

1. Question: *What is the purpose of the calibration curve?*

Answer: Most analytical instruments generate an electrical output signal such as a current or a voltage. A calibration curve establishes the relationship between the signal generated by a measurement instrument and the concentration of the substance being measured. Different chemical compounds and elements give different signals. When an unknown sample is measured, the signal from the unknown is converted into concentration using the calibration curve.

2. **Question:** *How do you make a calibration curve?*
Answer: You prepare a series of "standard solutions" of the substance that you intend to measure, measure the signal (e.g., absorbance, if you are doing absorption spectrophotometry), and plot the concentration on the x-axis and the measured signal for each standard on the y-axis. Draw a straight line as close as possible to the points on the calibration curve (or a smooth curve if a straight line will not fit), so that as many points as possible are right on or close to the curve.

3. **Question:** *How do you use a calibration curve to predict the concentration of an unknown sample? How do you determine concentration from a non-linear calibration plot?*
Answer: You can do that in two ways, graphically and mathematically. Graphically, draw a horizontal line from the signal of the unknown on the *y* axis over to the calibration curve and then straight down to the concentration (*x*) axis to the concentration of the unknown. Mathematically, fit an equation to the calibration data, and solve the equation for concentration as a function of signal. Then, for each unknown, just plug its signal into this equation and calculate the concentration. For example, for a linear equation, the curve fit equation is **Signal** = *slope* * **Concentration** + *intercept*, where *slope* and *intercept* are determined by a linear (first-order) least-squares curve fit to the calibration data. Solving this equation for **Concentration** yields **Concentration** = (**Signal** - *intercept) / slope*, where **Signal** is the signal reading (e.g., absorbance) of the unknown solution. (Click here for a fill-in-the-blank OpenOffice spreadsheet that does this for you. View screen shot).

4. **Question:** *How do I know when to use a straight-line curve fit and when to use a curved line fit like a quadratic or cubic?*
Answer: Fit a straight line to the calibration data and look at a plot of the "residuals" (the differences between the *y* values in the original data and the *y* values computed by the fit equation). Deviations from linearity will be much more evident in the residuals plot than in the calibration curve plot. (Click here for a fill-in-the-blank OpenOffice spreadsheet that does this for you. View screen shot). If the residuals are randomly scattered all along the best-fit line, then it means that the deviations are caused by random errors such as instrument noise or by random volumetric or procedural errors. In that case you can use a straight line (linear) fit. If the residuals have a smooth shape, like a "U" shape, this means that the calibration curve is curved, and you should use a non-linear curve fit, such as a quadratic or cubic fit. If the residual plot has an “S" shape, you should probably use a cubic fit. (If you are doing absorption spectrophotometry, see Comparison of Curve Fitting Methods in Absorption Spectroscopy).

5. **Question:** *What if my calibration curve is linear at low concentrations but curves off at the highest concentrations?*
Answer: You can't use a linear curve fit in that case, but if the curvature is not too severe, you might be able to get a good fit with a quadratic or cubic fit. If not, you could break the concentration range into two regions and fit a linear curve to the lower linear region and a quadratic or cubic curve to the higher non-linear region.

6. **Question:** *What is the difference between a calibration curve and a line of best fit? What is the difference between a linear fit and a calibration curve?*
Answer: The calibration curve is an experimentally measured relationship between concentration and signal. You do not ever really know the *true* calibration curve. You can only *estimate* it at a few points

by measuring a series of standard solutions. Then draw a line or a smooth curve that goes as much as possible through the points, with some points being a little higher than the line and some points a little lower than the line. That is what I mean by that is a "best fit" to the data points. The actual calibration curve might not be perfectly linear, so a linear fit is not always the best. A quadratic or cubic fit might be better if the calibration curve shows a gradual smooth curvature.

7. **Question:** *Why the slope line does not go through all points on a graph?*
Answer: That will only happen if you (1) are a perfect experimenter, (2) have a perfect instrument, and (3) choose the perfect curve-fit equation for your data. That is not going to happen. There are *always* little errors. The least-squares curve-fitting method yields a *best* fit, not a *perfect* fit, to the calibration data for a given curve shape (linear, quadratic, or cubic). Points that fall off the curve are assumed to do so because of random errors or because the actual calibration curve shape does not match the curve-fit equation.

There is one artificial way you can make the curve go through all the points, and that is to use *too few calibration standards*: for example, if you use only *two* points for a straight-line fit, then the best-fit line will obviously go through those two points *no matter what*. Similarly, if you use only *three* points for a quadratic fit, then the quadratic best-fit curve will go right through those three points, and if you use only *four* points for a cubic fit, then the cubic best-fit curve will go right through those four points. But that is not really recommended, because if one of your calibration points is off by a huge error, the curve fit *will still look perfect*, and you will have *no clue* that something is wrong. *You really must use more standards so that you will know when something has gone wrong.*

8. **Question:** *What happens when the absorbance reading is higher than any of the standard solutions?*
Answer: If you are using a curve-fit equation, you will still get a value of concentration calculated for *any* signal reading you put in, even above the highest standard. However, it is risky to do that, because you really do not know for sure what the shape of the calibration curve is above the highest standard. It could continue straight, or it could curve off in some unexpected way - how would you know for sure? It is best to add another standard at the high end of the calibration curve.

9. **Question:** *What's the difference between using a single standard and multiple standards?*
Answer: The single standard method is the simplest and quickest method, but it is accurate only if the calibration curve is known to be linear. Using multiple standards has the advantage that any non-linearity in the calibration curve can be detected and avoided (by diluting into the linear range) or compensated (by using non-linear curve fitting methods). Also, the random errors in preparing and reading the standard solutions are averaged over several standards, which is better than "putting all your eggs in one basket" with a single standard. On the other hand, an obvious disadvantage of the multiple standard method is that it requires much more time and uses more standard material than the single standard method.

10. **Question:** *What's the relationship between sensitivity in analysis and the slope of standard curve*?
Answer: Sensitivity is defined as the slope of the standard (calibration) curve.

11. **Question:** *How do you make a calibration curve in Excel or in OpenOffice?*
Answer: Put the concentration of the standards in one column and their signals (e.g., absorbances) in

another column. Then make an XY scatter graph, putting concentration on the X (horizontal) axis and signal on the Y (vertical) axis. Plot the data points with symbols only, not lines between the points. To compute a least-squares curve fit, you can either put in the least-squares equations into your spreadsheet, or you can use the built-in LINEST function in both Excel and OpenOffice Calc to compute polynomial and other curvilinear least-squares fits. For examples of OpenOffice spreadsheets that graphs and fits calibration curves, see Worksheets for Analytical Calibration Curves.

12. **Question:** *What's the difference in using a calibration curve in absorption spectrometry vs other analytical methods such a fluorescence or emission spectroscopy?*
Answer: The only difference is the units of the signal. In absorption spectroscopy you use *absorbance* (because it is the most nearly linear with concentration) and in fluorescence (or emission) spectroscopy you use the *fluorescence (or emission) intensity*, which is usually linear with concentration (except sometimes at high concentrations). The methods of curve fitting and calculating the concentration are basically the same.

13. **Question:** *If the solution obeys Beer's Law, is it better to use a calibration curve rather than a single standard?*
Answer: It might not make much difference either way. If the solution is known from previous measurements to obey Beer's Law exactly on the same spectrophotometer and under the conditions in use, then a single standard can be used (although it is best if that standard gives a signal close to the maximum expected sample signal or to whatever signal gives the best signal-to-noise ratio - an absorbance near 1.0 in absorption spectroscopy). The only real advantage of multiple standards in this case is that the random errors in preparing and reading the standard solutions are averaged over several standards, but the same effect can be achieved more simply by making up multiple copies of the same single standard (to average out the random volumetric errors) and reading each separately (to average out the random signal reading errors). And if the signal reading errors are much smaller than the volumetric errors, then a *single* standard solution can be measured repeatedly to average out the random measurement errors.

14. **Question:** *What is the effect on concentration measurement if the monochromator is not perfect?*
Answer: If the wavelength calibration if off a little bit, it will have no significant effect if the monochromator setting is left untouched between measurement of standards and unknown sample. The slope of the calibration curve will be different, but the calculated concentrations will be OK. (But if anything changes the wavelength between the time you measure the standards and the time you measure the samples, an error will result). If the wavelength has a poor stray light rating or if the resolution is poor (spectral bandpass is too big), the calibration curve may be affected adversely. In absorption spectroscopy, stray light and poor resolution may result in non-linearity, which requires a non-linear curve fitting method. In emission spectroscopy, stray light and poor resolution may result in a spectral interference which can result in significant analytical errors.

15. **Question:** *What does it mean if the intercept of my calibration curve fit is not zero?*
Answer: Ideally, the y-axis intercept of the calibration curve (the signal at zero concentration) should be zero, but there are several reasons why this might not be so. (1) If there is substantial random scatter in the calibration points above and below the best-fit line, then it is likely that the non-zero intercept is

just due to random error. If you prepared another separate set of standards, that standard curve would have different intercept, either positive or negative. There is nothing that you can do about this, unless you can reduce the random error of the standards and samples. (2) If the shape of the calibration curve does not match the shape of the curve fit, then it is very likely that you will get a non-zero intercept every time. For example, if the calibration curve bends down as concentration increases, and you use a straight-line (linear) curve fit, the intercept will be positive (that is, the curve fit line will have a positive y-axis intercept, even if the actual calibration curve goes through zero). This is an artifact of the poor curve fit selection. If you see that happen, try a different curve shape (quadratic or cubic). (3) If the instrument is not "zeroed" correctly, in other words, if the instrument gives a non-zero reading when the blank solution is measured. In that case you have three choices: you can zero the instrument (if that is possible). You can subtract the blank signal from all the standard and sample readings. Or you can just let the curve fit subtract the intercept for you (if your curve fit procedure calculates the intercept and you keep it in the solution to that equation, e.g., Concentration = (Signal - *intercept*) / *slope*).

16. **Question:** *How can I reduce the random scatter of calibration points above and below the best-fit line?*
Answer: Random errors like this could be due either to random volumetric errors (small errors in volumes used to prepare the standard solution by diluting from the stack solution or in adding reagents) or they may be due to random signal reading errors of the instrument, or to both. To reduce the volumetric error, use more precise volumetric equipment and practice your technique to perfect it (for example, use your technique to deliver pure water and weigh it on a precise analytical balance). To reduce the signal reading error, adjust the instrument conditions (e.g., wavelength, path length, slit width, etc.) for best signal-to-noise ratio and average several readings of each sample or standard.

17. **Question:** *What are interferences? What effect do interferences have on the calibration curve and on the accuracy of concentration measurement?*
Answer: When an analytical method is applied to complex real-world samples, for example the determination of drugs in blood serum, measurement error can occur due to *interferences*. Interferences are measurement errors caused by chemical components in the samples that influence the measured signal, for example by contributing their own signals or by reducing or increasing the signal from the analyte. Even if the method is well calibrated and is capable of measuring solutions of pure analyte accurately, interference errors may occur when the method is applied to complex real-world samples. One way to correct for interferences is to use "matched-matrix standards", standard solutions that are prepared to contain *everything that the real samples contain*, except that they have known concentrations of analyte. But this is very difficult and expensive to do exactly, so every effort is made to reduce or compensate for interferences in other ways. For more information on types of interferences and methods to compensate for them, see Comparison of Analytical Calibration Methods.

18. **Question:** *What are the sources of error in preparing a calibration curve?*
Answer: A calibration curve is a plot of analytical signal (e.g., absorbance, in absorption spectrophotometry) vs concentration of the standard solutions. Therefore, the main sources of error are the errors in the standard concentrations and the errors in their measured signals. Concentration errors depend mainly on the accuracy of the volumetric glassware (volumetric flasks, pipettes, solution

delivery devices) and on the precision of their use by the people preparing the solutions. In general, the accuracy and precision of handling large volumes above 10 mL is greater than that at lower volumes below 1 mL. Volumetric glassware can be calibrated by weighing water on a precise analytical balance (you can look up the density of water at various temperatures and thus calculate the exact volume of water from its measured weight). This would allow you to label each of the flasks, etc., with their actual volume. But precision may still be a problem, especially at lower volumes, and it is very much operator-dependent. It takes practice to get good at handling small volumes. Signal measurement error depends hugely on the instrumental method used and on the concentration of the analyte. It can vary from near 0.1% under ideal conditions to 30% near the detection limit of the method. Averaging repeat measurements can improve the precision with respect to random noise. To improve the signal-to-noise ratio at low concentrations, you may consider modifying the conditions, such as changing the monochromator slit width or the absorption path length or by using another instrumental method (such as a graphite furnace rather than flame atomizer in atomic absorption measurements).

19. **Question:** *How can I find the error in a specific quantity using a least squares fitting method? How can I estimate the error in the calculated slope and intercept?*
Answer: When using a simple straight-line (first-order) least-squares fit, the best fit line is specified by only two quantities: the *slope* and the *intercept*. The *random error* in the slope and intercept (specifically, their standard deviation) can be estimated mathematically from the extent to which the calibration points deviate from the best-fit line. The equations for doing this are given here and are implemented in the "spreadsheet for linear calibration with error calculation". *It is important to realize that these error computations are only estimates*, because they assume that the calibration data set is representative of all the calibration sets that would be obtained if you repeated the calibration a large number of times - in other words, the assumption is that the random errors (volumetric and signal measurement errors) in your particular data set are typical. If your random errors happen to be small when you run your calibration curve, you will get a deceptively *good*-looking calibration curve, but your estimates of the random error in the slope and intercept will be too *low*. If your random errors happen to be large, you will get a deceptively *bad*-looking calibration curve, and your estimates of the random error in the slope and intercept will be too *high*. These error estimates can be particularly poor when the number of points in a calibration curve is small. The accuracy of the estimates increases if the number of data points increases, but of course preparing many standard solutions is time consuming and expensive. The bottom line is that you can only expect these error predictions from a single calibration curve to be very rough. They could easily be off by a factor of two or more, as demonstrated by the simulation "Error propagation in the Linear Calibration Curve Method" (download OpenOffice version).

20. *How can I estimate the error in the calculated concentrations of the unknowns?*
Answer: You can use the slope and intercept from the least-squares fit to calculate the concentration of an unknown solution by measuring its signal and computing (**Signal** - *intercept*) / *slope*, where **Signal** is the signal reading (e.g., absorbance) of the unknown solution. The errors in this calculated concentration can then be estimated by the usual rules for the propagation of error. First, the error in (**Signal** - *intercept*) is computed by the rule for addition and subtraction. Second, the error in (**Signal** - *intercept*) / *slope* is computed by the rule for multiplication and division. The equations for doing this

are given here and are implemented in the "spreadsheet for linear calibration with error calculation". *It is important to realize that these error computations are only estimates*, for the reason given in Question #19 above, especially if the number of points in a calibration curve is small, as demonstrated by the simulation "Error propagation in the Linear Calibration Curve Method" (download OpenOffice version).

21. *What is the minimum acceptable value of the coefficient of determination (R^2) for calibration?*
Answer: It depends on the accuracy required. As a rough rule of thumb, if you need an accuracy of about 0.5%, you need an R^2 of 0.9998. If a 1% error is good enough, an R^2 of 0.997 will do. And if a 5% error is acceptable, an R^2 of 0.97 will do. The bottom line is that the R^2 must be very close to 1.0 for quantitative results in analytical chemistry.

Catalog of signal processing functions, scripts, and spreadsheet templates

This is a list of the functions, scripts, data files, and spreadsheets used in this book, collected according to topic, with brief descriptions. If you are reading this book online, on an Internet connected computer, you can **Ctrl-Click** on any of links and select "Save link as..." to download them to your computer. There are approximately 200 Matlab/Octave m-files (functions and demonstration scripts). Place these into the Matlab or Octave "path" so you can use them just like any other built-in feature. (Difference between scripts and functions). To display the built-in help for these functions and script, type "help <name>" at the command prompt (where "<name>" is the name of the script or function).

Many of the figures in this book are screen images of my software in action. Often the screen shots display a version number that is older than the current version, betraying the date when they were first made. There is almost certainly a newer version that includes additional functions. If you are unsure whether you have all the latest versions, the simplest way to update all my functions, scripts, tools, spreadsheets and documentation files is to download the latest site archive ZIP file (approx. 400 MBytes), then right-click on the zip file and click "Extract all". Then list the contents of the extracted folder *by date* and then drag and drop any *new or newly updated files* into a folder in your Matlab/Octave path. The ZIP file contains *all* the files used by this web site *in one directory*, so you can search for them by file name or sort them by date to determine which ones have changed since the last time you downloaded them.

If you try to run one of my scripts or functions and it gives you a "missing function" error, look for the missing item in this catalog, download it into your path, and try again.

Some of these functions have been requested by users, suggested by Google search terms, or corrected and expanded based on extensive user feedback. You could almost consider this an international "crowd-sourced" software project. *I wish to express my thanks and appreciation for all those who have made useful suggestions, corrected errors, and especially those who have sent me data from their work to test my programs on. These contributions have really helped to correct bugs and to expand the capabilities of my programs.*

Peak shape functions (for Matlab and Octave)



Most of these shape functions take *three* required input arguments: the independent variable ("x") vector, the peak position, "pos", and the peak width, "wid", usually the full width at half maximum. The functions marked '*variable shape*' require an additional fourth input argument that determines the exact peak shape. The sigmoidal and exponential shapes (alpha function, exponential pulse, up-sigmoid, down-sigmoid, Gompertz, FourPL, and OneMinusExp) have different variables names.

Gaussian **`y = gaussian(x,pos,wid)`**
Exponentially-broadened Gaussian (variable shape)
Triangle-broadened Gaussian (variable shape)
Bifurcated Gaussian (variable shape)
Flattened Gaussian (variable shape)
Clipped Gaussian (variable shape)
Lorentzian (aka 'Cauchy') **`y = lorentzian(x,pos,wid)`**
Exponentially-broadened Lorentzian (variable shape)
Clipped Lorentzian (variable shape)

Gaussian/Lorentzian blend (variable shape)
Voigt profile (variable shape)
Lognormal
Sech2pulse (intermediate between Gaussian and Lorentzian) The self-contained script Sech2ShapeComparison.m compares Gaussian, Lorentzian, and sech2 pulse shapes (graphic).
Logistic distribution (On the other hand, for the logistic *function*, see up-sigmoid)
Pearson 5 (variable shape)
Alpha function
Exponential pulse
Plateau (variable shape, symmetrical product of sigmoid and down sigmoid, similar to Flattened Gaussian)
Breit-Wigner-Fano resonance (BWF) (variable shape)
Triangle
Exponentially-broadened triangle (variable shape)
Gaussian/Triangle blend (variable shape)
Rectanglepulse
Tsallis distribution (variable shape, similar to Pearson 5)
up-sigmoid (logistic *function* or "S-shaped"). Simple upward going sigmoid.
Down-sigmoid ("Z-shaped") Simple downward going sigmoid.
Gompertz, 3-parameter logistic, a variable-shape sigmoidal:
`y=Bo*exp(-exp((Kh*exp(1)/Bo)*(L-t) +1))`
FourPL, 4-parameter logistic, `y = maxy+(miny-maxy)./(1+(x./ip).^slope)`
OneMinusExp, Asymptotic rise to flat plateau: `g = 1-exp(-wid.*(x-pos))`
peakfunction.m, a function that generates many different peak types specified by number.

modelpeaks, a function that simulates multi-peak time-series signal data consisting of any number of peaks of the same shape. Syntax is model= modelpeaks(x, NumPeaks, peakshape, Heights, Positions, Widths, extra), where 'x' is the independent variable vector, 'NumPeaks' is the number of peaks, 'peakshape' is the peak shape number, 'Heights' is the vector of peak heights, 'Positions' is the vector of peak positions, 'Widths' is the vector of peak widths, and 'extra' is the additional shape parameter required by the exponentially broadened, Pearson, Gaussian/Lorentzian blend, BiGaussian and BiLorentzian shapes. Type 'help modelpeaks'. To create noisy peaks, use one of the following noise functions to create some random noise to add to the modelpeaks array.

modelpeaks2, a function that simulates multi-peak time-series signal data consisting of any number of peaks of different shapes. Syntax is y=modelpeaks2(t, Shape, Height, Position, Width, extra) where 'shape' is a vector of peak type numbers and the other input arguments are the same as for modelpeaks.m. Type 'help modelpeaks2'.

ShapeDemo demonstrates 16 basic peak shapes graphically, showing the variable-shape peaks as multiple lines. (Graphic on page 424)

SignalGenerator.m is a script that uses the modelpeaks.m function above to create and plot realistic computer-generated signal consisting of multiple peaks on a variable baseline plus variable random noise. You may change the lines marked by "<<<" to modify the character of the signal peaks, baseline, and noise.

Signal Arithmetic

stdev.m Octave and Matlab compatible standard deviation function (because the regular built-in std.m function behaves differently in Matlab and in Octave). rsd.m is the relative standard deviation (the standard deviation divided by the mean).

PercentDifference.m A simple function that calculates the percent difference between two numbers or vectors, i.e., `100.*(b-a)./a,` where a and b can be scalar or vector.

halfwidth and tenthwidth: `[FWHM,slope1,slope2,hwhm1,hwhm2] = halfwidth(x,y,xo)` uses linear interpolation between points to compute the approximate FWHM (full width at half maximum) of any smooth peak whose maximum is at x=xo, has a zero baseline, and falls to below one-half of the maximum height on both sides. Not accurate if the peak is noisy or sparsely sampled. If the additional output arguments are supplied, it also returns the leading and trailing edge slopes, slope1 and slope2, and the leading and trailing edge half widths at half maximum, hwhm1 and hwhm2, respectively. If x0 is omitted, it determines the halfwidth of the largest peak. Example: `xo=500; width=100; x=1:1000; y=exp(-1.*((x-xo)/(0.60056120439323.*width)).^2); halfwidth(x,y,xo).` The analogous function [twidth,slope1,slope2,hwhm1,hwhm2] = tenthwidth(x,y,xo) computes the full width at 1/10 maximum, and just for the heck of it, hundredthwidth, `[hwidth,slope1,slope2] = hundredthwidth(x,y,xo)`, computes the full width at 1/100 maximum.

MeasuringWidth.m is a script that compares two methods of measuring the full width at half maximum of a peak: Gaussian fitting (using peakfit.m) and direct interpolation (using halfwidth.m). The two methods agree exactly for a finely-sampled noiseless Gaussian on a zero baseline but give slightly different answers if any of these conditions are not met. The halfwidth function works well for any finely-sampled smooth peak shape on a zero baseline, but the peakfit function is better at resisting random noise and it can correct for some types of baselines and it has a wide selection of peak shapes to use as a model. See the help file.

IQrange.m, estimates the standard deviation of a set of numbers by dividing its "interquartile range" (IQR) by 1.34896, an alternative to the usual standard deviation calculation that works better for computing the dispersion (spread) of a data set that contains outliers. Essentially it is the standard deviation with outliers removed. Syntax is b = IQrange(a).

rmnan(a), which stands for "**R**e**M**ove **N**ot **A** **N**umber", removes NaNs ("**N**ot **a** **N**umber") and Infs ("Infinite") from vectors, replacing with nearest real numbers and printing out the number of changes (if any are made). Use this to prevent subsequent operations from stopping on an error.

rmz(a) **R**e**M**oves **Z**eros from vectors, replacing with nearest non-zero numbers and printing out the number of changes (if any are made). Use this to remove zeros from vectors that will subsequently be used as the denominator of a division.

[a,changes]=nht(a,b); "no higher than" replaces any numbers in vector a that are above the scalar b with b. Optionally "changes" returns the number of changes. The similar function [a,changes]=nlt(a,b), "no lower than", replaces any numbers in vector a that are lower than the scalar b with b. Optionally "changes" returns the number of changes.

makeodd(a): Makes the elements of vector "a" the next higher odd integers. This can be useful in computing smooth widths to ensure that the smooth will not shift the maximum of peaks. For example, `makeodd([1.1 2 3 4.8 5 6 7.7 8 9]) = [1 3 3 5 5 7 9 9 9]`

condense(y,n), function to reduce the length of vector y by replacing each group of n successive values

by their average. The similar function condensem.m works for matrices. Use to re-sample an oversampled signal. Mentioned on Smoothing (page 40) and iSignal (page 390)

val2ind(x,val), returns the index and the value of the element of vector x that is closest to val. Example: **`if x=[1 2 4 3 5 9 6 4 5 3 1], then val2ind(x,6)=7 and val2ind(x,5.1)=[5 9].`** This is useful for accessing subsets of x, y data sets. For example, the code sequence **`x1=7;x2=8; irange = val2ind(x,x1):val2ind(x,x2); xx=x(irange); yy=y(irange); plot(xx,yy)`** will isolate the subset xx, yy and plot it only over the range of x values from 7 to 8. For some other examples of how this can be used, see page 251.

testcondense.m is a script that demonstrates of the effect of boxcar averaging using the condense.m function to reduce noise without changing the noise color. Shows that it reduces the measured noise, removing the high frequency components, resulting in a faster fitting execution time and a lower fitting error, but no more accurate measurement of peak parameters.

NumAT(m,threshold): "**Num**bers **A**bove **T**hreshold": Counts the number of adjacent elements in the vector 'm' that are greater than or equal to the scalar value 'threshold'. It returns a matrix listing each group of adjacent values, their starting index, the number of elements in that group, and the sum of that group, and the mean. Type "help NumAT" and try the example.

isOctave.m Returns 'true' if this code is being executed by Octave. It returns 'false' if this code is being executed by MATLAB, or any other MATLAB variant. Useful in those few cases where there is a small difference between the syntax or operation of Matlab and Octave functions, as for example trypoly(x,y), tablestats.m, and trydatatrans.m.

Data plotting

The Matlab/Octave scripts plotting.m and plotting2.m show how to plot multiple signals using matrices and subplots (multiple small plots in a single Figure window). The scripts realtimeplotautoscale.m and realtimeplotautoscale2.m demonstrate plotting in (Click for animated graphic).

plotit, version 2, is an easy-to-use function for plotting x,y data in matrices or in separate vectors. Syntax: **`[coef,RSquared,StdDevs,BootResults]=plotit(xi,yi,polyorder,datastyle, fitstyle)`**. It can also fit polynomials to the data and compute the errors. Click here or type "help plotit" at the Matlab/Octave prompt for some examples.

plotxrange extracts and plots values of vectors x,y only for x values between x1 and x2. Returns extracted values in vectors xx,yy and the range of index values in irange. Ignores values of x1 and x2 outside the range of x.

segplot, syntax **`[s,xx,yy]=segplot(x,y,NumSegs,seg)`**, divides y into "NumSegs" equal-length segments and plots the x, y data with segments marked by vertical lines, each labeled with a small segment number at the bottom. Returns a vector 's' of segment indexes, and the subset xx,yy, of values in the segment number 'seg'. If the 4th input argument, 'seg', is included, it plots this segment only.

Signals and Noise

whitenoise, pinknoise, bluenoise propnoise, sqrtnoise, bimodal: different types of random noise that might be encountered in physical measurements. Type "help whitenoise", etc., for help and examples.

noisetest.m is a self-contained Matlab/Octave function for demonstrating different noise types. It plots Gaussian peaks with four different types of added noise with the same standard deviation: constant white noise, constant pink (1/f) noise, proportional white noise, and square-root white noise. It then fits a Gaussian model to each noisy data set and computes the average and the standard deviation of the

peak height, position, width, and area for each noise type. See page 24. See also NoiseColorTest.m.

SubtractTwoMeasurements.m is a Matlab/Octave script demonstration of measuring the noise and signal-to-noise ratio of a stable waveform by subtracting two measurements of the signal waveform, m1 and m2 and computing the standard deviation of the difference. The signal must be stable between measurements (except for the random noise). The standard deviation of the measured noise is given by sqrt((std(m1-m2).^2)/2).

NoiseColorTest.m, a function that demonstrates the effect of smoothing white, pink, and blue noise. It displays a graphic of five noise color types both before and after smoothing, as well as their frequency spectra. All noise samples have a standard deviation of 1.0 before smoothing. You can change the smooth width and type in lines 6 and 7.

CurvefitNoiseColorTest.m, a function that demonstrates the effect of white, pink, and blue noise on curve fitting a single Gaussian peak.

RANDtoRANDN.m is a script that demonstrates how the expression 1.73*(RAND() - RAND() + RAND() - RAND()) approximates normally-distributed random numbers with zero mean and a standard deviation of 1. See page 24.

RoundingError.m. A script that demonstrates digitization (rounding) noise and shows that adding noise and then ensemble averaging multiple signals can reduce the overall noise in the signal. This is a rare example where adding noise is beneficial. See page 308.

DigitizedSpeech.m, an audible/graphic demonstration of rounding error on digitized speech. It starts with an audio recording of the spoken phrase "Testing, one, two, three", previously recorded at 44000 Hz and saved in WAV format (download link), rounds off the amplitude data progressively to 8 bits (256 steps), 4 bits (16 steps), and 1 bit (2 steps), and then the same with random white noise added before the rounding (2 steps + noise), plots the waveforms and plays the resulting sounds, demonstrating both the degrading effect of rounding and the remarkable improvement caused by adding noise. See page 308. See iFilterTesting123.m.

CentralLimitDemo.m, script that demonstrates that the more independent uniform random variables are combined, the probability distribution becomes closer and closer to normal (Gaussian). See page 24
EnsembleAverageDemo.m is a Matlab/Octave script that demonstrates ensemble averaging to improve the signal-to-noise ratio of a very noisy signal. Click for graphic. The script requires the "gaussian.m" function to be downloaded and placed in the Matlab/Octave path, or you can use any other peak shape function, such as lorentzian.m or rectanglepulse.m.

EnsembleAverageDemo2.m is a Matlab/Octave script that demonstrates the effect of *amplitude noise*, *frequency noise*, and *phase noise* on the ensemble averaging of a sine waveform.

EnsembleAverageFFT.m is a Matlab/Octave script that demonstration of the effect of *amplitude noise*, *frequency noise*, and *phase noise* on the ensemble averaging of a sine waveform signal. Shows that: (a) ensemble averaging reduces the white noise in the signal but not the frequency or phase noise, (b) ensemble averaging the Fourier transform has the same effect as ensemble averaging the signal itself, and (c) the effect of phase noise is reduced if the power spectra are ensemble averaged. EnsembleAverageFFTGaussian.m does the same for a Gaussian peak signal, where variation in peak width is frequency noise and variation in peak position is phase noise.

iPeakEnsembleAverageDemo.m is a self-contained demonstration of the iPeak function. In this example, the signal contains a repeated pattern of two overlapping Gaussian peaks of width 12, with a

2:1 height ratio. These patterns occur at random intervals, and the noise level is about 10% of the average peak height. Using iPeak's ensemble average function (**Shift-E**), the patterns can be averaged and the signal-to-noise ratio significantly improved. See page 27.

PeriodicSignalSNR.m is a Matlab/Octave script demonstrating the estimation of the peak-to-peak and root-mean-square signal amplitude and the signal-to-noise ratio of a periodic waveform, estimating the noise by looking at the time periods where its envelope drops below a threshold. See page 24.

LowSNRdemo.m is a script that compares several different methods of peak measurement with very low signal-to-noise ratios. It creates a single peak, with adjustable shape, height, position, and width, adds constant white random noise so the signal-to-noise ratio varies from 0 to 2, then measures the peak height and position by each method and computes the average error. Four methods are compared: (1) the peak-to-peak measure of the smoothed signal and background, (2) a peak finding method based on findpeakG, (3) unconstrained iterative least-squares fitting (INLS) based on the peakfit.m function, and (4) constrained classical least-squares fitting(CLS) based on the cls2.m function. See the appendix: How Low can you Go? Performance with very low signal-to-noise ratios.

RandomWalkBaseline.m simulates a Gaussian peak with randomly variable position and width superimposed on a drifting "random walk" baseline. Compare to WhiteNoiseBaseline.m. See page 318.

AmplitudeModulation.m is a Matlab/Octave script simulation of modulation and synchronous detection, demonstrating the noise reduction capability. See page 320.

DerivativeNumericalPrecisionDemo.m. Self-contained function that demonstrates how the *numerical precision limits* of the computer affects the first through fourth derivatives of a smooth ("noiseless") Gaussian band, showing both the waveforms (in Figure window 1) and their frequency spectra (in Figure window 2). The numerical precision limit of the computer creates random noise at very high frequencies, which is emphasized by differentiation, and by the fourth derivative that noise overwhelms the signal frequencies at lower frequencies. Most of the noise can be removed by smoothing with a p-spline (three passes of a sliding-average) with a smooth ratio of 0.2. With real experimental data, even the tiniest amounts of noise in the original data would be much greater than this. See page 340.

RegressionNumericalPrecisionTest.m is a Matlab/Octave script that demonstrates how the *numerical precision limits* of the computer effects the Classical Least-squares (multilinear regression) of two very closely-spaced "noiseless" overlapping Gaussian peaks. This uses three different mathematical formulation of the least-squares calculation that give different results when the numerical precision limits of the computer are reached. But practically, the difference between these methods is unlikely to be seen. Even the tiniest bit of added random noise (line 15) or signal instability produces a far greater error. Used in page 340.

RegressionADCbitsTest.m. Demonstration of the effect of analog-to-digital converter resolution (defined by the number of bits in line 9) on Classical Least-squares (multilinear regression) of two closely-spaced overlapping Gaussian peaks. Normally, the random noise (line 10) produces a greater error than the ADC resolution. Used on page 340.

CreateSimulatedSignal.m. Script that creates a simulated multi-peak signal that is meant to match an experimental signal, using a list of peaks in the experimental signal in matrix P, plus added noise and baseline. Used to test the accuracy of peak detection and area measurement methods with that type of signal.

Smoothing

fastsmooth, versatile function for fast data smoothing. The syntax is **`SmoothY=fastsmooth(Y,w, type, ends)`**. See page 40. Note: Greg Pittam has published a modification of the fastsmooth function that tolerates NaNs (Not a Number) in the data file (nanfastsmooth(Y,w,type,tol)) and a version for smoothing angle data (nanfastsmoothAngle(Y,w,type,tol)). Click for animated example.

SegmentedSmooth.m, segmented multiple-width data smoothing function based on the fastsmooth algorithm. The syntax is **`SmoothY = SegmentedSmooth(Y,smoothwidths,type,ends)`**. This function divides Y into several equal-length segments according to the length of the vector 'smoothwidths', then smooths each segment with a smooth of width defined by the sequential elements of vector 'smoothwidths' and smooth type 'type'. Type "help SegmentedSmooth" for examples. DemoSegmentedSmooth.m demonstrates the operation (click for graphic). See page 40.

medianfilter , median-based filter function for eliminating narrow spike artifacts. The syntax is mY=medianfilter(y, Width), where "Width" is the number of points in the spikes that you wish to eliminate. Type "help medianfilter" at the command prompt.

killspikes.m is a threshold-based filter function for eliminating narrow spike artifacts. The syntax is **`fy=killspikes(x,y,threshold,width).`** Each time it finds a positive or negative jump in the data between y(n) and y(n+1) that exceeds "threshold", it replaces the next "width" points of data with a *linearly interpolated segment* spanning x(n) to x(n+width+1), See `killspikesdemo`. Type "`help killspikes`" at the command prompt.

testcondense.m is a script that demonstrates of the effect of boxcar averaging using the condense.m function, which performs a non-overlapping boxcar averaging function, to reduce noise without changing the noise color. Shows that it reduces the measured noise, removing the high frequency components, resulting in a faster fitting execution time and a lower fitting error, but unfortunately *no more accurate measurement of peak parameters*.

SmoothWidthTest.m is a Matlab/Octave script that demonstrates the effect of smoothing on the peak height, random white noise, and signal-to-noise ratio of a noisy peak signal. Produces an animation showing the effect of progressively wider smooth widths, then draws a graph of peak height, noise, and signal-to-noise ratio vs smooth ratio. Click to see gif animation. You can change the peak *shape* and *width* in line 8 and the smooth *type* in line 9: 1=rectangle, 2=triangle, 3=p-spline. The script requires the "gaussian.m" function to be downloaded and placed in the Matlab/Octave path, or you can use any other peak shape function, such as lorentzian.m or rectanglepulse.m, etc.

SmoothExperiment.m, very simple script that demonstrates the effect of smoothing on the position, width, and height of a single Gaussian peak. Requires that the fastsmooth.m and peakfit.m functions be present in the path. See page 55.

smoothdemo.m, self-contained function that compares the performance and speed of four types of smooth operations: (1) sliding-average, (2) triangular, (3) p-spline (equivalent to three passes of a sliding-average), and (4) Savitzky-Golay. These smooth operations are applied to a single noisy Gaussian peak. The peak height of the smoothed peak, the standard deviation of the smoothed noise, and the signal-to-noise ratio are all measured as a function of smooth width. See page 55.

SmoothOptimization.m, script that shows why you do not need to smooth data prior to least-squares curve fitting. It compares the effect of smoothing on the signal-to-noise ratio of peak height of a noisy Gaussian peak, using three different measurement methods. Requires that the fitgauss2.m, gaussfit.m, gaussian.m, and fminsearch.m functions be present in the path. See page 233.

SmoothVsCurvefit.m, comparison of peak height measurement by taking the maximum of the smoothed signal and by curve fitting the original unsmoothed data. Requires peakfit.m and gaussian.m in the path.

DemoSegmentedSmooth.m demonstrates the operation of SegmentedSmooth.m with a signal consisting of noisy variable-width peaks that get progressively wider. Requires SegmentedSmooth.m and gaussian.m in the path.

DeltaTest.m. A simple Matlab/Octave script that demonstrates the shape of any smoothing algorithm can be determined by applying that smooth to a *delta function,* a signal consisting of all zeros except for one point. The result is called the *impulse response function*.

iSignal (page 390) performs several different kinds of smoothing, segmented smoothing, median filtering, and spike removal (as well as differentiation, peak sharpening, least-squares measurements of peak position, height, width, and area, signal and noise amplitudes, frequency spectra in selected regions of the signal, and signal-to-noise ratio of peaks). m-file link: isignal.m. Click here to download the ZIP file "iSignal8.zip". Click for animated example.

The script RealTimeSmoothTest.m demonstrates real-time smoothing, plotting the raw unsmoothed data as a black line and the smoothed data in red. In this case the script pre-calculates simulated data in line 28 and then accesses the data point-by-point in the processing loop (lines 30-51). The total number of data points is controlled by 'maxx' in line 17 (initially set to 1000) and the smooth width (in points) is controlled by 'SmoothWidth' in line 20. Animated graphic.

Data Smoothing Tool (download link: DataSmoothing.mlx) is an interactive Live Script that can apply several types of smoothing to experimental data stored on disk. It can perform spike removal, sliding average smooths with up to 5 passes, Savitsky-Golay and Fourier low-pass filtering, and wavelet denoising (which requires the Matlab Wavelet Toolkit). See page 59.

Differentiation and peak sharpening

deriv, deriv2, deriv3, deriv4, derivxy and secderivxy, simple functions for computing the derivatives of time-series data without smoothing. See page 75.

SmoothDerivative.m combines differentiation and smoothing. The syntax is SmoothedDeriv = SmoothedDerivative(x, y, DerivativeOrder, w, type, ends) where 'DerivativeOrder' determines the derivative order (0 through 5), 'w' is the smooth width, 'type' determines the smooth mode, and 'ends' controls how the "ends" of the signal (the first w/2 points and the last w/2 points) are handled.

SlopeAnimation.m is an animated Matlab/Octave demonstration that shows that the first derivative of a signal is the slope of the tangent to the signal at each point.

sharpen, peak sharpening by the even-derivative method. Syntax is **`SharpenedSignal = sharpen(signal, factor1, factor2, SmoothWidth)`**. See page 88. Related demos: SegmentedSharpen.m, DemoSegmentedSharpen.m (graphic), SharpenedGaussianDemo.m (graphic), SharpenedGaussianDemo4terms.m (graphic), SharpenedLorentzianDemo.m (graphic), SharpenedLorentzianDemo4terms.m.

symmetrize.m converts exponentially broadened peaks into symmetrical peaks by the weighted addition or subtraction of the first derivative. The syntax is ySym = symmetrize(t, y, factor, smoothwidth, type, ends), where t, y are the raw data vectors, 'factor' is the derivative weighting factor, and 'smoothwidth', 'type', 'ends' are the SegmentedSmooth arguments for smoothing the derivative. To perform a *segmented* symmetrization, "factor" and "smoothwidth" can be vectors. (In version 2, only the derivative is smoothed internally, not the entire symmetrized signal). SymmetrizeDemo.m runs all

five examples in the symmetrize.m help file, each in a different figure window.

First derivative symmetrization can be followed by an application of even derivative sharpening for further peak sharpening, as demonstrated for a single exponentially modified Gaussian (EMG) by the self-contained Matlab/Octave demo function EMGplusfirstderivative.m and for an exponentially modified Lorentzian (EML) by EMLplusfirstderivative.m. In both of these demos, Figure 1 shows the symmetrization and Figure 2 shows that the symmetrized peak can be further narrowed by additional 2nd and 4th derivative sharpening. SymmetizedOverlapDemo.m demonstrates the optimization of the first derivative symmetrization for the measurement of the areas of two overlapping exponentially broadened Gaussians. *Double* exponential symmetrization is performed by the function DEMSymm.m. It is demonstrated by the script DemoDEMSymm.m and its two variations (1, 2), which creates two overlapping double exponential peaks from Gaussian originals, then calls the function DEMSymm.m to perform the symmetrization, using a three-level plus-and-minus bracketing technique to help you to determine the best values of the two weighting factors by trial and error. The interactive function *iSignal* (page 390) can perform first derivative symmetrization interactively, with keystrokes to increase and decrease the "factor" while watching the effect on the signal. The script AsymmetricalOverlappingPeaks.m demonstrates the use of first-derivative symmetrization and curve fitting to analyze a complex "mystery" peak. See page 366).

ProcessSignal, a Matlab/Octave command-line multi-purpose function that includes smoothing, differentiation, peak sharpening, and median filtering on the time-series data set x,y (column or row vectors). Like iSignal, without the plotting and interactive keystroke controls. Type "help ProcessSignal". It returns the processed signal as a vector that has the same shape as x, regardless of the shape of y. The syntax is **`Processed= Processed=ProcessSignal(x, y, DerivativeMode, w, type, ends, Sharpen, factor1, factor2, Symize, Symfactor, SlewRate, MedianWidth).`**

derivdemo1.m, a function that demonstrates the basic shapes of derivatives. See page 64.

DerivativeShapeDemo.m is a function that demonstrates the first derivatives of 16 different peak shapes. (Graphic)

derivdemo2.m, a function that demonstrates the effect of peak width on the amplitude of derivatives. See page 64.

derivdemo3.m, a function that demonstrates the effect of smoothing on the *first* derivative of a noisy signal. See page 64.

derivdemo4.m, a function that demonstrates the effect of smoothing on the *second* derivative of a noisy signal. See page 64.

DerivativeDemo.m is a self-contained Matlab/Octave demo function that uses ProcessSignal.m and plotit.m to demonstrate an application of differentiation to the quantitative analysis of a peak buried in an unstable background (e.g. as in various forms of spectroscopy). The object is to derive a measure of peak amplitude that varies linearly with the actual peak amplitude and is minimally affected by the background and the noise. To run it, just type DerivativeDemo at the command prompt. You can change several of the internal variables (e.g., Noise, BackgroundAmplitude) to make the problem harder or easier. Note that, even though the magnitude of the derivative is numerically smaller than the original signal (because it has different units), the signal-to-noise ratio of the derivative is better, and the derivative signal is linearly proportional to the actual peak height, despite the interference of large background variations and random noise. See page 75.

iSignal or isignaloctave (page 390) is an interactive function that includes *differentiation and*

smoothing for time-series signals, up to the 5th derivative, automatically including the required type of smoothing. Simple keystrokes allow you to adjust the smoothing parameters (smooth type, width, and ends treatment) while observing the effect on your signal dynamically. It can also perform *interactive symmetrization* and *sharpening* of exponentially broadened peaks by the first-derivative addition technique (page 83). Click here to download the ZIP file "iSignal8.zip". Click for animated example.

demoisignal.m for Matlab is a self-running script that demonstrates several of the features of iSignal (and requires that the latest version of iSignal, and version 6 of plotit.m, be present in your Matlab path). Demonstrates panning and zooming, smoothing, differentiation, frequency spectrum, peak measurement, and derivative spectroscopy calibration (in conjunction with plotit.m version 6).

iSignalDeltaTest is a Matlab/Octave script that demonstrates the frequency response (power spectrum) of the smoothing and differentiation functions of iSignal by a applying them to a delta function. Change the smooth type, smooth width, and derivative order and see how the power spectrum changes.

The script RealTimeSmoothFirstDerivative.m demonstrates real-time smoothed differentiation, using a simple adjacent-difference algorithm (line 47) and plotting the raw data as a black line and the first derivative data in red. The script RealTimeSmoothSecondDerivative.m computes the smoothed *second* derivative by using a central difference algorithm (line 47). Both scripts pre-calculate the simulated data in line 28 and then access the data point-by-point in the processing loop (lines 31-52). In both cases the maximum number of points is set in line 17 and the smooth width is set in line 20.

The script RealTimePeakSharpening.m demonstrates real-time peak sharpening using the second derivative technique. It uses pre-calculated simulated data in line 30 and then accesses the data point-by-point in the processing loop (lines 33-55). In both cases the maximum number of points is set in line 17 and the smooth width is set in line 20 and the weighting factor (K1) is set in line 21. In this example the smooth width is 101 points, which accounts for the delay in the sharpened peak compared to the original.

Harmonic Analysis

FrequencySpectrum.m (syntax **`fs=FrequencySpectrum(x,y)`**) returns real part of the Fourier power spectrum of x,y as a matrix.

PlotFrequencySpectrum.m plots the frequency spectrum or periodogram of the signal x,y on linear or log coordinates. The syntax is **`PowerSpectrum= PlotFrequencySpectrum(x, y, plotmode, XMODE, LabelPeaks)`**. Type "help PlotFrequencySpectrum" for details. Try this example:

```
x= [0:.01:2*pi]'; y=sin(200*x)+randn(size(x));
subplot(2,1,1); plot(x,y); subplot(2,1,2);
PowerSpectrum=PlotFrequencySpectrum(x,y,1,0,1);
```

CompareFrequencySpectrum.m. A script that compares two signals (upper panel) and their frequency spectra (lower panel) with the original signal shown in blue and the modified signal in green. plotmode: =1: linear, =2: semilog X, =3: semilog Y; =4: log-log). XMODE: =0 for frequency Spectrum (x is frequency); =1 for periodogram (x is time). Define the signal modification in line 15. You can load a signal stored in .mat format or create a simulated signal for testing. You must have PlotFrequencySpectrum.m in the path.

PlotSegFreqSpect.m is a segmented Fourier spectrum (syntax PSM=(x,y, NumSegments, MaxHarmonic, LogMode)) breaks y into 'NumSegments' equal length segments, multiplies each by an apodizing Hanning window, computes the power spectrum of each segment, returns the power spectrum matrix (PSM), and plots the result of the first 'MaxHarmonic' Fourier components as a

contour plot. See page 105 for an example of its application to a signal that is completely buried in an excess of noise and interfering signals.

iSignalDeltaTest is a Matlab/Octave script that demonstrates the *frequency response* (power spectrum) of the smoothing and differentiation functions of iSignal by a applying them to a delta function. Change the smooth type, smooth width, and derivative order and see how the power spectrum changes.

SineToDelta.m. A demonstration animation (animated graphic) showing the waveform and the power spectrum of a rectangular pulsed sine wave of variable duration (whose power spectrum is a "sinc" function) changing continuously from a pure sine wave at one extreme (where its power spectrum is a delta function) to a single-point pulse at the other extreme (where its power spectrum is a flat line). GaussianSineToDelta.m is similar, except that it shows a *Gaussian* pulsed sine wave, whose power spectrum is a Gaussian function, but which is the same at the two extremes of pulse duration (animated graphic).

isignal.m or isignaloctave.m, (page 390) is a multi-purpose interactive signal processing that includes a **Frequency Spectrum mode**, toggled on and off by the **Shift-S** key. It computes frequency spectrum of the segment of the signal displayed in the upper window and displays it in the lower window (in red). You can use the pan and zoom keys to adjust the region of the signal to be viewed or press **Ctrl-A** to select the entire signal. Press **Shift-S** again to return to the normal mode. See page 94 for a relevant example. Click for animated example.

iPower, a keyboard-controlled interactive power spectrum demonstrator, useful for teaching and learning about the power spectra of different types of signals and the effect of signal duration and sampling rate. Single keystrokes allow you to select the type of signal (12 different signals included), the total duration of the signal, the sampling rate, and the global variables f1 and f2 which are used in different ways in the different signals. When the **Enter** key is pressed, the signal (y) is sent to the Windows WAVE audio device. Press **K** to see a list of all the keyboard commands. (m-file link: ipower.m). Slideshow of examples.

The script RealTimeFrequencySpectrumWindow.m computes and plots the Fourier frequency spectrum of a signal. It loads the simulated real-time data from a ".mat file" (in line 31) and then accesses the data point-by-point in the processing 'for' loop. A critical variable in this case is "WindowWidth" (line 37), the number of data points taken to compute each frequency spectrum. If the data stream is an audio signal, it is also possible to play the sound through the computer's sound system synchronized with the display of the frequency spectra (set "PlaySound" to 1).

Fourier convolution and deconvolution

ExpBroaden, exponential broadening function. Syntax is **`yb = ExpBroaden(y,t).`** Convolutes the vector *y* with an exponential decay of time constant *t*. Mentioned on pages 34 and 416.

GaussConvDemo.m, a script that demonstrates that a Gaussian of unit height, Fourier convoluted with a zero-centered Gaussian of the same width is a Gaussian with a height of 1/sqrt(2) and a width of sqrt(2) and of equal area to the original Gaussian. When you run this script, the top panel shows the convolution and the bottom panel shows how to recover the original y from the convoluted result (graphic). You can optionally add noise in line 9 to show how convolution smooths the noise and how Fourier deconvolution restores it. Requires gaussian.m in the path.

CombinedDerivativesAndSmooths.txt. Convolution coefficients for computing the first through fourth derivatives, with rectangular, triangular, and P-spline smooths.

Convolution.txt, simple examples of whole-number convolution vectors for smoothing and differentiation.

deconvolutionexample.m, a simple example script that demonstrates the use of the Matlab Fourier deconvolution 'deconv' function. See page 118.

DeconvDemo.m, a Fourier deconvolution demo script with a signal containing four Gaussians broadened by an exponential function (graphic). DeconvDemo2.m is a similar script for a single Gaussian (graphic).DeconvDemo3.m demonstrates deconvolution of a *Gaussian* convolution function from a rectangular pulse (animated graphic). DeconvDemo4.m (animated graphic) demonstrates "self-deconvolution" applied a signal consisting of a Gaussian peak that is broadened by the measuring instrument, and an attempt to recover the original peak width. DeconvDemo5.m (graphic) shows an attempt to resolve *two* closely-spaced underlying peaks that are *completely unresolved* in the observed signal. See page 306. Variation of this include versions with Lorentzian peaks and one with a triangular convolution function.

deconvgauss.m. ydc=deconvgauss(x,y,w) deconvolutes a Gaussian function of width 'w' from vector y, returning the deconvoluted result.

LorentzianSelfDeconvDemo.m. Demonstration of Lorentzian self-deconvolution. Requires lorentzian, halfwidth, and fastsmooth functions.

deconvexp.m. ydc=deconvexp(y,tc) deconvolutes an exponential function of time constant 'tc' from vector y, returning the deconvoluted result.

SegExpDeconv(x,y,tc) is a segmented version of deconvexp.m. It divides x,y into a number of equal-length segments defined by the length of the vector 'tc', then each segment is deconvoluted with an exponential decay of the form exp(-x./t) where 't' is the corresponding element of the vector 'tc'. *Any number and sequence of t values can be used.* Useful when the peak width and/or exponential tailing of peaks varies across the signal duration. SegExpDeconvPlot.m is the same except that it plots the original and deconvoluted signals and *shows the divisions between the segments by vertical magenta lines*. SegGaussDeconv.m and SegGaussDeconvPlot.m are the same except that they perform a symmetrical (zero-centered) Gaussian deconvolution. SegDoubleExpDeconv.m and SegDoubleExpDeconvPlot.m perform a symmetrical (zero-centered) exponential deconvolution.

P=convdeconv(x,y,vmode,smode,vwidth,DAdd), for Matlab or Octave, performs Gaussian, Lorentzian, or exponential convolution and deconvolution of the signal in x,y.

iSignal 8.3 (page 390) has a **Shift-V** keypress that displays the menu of Fourier convolution and deconvolution operations that allow you to convolute a Gaussian or exponential function with the signal, or to deconvolute a Gaussian or exponential function from the signal and allows you to adjust the width interactively. Click here to download the ZIP file "iSignal8.zip"Data convolution tool. The interactive Live Script DeconvoluteData.mlx can perform Fourier self-deconvolution on your own data stored in disk. See page 126.

Fourier Filter

FouFilter, Fourier filter function, with variable band-pass, low-pass, high-pass, or notch (band reject). The syntax is **`[ry,fy,ffilter,ffy] =FouFilter(y, samplingtime, centerfrequency, frequencywidth, shape, mode)`**. Version 2, March 2019. See page 128.

SegmentedFouFilter.m is a segmented version of FouFilter.m that applies different center frequencies and widths to different segments of the signal. The syntax is the same as FouFilter.m except that the

two input arguments “centerFrequency” and “FilterWidth” must be vectors with the values of centerFrequency of filterWidth for each segment. The signal is divided equally into several segments determined by the length of centerFrequency and filterWidth, which must be equal in length. Type “help SegmentedFouFilter” for help and examples.

iFilter, interactive Fourier filter. (m-file link: ifilter.m), which uses the pan and zoom keys to control the center frequency and the filter width (page 391). Click here for animated example. Select from low-pass, high-pass, band-pass, band-reject, harmonic comb-pass, or harmonic comb-reject filters. Click here to watch or download an mp4 video of iFilter filtering a noisy Morse code signal, with sound (watch the title of the figure as the video plays). *The Octave versions use the < and > keys (with and without shift).*

MorseCode.m is a script that uses iFilter to demonstrate the abilities and limitations of Fourier filtering. It creates a pulsed fixed frequency sine wave that spells out “SOS” in Morse code (dit-dit-dit/dah-dah-dah/dit-dit-dit), adds random white noise so that the SNR is very poor (about 0.1 in this example), then uses a Fourier bandpass filter tuned to the signal frequency, to isolate the signal from the noise. As the bandwidth is reduced, the signal-to-noise ratio begins to improve and the signal emerges from the noise until it becomes clear, but if the bandwidth is too narrow, the step response time is too slow to give distinct “dits” and “dahs”. Use the “?” and “ " ” keys to adjust the bandwidth. (The step response time is inversely proportional to the bandwidth). Press 'P' or the Spacebar to hear the sound. You must install iFilter.m in the Matlab path. Watch on YouTube at https://youtu.be/agjs1-mNkmY. (look at the explanation in the title of the figure as the video plays).

TestingOneTwoThree.wav is a 1.6 sec duration audio recording of the spoken phrase "Testing, one, two, three", recorded at a sampling rate of 44000 Hz and saved in WAV format. When loaded into iFilter(`v=wavread('TestingOneTwoThree.wav');`), set to bandpass mode, and tuned to a narrow segment that is well above the frequency range of most of the signal, it might seem as if though this passband would miss most of the frequency components in the signal, yet even in this case the speech is intelligible, demonstrating the remarkable ability of the ear-brain system to make do with a highly compromised signal. Press P or space to hear the filter's output. Different filter settings will change the timbre of the sound. See page 391. Click for graphic. You can also use the Live Script version FourierFilterTool.mlx; See iFilterTesting123.m.

The script RealTimeFourierFilter.m is a demonstration of a real-time Fourier filter. Like the other real-time signal processing scripts, this one pre-computes a simulated signal starting in line 38, then access the data point-by-point (lines 56, 57), and divides up the data stream into segments to compute each filtered section. In this demonstration, a bandpass filter is used to detect a 500 Hz ('f' in line 28) sine wave that occurs in the middle third of a very noisy signal (line 32), from about 0.7 sec to 1.3 sec. The filter center frequency (CenterFrequency) and width (FilterWidth) are set in lines 46 and 47.

Wavelets and wavelet denoising

Morelet.m demonstrates the application of the wavelet transform to unravel the components of a complicated signal. Code written by Michael X. Cohen, in “A better way to define and describe Morlet wavelets for time-frequency analysis”, *NeuroImage*, Volume 199, 1 October 2019, Pages 81-86. MorletExample2.m creates and analyzes the “buried peaks” signal consisting of three components: a pair of weak Gaussian peaks which are the desirable signal components, a strong interference by a variable-frequency sine wave, and an excess of random white noise. The Gaussian peaks are invisible in the raw signal.

Peak area measurement

PerpDropAreas.m **`[AreaVector]=PerpDropAreas(x,y,startx,endx,MaxVector)`** measures the peak areas of the peaks in x, y, starting an x value of startX and ending at endX, with specified peak positions in the vector MaxVector, which can be of any length. Uses the halfwaypoint method. Returns the areas in the vector PDMeasAreas and the midpoint indices in the optional second output argument.

HeightAndArea.m is a demonstration script that uses measurepeaks.m to measure the peaks in computer-generated signals consisting of a series of Gaussian peaks with gradually increasing widths that are superimposed in a curved baseline plus random white noise. It plots the signal and the individual peaks and compares the actual peak position, heights, and areas of each peak to those measured by measurepeaks.m using the absolute peak height, peak-valley difference, perpendicular drop, and tangent skim methods. Prints out a table of the relative percent difference between the actual and measured values for each peak and the average error for all peaks.

measurepeaks.m automatically detects peaks in a signal, similar to findpeaksSG. It returns a table of peak number, position, absolute peak height, peak-valley difference, perpendicular drop area, and tangent skim area of each peak. It can plot the signal and the individual peaks if the last (7th) input argument is 1. Type "help measurepeaks" and try the seven examples there or run HeightAndArea.m to run a test of the accuracy of peak height and area measurement with signals that have multiple peaks with noise, background, and some peak overlap. The script testmeasurepeaks.m will run all of the examples with a 1-second pause between each (requires measurepeaks.m and gaussian.m in the path).

The script SharpenedOverlapDemo.m (graphic) demonstrates the effect of sharpening on perpendicular drop area measurements of two overlapping Gaussians peaks with adjustable height, separation, and width, calculating the percent different between the area measured on the overlapping peak signal compared to the true areas of the isolated peaks.

SharpenedOverlapCalibrationCurve.m is a script that simulates quantitative measurement of mixtures of *three* overlapping Gaussian peaks. Even-derivative sharpening (the red line in the signal plots) is used to improve the resolution of the peaks to allow perpendicular drop area measurement. A straight line is fit to the calibration curve and the R^2 is calculated, to demonstrate (1) the linearity of the response, and (2) the independence of the overlapping adjacent peaks. Must have gaussian.m, derivxy.m, autopeaks.m, val2ind.m, halfwidth.m, fastsmooth.m, and plotit.m in the path.

ComparePDAreas.m compares the effect of digital processing on the areas of a set of peaks measured by the perpendicular drop method. Syntax is **`[P1,P2,coef,R2] = ComparePDAreas(x, orig, processed, PeakSensitivity)`**, where x=independent variable (e.g., time); orig = original signal y values; processed = processed signal y values; P1 = peak table of original signal; P2 = peak table of processed signal; PeakSensitivity = approximate number of peaks that would fit into the entire x-axis range (larger numbers > more peak detected). Displays a scatter plot of original areas vs processed areas for each peak and returns the peak tables, P1 and P2 respectively, and the slope, intercept, and R^2 values, which should ideally be 1,0, and 1, if the processing has no effect at all on peak area.

iSignal (page 390) is my downloadable Matlab function that performs various signal processing functions described in this tutorial, including one-at-a-time manual measurement of peak area using Simpson's Rule and the perpendicular drop method. Click to view or right-click > Save link as... here, or you can download the ZIP file with sample data for testing. The animated GIF iSignalAreaAnimation.gif (click to view) shows iSignal applying the perpendicular drop method to a series of four peaks of equal area. (Look at the bottom panel to see how the measurement intervals, marked by the vertical dotted magenta lines, are positioned at the valley minimum on either side of

each of the four peaks). It also has a built-in peak fitter, activated by the **Shift-F** key, based on peakfit.m, that measures the areas of overlapping peak of known shape. There is also an *automatic* peak finding function based on the autopeaks function, activated by the **J** or **Shift-J** keys, which displays a table listing the peak number, position, absolute peak height, peak-valley difference, perpendicular drop area, and tangent skim area of each peak in the signal.

peakfit, a command-line function for multiple peak fitting by iterative non-linear least-squares. It measures the peak position, height, width, and area of overlapping peaks, and it has several ways to correct for non-zero baselines. For best results, it requires that the peak shape of your peaks be among those listed here.

PeakCalibrationCurve.m is a Matlab/Octave simulation of the calibration of a flow injection or chromatography system that produces signal peaks that are related to an underlying concentration or amplitude ('amp'). The measurepeaks.m function is used to determine the absolute peak height, peak-valley difference, perpendicular drop area, and tangent skim area. The Matlab/Octave script PeakShapeAnalyticalCurve.m shows that, for a single isolated peak whose shape is constant and independent of concentration, if the wrong model shape is used, the peak heights measured by curve fitting will be inaccurate, but that error will be exactly the same for the unknown samples and the known calibration standards, so the error will "cancel out" and the measured concentrations will be accurate, provided you use the same inaccurate model for both the known standards and the unknown samples. See page 337.

PowerTransformTest.m is a simple script that demonstrates the power method of peak sharpening to aid in reducing in peak overlap. The scripts PowerMethodGaussian.m and PowerMethodLorentzian.m compare the power methods to deconvolution, for Gaussian and Lorentzian peak, respectively. PowerMethodCalibrationCurve is a variant of PeakCalibrationCurve.m that evaluates the power method in the context of a flow injection or chromatography measurement. The self-contained function PowerMethodDemo.m demonstrates the power method for measuring the area of small shouldering peak that is partly overlapped by a much stronger interfering peak (Graphic). It also demonstrates the effect of random noise, smoothing, and any uncorrected background under the peaks.

AsymmetricalAreaTest.m. Test of accuracy of peak area measurement methods for an asymmetrical peak, comparing (A) Gaussian estimation, (B) triangulation, (C) perpendicular drop method, and curve fitting by (D) exponentially broadened Gaussian, and (E) two overlapping Gaussians. Must have the following functions in the Matlab/Octave path: gaussian.m, expgaussian.m, findpeaksplot.m, findpeaksTplot.m, autopeaks.m, and peakfit.m. Related script AsymmetricalAreaTest2.m compares the standard deviations of those same methods with randomized noise samples.

SumOfAreas.m. Demonstrates that even drastically non-Gaussian peaks can be fit with up to five overlapping Gaussian components, and that the total area of the components approaches the area under the non-Gaussian peak as the number of components increases (graphic). In most cases only a few components are necessary to obtain a good estimate of the peak area.

Linear Least-squares

TestLinearFit effect of number of points.txt. Effect of sample size on least-square error estimates by Monte Carlo Simulation, Algebraic propagation-of-errors, and the bootstrap method, using the Matlab script TestLinearFit.m.

LeastSquaresCode.txt. Simple pseudocode for calculating the first-order least-square fit of y vs x, including the Slope and Intercept and the predicted standard deviation of the slope (SDslope) and intercept (SDintercept).

CalibrationQuadraticEquations.txt. Simple pseudocode for calculating the second order least-square fit of y vs x, including the constant, x, and x2 terms.

plotit, version 2, (previously named 'plotfit'), is a function for plotting x,y data in matrices or in separate vectors. It optionally fits the data with a polynomial of order *n* if *n* is included as the third input argument. In **version 6** the syntax is **`[coef, RSquared, StdDevs] =plotit(x,y) or plotit(x,y,n)`** or optionally **`plotit(x, y, n, datastyle, fitstyle)`**, where **`datastyle`** and **`fitstyle`** are optional strings specifying the line and symbol style and color, in standard Matlab convention. For example, **`plotit(x,y,3,'or','-g')`** plots the data as red circles and the fit as a green solid line (the default is red dots and a blue line, respectively). Plotit returns the best-fit coefficients **`'coeff'`**, in decreasing powers of x, the standard deviations of those coefficients **`'StdDevs'`** in the same order, and the R-squared value. Type "help plotit" at the command prompt for syntax options. See page 181. This function works in Matlab or Octave and has a built-in bootstrap routine that computes coefficient error estimates (STD and % RSD of each coefficient) by the bootstrap method and returns the results in the matrix "BootResults" (of size 5 x polyorder+1). The calculation is triggered by including a 4th output argument, e.g. **`[coef, RSquared, StdDevs, BootResults]= plotit(x,y,polyorder)`**. This works for any positive integer polynomial order. The variation plotfita animates the bootstrap process for instructional purposes. The variation logplotfit plots and fits log(x) vs log(y), for data that follows a power law relationship or that covers a very wide numerical range.

RSquared.m Computes the R^2 (Rsquared or correlation coefficient) in both Matlab and Octave. Syntax RS=RSquared(polycoeff, x,y).

trypoly(x,y) fits the data in x,y with a series of polynomials of degree 1 through length(x)-1 and returns the coefficients of determination (R^2) of each fit as a vector, allowing you to evaluate how polynomials of various orders fit your data. To plot as a bar graph, write `bar(trypoly(x,y)); xlabel('Polynomial Order'); ylabel('Coefficient of Determination (R2)')`. Click for an example. See related function testnumpeaks.m.

trydatatrans(x,y,polyorder) tries 8 different simple data transformations on the data x,y, fits the transformed data to a polynomial of order 'polyorder', displays results graphically in 3 x 3 array of small plots and returns all the R^2 values in a vector.

LinearFiMC.m, a script that compares standard deviation of slope and intercept for a first order least-squares fit computed by random-number simulation of 1000 repeats to predictions made by closed-form algebraic equations. See page 167.

TestLinearFit.m, a script that compares standard deviation of slope and intercept for a first-order least-squares fit computed by random-number simulation of 1000 repeats to predictions made by closed-form algebraic equations and to the bootstrap sampling method. Several different noise models can be selected by commenting/uncommenting the code in lines 20-26. See page 167.
GaussFitMC.m, a function that demonstrates Monte Carlo simulation of the measurement of the peak height, position, and width of a noisy x,y Gaussian peak. See page 173.

GaussFitMC2.m, a function that demonstrates measurement of the peak height, position, and width of a noisy x,y Gaussian peak, comparing the gaussfit parabolic fit to the fitgaussian iterative fit. See page 173.

SandPfrom1950.mat is a MAT file containing the daily value of the S&P 500 stock market index vs time from 1950 through September of 2016. These data are used by FitSandP.m a Matlab/Octave script that performs a least-squares fit of the compound interest equation to the daily value, V, of the S&P 500 stock market index vs time, T, from 1950 through September of 2016, by two methods:
(1) the iterative curve fitting method, and (2) by taking the logarithm of the values and fitting those to a

straight line. SnPsimulation.m. Matlab/Octave script that simulates the S&P 500 stock market index by adding proportional random noise to data calculated by the compound interest equation with a known annual percent return, then fits the equation to that noisy synthetic data by the two methods above. See page 326.

gaussfit.m function **`[Height, Position, Width]=gaussfit(x,y)`**. Takes the natural log of y, fits a parabola (quadratic) to the (x, ln(y)) data, then calculates the position, width, and height of the Gaussian from the three coefficients of the quadratic fit.

lorentzfit.m function **`[Height, Position, Width]=lorentzfit(x,y)`**. Takes the reciprocal of y, fits a parabola (quadratic) to the (x,1/y) data, then calculates the position, width, and height of the Lorentzian from the three coefficients of the quadratic fit.

OverlappingPeaks.m is a demo script that shows how to use gaussfit.m as a quick way to measure two overlapping partially Gaussian peaks. It requires careful selection of the optimum data regions around the top of each peak (lines 15 and 16). Try changing the relative position and height of the second peak or adding noise (line 3) and see how it affects accuracy. This function needs the gaussian.m and gaussfit.m functions in the path. Iterative methods work much better in such cases, but they are slower.

Peak Finding and Measurement

allpeaks.m. **`allpeaks(x,y)`** A super-simple peak detector for x,y, data sets that lists every *y* value that has lower *y* values on both sides. Allvalleys.m is the same for valleys, lists every *y* value that has *higher y* values on both sides. A related version, allpeaksw.m, also estimates the width of the peaks.

peaksat.m. (**peaks a**bove **t**hreshold) lists every y value that (a) has lower y values on both sides and (b) is above the specified threshold. Returns a 2 by *n* matrix P with the x and y values of each peak, where *n* is the number of detected peaks. A related version, peaksatw.m, also estimates the width of the peaks. The variation **peaksatG.m** ("Peaks **A**bove **T**hreshold/**G**aussian") additionally performs a least-squares fit to the top of each detected peak to estimate its width and area.

findpeaksx.m, **`P=findpeaksx(x,y, SlopeThreshold, AmpThreshold, SmoothWidth, FitWidth, smoothtype)`** is a simple command-line function to locate and count the positive peaks in noisy data sets. It is an alternative to the findpeaks function in the Signal Processing Toolkit. It detects peaks by looking for downward zero-crossings in the smoothed first derivative that exceed SlopeThreshold and peak amplitudes that exceed AmpThreshold and returns a list (in matrix P) containing the peak number and the position and height of each peak. It can find and count over 10,000 peaks per second in very large signals. Type "help findpeaksx.m". See PeakFindingandMeasurement.htm. The variant findpeaksxw.m additionally measures the width of the peaks. See the demonstration script demofindpeaksxw.m.

findpeaksG.m and findvalleys.m automatically find the peaks or valleys in a signal and measure their position, height, width, and area by curve fitting. The syntax is **`P= findpeaksG(x, y, SlopeThreshold, AmpThreshold, SmoothWidth, FitWidth, smoothtype)`**. It returns a matrix containing the peak parameters for each detected peak. For peak of Lorentzian shape, use findpeaksL.m instead. See page 234. There are many variations and extensions based on this basic function. See page 236.

findpeaksplot.m is a simple variant of findpeaksG.m that also plots the x,y data and numbers the peaks on the graph (if any are found). Syntax: **`findpeaksplot(x, y, SlopeThreshold, AmpThreshold, SmoothWidth, FitWidth, smoothtype)`**

OnePeakOrTwo.m is a demo script that creates a signal that might be interpreted as either one peak at x=3 on a curved baseline or as two peaks at x=.5 and x=3, depending on context. In this demo, the findpeaksG.m function was called twice, with two different values of SlopeThreshold to demonstrate.

iPeak (page 253), or its Octave version ipeakoctave.m, automatically finds and measures multiple peaks in a signal. (m-file link: ipeak.m). Check out the Animated step-by-step instructions. The ZIP file ipeak8.zip contains several demo scripts (ipeakdemo.m, ipeakdemo1.m, etc.) that illustrate various aspects of the iPeak function and how it can be used effectively. testipeak.m is a script that tests for the proper installation and operation of iPeak by running quickly through all eight examples and six demos for iPeak. Assumes that ipeakdata.mat has been loaded into the Matlab workspace. Click for slideshow of examples. The syntax is **`P=ipeak(DataMatrix, PeakD, AmpT, SlopeT, SmoothW, FitW, xcenter, xrange, MaxError, positions, names)`**

findpeaksSG.m is a *segmented* variant of findpeaksG with the same syntax, except that the peak detection parameters can be *vectors*, dividing up the signal into regions optimized for peaks of different widths. The syntax is **`P = findpeaksSG(x, y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, smoothtype)`**. This works better than findpeaksG when the peak widths vary greatly over the duration of the signal. The script TestPrecisionFindpeaksSG.m demonstrates the application. Graphic. See page 333.

findpeaksSGw.m is like the above except that is uses *wavelet denoising* (page 136) instead of smoothing. It takes the wavelet level rather than the smooth width as an input argument. The script TestPrecisionFindpeaksSGvsW.m compares the precision and accuracy for peak position and height measurement.

autofindpeaks.m (and autofindpeaksplot.m) are similar to findpeaksSG.m except that you can *leave out the peak detection parameters* and just write "autofindpeaks(x,y)" or autofindpeaks(x,y,*n*) where *n* is the peak capacity, roughly the number of peaks that would fit into that signal record (greater *n* looks for many narrow peaks. Smaller *n* looks for fewer wider peaks). It also prints out the input argument list for use with any of the findpeaks... functions. In version 1.1, you can call autofindpeaks with the output arguments [P,A] and it returns the calculated peak detection parameters as a 4-element row vector A, which you can then pass on to other functions such as measurepeaks, effectively giving that function the ability to calculate the peak detection parameters from a single number n . For example:

```
x=[0:.1:50];
y=5+5.*sin(x)+randn(size(x));
[P,A]=autofindpeaks(x,y,3);
P=measurepeaks(x,y,A(1),A(2),A(3),A(4),1);
```

Type "help autofindpeaks" and run the examples. The script testautofindpeaks.m runs all the examples in the help file, additionally plotting the data and numbering the peaks (like autofindpeaksplot.m). Graphic animation.

[M,A]=autopeaks.m and autopeaksplot.m. Peak detection and height and area measurement for peaks of arbitrary shape in x,y time series data. The syntax is **`[P, DetectionParameters] = autofindpeaks(x, y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, smoothtype)`**, but like autofindpeaks.m, the peak detection parameters SlopeThreshold, AmpThreshold, smoothwidth peakgroup, and smoothtype can be omitted and the function will calculate estimated initial values. Uses the measurepeaks.m algorithm for measurement, returning a table in the matrix M containing the peak number, position, absolute peak height, peak-valley difference, perpendicular drop area, and tangent skim area of each peak. Optionally returns the peak detection parameters that it calculates in vector A. Using the simple syntax M=autopeaks(x,y) works well in some cases, but if not try M=autopeaks(x,y,*n*), using different values of *n* (roughly the number

of peaks that would fit into the signal record) until it detects the peaks that you want to measure. For the most precise control over peak detection, you can specify all the peak detection parameters by typing M=autopeaks(x,y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup); autopeaksplot.m is the same but it also plots the signal and the individual peaks (in blue) with the maximum (red circles), valley points (magenta), and tangent lines (cyan) marked. The script testautopeaks.m runs all the examples in the autopeaks help file, with a 1-second pause between each one, printing out results in the command window and additionally plotting and numbering the peaks (Figure window 1) and each individual peak (Figure window 2). It requires gaussian.m and fastsmooth.m in the path. iSignal (page 390) has a peak finding function based on the autopeaks function, activated by the **J** or **Shift-J** keys, which displays a table of peak number, position, absolute peak height, peak-valley difference, perpendicular drop area, and tangent skim area of each peak in the signal.

findpeaksG2d.m is a variant of findpeaksSG that can be used to locate the positive peaks *and shoulders* in a noisy x-y time series data set. Detects peaks in the negative of the *second* derivative of the signal, by looking for downward slopes in the *third* derivative that exceed SlopeThreshold. See TestFindpeaksG2d.m. Syntax: **`P = findpeaksG2d(x, y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, smoothtype)`**

measurepeaks.m automatically detects peaks in a signal, like findpeaksSG. **`M = measurepeaks(x, y, SlopeThreshold, AmpThreshold, SmoothWidth, FitWidth, plots).`** It returns a table M of peak number, position, absolute peak height, peak-valley difference, perpendicular drop area, and tangent skim area of each peak. It can plot the signal and the individual peaks if the last (7th) input argument is 1. Type “help measurepeaks” and try the seven examples there or run HeightAndArea.m to run a test of the accuracy of peak height and area measurement with signals that have multiple peaks with noise, background, and some peak overlap. Generally, its values for perpendicular drop area are best for peaks that have no background, even if they are slightly overlapped, whereas its values for tangential skim area are better for isolated peaks on a straight or slightly curved background. Note: this function uses smoothing (specified by the SmoothWidth input argument) only for peak *detection*. It performs measurements on the *raw unsmoothed* y data. In some cases, it may be beneficial to smooth the y data yourself before calling measurepeaks.m, using any smooth function of your choice. The script testmeasurepeaks.m will run all the examples in the measurepeaks help file with a 1-second pause between each (requires measurepeaks.m and gaussian.m in the path). Graphic animation. The related functions wmeasurepeaks.m and testwmeasurepeaks.m utilize *wavelet denoising* (page 136) rather than smoothing.

findpeaksT.m and findpeaksTplot.m are variants of findpeaks that measure the peak parameters by constructing a triangle around each peak with sides tangent to the sides of the peak. Graphic example.

findpeaksb.m is a variant of findpeaksG.m that more accurately measures peak parameters by using iterative least-square curve fitting based on peakfit.m. This yields better peak parameter values than findpeaks alone, because it fits the entire peak, not just the top part, and because it has provision for 33 different peak shapes and for background subtraction (linear or quadratic). Works best with isolated peaks that do not overlap. Syntax is **`P = findpeaksb(x,y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, smoothtype, window, PeakShape, extra, BASELINEMODE).`** The first seven input arguments are exactly the same as for the findpeaksG.m function. If you have been using findpeaks or iPeak (page 253) to find and measure peaks in your signals, you can use those same input argument values for findpeaksb.m. The demonstration script DemoFindPeaksb.m shows how findpeaksb3 works with multiple overlapping peaks. Type "help findpeaksb" at the command prompt. See PeakFindingandMeasurement.htm. Compare this to the related findpeaksfit.m and findpeaksb3, next. Click for slideshow of examples.

findpeaksSb.m is a segmented variant of findpeaksb.m. It has the same syntax as findpeaksb.m, **`P = findpeaksb(x, y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, smoothtype, window, PeakShape, extra, NumTrials, BASELINEMODE)`**, except that the input arguments SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, window, width, PeakShape, extra, NumTrials, BaselineMode, and fixedparameters, can all optionally be scalars or vectors with one entry for each segment, in the same manner as findpeaksSG.m. It returns a matrix P listing the peak number, position, height, width, area, percent fitting error and "R2" of each detected peak. DemoFindPeaksSb.m demonstrates this function by creating a series of Gaussian peaks whose widths increase by a factor of 25-fold and that are superimposed in a curved baseline with random white noise that increases gradually. Four segments are used, changing the peak detection and curve fitting values so that all the peaks are measured accurately. Graphic. Printout. See page 333.

findpeaksb3.m is a variant of findpeaksb.m that fits each detected peak *together with the previous and following peaks* found by findpeaksG.m. It deals better with overlapping peaks than findpeaksb.m does, and it handles larger numbers of peaks better than findpeaksfit.m, but *it fits only those peaks that are found* by findpeaks. The syntax is **`P=findpeaksb3(x,y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, smoothtype, PeakShape, extra, NumTrials, BASELINEMODE, ShowPlots)`**. The first seven input arguments are exactly the same as for the findpeaksG.m function. If you have been using findpeaks or iPeak (page 253) to find and measure peaks in your signals, you can use those same input argument values for findpeaksb3.m. The demonstration script DemoFindPeaksb3.m shows how findpeaksb3 works with multiple overlapping peaks.

findpeaksfit.m is essentially a serial combination of findpeaksG.m and peakfit.m. It uses the number of peaks found by findpeaks and their peak positions and widths as input for the peakfit.m function, which then fits the entire signal with the specified peak model. This deals with non-Gaussian and overlapped peaks better than findpeaks alone. However, it fits only those peaks that are found by findpeaks. The syntax is **`[P, FitResults, LowestError, BestStart, xi, yi] = findpeaksfit(x, y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, smoothtype, peakshape, extra, NumTrials, BaselineMode, fixedparameters, plots)`**. The first seven input arguments are exactly the same as for the findpeaksG.m function. If you have been using findpeaks or iPeak (page 253) to find and measure peaks in your signals, you can use those same input argument values for findpeaksfit.m. The remaining six input arguments of findpeaksfit.m are for the peakfit function. If you have been using peakfit.m or ipf.m (page 416) to fit peaks in your signals, you can use those same input argument values for findpeaksfit.m. Type "help findpeaksfit" for more information. See page 234. Click for animated example.

peakstats.m uses the same algorithm as findpeaksG.m, but it computes and returns a table of summary statistics of the peak intervals (the x-axis interval between adjacent detected peaks), heights, widths, and areas, listing the maximum, minimum, average, and percent standard deviation of each, and optionally displaying the x, t data plot with numbered peaks in Figure window 1, the table of peak statistics in the command window, and the histograms of the peak intervals, heights, widths, and areas in Figure window 2. Type "help peakstats". See page 234. Version 2, March 2016, adds median and mode.

tablestats.m (**`PS=tablestats(P, displayit)`**) is similar to peakstats.m except that it accepts as input a peak table P such as generated by findpeaksG.m, findvalleys.m, findpeaksL.m, findpeaksb.m, findpeaksplot.m, findpeaksnr.m, findpeaksGSS.m, findpeaksLSS.m, or findpeaksfit.m, any function that return a table of peaks with at least 4 columns listing peak number, height, width, and area. Computes the peak intervals (the x-axis interval between adjacent detected peaks) and the maximum, minimum, average, and percent standard deviation of each, and optionally displaying the histograms of

the peak intervals, heights, widths, and areas in Figure window 2. The optional last argument displayit = 1 if the histograms are to be displayed, otherwise not.

findpeaksnr.m is a variant of findpeaksG.m that additionally computes the signal-to-noise ratio (SNR) of each peak and returns it in the 5th column of the peak table. The SNR is computed as the ratio of the peak height to the root-mean-square residual (difference between the actual data and the least-squares fit over the top part of the peak). See PeakFindingandMeasurement.htm.

findpeaksE.m is a variant of findpeaksG.m that additionally estimates the percent relative fitting error of each peak (assuming a Gaussian peak shape) and returns it in the 6th column of the peak table.

findpeaksGSS.m and findpeaksLSS.m, for Gaussian and Lorentzian peaks respectively, are variants of findpeaksG.m and findpeaksL.m that additionally compute the 1% start and end positions return them in the 6th and 7thcolumns of the peak table. See PeakFindingandMeasurement.htm.

findsquarepulse.m (syntax **`S=findsquarepulse(t, y, threshold)`**) locates the rectangular pulses in the signal t, y that exceed a y-value of "threshold" and determines their start time, average height (relative to the adjacent baseline) and width. DemoFindsquare.m creates a test signal and calls findsquarepulse.m to demonstrate.

findsteps.m **`P= findsteps(x, y, SlopeThreshold, AmpThreshold, SmoothWidth, peakgroup)`** locates positive transient steps in noisy x-y time series data, by computing the first derivative of y that exceed SlopeThreshold, computes the step height as the difference between the maximum and minimum y values over a number of data point equal to "Peakgroup". It returns list (P) with step number, x and y positions of the bottom and top of each step, and the step height of each step detected; "SlopeThreshold" and "AmpThreshold" control step sensitivity, Higher values will neglect smaller features. Increasing "SmoothWidth" ignores small sharp false steps caused by random noise or by "glitches" in the data acquisition. See findsteps.png for a real example. findstepsplot.m plots the signal and numbers the peaks.

idpeaks, peak identification function. The syntax is **`[IdentifiedPeaks, AllPeaks] = idpeaks(DataMatrix, AmpT, SlopeT, sw, fw, maxerror, Positions, Names)`**. Locates and identifies peaks in `DataMatrix` that match the position of peaks in the array "`Positions`" with matching names "`Names`". Type "`help idpeaks`" for more information. Download and extract idpeaks.zip for a working example or see **Example 8** on page 252.

idpeaktable.m **`[IdentifiedPeaks]=idpeaktable(P, maxerror, Positions, Names)`**. Compares the found peak positions in peak table "P" to a database of known peaks, in the form of a cell array of known peak maximum positions ("Positions") and matching cell array of names ("Names"). If the position of a found peak in the signal is closer to one of the known peaks by less than the specified maximum error ("maxerror"), that peak is considered a match and its peak position, name, error, and amplitude are entered into the output cell array "IdentifiedPeaks". The peak table may be one returned by any of my peak finder or peak fitting functions, having one row for each peak and columns for peak number, position, and height as the first three columns.

demoipeak.m is a simple demo script that generates a noisy signal with peaks, calls iPeak, and then prints out a table of the actual peak parameters and a list of the peaks detected and measured by iPeak for comparison. Before running this demo, ipeak.m (page 253) must be downloaded and placed in the Matlab path. The Octave version is demoipeakoctave.m.

DemoFindPeak.m, a demonstration script using the findpeaksG function on noisy synthetic data. Numbers the peaks and prints out the peak parameters in the command window. Requires that gaussian.m and findpeaksG.m be present in the path. See page 234.

TestFindpeaksG2d.m. Demonstration script for findpeaks2d.m, which shows that this function can locate peaks resulting in 'shoulders' that do not produce a distinct maximum in the original signal. Detects peaks in the negative of the smoothed second derivative of the signal (shown as the dotted line in the figure). Requires gaussian.m, findpeaksG.m, findpeaksG2d.m, fastsmooth.m, and peakfit.m in the path. Graphic. Also uses theTestFindpeaksG2d results as the "start" value for iterative peak fitting using peakfit.m, which takes longer to compute but gives more accurate results, especially for width and area:

DemoFindPeakSNR is a variant of DemoFindPeak.m that uses findpeaksnr.m to compute the signal-to-noise ratio (SNR) of each peak and returns it in the 5th column.

triangulationdemo.m is a demo function (screen graphic) that compares findpeaksG (which determines peak parameters by curve-fitting a Gaussian to the center of each peak) to findpeaksT, which determines peak parameters by the triangle construction method (drawing a triangle around each peak with sides that are tangent to the sides of the peak). Performs the comparison with 4 different peak shapes: plain Gaussian, bifurcated Gaussian, exponential modified Gaussian, and Breit-Wigner-Fano). In some cases, the triangle construction method can be more accurate than the Gaussian method if the peak shape is asymmetric.

findpeaksfitdemo.m, a demonstration script of findpeaksfit automatically finding and fitting the peaks in a set of 150 signals, each of which may have 1 to 3 noisy Lorentzian peaks in variable locations. Requires the findpeaksfit.m and lorentzian.m functions installed. This script was used to generate the GIF animation findpeaksfit.gif.

FindpeaksComparison.m. Which to use: findpeaksG, findpeaksb, findpeaksb3, or findpeaksfit? This script compares all four functions applied to a computer-generated signal with multiple peaks with variable types and amounts of baseline and random noise. (Requires all these functions, plus modelpeaks.m, findpeaksG, and findpeaksL.m, in the Matlab/Octave path. Type "help FindpeaksComparison" for details). Results are displayed graphically in Figure windows 1, 2, and 3 and printed out in a table of parameter accuracy and elapsed time for each method. You may change the lines in the script marked by <<< to modify the number and character and amplitude of the signal peaks, baseline, and noise. (Adjust the parameters to make the simulated signal like your experimental signal to discover which method works best for your type of signal). The best method depends mainly on the shape and amplitude of the baseline and on the extent of peak overlap.

iPeakEnsembleAverageDemo.m is a demonstration script for iPeak's ensemble average function. In this example, the signal contains a repeated pattern of two overlapping Gaussian peaks, 12 points apart, both of width 12, with a 2:1 height ratio. These patterns occur at random intervals throughout the recorded signal, and the random noise level is about 10% of the average peak height. Using iPeak's ensemble average function (**Shift-E**), the patterns can be averaged and the signal-to-noise ratio improved.

ipeakdata.mat, a data set for demonstrating idpeaks.m or the peak identification function of iPeak. It includes a high-resolution atomic spectrum and a table of known emission wavelenghs. See page 234.

Which to use: iPeak or Peakfit? Try these Matlab demo functions that compare iPeak.m (page 253) with peakfit.m (page 234) for signals with a few peaks and signals with many peaks and that shows how to adjust iPeak to detect broad or narrow peaks. These are self-contained demos that include all required sub-functions. Just place them in your path and type their name at the command prompt. You can download all these demos together in idemos.zip. They require no input or output arguments.

SpikeDemo1.m and SpikeDemo2.m are Matlab/Octave scripts that demonstrate how two measure

spikes (very narrow peaks) in the presence of serious interfering signals. See page 304.

PowerTransformTest.m is a simple script that demonstrates the power method of peak sharpening to aid in reducing in peak overlap. PowerMethodCalibrationCurve is a variant of PeakCalibrationCurve.m that evaluates the power method in the context of a flow injection or chromatography measurement. powertest2 is a self-contained function that demonstrates the power method for measuring the area of small shouldering peak (Graphic).

The script realtimepeak.m demonstrates simple real-time peak detection based on derivative zero-crossing, using mouse clicks to simulate data. Each time your mouse clicks form a peak (that is, go up and then down again), the program will register and label the peak on the graph (as illustrated on the right) and print out its *x* and *y* values. In this case, a peak is defined as any data point that has lower amplitude points adjacent to it on both sides, which is determined by the nested 'for' loops in lines 31-36. The more sophisticated script RealTimeSmoothedPeakDetectionGauss.m uses the technique described on page 234 that locates the positive peaks in a noisy data set that rise above a set amplitude threshold, performs a least-squares curve-fit of a Gaussian function to the top part of the raw data peak, computes the position, height, and width (FWHM) of each peak from that least-squares fit and prints out each peak found in the command window. (Animated graphic).

AreasOfIsolatedPeaks.m. Script to demonstrate the measurement of the areas of isolated peaks superimposed on a variable baseline, by the trapezoidal method, using the inbuilt trapz function.

Peak detection tool. PeakDetection.mlx (page 253) is an interactive Live Script for peak detection and measurement, including a selection of peak detectors, data smoothing, symmetrization, peak sharpening, and curve fitting, with interactive sliders and drop-down menus to control them interactively.

Multicomponent Spectroscopy

cls.m is a classical least-squares function that you can use to fit a computer-generated model, consisting of any number of peaks of known shape, width, and position, but of unknown height, to a noisy x,y signal. The syntax is **`heights= cls(x,y, NumPeaks, PeakShape, Positions, Widths, extra)`** where x and y are the vectors of measured signal (e.g. x might be wavelength and y might be the absorbance at each wavelength), 'NumPeaks' is the number of peaks, 'PeakShape' is the peak shape number (1=Gaussian, 2=Lorentzian, 3=logistic, 4=Pearson, 5=exponentially broadened Gaussian; 6=equal-width Gaussians; 7=Equal-width Lorentzians; 8=exponentially broadened equal-width Gaussian, 9=exponential pulse, 10=sigmoid, 11=Fixed-width Gaussian, 12=Fixed-width Lorentzian; 13=Gaussian/Lorentzian blend; 14=BiGaussian, 15=BiLorentzian), 'Positions' is the vector of peak positions on the x axis (one entry per peak), 'Widths' is the vector of peak widths in x units (one entry per peak), and 'extra' is the additional shape parameter required by the exponentially broadened, Pearson, Gaussian/Lorentzian blend, BiGaussian and BiLorentzian shapes. cls.m returns a vector of measured peak heights for each peak. See clsdemo.m. (Note: this method is now included in the non-linear iterative peak fitter peakfit.m (page 234) as peak shape 50. See the demonstration script peakfit9demo.m*)*

The cls2.m function is similar to cls.m, except that it also measures the baseline (assumed to be flat) and returns a vector containing the background B and measured peak heights H for each peak , e.g. [B H1 H2 H3...].

RegressionDemo.m, script that demonstrates the classical least-squares procedure for a simulated absorption spectrum of a 5-component mixture at 100 wavelengths. Requires that gaussian.m be present in the path. See page 188.

clsdemo.m is a demonstration script that creates a noisy signal, fits it using the Classical Least-squares method with cls.m, computes the accuracy of the measured heights, then repeats the calculation using iterative least-squares using peakfit.m (page 234) for comparison. (This script requires cls.m, modelpeaks.m, and peakfit.m in the Matlab/Octave path).

CLSvsINLS.m is a script that compares the classical least-squares (CLS) method with three different variations of the iterative method (INLS) method for measuring the peak heights of three Gaussian peaks in a noisy test signal, demonstrating that the fewer the number of unknown parameters, the faster and more accurate is the peak height calculation.

Non-linear iterative curve fitting and peak fitting

gaussfit, function that performs least-squares fit of a single Gaussian function to an x,y data set, returning the height, position, and width of the best-fit Gaussian. Syntax is `[Height, Position, Width] = gaussfit(x,y)`. The similar function lorentzfit.m performs the calculation for a Lorentzian peak shape. See page 172. The similar function plotgaussfit does the same thing as gaussfit.m but also plots the data and the fit. The data set cannot contain any zero or negative values.

bootgaussfit is an expanded version of gaussfit that provides optional plotting and error estimation. The syntax is **`[Height, Position, Width, BootResults] = bootgaussfit(x, y, plots)`**. If plots=1, plots the raw data as red dots and the best-fit Gaussian as a line. If the 4th output argument (`BootResults`) is supplied, computes peak parameter error estimates by the bootstrap method.

fitshape2.m, syntax **`[Positions, Heights, Widths, FittingError] = fitshape2(x, y, start)`**, is a simplified general-purpose Matlab/Octave *function* for fitting multiple overlapping model shapes to the data contained in the vector variables x and y. The model is linear combination of any number of basic functions that are defined mathematically as a function of x, with two variables that the program will independently determine positions and widths for each peak, in addition to the peak heights (i.e., the weights of the weighted sum). You must provide the first guess starting vector 'start', in the form [position1 width1 position2 width2 ...etc.], which specifies the first-guess position and width of each component (one pair of position and width for each peak in the model). The function returns the parameters of the best-fit model in the vectors **`Positions, Heights, Widths`**, and computes the percent error between the data and the model in `FittingError`. It also plots the data as dots and the fitted model as a line. The interesting thing about this function is that *the only part that defines the shape of the model is the last line*. In fitshape2.m, that line contains the expression for a *Gaussian peak of unit height*, but you could change that to *any other expression or algorithm* that computes *g* as a function of *x* with two unknown parameters 'pos' and 'wid' (position and width, respectively, for peak-type shapes, but they could represent anything for other function types, such as the exponential pulse, sigmoidal, etc.). Everything else in the fitshape.m function can remain the same. This makes fitshape a good platform for experimenting with different mathematical expressions as proposed models to fit data. There are also two other variations of this function for models with *one* iterated variable plus peak height (fitshape1.m) and *three* iterated variables plus peak height (fitshape3.m). Each has illustrative examples contained in the built-in help file (type “help <filename>”). An alternative version is FitMultipleShapes2, which allows you to specify any of 16 common peak shape functions by number. Type "help FitMultipleShapes2" for examples of use. Syntax:

```
[Positions,Heights,Widths,FittingError] = FitMultipleShapes2(x,y,shape,start,m)
```

peakfit (page 234) a versatile command-line function for multiple peak fitting by iterative non-linear least-squares. A Matlab File Exchange "Pick of the Week". The full syntax is **`[FitResults, GOF, baseline, coeff, BestStart, xi, yi, BootResults] = peakfit(signal, center, window, NumPeaks, peakshape, extra, NumTrials, start, BASELINEMODE, fixedwidth, plots, bipolar, minwidth)`**. Type "help peakfit". See page 396. Compared to

the fitshape.m function described previously, peakfit.m has a large number of *built-in* peak shapes selected by number, it does not require (although it can be given) the first-guess position and width of each component, and it has features for background correction and other useful features to improve the quality and estimate the reliability of fits. Test the installation on your computer by running the autotestpeakfit.m script, which runs through the whole gauntlet of fitting tests without pause, printing out what it is doing and the results, checking to see if the fitting error is greater than expected and printing out a WARNING if it is. This takes 17 seconds to run in Matlab 9.9 2020b on a 3.5Ghx i7 windows 10 machine. See the version history, page 396, for a brief description of the new features of each version of peakfit.m from 3.7 to the present.

testnumpeaks(x,y,peakshape,extra,NumTrials,MaxPeaks). Simple test to estimate the number of model peaks required to fit an x,y data set. Fits data x,y, with shape "peakshape", with optional extra shape factor "extra", with NumTrials repeat per fit, up to a maximum of "MaxPeaks" model peaks, displays each fit and graphs fitting error vs number of model peaks. If two or more numbers of peaks give about the same error, it's best to take the smaller number.

SmoothVsFit.m is a demonstration script that compares iterative least-square fitting to two simpler methods for the measurement of the peak height of a single peak of uncertain width and position and with a very poor signal-to-noise ratio of 1. The accuracy and precision of the methods are compared. SmoothVsFitArea.m does the same thing for the measurement of peak area. See page 173.

ipf.m (page 416) is an interactive multiple peak fitter (m-file link: ipf.m). It uses iterative nonlinear least-squares to fit any number of overlapping peaks of the same or different peak shapes to x-y data sets. Demoipf.m is a demonstration script for ipf.m, with a built-in simulated signal generator. The true values of the simulated peak positions, heights, and widths are displayed in the Matlab command window, for comparison to the FitResults obtained by peak fitting. Click for animated step-by-step instructions. You can also download a ZIP file containing ipf.m plus some examples and demos. Click for animated example.

SmallPeak.m is a demonstration of several curve-fitting techniques applied to the challenging problem of measuring the height of a small peak that is closely overlapped with and completely obscured by a much larger peak. It compares iterative fits by unconstrained, equal-width, and fixed-position models (using peakfit.m, page 234) to a classical least-squares fit in which *only* the peak heights are unknown (using cls.m). Spread out the four Figure windows so you can observe the dramatic difference in stability of the different methods. A final table of relative percent peak height errors shows that the more the constraints, the better the results (but *only if the constraints are justified*). See page 322.

BlackbodyDataFit.m, a script that demonstrates iterative least-squares fitting of the *blackbody equation* to a measured spectrum of an incandescent body, for the purpose of estimating its color temperature. See page 207.

Demofitgauss.m a script that demonstrates iterative fitting a *single* Gaussian function to a set of data, using the fminsearch function. Requires that gaussian.m and fmsearch.m (in the "Optim 1.2.1" package) be installed. Demofitgaussb.m and fitgauss2b.m illustrate a modification of this technique to accommodate shifting baseline (Demofitlorentzianb.m and fitlorentzianb.m for Lorentzian peaks). This modification is now incorporated to peakfit.m (version 4.2 and later), ipf.m (version 9.7 and later), findpeaksb.m (version 3 and later), and findpeaksfit, (version 3 and later). See page 207.

Demofitgauss2.m a script that demonstrates iterative fitting of *two* overlapping Gaussian functions to a set of data, using the fminsearch function. Requires that gaussian.m and fmsearch.m (in the "Optim 1.2.1" package) be installed. Demofitgauss2b.m is the baseline-corrected extension. See page 207.

VoigtFixedAlpha.m and VoigtVariableAlpha.m demonstrate two different ways to fit peaks

with *variable shapes*, such as the Voigt profile, Pearson, Gauss-Lorentz blend, and the bifurcated and exponentially-broadened shapes, which are defined not only by a peak position, height, and width, but also by an additional "shape" parameter that fine-tunes the shape of the peak. If that parameter is *equal* for all peaks in a group, it can be passed as an additional input argument to the shape function, as shown in VoigtFixedAlpha.m. If the shape parameter is allowed to be *different* for each peak in the group and is to be determined by iteration (just as is position and width), then the routine must be modified to accommodate *three*, rather than *two*, iterated variables, as shown in VoigtVariableAlpha.m. Although the *fitting error is lower* with variable alphas, the execution time is longer and the *alphas values so determined are not very stable*, with respect to noise in the data and the starting guess values, especially for multiple peaks. See page 204. The script VoigtShapeFittingDemonstration.m uses peakfit.m version 9.5 to fit a single Voigt profile and to calculate the Gaussian width component, Lorentzian width component, and alpha. It computes the theoretical Voigt profile and adds random noise for realism. VoigtShapeFittingDemonstration2.m does the same for two overlapping Voigt profiles, using both fixed alpha and variable alpha models (shape numbers 20 and 30). (Requires voigt.m, halfwidth.m, and peakfit.m in the path).

Demofitmultiple.m. Demonstrates an iterative fit to sets of computer-generated noisy peaks of different types, knowing only the shape types and variable shape parameters of each peak. Iterated parameters are shape, height, position, and width of all peaks. Requires the fitmultiple.m and peakfunction.m functions. View screen shot. See page 204.

BootstrapIterativeFit.m, a function that demonstrates bootstrap estimation of the variability of an iterative least-squares fit to a single noisy Gaussian peak. The syntax is: **`BootstrapIterativeFit (TrueHeight,TruePosition,TrueWidth,NumPoints,Noise,NumTrials).`** See page 170.

BootstrapIterativeFit2.m, a function that demonstrates bootstrap estimation of the variability of an iterative least-squares fit to two noisy Gaussian peaks. The syntax is: **`BootstrapIterativeFit2(TrueHeight1, TruePosition1, TrueWidth1, TrueHeight2, TruePosition2, TrueWidth2, NumPoints, Noise, NumTrials).`** See page 170.

DemoPeakfitBootstrap.m. Self-contained demonstration function for peakfit.m (page 234), with built-in signal generator. Demonstrates bootstrap error estimation. See page 170.

DemoPeakfit.m, Demonstration script (for peakfit.m) that generates an overlapping peak signal, adds noise, fits it with peakfit.m, then computes the accuracy and precision of peak parameter measurements. Requires that peakfit.m be present in the path. See page 413.

peakfit9demo. Demonstrates multilinear regression (shape 50) available in peakfit.m version 9 (Requires modelpeaks.m and peakfit.m in the Matlab path). Creates a noisy model signal of three peaks of known shapes, positions, and widths, but unknown heights. Compares multilinear regression in Figure window 1 with unconstrained iterative non-linear least-squares in Figure window 2.

DemoPeakFitTime.m is a simple script that demonstrates how to use peakfit.m to apply *multiple curve fits to a signal that is changing with time*. The signal contains two noisy Gaussian peaks in which the peak position of the *second* peak increases with time and the other parameters remain constant, except for the noise. (click to play animation.

isignal (page 390) can be used as a command-line function in Octave, but its *interactive features currently work only in Matlab*. The syntax is **`isignal(DataMatrix, xcenter, xrange, SmoothMode, SmoothWidth, ends, DerivativeMode, Sharpen, Sharp1, Sharp2, SlewRate, MedianWidth).`**

testpeakfit.m, a test script that demonstrates 36 different examples on page 416. Use for testing that

peakfit and related functions are present in the path. autotestpeakfit.m does the same without pausing between functions and waiting for a keypress (takes about 17 seconds to run).

Live Script peak fitting tool, similar to the other Live Script tools for smoothing, deconvolution, peak detection, is described on page 440. It is based on the peakfit.m function and can fit a variety of peak shapes with optional specified start position, baseline correction, and symmetrization.

Multiple peak fits with different profiles. ShapeTestS.m and ShapeTestA.m tests the data in its input arguments x,y, assumed to be a single isolated peak, fits it with *different candidate model peak shapes* using peakfit.m, plots each fit in a separate figure window, and prints out a table of fitting errors in the command window. ShapeTestS.m tries seven different candidate symmetrical model peaks, and ShapeTestA.m tries six different candidate asymmetrical model peaks. The one with the lowest fitting error (and R^2 closest to 1.000) is presumably the best candidate. *Try the examples in their help files*. But beware: if there is too much noise in your data, the results can be misleading
WidthTest.m is a script that demonstrates that constraining some of the peak parameters of a fitting model to fixed values, *if* those values are accurately known, improves that accuracy of measurement of the *other* parameters, even though it *increases* the fitting error. Requires installation of the GL.m and peakfit.m functions (version 7.6 or later) in the Matlab/Octave path.

The script NumPeaksDemo.m demonstrates one way to attempt to estimate the minimum number of model peaks needed to fit a set of data, plotting the fitting error vs the number of model peaks, and looking for the point at which the fitting error reaches a minimum. This script creates a noisy computer-generated signal containing a user-selected 3, 4, 5 or 6 underlying Lorentzian peaks and uses peakfit.m to fit the data to a series of models containing 1 to 10 model peaks. The correct number of underlying peaks is either the fit with the lowest fitting error, or, if two or more fits have about the same fitting error, the fit with the least number of peaks, as in this example, which actually has 4 underlying peaks. If the data are very noisy, however, the determination becomes unreliable. (To make this demo closer to your type of data, you could change Lorentzian to Gaussian or any other model shape, or change the peak width, number of data points, or the noise level). This script requires that peakfit.m and the appropriate shape functions (gaussian.m, lorentzian.m, etc.) be present in the path. The function testnumpeaks.m does this for your own x,y data.

Peakfit Time Tests. These are a series of scripts that demonstrate how the execution time of the peakfit.m function varies with the peak shape (PeakfitTimeTest2.m and PeakfitTimeTest2a.m, with number of peaks in the model (PeakfitTimeTest.m), and with the number of data points in the fitted region (PeakfitTimeTest3.m). This issue is discussed on page 433.

TwoPeaks.m is a simple 8-line script that compares findpeaksG.m and peakfit.m with a signal consisting to two noisy peaks. findpeaksG.m and peakfit.m must be in the Matlab/Octave path.

peakfitVSfindpeaks.m performs a direct comparison of the accuracy of findpeaksG vs peakfit. This script generates four very noisy peaks of different heights and widths, then applies findpeaksG.m and peakfit.m to measure the peaks and compares the results. The peaks detected by findpeaks are labeled "Peak 1", "Peak 2", etc. If you run this script several times, you'll find that both methods work well most of the time, with peakfit giving smaller errors in most cases, but occasionally findpeaks will miss the first (lowest) peak and rarely it will detect an extra peak that is not there if the signal is very noisy.

CaseStudyC.m is a self-contained Matlab/Octave demo function that demonstrates the application of several techniques described on this site to the quantitative measurement of a peak buried in an unstable background, a situation that can occur in the quantitative analysis applications of various forms of spectroscopy and remote sensing. See Case Studies C.

GaussVsExpGauss.m Comparison of alternative models for the unconstrained exponentially-broadened

Gaussians, shapes 31 and 39. Shape 31 (expgaussian.m) creates the shape by performing a Fourier convolution of a specified Gaussian by an exponential decay of specified time constant, whereas shape 39 (expgaussian2.m) uses a mathematical expression for the final shape so produced. Both result in the *same shape* but are parameterized differently. Shape 31 reports the peak height and position as that of the original Gaussian before broadening, whereas shape 39 reports the peak height of the broadened result. Shape 31 reports the width as the FWHM of the original Gaussian and shape 39 reports the standard deviation (sigma) of that Gaussian. Shape 31 reports the exponential factor on the *number of data points* and shape 39 reports the *reciprocal of time constant* in time units. See Figure windows **2** and **3**. You must have peakfit.m (version 8.4) gaussian.m, expgaussian.m, expgaussian2.m, findpeaksG.m, and halfwidth.m in the Matlab/Octave path. DemoExpgaussian.m is a script that gives a more detailed exploration of the effect of exponential broadening on a Gaussian peak (requires gaussian.m, expgaussian.m, halfwidth.m, val2ind.m, and peakfit.m in the Matlab/Octave path).

AsymmetricalOverlappingPeaks.m is a multi-step script that demonstrates the use of a combination of first-derivative symmetrization before curve fitting to analyze a complex mystery peak. See page 366).

Keystroke-operated *interactive* functions

The interactive functions **ipeak**, **isignal**, **ipf**, and **ifilter** run in the Figure window and use a simple set of single keystroke commands, rather than on-screen buttons or menus or sliders, to reduce screen clutter, minimize overhead, maximize processing speed, and allow you to explore data and try out various approaches easily and quickly. All have several keystroke commands in common: all share the same set of *pan and zoom* to adjust the portion of the signal that is displayed in the upper panel. (There are also *Octave* versions of all these, **ipeakoctave**, **isignaloctave**, **ipfoctave**, and **ifilteroctave**, all of which use *different keys for the pan and zoom* adjustments than the Matlab versions). All versions use the **K** key to display the list of keystroke commands. Double-click the figure window title bar to expand to full screen for a better view of small signal features. All use the **T** key to cycle through the baseline correction modes. All use the **Shift-Ctrl-S**, **Shift-Ctrl-F**, and **Shift-Ctrl-P** keys to transfer the current signal between **iSignal**, **ipf**, and **iPeak**, respectively. To make it easier to transfer settings from one of these functions to other related functions, all use the **W** key to print out the syntax of other related functions, with the pan and zoom settings and other numerical input arguments specified, ready for you to Copy, Paste and edit into your own scripts or back into the command window. For example, you can convert a curve fitting operation performed in **ipf.m** into the command-line **peakfit.m** function. Or you can convert a peak finding operation performed in ipeak.m into a command-line findpeaksG.m or findpeaksb.m function. The **W** key is useful with signals that require different signal processing in different regions of their x-axis ranges, by allowing you to create a series of command-line functions for each local region that, when executed in sequence, quickly processes each segment of the signal appropriately and can be repeated easily for any number of other examples of that same type of signal. To adjust continuously variable parameters, these programs use *pairs of adjacent keys* to increase or decrease each parameter in steps, often with the shift-key controlling the step size.

Hyperlinear Quantitative Absorption Spectrophotometry

tfit.m, a self-contained command-line Matlab/Octave function that demonstrates a computational method for quantitative analysis by multiwavelength absorption spectroscopy which uses convolution and iterative curve fitting to correct for spectroscopic non-linearity. The syntax is tfit(TrueAbsorbance). TFitStats.m is a script that demonstrates the reproducibility of the method. TFitCalCurve.m compares the calibration curves for single-wavelength, simple regression, weighted regression, and TFit methods. TFit3.m is a demo function for a mixture of 3 absorbing components. The syntax is TFit3(TrueAbsorbanceVector), e.g., TFit3([3 .2 5]). Download all these as a ZIP file. Click for animated example. TFitDemo.m is a keypress-operated *interactive* explorer for the Tfit method, applied

to the measurement of a single component with a Lorentzian (or Gaussian) absorption peak, with controls that allow you to adjust the true absorbance ("Peak A"), spectral width of the absorption peak ("AbsWidth"), spectral width of the instrument function ("InstWidth"), stray light, and the noise level ("Noise") continuously while observing the effects graphically and numerically. See page 276. Click for animated example. These functions and scripts also work in the latest version of Octave.

MAT files (for Matlab and Octave) and Text files (.txt)

DataMatrix2 is a computer-generated test signal consisting of 16 symmetrical Gaussian peaks with random white noise added. Can be used to test the peakfit.m function. See page 228.

DataMatrix3 is a computer-generated test signal consisting of 16 Gaussian peaks with random white noise that have been exponentially broadened with a time constant of 33 x-axis units. See page 228.

udx.txt: a text file containing the 2 x 1091 matrix that consists of two Gaussian peaks with different sampling intervals. It is used as an example in Smoothing and in Curve Fitting.

TimeTrial.txt, a text file comparing the speed of several different signal processing tasks, using the following different software configurations:

(a) Matlab 2020b on Windows 10, 64-bit, 3.6 GHz, core i7, 16 GBytes RAM
(b) Matlab 2009a, on older Windows machine
(c) Matlab 2017b Home, on older Windows machine
(d) Matlab Online, R2018b, in Google Chrome
(e) Matlab Mobile (on recent iPad)
(f) Octave 6.2.0 on Windows 10, 64-bit, 3.6 GHz, core i7, 16 GBytes RAM

The Matlab/Octave code that generated this is TimeTrial.m, which runs all of the tasks one after the other and prints out the elapsed times for your machine plus the times previously recorded for each tasks on each of the five software systems. TimeTrial.xlsx summarizes the comparison of Matlab to Octave.

Readability.txt. Report on the English language readability analysis of IntroToSignalProcessing.pdf performed by http://www.online-utility.org/english/readability_test_and_improve.jsp

Spreadsheets (for Excel or OpenOffice Calc)

Notes. These spreadsheets are self-contained and so do not rely on external files. You may transfer your data to them by using the Data tab and/or Copy and Paste.

If you see a yellow bar at the top of the spreadsheet window, click the "Enable Editing" button.

If your browser changes the file extension of these spreadsheets to .zip when they are downloaded, rename the files to their original file extensions (.ods, .xls, or .xlsx) before running them.

These spreadsheets have no protected cells, so there is nothing stopping you from changing the formulas accidentally. This means you can modify any aspect of these spreadsheets for your own purposes, which you are invited to do. If you mess up, just use the Undo function (**Ctrl-Z**) or you can download another copy.

Random numbers and noise (page 24). The spreadsheets RandomNumbers.xls (for Excel) and RandomNumbers.ods (for OpenOffice) demonstrate how to create a column of normally-distributed random numbers (like white noise) in a spreadsheet that has only a uniformly-distributed random number function. Also shows how to compute the interquartile range and the peak-to-peak value and how they compare to the standard deviation. See page 24. The same technique is used in the spreadsheet SimulatedSignal6Gaussian.xlsx, which computes and plots a simulated signal consisting of

up to 6 overlapping Gaussian bands plus random white noise.

Smoothing (page 42). The spreadsheets smoothing.ods (for Open office Calc) and smoothing.xls (for Microsoft Excel) demonstrate a 7-point rectangular (sliding average) in column C and a 7-point triangular smooth in column E, applied to the data in column A. You can type in (or Copy and Paste) any data you like into column A. You can extend the spreadsheet to longer columns of data by dragging the last row of columns A, C, and E down as needed. You can change the smooth width by changing the equations in columns C or E. The spreadsheet MultipleSmoothing.xls for Excel or Calc demonstrates a more flexible method that allows you to define various types of smooths by typing a few integer numbers. The spreadsheets UnitGainSmooths.xls and UnitGainSmooths.ods contain a collection of unit-gain convolution coefficients for rectangular, triangular, and P-spline smooths of width 3 to 29 in both vertical (column) and horizontal (row) format. You can Copy and Paste these into your own spreadsheets. Convolution.txt lists some simple whole-number coefficient sets for performing single and multi-pass smoothing. VariableSmooth.xlsx demonstrates an even more powerful and flexible technique, especially for very large and variable smooth widths, which uses the spreadsheet AVERAGE and INDIRECT functions (page 353). It allows you to change the smooth width simply by changing the value of a single cell. See page 53 for details. SegmentedSmoothTemplate.xlsx is a segmented multiple-width data smoothing spreadsheet template, which can apply individually specified different smooth widths to different regions of the signal, especially useful if the widths of the peaks or the noise level varies substantially across the signal. In this version there are 20 segments. SegmentedSmoothExample.xlsx is an example with data (graphic). A related sheet GradientSmoothTemplate.xlsx (graphic) performs a linearly increasing (or decreasing) smooth width across the entire signal, given only the start and end values, automatically generating as many segments are necessary.

Differentiation (page 63). DerivativeSmoothingOO.ods (for OpenOffice Calc) and DerivativeSmoothing.xls (for Excel) demonstrate the application of differentiation for measuring the amplitude of a peak that is buried in a broad curved background. Differentiation and smoothing are both performed together. Higher order derivatives are computed by taking the derivatives of previously computed derivatives. DerivativeSmoothingWithNoise.xlsx is a related spreadsheet that demonstrates the dramatic effect of smoothing on the signal-to-noise ratio of derivatives on a noisy signal. It uses the same signal as DerivativeSmoothing.xls, but adds simulated white noise to the Y data. You can control the amount of added noise. SecondDerivativeXY2.xlsx, demonstrates locating and measuring changes in the second derivative (a measure of curvature or acceleration) of a time-changing signal, showing the apparent increase in noise caused by differentiation and the extent to which the noise can be reduced by smoothing (in this case by two passes of a 5-point triangular smooth). The smoothed second derivative shows a large peak at the point where the acceleration changes and plateaus on either side showing the magnitude of the acceleration before and after the change (2 and 4, respectively). Convolution.txt lists simple whole-number coefficient sets for performing differentiation and smoothing. CombinedDerivativesAndSmooths.txt lists the sets of unit-gain coefficients that perform 1st through 4th derivatives with various degrees of smoothing. See page 63.

Peak sharpening (page 78). The derivative sharpening method with two derivative terms (2nd and 4th) is available in the form of an empty template (PeakSharpeningDeriv.xlsx and PeakSharpeningDeriv.ods) or with example data entered (PeakSharpeningDerivWithData.xlsx and PeakSharpeningDerivWithData.ods). You can either type in the values of the derivative weighting factors K1 and K2 directly into cells J3 and J4, or you can enter the estimated peak width (FWHM in number of data points) in cell H4 and the spreadsheet will calculate K1 and K2. There is a demo version with adjustable simulated peaks (PeakSharpeningDemo.xlsx and PeakSharpeningDemo.ods), as well as a version with clickable buttons

for convenient interactive adjustment of the K1 and K2 factors by 1% or by 10% for each click. There is also a 20-segment version where the sharpening constants can be specified for each of 20 signal segments (SegmentedPeakSharpeningDeriv.xlsx). For applications where the peak widths gradually increase (or decrease) with time, there is also a gradient sharpening template (GradientPeakSharpeningDeriv.xlsx) and an example with some data already entered (GradientPeakSharpeningDerivExample.xlsx). You need only set the starting and ending peak widths and the spreadsheet will apply the required sharpening factors K1 and K2. PeakSymmetricalizationDemo.xlsm (graphic) demonstrates the symmetrization of exponentially modified Gaussians (EMG) by the weighted addition of the first derivative (and also allows further second derivative sharpening of the resulting symmetrized peak). There is also an empty template PeakSymmetricalizationTemplate.xlsm (graphic) and an example application with sample data already typed in: PeakSymmetricalizationExample.xlsm. PeakDoubleSymmetrizationExample.xlsm performs the symmetrization of a *doubly* exponential broadened peak. It has buttons to interactively adjust the two stages of first-derivative weighting. Two variations (1, 2) include example data for two overlapping peaks, for which the areas after symmetrization are measured by perpendicular drop. ComparisonOfPerpendicularDropAreaMeasurements.xlsx (graphic) demonstrates the effect of the power sharpening method on perpendicular drop area measurements of Gaussian and exponentially broadened Gaussian peaks, including the effect of resolution, relative peak height, random noise, smoothing, and non-zero baseline has on the normal and power sharpening method. PowerSharpeningTemplate.xlsx is an empty template that preforms this method and PowerSharpeningExample.xlsx is the same with example data.

Convolution (page 109). Spreadsheets can be used to perform "shift-and-multiply" convolution for small data sets (for example, MultipleConvolution.xls or MultipleConvolution.xlsx for Excel and MultipleConvolutionOO.ods for Calc), which is essentially the same technique as the above spreadsheets for smoothing and differentiation. Use this spreadsheet to investigate convolution, smoothing, differentiation, and the effect of those operations on noise and signal-to-noise ratio. (For larger data sets the performance is slower than Fourier convolution, which is much easier done in Matlab or Octave than in spreadsheets). Convolution.txt lists simple whole-number coefficient sets for performing differentiation and smoothing.

Peak Area Measurement (page 132). EffectOfDx.xlsx demonstrates that the simple equation sum(y)*dx accurately measures the peak area of an isolated Gaussian peak if there are at least 4 or 5 points visibly above the baseline. EffectOfNoiseAndBaseline.xlsx demonstrates the effect of random noise and non-zero baseline, showing that the area is more sensitive to non-zero baseline that the same amount of random noise. PeakSharpeningAreaMeasurementDemo.xlsm (screen image) demonstrates the effect of derivative peak sharpening on perpendicular drop area measurements of two overlapping Gaussian peaks. Sharpening the peaks reduces the degree of overlap and can greatly reduce the peak area measurement error errors made by the *perpendicular drop* method (page 142). The spreadsheets listed under “Peak Sharpening” on the previous page include peak area measurement.

Curve Fitting (page 161). LeastSquares.xls and LeastSquares.odt perform polynomial least-squares fits to a straight-line model and QuadraticLeastSquares.xls and QuadraticLeastSquares.ods does the same for a quadratic (parabolic) model. There are specific versions of these spreadsheets that also calculate the concentrations of the unknowns (download complete set as CalibrationSpreadsheets.zip).

Multi-component spectroscopy (page 188). RegressionTemplate.xls and RegressionTemplate.ods (graphic with example data) perform multicomponent analysis using the matrix method for a *fixed* 5-component, 100 wavelength data set. RegressionTemplate2.xls uses a more advanced spreadsheet technique (page 353) that allows the template to *automatically adjust* to different numbers of components and wavelengths. Two examples show the *same* template with data entered for a mixture

of 5 components measured at 100 wavelengths (RegressionTemplate2Example.xls) and for 2 components at 59 wavelengths (RegressionTemplate3Example.xls).

Peak fitting (page 173). A set of spreadsheets using the Solver function to perform iterative non-linear peak fitting for multiple overlapping peak models is described here. There are versions for Gaussian and for Lorentzian peak shapes, with and without baseline, for 2-6 peak models and 100 wavelengths (with instructions for modification). All of these have file names beginning with "CurveFitter...".

Peak detection and measurement (page 234). The spreadsheet PeakAndValleyDetectionTemplate.xlsx (or PeakAndValleyDetectionExample.xlsx with sample data), is a simple peak and valley detector that defines a peak as any point with lower points on both sides and a valley as any point with higher points on both sides (see page 492). The spreadsheet PeakDetection.xls implements as more selective derivative zero-crossing peak detection method described on page 236. In both cases, the input x,y data are contained in Sheet1, columns **A** and **B**, starting in row 9. (You can paste your own data there). See PeakDetectionExample.xlsx/.xls) for an example with data already pasted in. PeakDetectionDemo2.xls/xlsx is a demonstration with a user-controlled computer-generated series of peaks. PeakDetectionSineExample.xls is a demo that generates a sinusoid with an adjustable number of peaks.

An extension of that method is made in PeakDetectionAndMeasurement.xlsx (screen image), which makes the assumption that the peaks are *Gaussian* and measures their height, position, and width on the *unsmoothed* data using a *least-squares technique*, just like "findpeaksG.m". The advantage of this technique is that it eliminates the peak distortion that might result from smoothing the data to prevent false peaks arising from random noise. For the first 10 peaks found, the x,y original unsmoothed data are copied to Sheets 2 through 11, respectively, where that segment of data is subjected to a Gaussian least-squares fit, using the same technique as GaussianLeastSquares.xls. The best-fit Gaussian parameter results are copied back to Sheet1, in the table in columns **AH-AK**. (In its present form. The spreadsheet is limited to measuring 10 peaks, although it can detect any number of peaks. Also, it is limited in Smooth Width and Fit Width by the 17-point convolution coefficients). The spreadsheet is available in OpenOffice (.ods) and in Excel (.xls) and (.xlsx) formats. They are functionally equivalent and differ only in minor cosmetic aspects. An example spreadsheet, with data, is available. A demo version, with a calculated noisy waveform that you can modify, is also available. See page 273. If the peaks in the data are too much overlapped, they may not make sufficiently distinct maxima to be detected reliably. If the noise level is low, the peaks can be artificially sharped by the derivative sharpening technique described previously. This is implemented by PeakDetectionAndMeasurementPS.xlsx and its demo version PeakDetectionAndMeasurementDemoPS.xlsx.

Spreadsheets for the TFit Method (page 276): Hyperlinear Quantitative Absorption Spectrophotometry. TransmissionFittingTemplate.xls (screen image) is an empty template for a single isolated peak;TransmissionFittingTemplateExample.xls (screen image) is the same template with example data entered. TransmissionFittingDemoGaussian.xls (screen image) is a demonstration with a simulated Gaussian absorption peak with variable peak position, width, and height, plus added stray light, photon noise, and detector noise, as viewed by a spectrometer with a triangular slit function. You can vary all the parameters and compare the best-fit absorbance to the true peak height and to the conventional log(1/T) absorbance.

TransmissionFittingCalibrationCurve.xls (screen image) includes an Excel macro (page 315) that automatically constructs a calibration curve comparing the TFit and conventional log(1/T) methods, for a series of 9 standard concentrations that you can specify.

Special spreadsheet techniques (page 353): "SpecialFunctions.xlsx" (Graphic) demonstrates the applications of the MATCH, INDIRECT, COUNT, IF, and AND functions when dealing with data arrays of variable size. "IndirectLINEST.xls" (Graphic link) demonstrates the particular benefit of using the INDIRECT function in conjunction with *array* functions such as INV and LINEST.

Afterword

How this book came to be.

At the University of Maryland in the Department of Chemistry and Biochemistry, I did research in analytical chemistry and developed and taught several courses, including an upper-division undergraduate lab course in "Electronics for Chemists", which by the 1980s included a laboratory computer component and an experiment in digital data acquisition and processing dealing with the use of mathematical and numerical techniques used in the processing of experimental data from scientific instruments. Analytical chemists like myself are basically tool builders. In the early days of our profession, the tools were mainly chemical (e.g., color reagents), but in later years they included instruments (e.g., spectroscopy and chromatography), and by the late 20th century included *software* tools. When the Web became available to the academic community in the early 90s, like many instructors I set up a website with a syllabus, experiments, and other reading material for this and for my other courses for students to access.

When I retired from the University in 1999, after 30 years of service, I noticed that I was getting pageviews on that course site that came from outside the University and from outside the country, especially directed to the lab experiment in digital data processing that I had developed in the 80's, when computers were relatively new in chemistry laboratories. I started getting an increasing number of emails with questions, suggestions, and comments from people in widely varied scientific fields. Ultimately, I decided to make this a long-term retirement project and to try to broaden this beyond chemistry and my specific course. My aim is to help science workers learn and apply computer-based mathematical data processing techniques, by producing free tutorial materials that explain things intuitively rather than in mathematical formality, with coding examples, practical software, and guidance/consulting on specific projects. To make this work useful for the widest possible audience, including those with limited funding, I have made several choices:

a. Everything is free: the book (in electronic form), the software, and the help and consulting. Only the paperback version of the book, available from Amazon, must be purchased.
b. The book and documentation are available in multiple formats: HTML, PDF, and DOCX.
c. The writing is at the 11th grade level, in a plain style, devoid of unnecessary jargon, idioms, figures of speech, metaphors, cultural references, sarcasm, irony, and humor, all of which might be difficult for those with limited English skills or for machine translators.
d. Formal mathematics is minimized. I rely more on logic, practical examples, graphics, analogies, and animations to explain concepts.
e. Multiple hardware platforms can be used: PCs, Macs, Unix, and portable devices.
f. Multiple software platforms are used: Matlab, Octave, Python, Excel and Open office spreadsheets. Some are free.

Who needs this software?

Isn't software already included in every modern scientific instrument hardware purchase? This is true, especially for those who are using conventional instruments in standard ways. But many scientists are working in new research areas for which there are no commercial instruments, or they are using modifications of existing systems for which there is no software, or they are building completely new types of instruments. In some cases, the software provided with commercial instruments is inflexible, inadequately documented, or hard to use. Not every researcher or science worker likes programming, or has time for it, or is good at it. Hired programmers typically do not understand the science and in any case sooner or later move on and no longer maintain their code. Well-documented code is more important than ever. I enjoy writing and coding, so this seemed to be a niche I could fit into.

Organization

My project has five parts:

- A book, entitled "A Pragmatic Introduction to Signal Processing", available in both paper, Kindle, and in DOCX and PDF printable online formats;
- A Web site (.edu domain), with essentially the same material as the book. No sign-in or registration is required.
- Downloadable free software in several different forms, listed on the web site and page 479.
- Help and consulting via email (optionally with data attachments).
- A Facebook group and the Matlab File Exchange for announcements and public discussion.

Although the complete book is available freely in DOCX and PDF format, several readers have found it too long to print themselves and have requested a pre-printed version, which is now sold through Amazon. The on-line materials, software, help, and consulting are all free. Open-source software alternatives are available, namely Octave, Python, and OpenOffice/LibreOffice.

Methodology

My policy is that contact with users ("clients") is initiated only from the clients and is strictly in written form, in English, mostly by email or Facebook group message - not phone or *Facetime* or *Skype*. Requests for direct real-time voice or video communication are politely deflected. This is done to allow extended conversations between time zones, to preserve communications in written form, for future reference and to avoid language problems and my own age-related hearing difficulties (readers have come from at least 162 different countries). Written communications via email also allow the use of machine translation apps such as Google Translate. Moreover, clients can send examples of their data via email attachment or via Google drive.

Information about the affiliation of the client and the nature of the project is not solicited and is strictly at the discretion of the client. Client information and data are kept confidential. In many cases, I know nothing about the origin of their data and must treat it as abstract numbers. I usually do not know the age, gender, race, country of origin, level of education, experience, or employment of clients unless they tell me. I must look for clues in their writing to gauge their level of knowledge and experience and to avoid insulting them on the one hand or confusing them on the other. Everyone is welcome.

I have attempted to minimize the use of fancy formatting and special effects on my web site, to make it compatible with older operating systems and browsers. No account or registration is needed. I allow no advertising on my web pages. I minimize the use of video, but I do use simple GIF animations where it would be useful, as these can be viewed right on the web page, or from within the ".docx" version in Microsoft Word 365, without downloading any addition plug-ins or software. I test my formatting to make sure it is viewable on mobile devices (tablets, smartphones).

Influence of the Internet

There are many different countries, states, universities, departments, specialties, and journals, but only one global Internet. Most, but not all, of it is accessible to anyone with an internet connection and a computer, tablet, or smartphone. Search engines look at the entire internet, except what is behind paywalls, irrespective of academic specialization, leading to the possibility that a solution arising in one corner of scholarship will be discovered by a need in another corner. Why, for example, would a neuroscientist, or a cancer researcher, or an economist, or a linguist, or a music scholar for that matter, know anything about my work? They would surely *not* if I published only in the scientific journals of my specialty. They understandably do not read those journals. But in fact, all those types of researchers, and hundreds more from other diverse fields, have found my work by "stumbling across it" *in a search engine query*, rather than by reading scholarly publications, and some of them have found it useful enough to *cite in their own publications* (page 521). In my own academic career, I published research only in analytical chemistry journals, which are read mostly by other analytical chemists. In contrast, my Web hits, emails, and citations have come from a much wider range of scientists, engineers, researchers, instructors, and students working in academia, industry, environmental fields, medical, engineering, earth science, space, military, financial, agriculture, communications, and even in a few cases to language and musicology.

Writing

I intended my writing to be instructional, not especially scholarly or rigorous. It is unashamedly *pragmatic*, defined as "Relating to matters of fact or practical affairs, often to the exclusion of intellectual or artistic matters; practical as opposed to idealistic." For many people, too much abstract mathematics can be a barrier to understanding. I make only basic assumptions about prior knowledge beyond the usual college science-major level: minimal math background and an 11th grade (USA high school) reading level, according to several automated readability indexes. I have tried to minimize forms of speech that might confuse translators (machine and human), and I even try to minimize the use of the passive voice. I often explain the same concept more than once in different contexts because I believe that can help to make some ideas "stick" better. An important part of my writing process is *feedback from users*, by email, social media, search engine terms, questions, corrections, etc. Moreover, I also regularly re-read older sections with "fresh eyes", correcting errors, and making improvements in phrasing. Questions from readers, and even search terms in Google searches, can also suggest areas where improvements are possible.

To make access easier, I make my writing available in multiple formats: Web (Simple HTML, with graphics and silent self-running GIF animations, and a site-specific search); DOCX (editable Microsoft Word), the latest version of which displays the GIF animations running right on the page; PDF

(Portable Document Format) for printing, and paperback and Kindle versions, through Amazon's Kindle Direct Publishing program. All except for the web version has a detailed table of contents. All except the paperback and Kindle versions are free.

A paper book is usually read starting from the beginning: the table of contents and the introduction. But web site access, especially via search engines (Google, Bing, etc.), is not related to the order of pages. This is evident in the data our web server collects for web page accesses: the table of contents and introduction are *not* the most accessed. In fact, on most days there are *no visits at all* to the table of contents or to the introduction pages. This can cause a problem with sequencing the topics, which is partially reduced by including "hot links" to the table of contents and to related previous and following material. (There is an average of three internal page references per page, plus a table of contents with over 200 entries). Also, to facilitate communication, I have added a "mail-to" link to each Web page that includes my email address *and the title of the page as the subject line* (so I can tell from the email's subject line what page they were on when they clicked the mail-to link).

Software platform selection criteria

As for software platforms, I chose two types: spreadsheets (page 16) and scripted languages Matlab (page 17), Octave (page 22), Python (page 439) and a brief dip into Mathematica (page 462). All have the advantage of being multi-platform. They run on PC, Mac, Unix, even on mobile devices (tablets/iPad) and on miniature deployable devices (e.g., Python on Raspberry Pi, page 344). These are popular development environments that have large user communities with multiple contributors and are widely used in science applications. All have a degree of backward compatibility that allows for interoperability with older legacy versions. Octave and Python are free. Companies, organizations, and college campuses often have site licenses for Excel (or for Microsoft Office, which includes Excel) and for Matlab. These platforms also have the advantage that they avoid secret algorithms, that is, their algorithms can be viewed in detail by any user. Their code is distributed in "open source" and "open document" formats that are either in plain text format (such as Matlab ".m" files or Python ".py" files) or in a format that could be opened and inspected using even free software (e.g., Microsoft Excel .xls and .xlsx spreadsheets can be opened with OpenOffice or LibreOffice). For those who cannot afford expensive software, Python, OpenOffice Calc (page 16) and Octave (page 22) can be downloaded without cost.

Most of the Matlab/Octave programs in my signal processing toolkit are "functions", which are essentially modular bits of code that can be assembled together by the user in different ways, a bit like high-tech *Lego* bricks, rather than self-contained stand-alone programs with elaborate graphical user interfaces, like commercial programs. User-developed functions like mine can be downloaded and used on their own, but they can also be used just like the in-built functions that come with the language, as *components to construct something bigger*. You can write your own functions or, if you wish, you can download and use functions written by others. When using functions, you can simply ignore the internal code and use the well-defined standard inputs and outputs. This is analogous to assembling customized entertainment or home security systems using separate commercial components, which are connected with USB and HDMI ports and cables, or Bluetooth and Wi-Fi connections between smartphone, tablets, computers, security cameras, doorbells, thermostats, and printers/ earphones/

speakers, etc., *all without worrying about the internal design of each component*. I use Matlab/Octave because of its high performance, very wide popularity, and its similarity to other languages that have often been used by scientists, such as Fortran, Basic, and Pascal. I have also given many examples in Python, another function-based language that is very capable and powerful. Even so, there are other languages that have their champions and might have been valid alternatives, such as R, Mathematica, Julia, and Scilab. In the interests of time and sanity, I have limited myself, for the time being, to Matlab/Octave and Python.

I have tried to strike a balance between cost, speed, ease of use, and learning curve, and making my software usable even to those who do not read all the documentation, by providing lots of examples and demos, including animated GIFs that will play in place on any web browser or in Microsoft Word 365. Every script or function has *built-in* help that is internal to the software. You can display this built-in help simply by typing "help __" in Matlab/Octave, or "help(__)" in Python, where __ is the name of the script or function. These help files contain not only instructions but often have simple *examples of use* and in many cases include references to other similar functions. Matlab/Octave and Python (with the addition of the Spyder desktop) have code editors for inspecting and editing code, with automatic error detection. Even this is not necessary if the existing action and inputs and outputs provide all that you need. (The spreadsheet templates and their examples and demos also have built-in instructions, and most of the spreadsheet have pop-up "cell comments" on certain cells, marked by a red dot, which pop up when the mouse pointer is hovered over them, providing an explanation for the function of that cell).

Outcomes

By 2016 website had already received over 2 million page views and over 100,000 downloads of my software programs (a few hundred per month), from either my web site or from the Matlab File Exchange. By now I have received thousands of emails with comments, suggestions, corrections, questions, offers to translate, etc. Comments from readers have been overwhelmingly positive, even enthusiastic, as indicated by these verbatim excerpts from emails about the website and about my software. In fact, many of these comments are so enthusiastic that one wonders*: why, for such a nerdy topic?* After all, most people do not take the time to write to the authors of web sites. One factor is that the number of users of the global Internet is so huge that even highly specialized topics can gather a substantial audience. As they say, "A wide net catches even the rarest fish". But I also believe that part of the reason for the enthusiastic response is that software documentation is sometimes poorly written and hard to understand. I believe more effort is needed in better explaining software and how it works and where it cannot be expected to work. I try to be responsive, answering each email and acting on their suggestions and corrections. The growth in social media is also a contributing factor, because it provides for the possibility of interacting with users in developing and improving software. You can see a specific example of that for my iPeak function in the Matlab File Exchange, see https://blogs.mathworks.com/pick/2016/09/09/most-activeinteractive-file-exchange-entry/. Look at the bottom of this page to see what I mean.

Impact

Positive comments and lots of downloads are nice, but not everyone who downloads something tries it in their work, and not everyone who does try it finds it valuable enough to cite it in their publications. Most gratifyingly, as of December 2024, *over 800 publications had cited my website and programs*, based on *Google Scholar* searches, covering an extraordinarily wide range of topics in industry, environment, medical, engineering, earth science, space, military, financial, agriculture, communications, and even occasionally language and musicology. (These citations are listed beginning on page 480 in the PDF version of the book and in https://terpconnect.umd.edu/~toh/spectrum/citations.pdf)

The Future

Where will all this go in the long run? As for my own retirement project, I will eventually move on to other projects and my work will fade away after I am gone, to join the yottabytes of forgotten material on the Internet. Unless it is specifically deleted, it will probably hang around forever, like moldy books in abandoned library shelves. The techniques that I have written about may become part of the education of all science students, or they may be replaced by more sophisticated methods, or the processing of scientific data may be entrusted to artificial intelligences (AI). At the very least, some application programming will be replaced by, or aided by, AI, as described on *(page 442)*. It's now possible for complete novices to write working programs using AI. This is called "vibe coding".

References

1. Douglas A. Skoog, *Principles of Instrumental Analysis*, 3rd Edition, Saunders, Philadelphia, 1984. Pages 73-76.

2. Gary D. Christian and James E. O'Reilly, *Instrumental Analysis*, Second Edition, Allyn and Bacon, Boston, 1986. Pages 846-851.

3. Howard V. Malmstadt, Christie G. Enke, and Gary Horlick, *Electronic Measurements for Scientists*, W. A. Benjamin, Menlo Park, 1974. Pages 816-870.

4. Stephen C. Gates and Jordan Becker, *Laboratory Automation using the IBM PC*, Prentice Hall, Englewood Cliffs, NJ, 1989.

5. Muhammad A. Sharaf, Deborah L Illman, and Bruce R. Kowalski, *Chemometrics*, John Wiley and Sons, New York, 1986.

6. Peter Wentzell and Christopher Brown, Signal Processing in Analytical Chemistry, in *Encyclopedia of Analytical Chemistry*, R.A. Meyers (Ed.), p. 9764–9800, John Wiley & Sons, Chichester, 2000 (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.124.2407&rep=rep1&type=pdf)

7. Constantinos E. Efstathiou, Educational Applets in Analytical Chemistry, Signal Processing, and Chemometrics. (http://www.chem.uoa.gr/Applets/Applet_Index2.htm)

8. A. Felinger, Data Analysis and Signal Processing in Chromatography, Elsevier Science (19 May 1998).

9. Matthias Otto, Chemometrics: Statistics and Computer Application in Analytical Chemistry, Wiley-VCH (March 19, 1999). Some parts viewable in Google Books.

10. Steven W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing.* (Downloadable chapter by chapter in PDF format from http://www.dspguide.com/pdfbook.htm). This is a much more general treatment of the topic.

11. Robert de Levie, *How to use Excel in Analytical Chemistry and in General Scientific Data Analysis*, Cambridge University Press; 1 edition (February 15, 2001), **ISBN-10:**0521644844. PDF excerpt .

12. Scott Van Bramer, Statistics for Analytical Chemistry, http://science.widener.edu/svb/stats/stats.html.

13. Taechul Lee, Numerical Analysis for Chemical Engineers.

14. Educational Matlab GUIs, Georgia Institute of Technology. (http://spfirst.gatech.edu/matlab/)

15. Jan Allebach, Charles Bouman, and Michael Zoltowski, Digital Signal Processing Demonstrations in Matlab, Purdue University (http://www.ecn.purdue.edu/VISE/ee438/demos/Demos.html)

16. Chao Yang , Zengyou He and Weichuan Yu, Comparison of public peak detection algorithms for MALDI mass spectrometry data analysis, http://www.biomedcentral.com/1471-2105/10/4

17. Michalis Vlachos, A practical Time-Series Tutorial with MATLAB.

18. Laurent Duval , Leonardo T. Duarte , Christian Jutten, *An Overview of Signal Processing Issues in Chemical Sensing*.

19. Nicholas Laude, Christopher Atcherley, and Michael Heien, *Rethinking Data Collection and Signal Processing. 1. Real-Time Oversampling Filter for Chemical Measurements,* https://pubs.acs.org/doi/abs/10.1021/ac302169y

20. P. E. S. Wormer, Matlab for Chemists, http://www.math.ru.nl/dictaten/Matlab/matlab_diktaat.pdf

21. Martin van Exter, Noise and Signal Processing, http://molphys.leidenuniv.nl/~exter/SVR/noise.pdf

22. Scott Sinex, Developer's Guide to Excelets, http://academic.pgcc.edu/~ssinex/excelets/

23. R. de Levie, Advanced Excel for scientific data analysis, Oxford University Press, New York (2004)

24. S. K. Mitra, Digital Signal Processing, a computer-based approach, 4th ed, McGraw-Hill, New York, 2011.

25. “Calibration in Continuum-Source AA by Curve Fitting the Transmission Profile”, T. C. O'Haver and J. Kindervater, *J. of Analytical Atomic Spectroscopy* 1, 89 (**1986**)

26. “Estimation of Atomic Absorption Line Widths in Air-Acetylene Flames by Transmission Profile Modeling”, T. C. O'Haver and Jing-Chyi Chang, *Spectrochim. Acta* 44B, 795-809 (**1989**)

27. “Effect of the Source/Absorber Width Ratio on the Signal-to-Noise Ratio of Dispersive Absorption Spectrometry”, T. C. O'Haver, *Anal. Chem*. 68, 164-169 (**1991**).

28. “Derivative Luminescence Spectrometry”, G. L. Green and T. C. O'Haver, *Anal. Chem. 46*, 2191 (**1974**).

29. “Derivative Spectroscopy”, T. C. O'Haver and G. L. Green, *American Laboratory 7*, 15 (**1975**).

30. “Numerical Error Analysis of Derivative Spectroscopy for the Quantitative Analysis of Mixtures”, T. C. O'Haver and G. L. Green, *Anal. Chem. 48*, 312 (**1976**).

31. “Derivative Spectroscopy: Theoretical Aspects”, T. C. O'Haver, *Anal. Proc. 19*, 22-28 (**1982**).

32. “Derivative and Wavelength Modulation Spectrometry," T. C. O'Haver, *Anal. Chem. 51*, 91A (**1979**).

33. “A Microprocessor-based Signal Processing Module for Analytical Instrumentation”, T. C. O'Haver and A. Smith, *American Lab. 13*, 43 (**1981**).

34. “Introduction to Signal Processing in Analytical Chemistry”, T. C. O'Haver, *J. Chem. Educ*. 68 (**1991**)

35. “Applications of Computers and Computer Software in Teaching Analytical Chemistry”, T. C.

O'Haver, *Anal. Chem.* **68**, 521A (**1991**).

36. “The Object is Productivity”, T. C. O'Haver, *Intelligent Instruments and Computers* Mar-Apr, **1992**, p 67-70.

37. Analysis software for spectroscopy and mass spectrometry, Spectrum Square Associates (http://www.spectrumsquare.com/).

38. *Fityk*, a program for data processing and nonlinear curve fitting. (http://fityk.nieto.pl/)

39. Peak fitting in *Origin* (http://www.originlab.com/index.aspx?go=Products/Origin/DataAnalysis/PeakAnalysis/PeakFitting)

40. *IGOR Pro 6*, software for signal processing and peak fitting (http://www.wavemetrics.com/index.html)

41. *PeakFIT*, automated peak separation analysis, Systat Software Inc..

42. *OpenChrom*, open-source software for chromatography and mass spectrometry. (http://www.openchrom.net/main/content/index.php)

43. W. M. Briggs, *Do not smooth times series, you hockey puck!*, http://wmbriggs.com/blog/?p=195

44. Nate Silver, *The Signal and the Noise*: *Why So Many Predictions Fail-but Some Do not* , Penguin Press, 2012. ISBN 159420411X . A much broader look at "signal" and "noise", aimed at a general audience, but still worth reading.

45. David C. Stone, Dept. of Chemistry, U. of Toronto, Stats Tutorial - Instrumental Analysis and Calibration.

46. Streamlining Digital Signal Processing: A Tricks of the Trade Guidebook, Richard G. Lyons, John Wiley & Sons, 2012.

47. Atomic spectra lines database. http://physics.nist.gov/PhysRefData/ASD/ and http://www.astm.org/Standards/C1301.htm

48. Curve fitting to get overlapping peak areas (http://matlab.cheme.cmu.edu/2012/06/22/curve-fitting-to-get-overlapping-peak-areas)

49. Tony Owen, Fundamentals of Modern UV-Visible Spectroscopy, Agilent Corp, 2000.

50. Nicole K. Keppy, Michael Allen, Understanding Spectral Bandwidth and Resolution in the Regulated Laboratory, Thermo Fisher Scientific Technical Note: 51721. http://www.analiticaweb.com.br/newsletter/02/AN51721_UV.pdf

51. Martha K. Smith, "Common mistakes in using statistics", http://www.ma.utexas.edu/users/mks/statmistakes/TOC.html

52. Jan Verschelde, “Signal Processing in MATLAB”, http://homepages.math.uic.edu/~jan/mcs320s07/matlec7.pdf

53. H. Mark and J. Workman Jr, “Derivatives in Spectroscopy”, Spectroscopy 18 (12). p.106.

54. Jake Blanchard, Comparing Matlab to Excel/VBA, https://blanchard.ep.wisc.edu/PublicMatlab/Excel/Matlab_VBA.pdf

55. Ivan Selesnick, "Least-squares with Examples in Signal Processing", http://eeweb.poly.edu/iselesni/lecture_notes/least_squares/

56. Tom O'Haver, "Is there Productive Life after Retirement?", *Faculty Voice*, University of Maryland, April 24, 2014. **DOI:** 10.13140/2.1.1401.6005; URL: https://terpconnect.umd.edu/~toh/spectrum/Retirement.pdf

57. http://www.dsprelated.com/, the most popular independent internet resource for Digital Signal Processing (DSP) engineers around the world.

58. John Denker, "Uncertainty as Applied to Measurements and Calculations", http://www.av8n.com/physics/uncertainty.htm. Excellent.

59. T. C. O'Haver, Teaching and Learning Chemometrics with Matlab, *Chemometrics and Intelligent Laboratory Systems* 6, 95-103 (1989).

60. Allen B. Downey, "Think DSP", Green Tree Press, 2014. (153-page PDF download). Python code instruction using sound as a basis.

61. Purnendu K. Dasgupta, et. al, "Black Box Linearization for Greater Linear Dynamic Range: The Effect of Power Transforms on the Representation of Data", *Anal. Chem.* 2010, 82, 10143–10150.

62. Joseph Dubrovkin, Mathematical Processing of Spectral Data in Analytical Chemistry: A Guide to Error Analysis, Cambridge Scholars Publishing, 2018 and 2019, 379 pages. ISBN 978-1-5275-1152-1. Link.

63. Power Law Approach as a Convenient Protocol for Improving Peak Shapes and Recovering Areas from Partially Resolved Peaks, M. Farooq Wahab, et. al., *Chromatographia* (2018). https://doi.org/10.1007/s10337-018-3607-0.

64. T. C. O'Haver, *Interactive Computer Models for Analytical Chemistry Instruction*, https://terpconnect.umd.edu/~toh/models/, **1995**.

65. T. C. O'Haver, *Interactive Simulations of Basic Electronic and Operational Amplifier Circuits*, https://terpconnect.umd.edu/~toh/ElectroSim, (**1996**)

66. Signal Processing at Rice University. (http://dsp.rice.edu/software/)

67. Steven Pinker, The Sense of Style: The Thinking Person's Guide to Writing in the 21st Century, New York, NY: Penguin, **2004**.

68. Joseph Dubrovkin, https://www.researchgate.net/profile/Joseph-Dubrovkin

69. Separations at the Speed of Sensors, D. C. Patel, M. Farooq Wahab, T. C. O'Haver, and Daniel W. Armstrong, Analytical Chemistry **2018** 90 (5), 3349-3356, DOI: 10.1021/acs.analchem.7b04944

70. MF Wahab, TC O'Haver, F. Gritti, G. Hellinghausen, and DW Armstrong, "Increasing chromatographic resolution of analytical signals using derivative enhancement approach," Talanta, vol. 192, pp. 492–499, **2019**

71. Yuri Kalambet, "Reconstruction of exponentially modified functions", **2019**. DOI: 10.13140/RG.2.2.12482.84160. Link.

72. Yuri Kalambet, Yuri Kozmin, Andrey Samokhin, "Comparison of integration rules in the case of very narrow chromatographic peaks", *Chemometrics and Intelligent Laboratory Systems* 179, May **2018**. DOI: 10.1016/j.chemolab.2018.06.001

73. Yuri Kalambet, et. al., "Reconstruction of chromatographic peaks using the exponentially modified Gaussian function", *Journal of Chemometrics* June **2011**, 25(7):352 - 356. DOI: 10.1002/cem.1343

74. Allen, L. C., Gladney, H. M., Glarum, S. H., J. Chem. Phys. 40, 3135 (**1964**)

75. J. W. Ashley, Charles N. Reilley, "De-Tailing and Sharpening of Response Peaks in Gas Chromatography", *Anal. Chem.*, 37, 6, 626-630, **1965**.

76. M. Johansson, M. Berglund and D. C. Baxter, "Improving accuracy in the quantitation of overlapping, asymmetric, chromatographic peaks by deconvolution: theory and application to coupled gas chromatography atomic absorption spectrometry", *Spectrochemica Acta*, Vol 48B, p. 1393-1409, **1993**.

77. S. Sterlinski, "A Method for Resolution Enhancement of Interfering Peaks in Ge(Li) Gamma-Ray Spectra", *J. of Radioanalytical Chemistry*, 31, 195-226, **1976**.

78. "Importance of academic blogs", Teachers Insurance and Annuity Association of America-College Retirement Equities Fund, New York, NY. https://careerpurpose.com/industries/education/academic-blogs.

79. Robi Polikar, The Wavelet Tutorial, http://web.iitd.ac.in/~sumeet/WaveletTutorial.pdf

80. C. Valens, "A Really Friendly Guide to Wavelets", http://agl.cs.unm.edu/~williams/cs530/arfgtw.pdf

81. Brani Vidakovic and Peter Mueller, "Wavelets for Kids",http://www.gtwavelet.bme.gatech.edu/wp/kidsA.pdf

82. Amara Graps, "An Introduction to Wavelets" https://www.eecis.udel.edu/~amer/CISC651/IEEEwavelet.pdf

83. Muhammad Ryan, "What is Wavelet and How We Use It for Data Science", https://towardsdatascience.com/what-is-wavelet-and-how-we-use-it-for-data-science-d19427699cef

84. Michael X. Cohen, "A better way to define and describe Morlet wavelets for time-frequency analysis", *NeuroImage*, Volume 199, 1 October **2019**, Pages 81-86.

85. Wahab M. F, O'Haver T. C., "Wavelet transforms in separation science for denoising and peak overlap detection." *J Sep Sci*. 43 (9-10) 1615–2012 (**2020**). ISSN 1615-9306; https://doi.org/10.1002/jssc.202000013

86. G. K. Wertheim, *J. of Electron Spectroscopy and Related Phenomena*, 6 (**1975**) 239-251.

87. R. E. Sturgeon, et. al., Atomization in graphite-furnace atomic absorption spectrometry. Peak-height method vs. integration method of measuring absorbance. *Anal. Chem*. 47, 8, 1240–1249 (1975) https://doi.org/10.1021/ac60358a039

88. Sunaina et al, "Calculating numerical derivatives using Fourier transform: some pitfalls and how to avoid them", *Eur. J. Phys.* 39 ,065806, **2018**

89. Sinex, Scott A, Investigating types of errors. *Spreadsheets in Education* 2.1 (**2005**): 115-124.

90. Catherine Perrin, Beata Walczak, and Désiré Luc Massart, "Quantitative Determination of the Components in Overlapping Chromatographic Peaks Using Wavelet Transform", *Analytical Chemistry* **2001** 73 (20), 4903-4917 DOI: 10.1021/ac010416a

91. F. Gritti, S. Besner, S. Cormier, M. Gilar, Applications of high-resolution recycling liquid chromatography: from small to large molecules, *Journal of Chromatography* A 1524 (**2017**) 108-120.

92. Desimoni E. and Brunetti B., "About Estimating the Limit of Detection by the Signal to Noise Approach", *Pharmaceutica Analytica Acta* 67, 4, **2015**. DOI: 10.4172/2153-2435.100035. PDF link.

93. Royal Society of Chemistry Analytical Methods Committee, "Recommendations for the Definition, Estimation and Use of the Detection Limit", *Analyst*, Feb. **1987**, vol.112, p. 199.

94. "MATLAB vs Python: Why and How to Make the Switch", https://realpython.com/matlab-vs-python/

95. MLAB, an advanced mathematical and statistical modeling system, by Gary Knott.

96. NIST Engineering Statistics Handbook: https://www.itl.nist.gov/div898/handbook/index.htm

97. "Why and How Savitzky–Golay Filters Should Be Replaced", Michael Schmid, David Rath, and Ulrike Diebold, *ACS Measurement Science* Au **2022** 2 (2), 185-196. DOI: 10.1021/acsmeasuresciau.1c00054

98. Farooq Wahab and Thomas C. O'Haver, "Peak deconvolution with significant noise suppression and stability using a facile numerical approach in in Fourier space", *Chemometrics and Intelligent Laboratory Systems* 235, **2023.** https://authors.elsevier.com/c/1gVwgcc6MExCW

99. M.F. Wahab, F. Gritti, T.C. O'Haver, Discrete Fourier transform techniques for noise reduction and digital enhancement of analytical signals, *TrAC, Trends Anal. Chem*., 143, Article 116354 (**2021**)

100. Aditi Gupta, *et. al*., A-TSPD: autonomous-two stage algorithm for robust peak detection in online time series | Cluster Computing (springer.com)

101. Nick Bilton, "Future Tense", Vanity Fair, Oct. 2013. Future Tense | Vanity Fair | October 2023

102. M. Farooq Wahab, et. al, "Solving Advanced Task-Specific Problems in Measurement Sciences with Generative AI", *Analytical Chemistry*, 96, (**2024**)

Publications that cite the use of my book, programs and/or documentation

Updated annually. Last updated December 2024.

If you have published a paper using these programs that you would like me to include here, please email the paper, or a citation to it, to Tom O'Haver at toh@umd.edu

1. Poppi, R. J., Vazquez, P. A., & Pasquini, C. (**1992**). Fast scanning Hadamard spectrophotometer. *Applied Spectroscopy*, 46(12), 1822-1827.

2. Ghatee, M. H., and A. Boushehri. "Modulation of the integrated rate equation of a composite system for the kinetic parameters." *Chemometrics and intelligent laboratory* systems 25.1 (**1994**): 43-49.

3. C.W.K. Chow, D.E. Davey, Dennis Mulcahy, T.C.W. Yeow , Signal enhancement of potentiometric stripping analysis using digital signal processing, *Analytica Chimica* Acta 307(1):15-26, April **1995**
DOI: 10.1016/0003-2670(95)00023-S

4. Ringe, Steven A. "Hydrogen-extended defect interactions in heteroepitaxial InP materials and devices." *Solid-State Electronics* 41.3 (**1997**): 359-380.

5. Chow, Christopher WK, David Edward Davey, and D. E. Mulcahy. "Signal filtering of potentiometric stripping analysis using Fourier techniques." *Analytica chimica acta* 338.3 (**1997**): 167-178.

6. Leung, Alexander Kai-man, Foo-Tim Chau, and Jun-bin Gao. "Wavelet transform: a method for derivative calculation in analytical chemistry." *Analytical* Chemistry 70.24 (**1998**): 5222-5229.

7. Harris, D. C. (**1998**). “Spektralphotometer”. In *Lehrbuch der Quantitativen Analyse* (pp. 695-746). Vieweg+ Teubner Verlag. Link.

8. Keyhani, Ali, Wenzhe Lu, and Gerald T. Heydt. "Neural network based composite load models for power system stability analysis." IEEE International Conference on Computational Intelligence for Measurement Systems and Applications. **2005**.

9. Fernández, Mario, and J. Ricardo Pérez-Correa. "Instrumentation for Monitoring SSF Bioreactors." Solid-State Fermentation Bioreactors. Springer Berlin Heidelberg, **2006**. 363-374

10. Richard Graham , Ring Laser Gain Media, Thesis, http://ir.canterbury.ac.nz/bitstream/10092/1377/1/thesis_fulltext. pdf (**2006**)

11. Sheaff, Chrystal N., Delyle Eastwood, and Chien M. Wai. "Increasing selectivity for TNT-based explosive detection by synchronous luminescence and derivative spectroscopy with quantum yields of selected aromatic amines." *Applied spectroscopy* 61.1 (**2007**): 68-73.

12. Hovick, James W., Michael Murphy, and J. C. Poler. "" Audibilization" in the chemistry laboratory: An introduction to correlation techniques for data extraction." *J. Chem. Educ* 84.8 (**2007**): 1331.

13. de Aragão, Bernardo José Guilherme, and Younes Messaddeq. "PEAK SEPARATION IN SPECTRAL ANALYSIS." (**2007**). Link.

14. Ingersoll, Justin Edward. A Regularization Technique for the Analysis of Photographic Data Used in Chemical Release Wind Measurements. ProQuest, **2008**.

15. Dinesh, S. "The Effect of Smoothing on the Extraction of Drainage Networks from Simulated Digital Elevation Models." *Journal of Applied Sciences Research* 4.11 (**2008**): 1356-1360.

16. Jed A. Meltzer, Hitten P. Zaveri, Irina I. Goncharova, Marcello M. Distasio, Xenophon Papademetris, Susan S. Spencer, Dennis D. Spencer and R. Todd Constable, ”Effects of Working Memory Load on Oscillatory Power

in Human Intracranial EEG", *Cereb. Cortex* (**2008**) 18 (8): 1843-1855. doi: 10.1093/cercor/bhm213

17. Sheaff, Chrystal N., et al. "Fluorescence detection and identification of tagging agents and impurities found in explosives." *Applied spectroscopy* 62.7 (**2008**): 739-746.

18. "A regularization technique for the analysis of photographic data used in chemical release wind measurements", JE Ingersoll, **2008**, books.google.com

19. "An application of detection function for the eye blinking detection", Pander, T. Przybyla, T. ; Czabanski, Human System Interactions **2008** Conference: 25-27 May 2008, Page(s): 287- 291

20. "Isotopically labeled oxygen studies of the NOx exchange behavior of La2CuO4 to determine potentiometric sensor response mechanism" F.M. Van Assche IV, E.D. Wachsman, *Solid State Ionics,* Volume 179, Issue 39, 15 December **2008**, Pages 2225–2233

21. "High-speed laryngoscopic evaluation of the effect of laryngeal parameter variation on aryepiglottic trilling." Moisik, Scott R., John H. Esling, and Lise CrevierBuchman. poster, http://www.ncl.ac.uk/linguistics/assets/documents/ MoisikEslingBuchman_NewcastleP haryngealsPoster_2009. Pdf (**2009**).

22. Tricas, Marazico, and Juan Ignacio. "Auto configuration dans LTE: procédés de mesure de l'occupation du canal radio pour une utilisation optimisée du spectre." ,"Auto configuration in LTE: measuring the occupancy of the radio channel for optimized use of the spectrum" (**2009**). PDF link.

23."Early age concrete strength monitoring with piezoelectric transducers by the harmonic frequencies method", Thomas J. Kelleher, **2009**.http://www.engin.swarthmore.edu/e90/2008/reports/Thomas%20Kelleher.pdf

24. "Information management for high content live cell imaging", Daniel Jameson, David A Turner, John Ankers, Stephnie Kennedy, Sheila Ryan, Neil Swainston, Tony Griffiths, David G Spiller, Stephen G Oliver, Michael RH White, Douglas B Kell and Norman W Paton, *BMC Bioinformatics* **2009**, 10:226 doi:10.1186/1471-2105-10-2263

25. "Human-Computer Systems Interaction: Backgrounds and Applications", edited by Zdzislaw S. Hippe, Juliusz Lech Kulikowski, Springer (Sep 30, **2009**), page 191.

26. "Multiplexed DNA detection using spectrally encoded porous SiO2 photonic crystal particles", SO Meade, MY Chen, MJ Sailor, *Anal. Chem*., **2009**, 81 (7), pp 2618–2625. DOI: 10.1021/ac802538x

27. "Prolonged stimulus exposure reveals prolonged neurobehavioral response patterns, Brett A. Johnson, Cynthia C. Woo, Yu Zeng, Zhe Xu, Edna E. Hingco, Joan Ong, Michael Leon. *The Journal of Comparative Neurology*, Volume 518, Issue 10, pages 1617–1629, 15 May **2010**

28."Alternative Measures of Phonation: Collision Threshold Pressure and Electroglottographic Spectral Tilt: Extra: Perception of Swedish Accents." Enflo, Laura. (**2010**). Full Text.

29. Botcharova, Maria. "Changes in structure of EEG-EMG coherence during brain development: analysis of experimental data and modeling of putative mechanisms." (**2010**) [PDF] from ucl.ac.uk

30."Vowel Dependence for Electroglottography and Audio Spectral Tilt", L Enflo, Proceedings of Fonetik, **2010**.
31. "Rapid and accurate detection of plant miRNAs by liquid northern hybridization.", Wang, Xiaosu, Yongao Tong, and Shenghua Wang. International journal of molecular sciences 11.9 (**2010**): 3138-3148.

32. Nusz, G. J. (**2010**). Label-free biodetection with individual plasmonic nanoparticles (Doctoral dissertation, Duke University).

33. Khudaish, Emad A., and Aysha A. Al Farsi. "Electrochemical oxidation of dopamine and ascorbic acid at a palladium electrode modified with in situ fabricated iodine-adlayer in alkaline solution." Talanta 80.5 (**2010**): 1919-1925.

34. "Advances in Music Information Retrieval", edited by Zbigniew W. Ras, Alicja Wieczorkowska, Springer, **2010**, page 135.

35. Bilal, M., Sharif, M., Jaffar, M. A., Hussain, A., & Mirza, A. M. (2010, May). Image restoration using modified hopfield fuzzy regularization method. In Future Information Technology (FutureTech), **2010** 5th International Conference on (pp. 1-6). IEEE.

36. Rim, Jung Ho. "Preparation and Characterization of Sources for Ultra-high Resolution Microcalorimeter Alpha Spectrometry." The Pennsylvania State University (**2010**). PDF link.

37. Xiaosu Wang , Yongao Tong and Shenghua Wang, Rapid and Accurate Detection of Plant miRNAs by Liquid Northern Hybridization, Int. J. Mol. Sci. **2010**, 11(9), 3138-3148; doi:10.3390/ijms11093138

38. “Radio Frequency Fuel Gauging with Neuro-Fuzzy Inference Engine For Future Spacecrafts”. Kumagai, A., Liu, T. I., & Sul, D. In Proceedings of the 10th IASTED, International Conference, **2010** (Vol. 674, No. 020, p. 243).

39. "Automatic Seizure Detection in ECoG by Differential Operator and Windowed Variance," Majumdar, K.K.; Vardhan, P., Neural Systems and Rehabilitation Engineering, IEEE Transactions on, vol.19, no.4, pp.356,365, Aug. **2011**

40. "Genetic algorithm with peaks adaptive objective function used to fit the EPR powder spectrum", Sebastian Grzegorz Żurek, Applied Soft Computing, Volume 11, Issue 1, January **2011**, Pages 1000–1007

41. "Determination of sea conditions for wave energy conversion by spectral analysis", B Yagci, P Wegener, EEE Transactions on Power Delivery, 18(2): 372–376, **2011**.

42. "Push-broom hyperspectral imaging for elemental mapping with glow discharge optical emission spectrometry", G Gamez, D Frey, J Michler - J. Anal. At. Spectrom., **2011**, 65, 85–98

43."Dual-order snapshot spectral imaging of plasmonic nanoparticles", Gregory J. Nusz, Stella M. Marinakos, Srinath Rangarajan, and Ashutosh Chilkoti, Applied Optics, Vol. 50, Issue 21, pp. 4198-4206 (**2011**) http://dx.doi.org/10.1364/AO.50.004198

44. Sugandharaju, Ravi Kumar Chatnahalli. “Gaussian Deconvolution and MapReduce Approach for Chipseq Analysis”. Dissertation. University of Cincinnati, **2011**.

45. "Parallel Deconvolution Algorithm in Perfusion Imaging" F Zhu, DR Gonzalez, T Carpenter, Healthcare Informatics, Imaging and Systems Biology (HISB), 2011 First IEEE International Conference, 26-29 July **2011**

46. "Field observations of infragravity waves and their behaviour on rock shore platforms" Edward P. Beetham, Paul S. Kench, Earth Surface Processes and Landforms, Volume 36, Issue 14, pages 1872–1888, November **2011**

47. "Majority Voting: Material Classification by Tactile Sensing Using Surface Texture", Jamali, N., Sammut, C., IEEE Transactions on Robotics, Volume: 27, Issue: 3, Page(s): 508 - 521, June **2011**

48. Yuan, Yuan, Yishan Luo, and Albert Chung. "VE-LLI-VO: Vessel enhancement using local line integrals and variational optimization." IEEE Transactions on Image Processing 20.7 (**2011**): 1912-1924.

49. ”Demand Estimation with Automated Meter Reading in a Distribution Network”,Aksela, K. and Aksela, M. , J. Water Resour. Plann. Manage., 137(5), 456–467 (**2011**). doi: 10.1061/(ASCE) WR.1943-5452.0000131

50. Ochoa, Jeimy Catherine Millán. Design and Development of a Localization System for a Sensor Network in Collective Symbiotic Organisms. Diss. Universitätsbibliothek der Universität Stuttgart, **2011**.

51. "Genetic algorithm with peaks adaptive objective function used to fit the EPR powder spectrum", Sebastian Grzegorz Żurek, Journal Applied Soft Computing archive. Volume 11, Issue 1, January **2011**, pages 1000-1007

52. Hornung, J. P. (**2011**). Exploring the potential for using deep-sea bamboo corals (Isidella sp.) for paleoceanographic reconstructions (Doctoral dissertation).

53. Boll, Marie-Theres. Ein neues Konzept zur automatisierten Bewertung von Fertigkeiten in der minimal invasiven Chirurgie für Virtual-Reality-Simulatoren in GridUmgebungen. Vol. 38. KIT Scientific Publishing, **2011**. Link.

54. "Development of ECG signal interpretation software on Android 2.2, Hermawan, K.; Iskandar, A.A.; Hartono, R.N., "Instrumentation, Communications, Information Technology, and Biomedical Engineering (ICICI-BME), 2011 2nd International Conference, vol., no., pp.259,264, 8-9 Nov. **2011** doi: 10.1109/ICICI-BME.2011.6108621

55. Choi, Sheng Heng. "Signal processing and amplifier design for inexpensive genetic analysis instruments." (**2011**). https://era.library.ualberta.ca/files/qr46r139p#.WifTkEqnGUk

56. Hoffman, Galen Brandt. Direct Write of Chalcogenide Glass Integrated Optics Using Electron Beams. Diss. The Ohio State University, **2011**. Direct link.

57. Bai, Er-Wei, et al. "Detection of radionuclides from weak and poorly resolved spectra using Lasso and subsampling techniques." Radiation Measurements 46.10 (**2011**): 1138-1146.

58. Sugandharaju, Ravi Kumar Chatnahalli. Gaussian Deconvolution and MapReduce Approach for Chipseq Analysis. Diss. University of Cincinnati, **2011**.

59."Automated peak alignment for nucleic acid capillary electrophoresis data by dynamic programming". Fethullah Karabiber, Kevin Weeks, and Oleg V. Favorov. In Proceedings of the 2nd ACM Conference on Bioinformatics, Computational Biology and Biomedicine (BCB '11). ACM, New York, NY, USA, **2011**. pages 544-546. DOI=10.1145/2147805.2147895 http://doi.acm.org/10.1145/2147805.2147895

60. Shin, Sung-Hwan, et al. "Mass estimation of impacting objects against a structure using an artificial neural network without consideration of background noise." Nuclear Engineering and Technology 43.4 (**2011**): 343-354.

61. Taibo, María Luisa Gómez, et al. "Matching needs and capabilities with assistive technology in an amyotrophic lateral sclerosis patient." Accessibility, Inclusion and Rehabilitation using Information Technologies (**2011**): 21.

62. Paul, Ruma R., Victor C. Valgenti, and Min Sik Kim. "Real-time Netshuffle: Graph distortion for on-line anonymization." Network Protocols (ICNP), 2011 19th IEEE International Conference on. IEEE, **2011**.

63. Lopez-Castellanos, V. (**2011**). Ultrawideband time domain radar for time reversal applications (Doctoral dissertation, The Ohio State University).

64. "Electricity gain via integrated operation of turbine generator and cooling tower using local model network." Pan, Tian-Hong, et al. Energy Conversion, IEEE Transactions on 26.1 (**2011**): **245**-255.

65. "Dynamic analysis of electronic devices' power signatures, Marcu, M.; Cernazanu, C., " Instrumentation and Measurement Technology Conference (I2MTC), **2012** IEEE International , vol., no., pp.117,122, 13-16 May **2012.** doi: 10.1109/I2MTC.2012.6229562

66. "Experimental comparison among pileup recovery algorithms for digital gamma ray spectroscopy" El-Tokhy, M.S. Mahmoud, I.I. ; Konber, H.A. Informatics and Systems (INFOS), 2012 8th International Conference on 14-16, May **2012**

67. Kwon, Soonil. "Voice-driven sound effect manipulation." International Journal of Human-Computer Interaction 28.6 (**2012**): 373-382.

68. "Distributed representation of chemical features and tunotopic organization of glomeruli in the mouse olfactory bulb" Limei Maa, Qiang Qiua, Stephen Gradwohla, Aaron Scotta, Elden Q. Yua, Richard Alexandera, Winfried Wiegraebea, and C. Ron Yu, Proceeding of the National Academy of Sciences, April 3, **2012** vol. 109, no. 14, pages 5481-5486.

69. Hofler, Alicia S. Optimization Framework for a Radio Frequency Gun Based\ Injector. Old Dominion University, PhD dissertation, **2012**.

70. "A Robust Heart Sound Segmentation and Classification Algorithm using Wavelet Decomposition and Spectrogram." Deng, Yiqi, and Peter J. Bentley. **2012**. Full text:

http://www.peterjbentley.com/heartworkshop/challengepaper3.pdf

71.“Detecting STR peaks in degraded DNA samples”. Marasco, E., Ross, A., Dawson, J., Moroose, T., & Ambrose, T.Proc. of 4th International Conference on Bioinformatics and Computational Biology (BICoB), (Las Vegas, USA), March **2012**. Full text: http://www.cse.msu.edu/~rossarun/pubs/RossDNAEnhanement_BICoB2011.pdf

72. "Saccades detection in optokinetic nystagmus-a fuzzy approach." PANDER, Tomasz, et al. , Journal of Medical Informatics & Technologies 19 (**2012**): 33-39.

73. "Grain-size properties and organic-carbon stock of Yedoma Ice Complex permafrost from the Kolyma lowland, northeastern Siberia", J Strauss, L Schirrmeister, S Wetterich, Andreas Borchers, Sergei P. Davydov, Global Biogeochemical Cycles, Volume 12, **2012**.

74."An Early Prediction of Cardiac Risk using Augmentation Index Developed based on a Comparative Study." Manimegalai, P., Delpha Jacob, and K. Thanushkodi., International Journal of Computer Applications 50 (2012). Abstract.

75. “Determinação Da Estabilidade Oxidativa De Biocombustíveis,” Bruno A. F. Vitorino, Franz H. Neff, Elmar U. K. Melcher, Antonio M. N. Lima, Anais do XIX Congresso Brasileiro de Automática, CBA 2012. http://www.eletrica.ufpr.br/anais/cba/2012/Artigos/100018.pdf

76."Efficacy of Differential Operators in Brain Electrophysiological Signal Processing: A Case Study in Epilepsy."Majumdar, Kaushik, and Pratap Vardhan. 2012 Full text.

77. Snider, W. (**2012**). Electro-optically Tunable Microring Resonators for Non-Linear Frequency Modulated Waveform Generation (Doctoral dissertation, Texas A & M University).

78."9.0 Experimental–Two-Dimensional GCxGC." Technologies towards the Development of a Lab-on-a-Chip GCxGC for Environmental Research (**2012**). Full text. A Thesis by Jaydene Halliday, BSc MRSC

79. "BaNa: A hybrid approach for noise resilient pitch detection," He Ba; Na Yang; Demirkol, I.; Heinzelman, W., Statistical Signal Processing Workshop (SSP), **2012** IEEE , vol., no., pp.369,372, 5-8 Aug. 2012. doi: 10.1109/SSP.2012.6319706

80. Tripathi, Ashish. THE NEW IMAGE PROCESSING ALGORITHM FOR\ QUALITATIVE AND QUANTITATIVE STM DATA ANALYSIS. Diss. **2012**.

81. Skelton, Martin. "Diffusion of Innovation System Elements-A Novel Method to Study Technology Development and Its Application to Wind Power." (**2012**). [PDF] from chalmers.se

82. Pander, T., et al. "A new method of saccadic eye movement detection for optokinetic nystagmus analysis." Engineering in Medicine and Biology Society (EMBC), **2012** Annual International Conference of the IEEE. IEEE, **2012**.

83. Mahmoud, I. I., M. S. El_Tokhy, and H. A. Konber. "Pileup recovery algorithms for digital gamma ray spectroscopy." Journal of Instrumentation 7.09 (**2012**): P09013.

84. Zhu, Fan, et al. "Parallel perfusion imaging processing using GPGPU." Computer methods and programs in biomedicine 108.3 (**2012**): 1012-1021.

85. Cuss, C. W., and Celine Guéguen. "Determination of relative molecular weights of fluorescent components in dissolved organic matter using asymmetrical flow fieldflow fractionation and parallel factor analysis." Analytica chimica acta 733 (**2012**): 98- 102.

86. Olugboji, Oluwafemi A., and Jack M. Hale. "Development of Damage Reconstruction Techniques from Impulsive Events Based on Measurements Made Remotely." ASME 2012 International Mechanical Engineering Congress and Exposition. American Society of Mechanical Engineers, **2012**.

87. Dickson, B., and M. Craig. "Deconvolving gamma-ray logs by adaptive zone refinement." Geophysics 77.4 (**2012**): D159-D169.

88. “SmartBells: RFID-Enhanced System to Monitor Free Weight Exercises. "Chaudhri, Rohit, and Gaetano Borriello. **2012** Full text.

89. "Diffusion of Innovation System Elements-A Novel Method to Study Technology Development and Its Application to Wind Power." Skelton, Martin. (**2012**). Fulltext.

90. Grotenhuis, Michael Gary. "An Overview of the Maximum Entropy Method of Image Deconvolution." A University of Minnesota–Twin Cities “Plan B” Master’s paper, **2012**.

91. "On comet attitude determination of Rosetta lander Philae through nonlinear optimal system identification." Caputo, Gianluca. (**2012**). Full text.

92. Valadares[1], D. C., Vitorino, B. A., Neta, M. L. N., Batista, E. S., Santos, M. V., Neff, F. H., & Melcher, E. N. (**2012**). System for Analysis of the Biodiesel Quality.

93. Mukhopadhyay, C. K., et al. "Acoustic emission during fracture toughness tests of SA333 Gr. 6 steel." Engineering Fracture Mechanics 96 (**2012**): 294-306.

94. Huang, Zifang. "Knowledge-Assisted Sequential Pattern Analysis: An Application in Labor Contraction Prediction." (**2012**). PDF link.

95. van de Voort, Frederik R., and David Pinchuk. "System and Method for Determining Base Content of a Hydrophobic Fluid." U.S. Patent Application 13/171,566.

96. Hoerndli, Frédéric J., et al. "Kinesin-1 regulates synaptic strength by mediating the delivery, removal, and redistribution of AMPA receptors." Neuron 80.6 (**2013**): 1421-1437.

97. Brockie, Penelope J., et al. "Cornichons control ER export of AMPA receptors to regulate synaptic excitability." Neuron 80.1 (**2013**): 129-142.

98. Žáčik, Michal. Šumová spektroskopie pro biologii. Diss. Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií, **2013**.

99. Phillips, James William, and Yi Jin. "Systems and methods for modulating the electrical activity of a brain using neuro-EEG synchronization therapy." U.S. Patent No. 8,465,408. 18 Jun. **2013**.

100. Moon, Jim, et al. "Body-worn vital sign monitor." U.S. Patent No. 8,364,250. 29 Jan. **2013**.
101. Hao, Manzhao, et al. "Corticomuscular Transmission of Tremor Signals by Propriospinal Neurons in Parkinson's Disease." PloS one 8.11 (**2013**): e79829.

102. McCOMBIE, Devin, Marshal Dhillon, and Matt Banet. "Method for measuring patient motion, activity level, and posture along with PTT-based blood pressure." U.S. Patent No. 8,475,370. 2 Jul. 2013.

103. Banet, Matt, Devin McCombie, and Marshal Dhillon. "Body-worn monitor for measuring respiration rate." U.S. Patent No. 8,545,417. 1 Oct. **2013**.

104. Banet, Matt, and Jim Moon. "Body-worn vital sign monitor." U.S. Patent No. 8,591,411. 26 Nov. **2013**.

105. Mccombie, Devin, et al. "Alarm system that processes both motion and vital signs using specific heuristic rules and thresholds." U.S. Patent No. 8,594,776. 26 Nov. **2013**.

106. Banet, Matt, Marshal Dhillon, and Devin McCombie. "Body-worn system for measuring continuous non-invasive blood pressure (cNIBP)." U.S. Patent No. 8,602,997. 10 Dec. **2013**.

107. Moon, Jim, et al. "Body-worn pulse oximeter." U.S. Patent No. 8,437,824. 7 May **2013**.

108. Cheng, Chunmei, et al. "Remote sensing estimation of Chlorophyll and suspended sediment concentration in turbid water based on spectral separation." Optik-International Journal for Light and Electron Optics 124.24 (**2013**): 6815-6819.

109. Phillips, James William, and Yi Jin. "Systems and methods for neuro-EEG synchronization therapy." U.S. Patent No. 8,585,568. 19 Nov. **2013**.

110. Khvostichenko, Daria S., et al. "An X-ray transparent microfluidic platform for screening of the phase behavior of lipidic mesophases." Analyst 138.18 (**2013**): 5384- 5395.

111. “A signal alignment method based on DTW with new modification”, Karabiber, F. ; Bilgisayar Muhendisligi Bolumu ; Balcilar, M. Signal Processing and Communications Applications Conference (SIU), **2013** 21st , 24-26 April 2013 . ISBN: 978-1-4673-5562-9; DOI: 10.1109/SIU.2013.6531176

112. “An automated signal alignment algorithm based on dynamic time warping for capillary electrophoresis data”, Turkish Journal of Electrical Engineering & Computer Sciences , Fethullah KARABİBER , 21, (**2013**), 851-863. Full text: pdf

113. "Traditional Asymmetric Rhythms: A Refined Model of Meter Induction Based On Asymmetric Meter Templates”, Fouloulis, Thanos, Aggelos Pikrakis, and Emilios Cambouropoulos, Proceedings of the Third International Workshop on Folk Music Analysis (FMA2013). **2013**. ISBN 978-90-70389-78-9

114. "Comparison of two methods for measuring γ-H2AX nuclear fluorescence as a marker of DNA damage in cultured human cells: applications for microbeam radiation therapy." Anderson, D., et al. , Journal of Instrumentation 8.06 (**2013**): C06008. Full text PDF.

115. Ayodeji, Olugboji Oluwafemi, Jonathan Yisa Jiya, and Jack M. Hale. "Event Reconstruction by Digital Filtering." Advances in Signal Processing 1.3 (**2013**): 48-56.

116. “A conserved aromatic residue regulating photosensitivity in short-wavelength sensitive cone visual pigments”. Kuemmel, C. M., Sandberg, M. N., Birge, R. R., & Knox, B. E. Biochemistry, 52(30), 5084-5091 (**2013**).

117. “Measurement of The Lightweight Rotor Eigenfrequencies And Tuning Of Its\ Model Parameters,” Luboš SMOLÍK, Michal HAJŽMAN, Transactions of the VŠB – Technical University of Ostrava, Mechanical Series, No. 1, **2013**, vol. LIX. FullEnglish text.

118. "Investigation of the phase separation of PNIPAM using infrared spectroscopy together with multivariate data analysis." Munk, Tommy, et al. , Polymer 54.26 (**2013**): 6947-6953. Abstract.

119. “Phase separation in InxGa1 xN (0.10< x< 0.40).” Belyaev, K. G., Rakhlin, M. ‐ V., Jmerik, V. N., Mizerov, A. M., Kuznetsova, Y. V., Zamoryanskaya, M. V., ... & Toropov, A. A. (2013). Physica Status Solidi (c), 10 (3), 527-531.

120. "Corticomuscular Transmission of Tremor Signals by Propriospinal Neurons in Parkinson's Disease." Hao, Manzhao, et al. , PloS one 8.11 (**2013**): e79829.

121. "Sickle-shaped voxel approach to enhance automatic reclaiming operation using bucket wheel reclaimer," Maung Thi Rein Myo; Tien-Fu Lu, Industrial Electronics and Applications (ICIEA), 2013 8th IEEE Conference on , vol., no., pp.1700,1705, 19- 21 June **2013**. doi: 10.1109/ICIEA.2013.6566642

122. "Review of software tools for design and analysis of large-scale MRM proteomic datasets." Colangelo, Christopher M., et al., Methods 61.3 (**2013**): 287-298.

123. Carabetta, Valerie J., et al. "A complex of YlbF, YmcA and YaaT regulates sporulation, competence and biofilm formation by accelerating the phosphorylation of Spo0A." Molecular microbiology 88.2 (**2013**): 283-300.

124. Web, N. L. P. M. L., and Andrew Rosenberg. "Ensemble Methods." (**2013**).

125. Cannon, Robert William, "Automated Spectral Identification of Materials using Spectral Identity Mapping", **2013**, Master of Science in Chemistry, Cleveland State University, College of Sciences and Health Professions.

126. MS Freeman, ZI Cleveland, Y Qi , Enabling hyperpolarized 129Xe MR spectroscopy and imaging of pulmonary gas transfer to the red blood cells in transgenic mice expressing human hemoglobin", Magnetic Resonance in Medicine, Volume 70, Issue 5, pages 1192–1199, November **2013**

127. SMOLÍK, Luboš, and Michal HAJ ˇZMAN. "MEASUREMENT OF THE LIGHTWEIGHT ROTOR EIGENFREQUENCIES AND TUNING OF ITS MODEL PARAMETERS . Transactions of the VSB – Technical University of Ostrava, Mechanical Series ˇ No. 1, **2013**, vol. LIX article No. 1942

128. Kumssa, Aida Meredassa. "Tablet User Interface Evaluation for a Portable Ultrasound System and Real-time Doppler Spectrum Processing." (**2013**).

129. Circuit level defects in the developing neocortex of Fragile X mice, J Tiago Gonçalves, James E Anstey, Peyman Golshani , Carlos Portera-Cailliau, Nature Neuroscience 16, 903–909 (**2013**) doi:10.1038/nn.3415

130. A Baradarani, J Sadler, JRB Taylor , High-resolution blood flow imaging through the skull, Electronics Letters, vol. 40, no. 13, **2014**, pp. 798–799.

131. Singh, R. (**2014**). Tune Measurement at GSI SIS-18: Methods and Applications (Doctoral dissertation, Technische Universität).

132. Pander, Tomasz, et al. "An automatic saccadic eye movement detection in an optokinetic nystagmus signal." Biomedical Engineering/Biomedizinische Technik 59.6 (**2014**): 529-543.

133. "Demonstration of Large Coupling-Induced Phase Delay in Silicon Directional Cross-Couplers," Westerveld, W.J.; Pozo, J.; Leinders, S.M.; Yousefi, M.; Urbach, H.P., Selected Topics in Quantum Electronics, IEEE Journal of , vol.20, no.4, pp.1,6, July-Aug. **2014**, doi: 10.1109/JSTQE.2013.2292874

134. "Probabilistic peak detection for first-order chromatographic data", M. Lopatka, G. Vivo-Truyols, M.J. Sjerps, Analytical Chemica Acta, **2014** DOI: http://dx.doi.org/10.1016/j.aca.2014.02.015

135. "A recursive algorithm for optimizing differentiation." Mashreghi, Ali, and Hadi Sadoghi Yazdi. Journal of Computational and Applied Mathematics 263 (**2014**): 1-13.

136. Cade, D. E. (**2014**). Detection, classification and ecology of acoustic scattering layers (Doctoral dissertation).

137. Grubišić, Vladimir, et al. "Heterogeneity of myotubes generated by the MyoD and E12 basic helix-loop-helix transcription factors in otherwise non-differentiation growth conditions." Biomaterials 35.7 (**2014**): 2188-2198.

138. “Comparison of Signal Smoothing Techniques for Use in Embedded System for Monitoring and Determining the Quality of Biofuels”, Dalton Cézane Gomes Valadares , Rute Cardoso Drebes, Elmar Uwe Kurt Melcher, Sérgio de Brito Espínola, Joseana Macêdo Fechine Régis de Araújo, Applied Mechanics and Materials, Vols. 448-453, pages 1679-1688, Trans Tech Publications, Switzerland, **2014**. DOI: 10.4028/www.scientific.net/AMM.448-453.1679

139. “Characterization of Integrated Optical Strain Sensors Based on Silicon Waveguides," Westerveld, W.J.; Leinders, S.M.; Muilwijk, P.M.; Pozo, J.; van den Dool, T.C.; Verweij, M.D.; Yousefi, M.; Urbach, H.P., , Selected Topics in Quantum Electronics, IEEE Journal of , vol.20, no.4, pp.1,10, July-Aug. **2014**. doi: 10.1109/JSTQE.2013.2289992

140. “Gaussian-function-based deconvolution method to determine the penetration ability of petrolatum oil into in vivo human skin using confocal Raman microscopy”, Chun-Sik Choe, Jürgen Lademann, and Maxim E Darvin, Laser Phys. 24 10560, **2014**. (http://iopscience.iop.org/1555-6611/24/10/105601)

141. “Borosilicate Glass Containing Bismuth and Zinc Oxides as a Hot Cell Material for Gamma-Ray Shielding”. H. A. Saudi, H. A. Sallam, K. Abdullah. Physics and Materials Chemistry. **2014**; 2(1):20-24. doi: 10.12691/pmc-2-1-4.

142. “Theta-Burst Stimulation of Hippocampal Slices Induces Network-Level Calcium Oscillations and Activates Analogous Gene Transcription to Spatial Learning”, Graham K. Sheridan , Emad Moeendarbary, Mark Pickering, John J. O'Connor, and Keith J. Murphy, PLOS One, June 20, **2014**. DOI: 10.1371/journal.pone.0100546

143. Mahmoud, Imbaby I., and Mohamed S. El_Tokhy. "Development of coincidence summing and resolution enhancement algorithms for digital gamma ray spectroscopy." Journal of Analytical Atomic Spectrometry 29.8 (**2014**): 1459-1466.

144. M. Rahmat, W. Maulina, Isnaeni, Miftah, N. Sukmawati, E. Rustami, M. Azis, K.B. Seminar, A.S. Yuwono, Y.H. Cho, H. Alatas, Development of a novel ozone gas sensor based on sol–gel fabricated photonic crystal, Sensors and Actuators A: Physical, Volume 220, 1 December **2014**, Pages 53–61

145. “Bacteria-instructed synthesis of polymers for self-selective microbial binding and labelling”, E. Peter Magennis,Francisco Fernandez-Trillo,Cheng Sui, Sebastian G. Spain, David J. Bradshaw, David Churchley, Giuseppe Mantovani, Klaus Winzer & Cameron Alexander, Nature Materials 13, 748–755 (**2014**) doi:10.1038/nmat3949 (http://www.nature.com/nmat/journal/v13/n7/extref/nmat3949-s1.pdf)

146. A COMPUTERIZED DATABASE FOR BULLET COMPARISON BY CONSECUTIVE MATCHING, Ashley Chu, David Read and David Howitt, Federally funded grant report, U.S. Department of Justice, Document No. 247771, July **2014**. (http://www.crime-scene-investigator.net/computerized-database-for-bullet-comparisonby-consecutive-matching.pdf)

147. Cade David E., Benoit-Bird Kelly J., (**2014**), An automatic and quantitative approach to the detection and tracking of acoustic scattering layers, Limnology and Oceanography: Methods, 12, doi: 10.4319/lom. **2014**.12.742.

148. Blake, Phillip, et al. "Diffraction in nanoparticle lattices increases sensitivity of localized surface plasmon resonance to refractive index changes." Journal of Nanophotonics 8.1 (**2014**): 083084-083084.

149. Sprinkhuizen, Sara M., Jerome L. Ackerman, and Yi Qiao Song. "Influence of ‐ bone marrow composition on measurements of trabecular microstructure using decay due to diffusion in the internal field MRI: Simulations and clinical studies." Magnetic Resonance in Medicine 72.6 (**2014**): 1499-1508.

150. Canlas, Reich Rechner D., Carlo Noel E. Ochotorena, and Elmer P. Dadios,"Fuzzy-genetic photoplethysmograph peak detection." Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management\(HNICEM), 2014 International Conference on. IEEE, **2014**.

151. Duenas, J. A., et al. "Dependency on the silicon detector working bias for proton–deuteron particle identification at low energies." Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 714 (**2013**): 48-52.

152. Sterling, Ryan, and Nathaniel Todd. "USING NEURAL SIGNALS TO PROVIDE INPUT FOR COMPUTING APPLICATIONS IN AUTONOMOUS PROSTHETICS." [PDF] from 136.142.82.187

153. Wang, Xiao, Yi-Qing Ni, and Ke-Chang Lin. "Comparison of statistical counting methods in SHM-based reliability assessment of bridges." Journal of Civil Structural Health Monitoring: 1-12.

154. González-Sáiz, J. M., et al. "Modulation of the phenolic composition and colour of red wines subjected to accelerated ageing by controlling process variables", Food chemistry 165 (**2014**): 271-281.

155. Kurniawan, Itmy Hidayat, and Sahat Simbolon. "Deteksi dan Pengukuran Spektra dalam Analisis Spektrografi Emisi dengan Pengolahan Citra." Jurnal Nasional Teknik Elektro dan Teknologi Informasi (JNTETI) 3.1 (**2014**).

156. Souri, Zoha. EEG-BASED ASSESSMENT OF DRIVER'S PERCEPTION OF TRAFFIC ENVIRONMENT. Diss. Lamar University, **2014**.

157. Lin, Junfang, et al. "Novel method for quantifying the cell size of marine phytoplankton based on optical measurements." Optics express 22.9 (**2014**): 10467-10476.

158. Hammonds Jr, James S., Kimani A. Stancil, and Charlezetta E. Stokes. "Quality factor temperature dependence of a surface phonon polariton resonance cavity." Applied Physics Letters 105.11 (**2014**): 114107.

159. Mall, U., et al. "Characterization of lunar soils through spectral features extraction in the NIR." Advances in Space Research 54.10 (**2014**): 2029-2040.

160. Bucur, R. V. "Structure of the Voltammograms of the Platinum-Black Electrodes: Derivative Voltammetry and Data Fitting Analysis." Electrochimica Acta 129 (**2014**): 76-84.

161. Teixeira, Carlos Esteves. "Sobre a teoria da difração de raios-X em estruturas tridimensionais." (**2014**). [PDF] from ufmg.br

162. Moon, Jim, et al. "Cable system for generating signals for detecting motion and measuring vital signs." U.S. Patent No. 8,738,118. 27 May **2014**.

163. Thompson, D. Brian, et al. "A Comparison of R-line Photoluminescence of Emeralds from Different Origins." The Journal of Gemmology 34.4 (**2014**): 334.

164. Oliveira, Raphael Rocha de. "Modelos de calibração multivariada por NIRS para a predição de características de qualidade da carne bovina." (**2014**). PDF] from ufg.br

165. Kirley, M. P. (**2014**). Electrical conductivity of metal surfaces at terahertz frequencies (Doctoral

dissertation, The University of Wisconsin-Madison).

166. Anderson, Danielle L., et al. "Spatial and temporal distribution of γH2AX fluorescence in human cell cultures following synchrotron-generated X-ray microbeams: lack of correlation between persistent γH2AX foci and apoptosis. "Synchrotron Radiation 21.4 (**2014**).

167. Maxfield, Dane Arthur. KINESIN-1 REGULATES SYNAPTIC STRENGTH BY MEDIATING DELIVERY, REMOVAL AND REDISTRIBUTION OF AMPARS. Diss. The University of Utah, **2014**.

168. Zou, Xiaoyu, Magneto-optical properties of ferromagnetic nanostructures on modified nanosphere templates. Thesis, CALIFORNIA STATE UNIVERSITY, LONG BEACH, **2014**, 87 pages; 1591619

169. A Carrasco, TA Brown, SG Lomber, Spectral and Temporal Acoustic Features Modulate Response Irregularities within Primary Auditory Cortex Columns, PloS one, **2014**, DOI: 10.1371/journal.pone.0114550

170. Sirotin, Yevgeniy B., Martín Elias Costa, and Diego A. Laplagne. "Rodent ultrasonic vocalizations are bound to active sniffing behavior." *Frontiers in behavioral neuroscience* 8 (**2014**).

171. Luo, Changtong, et al. "Wave system fitting: A new method for force measurements in shock tunnels with long test duration." Mechanical Systems and Signal Processing (**2015**).

172. Bleecker, J. V. (**2015**). Relating phase separation and thickness mismatch in model lipid membranes (Doctoral dissertation).

173. Möbius, Klaus, et al. "Möbius–Hückel topology switching in an expanded porphyrin cation radical as studied by EPR and ENDOR spectroscopy." Physical Chemistry Chemical Physics 17.9 (**2015**): 6644-6652.

174. Tariq, Humera, and SM Aqil Burney. "Low Level Segmentation of Brain MR Slices and Quantification Challenges.", NCMCS'15 (**2015**). Link.

175. Nystad, Helle Emilia. Comparison of Principal Component Analysis and Spectral Angle Mapping for Identification of Materials in Terahertz Transmission Measurements. Diss. Master's thesis, Norwegian University of Technology and Science, **2015**.

176. Hahn, Christian, et al. "Adjusting rheological properties of concentrated microgel suspensions by particle size distribution." *Food Hydrocolloids* 49 (**2015**): 183-191.

177. Chiuchiú, D. "Time-dependent study of bit reset." EPL (Europhysics Letters)109.3 (**2015**): 30002.

178. Taghizadeh, Mohammad Taghi, Nazanin Yeganeh, and Mostafa Rezaei. "The investigation of thermal decomposition pathway and products of poly (vinyl alcohol) by TG FTIR." - *Journal of Applied Polymer Science* 132.25 (**2015**).

179. P Sevusu , Real-time air quality measurements using mobile platforms, **2015**, Thesis, [PDF] from rutgers.edu

180. D. S. Khvostichenko, J. D. D. Ng, S. L. Perry, M. Menon, P. J. A. Kenis, Effects of detergent β-octyglucoside and phosphate salt solutions on phase behavior of monoolein mesophases , [PDF] from researchgate.net

181. Mahmoud, Imbaby I., and Mohamed S. El_Tokhy. "Advanced signal separation and recovery algorithms for digital x-ray spectroscopy." *Nuclear Instruments and Methods in Physics Research Section A: Accelerators,*

Spectrometers, Detectors and Associated Equipment 773 (**2015**): 104-113.

182. Morrow, Justin D. Surface Microstructure and Properties of Pulsed Laser Micro Melted S7 Tool Steel. The University of Wisconsin-Madison, **2015**.

183. Ubnoske, Stephen M., et al. "Role of nanocrystalline domain size on the electrochemical double-layer capacitance of high edge density carbon nanostructures." *MRS Communications* (**2015**): 1-6.

184. MUHAMMAD MUFTI AZIS, "Experimental and kinetic studies of H2 effect on lean exhaust after treatment processes: HC-SCR and DOC" CHALMERS UNIVERSITY OF TECHNOLOGY Göteborg, Sweden **2015**

185. Umesh Rudrapatna, S., et al. "Measurement of distinctive features of cortical spreading depolarizations with different MRI contrasts." NMR in Biomedicine 28.5 (**2015**): 591-600.

186. Kühbach, Markus, Brüggemann, Thiemo, Molodov, Konstantin, Gottstein, Günter. "On a Fast and Accurate In-Situ Measuring Strategy for Recrystallization Kinetics and Its Application to an Al-Fe-Si Alloy", *Metallurgical and Materials Transactions* A, March **2015**, Volume 46, Issue 3, pp 1337-1348

187. D. Y. Lipatov, Y. R. Shaltaeva, V. V. Belyakov, A. V. Golovin, V. S. Pershenkov, V. V. Shurenkov, D. Y. Yakovlev, "Modeling of IMS Spectra in Medical Diagnostic Purposes", 3rd International Conference on Nanotechnologies and Biomedical Engineering, Volume 55 of the series IFMBE Proceedings, **2015**, pp 404-408

188. Y. Meerten, , Y. Swolfs , J. Baets , L. Gorbatikh , I. Verpoebucurst , "Penetration impact testing of self-reinforced composites", Composites Part A: Applied Science and Manufacturing, Volume 68, January **2015**, Pages 289–295

189. Ivanov, I , Optimal filtering of synchronized current phasor measurements in asteady state, 2015 IEEE International Conference on Industrial Technology (ICIT), Pages 1362 - 1367 , 17-19 March **2015**

190. L Farge, J Boisse, J Dillet, S André, Wide angle X ray scattering study of the lamellar/fibrillar transition for a semi crystalline polymer deformed in tension in relation with the evolution of volume strain*, Journal of Polymer Science* B, Volume 53, Issue 20, 15 October **2015**, Pages 1470–1480

191. Patrick Schloth , Precipitation in the high strength AA7449 aluminium alloy: implications on internal stresses on different length scales, Thesis, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, June **2015**.

192. Guzman, P. (**2015**). Studying the Physical Stability of BSA at the Bulk Solution and Oil/Water Interface (Doctoral dissertation, University of Otago).

193. FlavonQ: An Automated Data Processing Tool for Profiling Flavone and Flavonol Glycosides with Ultra-High-Performance Liquid Chromatography–Diode Array Detection–High Resolution Accurate Mass–Mass Spectrometry, Mengliang Zhang, Jianghao Sun, and Pei Chen*, *Anal. Chem*., **2015**, 87 (19), pp 9974–9981, DOI: 10.1021/acs.analchem.5b02624

194. Schulze, H. Georg, and Robin FB Turner. "Development and Integration of Block Operations for Data Invariant Automation of Digital Preprocessing and Analysis of Biological and Biomedical Raman Spectra." *Applied spectroscopy* 69.6 (**2015**): 643-664.

195. Hutchison, Richard Stephen. Novel high refractive index, thermally conductive additives for high brightness white LEDs. Diss. Rensselaer Polytechnic Institute, **2015**.

196. Magnotti, G., et al. "Raman spectra of methane, ethylene, ethane, dimethyl ether, formaldehyde and

propane for combustion applications." *Journal of Quantitative Spectroscopy and Radiative Transfer* 163 (**2015**): 80-101.

197. Chen, Rex Chin-Hao. "Spectral and Temporal Interrogation of Cerebral Hemodynamics Via High-Speed Laser Speckle Contrast Imaging." (**2015**).

198. Maistry, N. (**2015**). Investigating the concept of Fraunhofer lines as a potential method to detect corona in the wavelength region 338nm-405nm during the day (Doctoral dissertation).

199. Parker, Michael J. Coupling Nuclear Induced Phonon Propagation with Conversion Electron Moessbauer Spectroscopy. No. AFIT-ENP-MS-15-J-054. AIR FORCE INSTITUTE OF TECHNOLOGY WRIGHT-PATTERSON AFB OH GRADUATE SCHOOL OF ENGINEERING AND MANAGEMENT, **2015**.

200. Maistry, Nattele. Investigating the concept of Fraunhofer lines as a potential method to detect corona in the wavelength region 338nm-405nm during the day. Diss. **2015**.

201. Liu, Yanping, et al. "Applications of Savitzky-Golay Filter for Seismic Random Noise Reduction." *Acta Geophysica* (**2015**).

202. Kojimoto, N. C. (**2015**). Ultrasonic inspection methods for defect detection and process control in roll-to-roll flexible electronics manufacturing (Doctoral dissertation, Massachusetts Institute of Technology).

203. Sheehan, Terry L., and Richard A. Yost. "What's the most meaningful standard for mass spectrometry: instrument detection limit or signal-to-noise ratio" *Current Trends Mass Spectrometry* 13 (**2015**): 16-22.

204. Bleecker, J. V. (**2015**). Relating phase separation and thickness mismatch in model lipid membranes (Doctoral dissertation).

205. Ilewicz, Witold, et al. "Comparison of baseline estimation algorithms for chromatographic signals." Methods and Models in Automation and Robotics (MMAR), 2015 20th International Conference on. IEEE, **2015**.

206. Massimi, Federico. Sviluppo di metodi integrati basati sulle tecniche di nanoindentazione e del fascio ionico focalizzato (FIB) per la caratterizzazione, risolta nello spazio, delle proprietà meccaniche dei materiali", ArcAdiA." (**2015**). http://hdl.handle.net/2307/5329

207. Swoboda, Daniel Maximilian, et al. "A Comprehensive Study of Simple Digital Filters for Botball IR Detection Techniques." PDF link.

208. CE Funes, EF Cromwell System and method for determining a baseline measurement for a biological response curve, US Patent App. 13/308,021, 2

209. AD Beyene, R Bluffstone, Z Gebreegziabher , The Improved Biomass Stove Saves Wood, But How Often Do People Use It?, [TXT] from worldbank.com

210. Raunio, Saida. "IMMUNOASSAY TEST FOR A QVANTITATIVE DETERMINATION OF HELICOBACTER PYLORI ANTIBODY IN BLOOD DONORS" (**2015**). PDFAlt, Daniel M. Design and Commissioning of a 16.1 MHz Multiharmonic Buncher for the ReAccelerator at NSCL. ProQuest, **2016**.

211. Coy, A., Rankine, D., Taylor, M., Nielsen, D. C., & Cohen, J. (**2016**). Increasing the accuracy and automation of fractional vegetation cover estimation from digital photographs. Remote Sensing, 8(7), 474.

212. Nguyen, Tuan Ngoc. "An algorithm for extracting the PPG Baseline Drift in realtime."

(**2016**). PDF link.

213. Maitre, Léa. "Metabonomic and epidemiological analyses of maternal parameters and exposures during pregnancy and their influence on fetal growth amongst the INMA birth cohort." (**2016**). PDF link.

214. Lipatov, D. Y., et al. "Modeling of IMS Spectra in Medical Diagnostic Purposes." 3rd International Conference on Nanotechnologies and Biomedical Engineering. Springer Singapore, **2016**.

215. Tong, Xia, et al. "Recursive Wavelet Peak Detection of Analytical Signals." Chromatographia 79.19-20 (**2016**): 1247-1255.

216. Wang, Xing. Effects of Interfaces on Properties of Cladding Materials for Advanced Nuclear Reactors. The University of Wisconsin-Madison, **2016**. PDF link.

217. Dang, Hue, Marian Dekker, Jason Farquhar, and Tom Heskes. "Processing and analyzing functional near-infrared spectroscopy data." (**2016**).

218. Damavandi, H. G. (**2016**). Data analytics, interpretation and machine learning for environmental forensics using peak mapping methods (Doctoral dissertation, The University of Iowa).

219. Gill, Ruby K., et al. "The effects of laser repetition rate on femtosecond laser ablation of dry bone: a thermal and LIBS study." Journal of biophotonics 9.1-2 (**2016**): 171-180.

220. Performance evaluation and optimization of X-ray stress measurement for nickel aluminium bronze based on the Bayesian method. Journal of Applied Crystallography, **2016** – scripts.iucr.org

221. Top-down modulation of stimulus drive via beta-gamma cross-frequency interaction. CG Richter, WH Thompson, CA Bosman, P Fries - bioRxiv, **2016** – biorxiv.org

222. Azpúrua, Marco A., Marc Pous, and Ferran Silva. "Decomposition of Electromagnetic Interferences in the Time-Domain." (**2016**).

223. Barros, Rodrigo Emanoel de Britto Andrade. SISTEMA DE INTERROGAÇÃO DE REDES DE BRAGG: PRIMEIROS PASSOS NA CRIAÇÃO DE UM PROTÓTIPO. Diss. Universidade Federal do Rio de Janeiro, **2016**.

224. Li, Yuanlu, et al. "A novel signal enhancement method for overlapped peaks with noise immunity." *Spectroscopy Letters* 49.4 (**2016**): 285-293.

225. Hatterschide, Joshua. "Retroviral-RNA Structure and Function: Investigating the role of aminoacyl-tRNA synthetases and retroviral-RNA structural elements in the initiation of reverse transcription." (**2016**).

226. Guizani, Chamseddine, et al. "Biomass char gasification by H 2 O, CO 2 and their mixture: Evolution of chemical, textural and structural properties of the chars." Energy 112 (**2016**): 133-145.

227. Wagner, C. F. (**2016**). Transition from transparency to hole-boring in relativistic laser-solid interactions at the Texas Petawatt (Doctoral dissertation).

228. Bocaege, E., and S. Hillson. "Disturbances and noise: Defining furrow form enamel hypoplasia." *American journal of physical anthropology* 161.4 (**2016**): 744-751

229. Merla, Yu, et al. "Extending battery life: A low-cost practical diagnostic technique for lithium-ion batteries." *Journal of Power Sources* 331 (**2016**): 224-231.

230. Besemer, Matthieu, et al. "Identification of Multiple Water–Iodide Species in Concentrated NaI Solutions Based on the Raman Bending Vibration of Water." *The Journal of Physical Chemistry* A 120.5 (**2016**): 709-714.

231. Cairone, Fabiana, Salvina Gagliano, and Maide Bucolo. "Experimental study on the slug flow in a serpentine microchannel." *Experimental Thermal and Fluid Science* 76 (**2016**): 34-44.

232. Davison, Adrian K., et al. "Objective Micro-Facial Movement Detection Using FACS-Based Regions and Baseline Evaluation." arXiv preprint arXiv:1612.05038 (**2016**).

233. Ninh, Giang Nguyen, et al. "Radioisotope identification method for poorly resolved gamma-ray spectrum of nuclear security concern." *AIP Conference Proceedings*. Vol. 1704. No. 1. AIP Publishing, **2016**.

234. Brachi, Paola, et al. "Pseudo-component thermal decomposition kinetics of tomato peels via isoconversional methods." *Fuel Processing Technology* 154 (**2016**): 243-250.

235. Lee, Hansol, et al. "Flow suppressed hyperpolarized 13C chemical shift imaging using velocity optimized bipolar gradient in mouse liver tumors at 9.4 T.", *Magnetic resonance in medicine* (**2016**).

236. Wu, B., et al. "Novel application of differential thermal voltammetry as an in-depth state-of-health diagnosis method for lithium-ion batteries." PDF file.

237. Creese, Andrew J., and Helen J. Cooper. "Separation of cis and trans Isomers of Polyproline by FAIMS Mass Spectrometry." *Journal of The American Society for Mass Spectrometry* 27.12 (**2016**): 2071-2074.

238. Kvyetnyy, Roman, et al. "Improving the quality perception of digital images using modified method of the eye aberration correction." Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2016. Vol. 10031. *International Society for Optics and Photonics*, **2016**.

239. Myers, G. A., Turner, L. G., Morgan, Q., & Pearce, J., "Raman Spectroscopy-detecting SOx and NOx in the Precipice Sandstone". (**2016**)

240. Pancholi, Manthan, et al. "Relative Translation and Rotation Calibration Between Optical Target and Inertial Measurement Unit." International Conference on Sensor Systems and Software. Springer, Cham, **2016**.

241. Ferriss, Elizabeth, Terry Plank, and David Walker. "Site-specific hydrogen diffusion rates during clinopyroxene dehydration." *Contributions to Mineralogy and Petrology* 171.6 (**2016**): 55.

242. Roy, Sujan Kumar, Wei-Ping Zhu, and Benoit Champagne. "Single channel speech enhancement using subband iterative Kalman filter." Circuits and Systems (ISCAS), **2016** IEEE International Symposium on. IEEE, **2016**.

243. Langaas, Gjertrud Louise. "Measurements of radioactivity in plant and soil samples taken near a nuclear power plant." (**2016**). PDF link.

244. Benigni, Paolo, and Francisco Fernandez-Lima. "Oversampling selective accumulation trapped ion mobility spectrometry coupled to FT-ICR MS: fundamentals and applications." *Analytical chemistry* 88.14 (**2016**): 7404-7412.

245. Geiger, Matthew, and Michael T. Bowser. "Effect of fluorescent labels on and amino acid sample dimensionality in two dimensional nLC× μFFE separations." *Analytical chemistry* 88.4 (**2016**): 2177-2187.

246. Aldokhail, A. M. (**2016**). Automated Signal to Noise Ratio Analysis for Resonance Imaging Using a Noise Distribution Model (Doctoral dissertation, University of Toledo).

247. Fasching, Joshua, et al. "Automated coding of activity videos from a study." Robotics and Automation (ICRA), 2016 IEEE International Conference. IEEE, **2016**.

248. Bleecker, J. V., Cox, P. A., Foster, R. N., Litz, J. P., Blosser, M. C., Castner, D. G., & Keller, S. L. (20**1**6). Thickness Mismatch of Coexisting Liquid Phases in Non-Canonical Lipid Bilayers. *The journal of physical chemistry*. B, 120(10), 2761.

249. Joshi, Bijal, and Nitu Anil Kumar. "Computationally efficient data rate mismatch compensation for telephony clocks." U.S. Patent No. 9,514,766. 6 Dec. **2016**.

250. Vallet, Aurélien, et al. "A multi-dimensional statistical rainfall threshold for deep landslides based on groundwater recharge and support vector machines." *Natural Hazards* 84.2 (**2016**): 821-849.

251. Wang, Lili, Paul DeRose, and Adolfas K. Gaigalas. "Assignment of the number of equivalent reference fluorophores to dyed microspheres." *J. Res. Nat. Ins. Stand. Technol*. 121 (**2016**): 269-286.

252. Skaret, H. B. (**2016**). The Arctic Sea Ice-Melting During Summer or not Freezing in Winter? (Master's thesis, The University of Bergen). PDF link.

253. Tuan T. Tran, et. al., Synthesis of Ge1−xSnx alloys by ion implantation and pulsed laser melting: Towards a group IV direct bandgap material, *Journal of Applied Physics* 119(18):183102, **2016**, DOI: 10.1063/1.4948960

363. Choi, Jae Sung, et al. "A New Automated Cell Counting Program by Using Hough Transform-Based Double Edge." *Advances in Computer Science and Ubiquitous Computing*. Springer, Singapore, **2016**. 712-716.

253. Van der Rest, Guillaume, Human Rezaei, and Frédéric Halgand. "Monitoring Conformational Landscape of Ovine Prion Protein Monomer Using Ion Mobility Coupled to Mass Spectrometry." *Journal of The American Society for Mass Spectrometry* 28.2 (**2017**): 303-314.

254. Mirsafavi, Rustin Y., et al. "Detection of papaverine for the possible identification of illicit opium cultivation." *Analytical Chemistry* 89.3 (**2017**): 1684-1688.

255. Myers, Grant A., Kelsey Kehoe, and Paul Hackley. "Analysis of Artificially Matured Shales with Confocal Laser Scanning Raman Microscopy: Applications to Organic Matter Characterization." Unconventional Resources Technology Conference (URTEC), **2017**.

256. Torres, Andrei BB, José Adriano Filho, Atslands R. da Rocha, Rubens Sonsol Gondim, and José Neuman de Souza. "Outlier detection methods and sensor data fusion for precision agriculture", **2017**, PDF link.

257. Desmet, F., Lesaffre, M., Six, J., Ehrlé, N., & Samson, S. (**2017**). Multimodal analysis of synchronization data from patients with dementia. In ESCOM 2017.

258. Seeber, Renato, and Alessandro Ulrici. "Analog and digital worlds: Part 2. Fourier analysis in signals and data treatment." *ChemTexts* 3.2 (**2017**): 8.

259. Mustafa, M. A., et al. "Nonintrusive Freestream Velocity Measurement in a Large-Scale Hypersonic Wind Tunnel." *AIAA Journal* (**2017**).

260. Suárez-Cortés, Pablo, et al. "Ned-19 inhibition of parasite growth and multiplication suggests a role for NAADP mediated signaling in the asexual development of Plasmodium falciparum." *Malaria Journal* 16.1 (**2017**): 366.

261. Catalbas, M. C., & Dobrisek, S., 3D Moving Sound Source Localization via Conventional Microphones. *Elektronika ir Elektrotechnika*, 23(4), 63-69. (**2017**).

262. Du, Zhenhui, et al. "High-sensitive carbon disulfide sensor using wavelength modulation spectroscopy in the mid-infrared fingerprint region." *Sensors and Actuators B: Chemical* 247 (**2017**): 384-391.

263. Hamilton, N. E., Mahjoub, R., Laws, K. J., & Ferry, M. (**2017**). A blended NPT/NVT scheme for simulating metallic glasses. *Computational Materials Science*, 130, 130-137.

264. Sun, Y. C., Huang, C., Xia, G., Jin, S. Q., & Lu, H. B. (**2017**). Accurate wavelength calibration method for compact CCD spectrometer. JOSA A, 34(4), 498-\505.

265. Mikhailov, I. F., et al. "Rapid diagnostics of urinary iodine using a portable EDXRF spectrometer." *Journal*

of X-Ray Science and Technology Preprint (**2017**): 1-7. PDF link.

266. Bianchi, Davide, et al. "A wavelet filtering method for cumulative gamma spectroscopy used in wear measurements." Applied Radiation and Isotopes 120 (**2017**): 51-59.

267. Xiong, Zheng, et al. "Automated Phase Segmentation for Large-Scale X-ray Diffraction Data Using a Graph-Based Phase Segmentation (GPhase) Algorithm." ACS Combinatorial Science 19.3 (**2017**): 137-144

268. Jiménez-Carvajal, C., et al. "Weighing lysimetric system for the determination of the water balance during irrigation in potted plants." Agricultural Water Management 183 (**2017**): 78-85.

269. Acciarri, R., et al. "Noise Characterization and Filtering in the MicroBooNE Liquid Argon TPC." arXiv preprint arXiv:1705.07341 (**2017**). PDF link.

270. Mathault, Jessy, Hamza Landari, Frederic Tessier, Paul Fortier, and Amine Miled. "Biological Modeling Challenges in a Multiphysics Approach." Circuits and Systems (MWSCAS), **2017** IEEE 60th International Midwest Symposium

271. Weiss, Charles J. "Scientific Computing for Chemists: An Undergraduate Course in Simulations, Data Processing, and Visualization." Journal of Chemical Education 94.5 (**2017**): 592-597.

272. Kianifar, Rezvan, et al. "Automated Assessment of Dynamic Knee Valgus and Risk of Knee Injury During the Single Leg Squat." *IEEE Journal of Translational Engineering in Health and Medicine* 5 (**2017**): 1-13.

273. Willem deGroot, A., et al. "Molecular Structural Characterization of Polyethylene." Handbook of Industrial Polyethylene and Technology: Definitive Guide to Manufacturing, Properties, Processing, Applications and Markets (**2017**): 139.

274. Mertens, Andreas, and Josef Granwehr. "Two-dimensional impedance data analysis by the distribution of relaxation times." *Journal of Energy Storage* 13 (**2017**): 401-408.

275. Wu, Yingwen, and Long Chen. "Comparison of spectra processing methods for SERS based quantitative analysis." Information, Cybernetics and Computational Social Systems (ICCSS), 2017 4th International Conference on. IEEE, **2017**.

276. Dehnavi, Sahar, Yasser Maghsoudi, and Mohammadjavad Valadanzoej. "Using spectrum differentiation and combination for target detection of minerals." *International Journal of Applied Earth Observation and Geoinformation* 55 (**2017**): 9-20.

278. Jia, Zhenhua, et al. "HB-phone: a bed-mounted geophone-based heartbeat monitoring system: demo abstract." Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks. ACM, **2017**.

279. Gozé, Perrine, et al. "Effects of ozone treatment on the molecular properties of wheat grain proteins." Journal of Cereal Science 75 (**2017**): 243-251.

280. Giron-Sierra, Jose Maria. "Periodic Signals." Digital Signal Processing with Matlab Examples, Volume 1. Springer Singapore, **2017**. 3-28.

281. Shojaosadati, Seyed Abbas, Sajjad Naeimipour, and Ahmad Fazeli. "FTIR Investigation of secondary structure of Reteplase inclusion bodies produced in Escherichia coli in terms of urea concentration (Spring 2017)." *Iranian Journal of Pharmaceutical Research* (**2017**).

282. Peng, Jiyu, et al. "Rapid Identification of Varieties of Walnut Powder Based on Laser-Induced Breakdown Spectroscopy." (**2017**): 19-28. Abstract.

283. Sun, Lili, et al. "Comprehensive evaluation of chemical stability of Xuebijing injection based on multiwavelength chromatographic fingerprints and multivariate chemometric techniques." *Journal of Liquid*

Chromatography & Related Technologies 40.14 (**2017**): 715-724.

284. Thompson, D. Brian, et al. "Photoluminescence Spectra of Emeralds from Colombia, Afghanistan, and Zambia." *Gems & Gemology* 53.3 (**2017**): 296-311.

285. Choorat, P., et al. "Applied integral intensity projection to find the numbers of the parking spots." Knowledge and Smart Technology (KST), **2017** 9th International Conference on. IEEE, 2017.

286. Phillips, James William, and Yi Jin. "Devices and methods of low frequency magnetic stimulation therapy." U.S. Patent No. 9,649,502. 16 May **2017**.

287. Augustyns, Valérie, et al. "Evidence of tetragonal distortion as the origin of the ferromagnetic ground state in γ − Fe nanoparticles." *Physical Review* B 96.17 (**2017**):174410.

288. Sprague-Klein, Emily A., et al. "Observation of Single Molecule Plasmon-Driven Electron Transfer in Isotopically Edited 4, 4-Bipyridine Gold Nanosphere Oligomers." *Journal of the American Chemical Society* 139.42 (**2017**): 15212-15221.

289. Mohan, Varun, and Prashant K. Jain. "Spectral Heterogeneity of Hybrid Lead Halide Perovskites Demystified by Spatially Resolved Emission." *The Journal of Physical Chemistry* C121.35 (**2017**): 19392-19400.

290. Cuss, Chad W., Iain Grant-Weaver, and William Shotyk. "AF4-ICPMS with the 300 Da Membrane to Resolve Metal-Bearing “Colloids”< 1 kDa: Optimization, Fractogram Deconvolution, and Advanced Quality Control." *Analytical Chemistry* 89.15 (**2017**): 8027-8035.

291. Shi, Xiaoyu, et al. "Super-Resolution Microscopy Reveals That Disruption of Ciliary Transition Zone Architecture Is a Cause of Joubert Syndrome." bioRxiv (**2017**): 142042.

292. Robinson, M. T., et al. "Photocatalytic photosystem I/PEDOT composite films prepared by vapor-phase polymerization." *Nanoscale* 9.18 (**2017**): 6158-6166.

293. Ros Martí, Marc. Deep convolutional neural network architecture for effective Image analysis. MS thesis. Universitat Politècnica de Catalunya, **2017**.

294. Jackson, Philip J., et al. "Identification of protein W, the elusive sixth subunit of the Rhodopseudomonas palustris reaction center-light harvesting 1 core complex." *Biochimica et Biophysica Acta* (BBA)-Bioenergetics (**2017**).

295. Johnson, Alexander C., and Michael T. Bowser. "High-Speed, Comprehensive, Two-Dimensional Separations of Peptides and Small Molecule Biological Amines Using Capillary Electrophoresis Coupled with Micro Free Flow Electrophoresis." *Analytical chemistry* 89.3 (**2017**): 1665-1673.

296. Toose, Peter, et al. "Radio-frequency interference mitigating hyperspectral L band radiometer." *Geoscientific Instrumentation, Methods and Data Systems* 6.1 (**2017**): 39.

297. Pajankar, Ashwin. "Filters and Their Application." Raspberry Pi Image Processing Programming. Apress, **2017**. 99-110.

298. Taraszewski, Michał, and Janusz Ewertowski. "Complex experimental analysis of rifle-shooter interaction." Defence Technology (**2017**).

299. Manlises, Cyrel Ontimare, et al. "Characterization of an ISFET with Built-in Calibration Registers through Segmented Eight-Bit Binary Search in Three-Point Algorithm Using FPGA." *Journal of Low Power Electronics and Applications* 7.3 (**2017**):19.

300. Kim, Geonha, et al. "Soil sampling strategies for site assessments in petroleum contaminated areas." Environmental geochemistry and health 39.2 (**2017**): 293-305.

301. Lanevski, Dmitri, Koit Mauring, and Eric Tkaczyk. "Interference filter tilting to detect a polycyclic aromatic hydrocarbon at the second harmonic of wavelength modulation frequency." Applied Optics 56.11 (**2017**): 3155-3161.

302. Hong, Tae-Kee, Iason Rusodimos, and Myung-Hoon Kim. "Higher order derivative voltammetry for reversible and irreversible electrode processes under spherical diffusion." Journal of Electroanalytical Chemistry 785 (**2017**): 255-264.

303. Root, Katharina, et al. "Insight into Signal Response of Protein Ions in Native ESI-MS from the Analysis of Model Mixtures of Covalently Linked Protein Oligomers." Journal of The American Society for Mass Spectrometry (**2017**): 1-13.

304. Du, Zhenhui, et al. "High-sensitive carbon disulfide sensor using wavelength Modulation spectroscopy in the mid-infrared fingerprint region." Sensors and Actuators B: Chemical 247 (**2017**): 384-391.

305. Elzanfaly, Eman S., et al. "Zero and second derivative synchronous fluorescence spectroscopy for the quantification of two non-classical β lactams in pharmaceutical vials: Application to stability studies." Luminescence (**2017**).

306. Ferraz de Menezes, Rebeca, et al. "Fs laser ablation of teeth is temperature limited and provides information about the ablated components." Journal of Biophotonics (**2017**).

307. Huang, Yi-Fan, et al. "Label-free, ultrahigh-speed, 3D observation of bidirectional and correlated intracellular cargo transport by coherent brightfield microscopy." *Nanoscale* 9.19 (**2017**): 6567-6574.

308. Mahmud, Akib. "Hardware in the Loop (HIL) Rig Design and Electrical Architecture." (**2017**).

309. Beyerl, Thomas, et al. *Reducing Complexity in Routing of Non-Standard Intersections, to Aid in Autonomous Vehicle Navigation*. No. 2017-01-0103. SAE Technical Paper, **2017**.

310. Lee, Hansol, et al. "Flow-suppressed hyperpolarized 13C chemical shift imaging using velocity-optimized bipolar gradient in mouse liver tumors at 9.4 T." *Magnetic resonance in medicine* 78.5 (**2017**): 1674-1682.

311. Haines, Grant E., and S. Laurie Sanderson. "Integration of swimming kinematics and ram suspension feeding in a model American paddlefish, Polyodon spathula." *Journal of Experimental Biology* (**2017**): jeb-166835.

312. Zhang, Huajun, and Y. I. N. G. Ning. "Method for Analyzing Mixture Components." U.S. Patent Application 15/120,974, filed March 2, **2017**.

313. Soto Morras, Marta. "Implementation and Analysis of Real Time Optical Flow Solutions for GPU architectures." (**2017**).

314. Pajankar, Ashwin. *Raspberry Pi Image Processing Programming*. Apress, **2017**.

315. Vintila, Florentin, Thomas C. Kübler, and Enkelejda Kasneci. "Pupil response as an indicator of hazard perception during simulator driving." *Journal of Eye Movement Research* 10.4 (**2017**): 3.

316. Humera Tariq, Abdul Muqeet, S.M.Aqil Burney, Humera Azam, "Otsu's Segmentation….", *J. Theoretical and Applied Information Technology*, Vol.95. No 22, **2017**

317 Liu, Yu, et al. "Supersonic transient magnetic resonance elastography for quantitative assessment of tissue elasticity." *Physics in Medicine & Biology* 62.10 (**2017**): 4083.

318. Manar M. Ouda, et. Al., Development of Pileup Recovery Algorithms by Peak Detection Method of Digital Gamma Ray Spectroscopy, 34th National Radio Science Conference (NRSC), **2017**. Link to full paper.

319. Xu, Jun-Li, Aoife A. Gowen, and Da-Wen Sun. "Time series hyperspectral chemical imaging (HCI) for investigation of spectral variations associated with water and plasticizers in casein-based biopolymers." Journal of Food Engineering 218 (**2018**): 88-105.

320. Smith, Brad C., Bachana Lomsadze, and Steven T. Cundiff. "Optimum repetition rates for dual-comb spectroscopy." *Optics express* 26.9 (**2018**): 12049-12056.

321. Butler, C. W., et al. "Neurons Specifically Activated by Fear Learning in Lateral Amygdala Display Increased Synaptic Strength." *eNeuro* 5.3 (**2018**).

322. Pukhlyakova, Ekaterina, et al. "β-Catenin–dependent mechanotransduction dates back to the common ancestor of Cnidaria and Bilateria." *Proceedings of the National Academy of Sciences* 115.24 (**2018**): 6231-6236.

323. Cheng, Jie. *Peak Detection to Count Gold Nanoparticles Translocations in Nanopipette*. Diss. UC Santa Cruz, **2018**.

324. Bonde, Amelie, et al. "VVRRM: Vehicular Vibration-Based Heart RR-Interval Monitoring System. "*Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*. ACM, **2018**.

335. Paige, Cristen, et al. "Characterizing the Normative Voice Tremor Frequency in Essential Vocal Tremor." *JAMA Otolaryngology–Head & Neck Surgery* (**2018**).

326. Myers, Grant A., Kelsey Kehoe, and Paul Hackley. "Development of Raman Spectroscopy as a Thermal Maturity Proxy in Unconventional Resource Assessment." *Unconventional Resources Technology Conference, Houston, Texas, 23-25 July 2018*. Society of Exploration Geophysicists, American Association of Petroleum Geologists, Society of Petroleum Engineers, **2018**.

327. Taraszewski, Michal, and Janusz Ewertowski. "Small-Caliber Grenade Projectile Applicable to Individual Grenade Launchers." *Defence Science Journal* 68.5 (**2018**).

328. Trinh, N. D., et al. "Double differential neutron spectra generated by the interaction of a 12 MeV/nucleon 36S beam on a thick natCu target." *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 896 (**2018**): 152-164.

329. Swainsbury, David JK, et al. "Probing the local lipid environment of the Rhodobacter sphaeroides cytochrome bc1 and Synechocystis sp. PCC 6803 cytochrome b6f complexes with styrene maleic acid." *Biochimica et Biophysica Acta (BBA)-Bioenergetics* 1859.3 (**2018**): 215-225.

330. Reynes, Julien, et al. "Experimental constraints on hydrogen diffusion in garnet." *Contributions to Mineralogy and Petrology* 173.9 (**2018**): 69.

331. Omer, Muhammad, and Elise C. Fear. "Automated 3D method for the construction of flexible and reconfigurable numerical breast models from MRI scans." *Medical & biological engineering & computing* 56.6 (**2018**): 1027-1040.

332. Klein, Tobias, et al. "Influence of Liquid Structure on Fickian Diffusion in Binary Mixtures of n-Hexane and Carbon Dioxide Probed by Dynamic Light Scattering, Raman Spectroscopy, and Molecular Dynamics Simulations." *The Journal of Physical Chemistry B* (**2018**).

333. Kielar, A., T. Deschamps, R. Jokel, and J. A. Meltzer. "Abnormal language-related oscillatory responses in primary progressive aphasia." *NeuroImage: Clinical* 18 (**2018**): 560-574.

334. Prodanov, Milana, et al. "Software Module for Processing EEG Signals in a Biofeedback Based System." *2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*. IEEE, **2018**.

335. Fratini, Marta, et al. "Surface Immobilization of Viruses and Nanoparticles Elucidates Early Events in Clathrin-Mediated Endocytosis." *ACS infectious diseases* (**2018**).

336. Siliņš, Kaspars. *Plasma Enhanced Chemical-and Physical-Vapor Depositions Using Hollow Cathodes*. Diss. Acta Universitatis Upsaliensis, **2018**.

337 Schito, Andrea, and Sveva Corrado. "An automatic approach for characterization of the thermal maturity of dispersed organic matter Raman spectra at low diagenetic stages." *Geological Society, London, Special Publications* 484 (**2018**): SP484-5.

338. Krystal T. Vasquez, et. al., Low-pressure gas chromatography with chemical ionization mass Spectrometry for quantification of multifunctional organic compounds in the atmosphere, *Atmos. Meas. Tech*. **2018**. PDF.

339. Pushkarsky, I., Tseng, P., Black, D., France, B., Warfe, L., Koziol-White, C. J., ... & Damoiseaux, R. (2018). Elastomeric sensor surfaces for high-throughput single-cell force cytometry (vol 2, pg 124, **2018**).

340. Ismail, Omar, et al. "The Way to Ultrafast, High-Throughput Enantioseparations of Bioactive Compounds in Liquid and Supercritical Fluid Chromatography." *Molecules* 23.10 (**2018**): 2709.

338. Hellinghausen, Garrett, M. Farooq Wahab, and Daniel W. Armstrong. "Improving visualization of trace components for quantification using a power law-based integration approach." *Journal of Chromatography A* 1574 (**2018**): 1-8.

339. Khundadze, Nana, et al. "On our way to sub-second separations of enantiomers in high-performance liquid chromatography." *Journal of Chromatography A* 1572 (**2018**): 37-43.

344. Roy, Daipayan, et al. "Frontiers in Ultrafast Chiral Chromatography." *LC• GC Europe* (**2018**): 308.

345. Maddalena, Riccardo, Christopher Hall, and Andrea Hamilton. "Effect of silica particle size on the formation of calcium silicate hydrate using thermal analysis." *Thermochimica Acta* (**2018**).

346. Darweesh, Samar Ahmed, et al. "Advancement and Validation of New Derivative Spectrophotometric Method for Individual and Simultaneous Estimation of Diclofenac sodium and Nicotinamide." *Oriental Journal of Chemistry* 34.3 (**2018**).

347. Li, Yuanlu, and Min Jiang. "Spatial-fractional order diffusion filtering." *Journal of Mathematical Chemistry* 56.1 (**2018**): 257-267.

348. Huang, Dian, et al. "High-Speed Live-Cell Interferometry: A New Method for Quantifying Tumor Drug Resistance and Heterogeneity." *Analytical chemistry* 90.5 (**2018**): 3299-3306.

349. Wu, Rihan, et al. "Demonstration of time-of-flight technique with all-optical modulation and MCT detection in SWIR/MWIR range." *Emerging Imaging and Sensing Technologies for Security and Defence III; and Unmanned Sensors, Systems, and Countermeasures*. Vol. 10799. International Society for Optics and Photonics, **2018**.

350. Pontremoli, Carlotta, et al. "Insight into the interaction of inhaled corticosteroids with human serum albumin: A spectroscopic-based study." *Journal of pharmaceutical analysis* 8.1 (**2018**): 37-44.

351. Zhao, Chenjiang. *Signal Processing: Peak Detection*. Diss. UC Santa Cruz, **2018**.

352. Coelho, Alan A. "Deconvolution of instrument and Kα2 contributions from X-ray powder diffraction patterns using nonlinear least-squares with penalties." *Journal of Applied Crystallography* 51.1 (**2018**): 112-123.

353. Al-gawwam, Sarmad, and Mohammed Benaissa. "Robust Eye Blink Detection Based on Eye Landmarks and Savitzky–Golay Filtering." *Information* 9.4 (**2018**): 93.

354. Yilmaz, Cagatay Murat, Cemal Kose, and Bahar Hatipoglu. "A Quasi-probabilistic distribution model for EEG Signal classification by using 2-D signal representation." *Computer methods and programs in biomedicine* 162 (**2018**): 187-196.

355. Gou, Yonggang, et al. "Motion parameter estimation and measured data correction derived from blast-induced vibration: new insights." *Measurement* (**2018**).

356. Hakala, Teemu, et al. "Direct Reflectance Measurements from Drones: Sensor Absolute Radiometric Calibration and System Tests for Forest Reflectance Characterization." *Sensors (Basel, Switzerland)* 18.5 (**2018**).

357. Mihálik, A., R. Ďurikovič, and M. Sejč. "Application of Motion Capture Attributes to Individual Identification under Corridor Surveillance." *Journal of Applied Mathematics, Statistics and Informatics* 14.1 (**2018**): 37-56.

358. Kianifar, Rezvan, and Dana Kulic. "Automatic assessment of the squat quality and risk of knee injury in the single leg squat." U.S. Patent Application 15/826,259, filed October 11, **2018**.

359. Parziale, Nick J., et al. "Amplification and Structure of Streamwise-Velocity Fluctuations in Four Shock-Wave/Turbulent Boundary-Layer Interactions." *2018 Fluid Dynamics Conference*. **2018**.

360. Simon, David M., and Mark T. Wallace. "Integration and Temporal Processing of Asynchronous Audiovisual Speech" *Journal of cognitive neuroscience* 30.3 (**2018**): 319-337.

361. Richter, Craig G., Richard Coppola, and Steven L. Bressler. "Top-down beta oscillatory signaling conveys behavioral context in early visual cortex." *Scientific reports* 8.1 (**2018**): 6991.

362. Dinç, Erdal, and Zehra Yazan. "Wavelet transform-based UV spectroscopy for pharmaceutical analysis" *Frontiers in Chemistry* 6 (**2018**).

363. Bartussek, Jan, and Fritz-Olaf Lehmann. "Sensory processing by motoneurons: a numerical model for low-level flight control in flies." *Journal of The Royal Society Interface* 15.145 (**2018**): 20180408.

364. Ben Hendrickson, Ralf Widenhorn, Paul R. DeStefano and Erik Bodegom, Detection and Reconstruction of Random Telegraph Signals Using Machine Learning, *Image Processing (ICIP),* Athens, **2018**, pp. 2441-2445. Link.

365. Oeltzschner, Georg, et al. "Hadamard editing of glutathione and macromolecule-suppressed GABA." *NMR in Biomedicine* 31.1 (**2018**): e3844.

364. Nocco, Mallika A., Matthew D. Ruark, and Christopher J. Kucharik. "Apparent electrical conductivity predicts physical properties of coarse soils." *Geoderma* 335 (**2019**): 1-11.

366. Лубов, Д. П., М. В. Катков, and Ю. В. Першин. "Вольт–амперные характеристики коммерческих сегнетоэлектрических конденсаторов: отклонения от модели Прейзаха." *ՀՀ ԳԱԱ Տեղեկագիր. Ֆիզիկա* 53.1 (**2018**): 86-95. (Machine translation: Voltage – ampere characteristics of commercial ferroelectric capacitors: deviations from the Preisach model. Armenian NAS RA Bulletin: Physics).

367. Thanos Papanicolaou, Achilleas G. Tsakiris, Micah A Wyssmann, Casey Kramer, Boulder Array Effects on Bedload Pulses and Depositional Patches, *Journal of Geophysical Research: Earth Surface 123(11),* **2018**.

368. Manuja Sharma, et. al., "Optical pH measurement system using a single fluorescent dye for assessing susceptibility to dental caries", *Journal of Biomedical Optics* 24(01):1, **2019**.

369. Mustafa, Muhammad A., David Shekhtman, and Nick J. Parziale. "Single-Laser Krypton Tagging Velocimetry Investigation of Supersonic Air and N 2 Boundary-Layer Flows over a Hollow Cylinder in a Shock Tube." *Physical Review Applied* 11.6 (**2019**): 064013.Link.

370. Karl Auerswald, Franziska K. Fischer, Tanja Winterrath, Robert Brandhuber, "Rain erosivity map for Germany derived from contiguous radar rain data", *Hydrology and Earth System Sciences* 23(4):1819-1832, April **2019** , DOI: 10.5194/hess-23-1819-2019

371. Martin Leblanc, et. al., Actinide mixed oxide conversion by advanced thermal denitration route, *Journal of Nuclear Materials* 519:157-165, March **2019**, DOI: 10.1016/j.jnucmat.2019.03.049

372. Sujan Kumar Roy and Kuldip K. Paliwal, An Iterative Kalman Filter with Reduced-Biased Kalman Gain for Single Channel Speech Enhancement in Non-stationary Noise Condition, *International Journal of Signal Processing Systems* Vol. 7, No. 1, March **2019**. DOI: 10.18178/ijsps.7.1.7-13

373. J. Chen, C. Yang, H. Zhu & Y. Li, Adaptive signal enhancement for overlapped peaks based on weighting factor selection, *Spectroscopy Letters*, Volume 52, **2019**. DOI: 10.1080/00387010.2018.1556219

374. Delfino, I., S. Cavella, and M. Lepore. "Scattering-based optical techniques for olive oil characterization and quality control." *Journal of Food Measurement and Characterization* 13.1 (**2019**): 196-212.

375. Zhang, G. W., et al. "Decomposition of overlapped ion mobility peaks by sparse representation. "*International Journal of Mass Spectrometry* 436 (**2019**): 147-152.

376. Demetriou, Demetris. *An Investigation into Nonlinear Random Vibrations based on Wiener Series Theory*. Diss. University of Cambridge, **2019**.

377. McLaren, Timothy I., René Verel, and Emmanuel Frossard. "The structural composition of soil phosphomonoesters as determined by solution 31P NMR spectroscopy and transverse relaxation (T2) experiments." *Geoderma* 345 (**2019**): 31-37.

378. Yan, Qi, Rui Yang, and Jiwu Huang. "Detection of Speech Smoothing on Very Short Clips." *IEEE Transactions on Information Forensics and Security* (**2019**).

379. Zhang, Weifang, et al. "The Analysis of FBG Central Wavelength Variation with Crack Propagation Based on a Self-Adaptive Multi-Peak Detection Algorithm." *Sensors* 19.5 (**2019**): 1056.

380. Fayolle, Clemence, Mélody Labrune, and Jean-Philippe Berteau. "Raman spectroscopy investigation shows that mineral maturity is greater in CD-1 than in C57BL/6 mice distal femurs after sexual maturity." *Connective Tissue Research* (**2019**).

381. Catlow, C. Richard A., et al. "Synthesis, characterisation and water-gas shift activity of nano-particulate mixed-metal (Al, Ti) cobalt oxides.", **2019**. researchgate.net.

383. Hansen, Lars N., et al. "Low-temperature plasticity in olivine: Grain size, strain hardening, and the strength of the lithosphere." *Journal of Geophysical Research: Solid Earth* (**2019**).

384. Shao, Xueguang, et al. "High order derivative to investigate the complexity of the near infrared spectra of aqueous solutions." *Spectrochimica Acta Part A: Molecular and Biomolecular Spectroscopy* 213 (**2019**): 83-89.

385. Choresh, Yael, et al. "Long-term griffon vulture population dynamics at Gamla Nature Reserve." *The Journal of Wildlife Management* 83.1 (**2019**): 135-144.

386. DeFelice, Mialy M., et al. "NF-κB signaling dynamics is controlled by a dose-sensing autoregulatory loop." *Sci. Signal.*12.579 (**2019**): eaau3568. PDF.

387. Xiaoxiao Ge , et al., Mechanism studies and fabrication for the incorporation of carbon into Al alloys by the electro-charging assisted process, *Carbon*. Page: 203-212, April **2019**. DOI: 10.1016/j.carbon.2019.04.049

388. Natalia Molinero. Et. al, "The human gallbladder microbiome is related to the physiological state and the biliary metabolic profile", *Microbiome* 7(1), **2019**. DOI: 10.1186/s40168-019-0712-8

389. Ci Song and Tao Pei, "Decomposition of Repulsive Clusters in Complex Point Processes with Heterogeneous Components", *International Journal of Geo-Information 8(8):326, **2019**. DOI: 10.3390/ijgi8080326*

390. Stephanie Zaleski, et. al., "Application of Fiber Optic Reflectance Spectroscopy for the Detection of Historical Glass Deterioration", *Journal of the American Ceramic Society,* June **2019**. DOI: 10.1111/jace.16703

391. M A Mustafa and Nick Parziale, "Proper Orthogonal Decomposition of Streamwise-Velocity Fluctuations in a Compression-Corner Shock-Wave/Turbulent Boundary-Layer Interaction", Conference: 32nd International Symposium on Shock Waves (ISSW32), June **2019**, DOI: 10.3850/978-981-11-2730-4_0473-cd

392. Antonio Matus-Vargas, et. al, "Aerodynamic Disturbance Rejection Acting on a Quadcopter Near Ground", Conference: 2019 6th International Conference on Control, Decision and Information Technologies (CoDIT), March **2019**. DOI: 10.1109/CoDIT.2019.8820321

393. Wenqi Cai, "Modeling and Experimental Study of the Vibration Effects in Urban Free-Space Optical Communication Systems", *IEEE Photonics Journal* PP(99):1-1, October **2019**. DOI: 10.1109/JPHOT.2019.2945695

394. Wei, Lingxiao, et al. "I know what you see: Power side-channel attack on convolutional neural network

accelerators." Proceedings of the 34th Annual Computer Security Applications Conference. ACM, **2018**.

395. Shewcraft, Ryan A., et al. "Coherent neuronal dynamics driven by optogenetic stimulation in the primate brain." *bioRxiv* (**2019**): 437970.

396. Burke, I. V., and William Johan. "A robust and automated deconvolution algorithm of peaks in spectroscopic data." (**2019**).

397. Tu, Shen, et al. "Enhanced Formation of Solvent-Shared Ion Pairs in Aqueous Calcium Perchlorate Solution toward Saturated Concentration or Deep Supercooling Temperature and Its Effects on the Water Structure." *The Journal of Physical Chemistry B* 123.45 (**2019**): 9654-9667.

398. Noori, Ansara, et al. "Portable Device for Continuous Sensing with Rapidly Pulsed LEDs–Part 1: Rapid On-the-fly Processing of Large Data Streams using an Open-Source Microcontroller with Field Programmable Gate Array." *Measurement* (**2019**).

399. Kim, Tae Hyong, et al. "Machine learning-based pre-impact fall detection model to discriminate various types of fall." *Journal of biomechanical engineering* 141.8 (**2019**): 081010.

400. Suresh, P. S., Niranjan Kumar Sura, and K. Shankar. "Landing Response Analysis on High-Performance Aircraft Using Estimated Touchdown States." *SAE International Journal of Aerospace* 12.1 (**2019**): 23-40.

401. Muirhead, David K., et al. "Raman Spectroscopy: an effective thermal marker in low temperature carbonaceous fold-thrust belts." *Geological Society, London, Special Publications* 490 (**2019**): SP490-2019.

402. Moreira, Mateus Perrisé, Manuel Castro Carneiro, and Andrey Linhares Bezerra de Oliveira. "Desenvolvimento de um programa para modelagem da curva de titulação de traços de carbonato em solução de hidróxido de lítio concentrado em sistema fechado." (**2019**).

403. Paruzzo, Federico Maria. *New Approaches to NMR Crystallography*. No. THESIS. EPFL, **2019**.

404. Paruzzo, Federico M., and Lyndon Emsley. "High-resolution 1H NMR of powdered solids by homonuclear dipolar decoupling." *Journal of Magnetic Resonance* 309 (**2019**): 106598.

405. Alzamil, Yasser. Optimising the quantitative analysis in functional pet brain imaging. Diss. Cardiff University, **2019**.

406. Walker, Patrick William, "War without oversight: Challenges to the development of autonomous weapons systems.", Thesis, University of Buckingham (**2019**).

407. Yang, Guofeng, et al. "Multiple Constrained Reweighted Penalized Least Squares for Spectral Baseline Correction." *Applied Spectroscopy* (**2019**): 0003702819885002.

408. Zhang, Jing, Shuai Chen, and Guoxiang Sun. "Spectral and chromatographic overall analysis: An insight into chemical equivalence assessment of traditional Chinese medicine." *Journal of Chromatography A* (**2019**): 460556.

409. Richter, Craig G., et al. "Brain rhythms shift and deploy attention." *bioRxiv* (**2019**): 795567.

410. Chen, Weiqi, et al. "An automated microfluidic system for the investigation of asphaltene deposition and dissolution in porous media." *Lab on a Chip* 19.21 (**2019**): 3628-3640.

411. Mukherjee, Soma, Soumi Betal, and Asoke Prasun Chattopadhyay. "Dual sensing and synchronous fluorescence spectroscopic monitoring of Cr3+ and Al3+ using a luminescent Schiff base: Extraction and DFT studies." *Spectrochimica Acta Part A: Molecular and Biomolecular Spectroscopy* (**2019**): 117837.

412. Roy Daipayan, and Daniel W. Armstrong. "Fast super/subcritical fluid chromatographic enantioseparations on superficially porous particles bonded with broad selectivity chiral selectors relative to fully porous particles." *Journal of Chromatography A* 1605 (**2019**): 360339.

413. Kang, Yuhao, and Azriel Z. Genack. "Time delay in a disordered topological system." *arXiv preprint*

arXiv:1912.05151 (**2019**).

414. Chankvetadze, Bezhan. "Recent trends in preparation, investigation and application of polysaccharide-based chiral stationary phases for separation of enantiomers in high-performance liquid chromatography." *TrAC Trends in Analytical Chemistry* (**2019**): 115709.

415. de Paula Pedroza, Ricardo Henrique. *Development of methods based on NIR and Raman spectroscopies together with chemometric tools for the qualitative and quantitative analysis of gasoline*. MS thesis. The University of Bergen, **2019**.

416. Wolf, Moritz, et al. "Synthesis, characterisation and water–gas shift activity of nano-particulate mixed-metal (Al, Ti) cobalt oxides." *Dalton Transactions* 48.36 (**2019**): 13858-13868.

417. Welch, Christopher J. "High throughput analysis enables high throughput experimentation in pharmaceutical process research." *Reaction Chemistry & Engineering* 4.11 (**2019**): 1895-1911.

418. Mironov, N. A., et al. "Methods for Studying Petroleum Porphyrins." *Petroleum Chemistry* 59.10 (**2019**): 1077-1091.

419. Yang, Guofeng, et al. "Spectral features extraction based on continuous wavelet transform and image segmentation for peak detection." *Analytical Methods* (**2019**).

420. Briggs, Tokini Kiki. *An Auto-Picking Algorithm for the Detection of Clay Seams in Potash Mines Using GPR Data*. Diss. Faculty of Graduate Studies and Research, University of Regina, **2019**.

421. Hong, Ning, et al. "High-Speed Rail Suspension System Health Monitoring Using Multi-Location Vibration Data." *IEEE Transactions on Intelligent Transportation Systems* (**2019**).

422. Hu, Jennifer F., et al. "Sequencing-based quantitative mapping of the cellular small RNA landscape." *bioRxiv* (**2019**): 841130.

423. Elsayad, Kareem. "Spectral Phasor Analysis for Brillouin Microspectroscopy." *Frontiers in Physics* 7 (**2019**): 62.

424. Huang, Weinan, et al. "Morphology and cell wall composition changes in lignified cells from loquat fruit during postharvest storage." *Postharvest Biology and Technology* 157 (**2019**): 110975.

425. Kupka, Teobald, et al. "Local aromaticity mapping in the vicinity of planar and nonplanar molecules." *Magnetic Resonance in Chemistry* 57.7 (**2019**): 359-372.

426. Parigger, Christian G. "Measurements of Gaseous Hydrogen–Nitrogen Laser-Plasma." *Atoms* 7.3 (**2019**): 61.

427. Venara, J., et al. "Design and development of a portable β-spectrometer for 90Sr activity measurements in contaminated matrices." *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* (**2019**).

428. Gallmeier, Esther A., et al. "Real time monitoring of the chemistry of hydroxylamine nitrate and iron as surrogates for nuclear materials processing." *Separation Science and Technology* (**2019**): 1-9.

429. Hong, Ning, et al. "High-Speed Rail Suspension System Health Monitoring Using Multi-Location Vibration Data." *IEEE Transactions on Intelligent Transportation Systems* (**2019**).

430. Kielar, Aneta, et al. "Slowing is slowing: Delayed neural responses to words are linked to abnormally slow resting state activity in primary progressive aphasia." *Neuropsychologia* 129 (**2019**): 331-347.

431. Wu, Wenchang, et al. "Diffusivities in 1-Alcohols Containing Dissolved H2, He, N2, CO, or CO2 Close to Infinite Dilution." *The Journal of Physical Chemistry B* 123.41 (**2019**): 8777-8790.

432. Park, Sungchan, K. A. N. G. Jooyoung, and Jungho Kim. "Ultrasound imaging apparatus and method for controlling the same." U.S. Patent Application 10/2 47,824, filed April 2, **2019**.

433. Renda, Fioranna, et al. "kSHREC 'Delta' reflects the shape of kinetochore rather than intrakinetochore

tension." *BioRxiv* (**2019**): 811075.

434. Santiago, Ruben, et al. "Methanol-Promoted Oxidation of Nitrogen Oxide (NO x) by Encapsulated Ionic Liquids." *Environmental science & technology* 53.20 (**2019**): 11969-11978.

435. Hu, Jennifer Fan. A systems-level view of the tRNA epitranscriptome: defining the role of tRNA abundance, stability, and modifications in the bacterial stress response. Diss. Massachusetts Institute of Technology, **2019**.

436. Weaver, Jordan S., Veronica Livescu, and Nathan A. Mara. "A comparison of adiabatic shear bands in wrought and additively manufactured 316L stainless steel using nano-indentation and electron backscatter diffraction." *Journal of Materials Science* 55.4 (**2020**): 1738-1752.

437. Li, Dongmei, Zhiwei Zhu, and Da-Wen Sun. "Visualization of the in-situ distribution of contents and hydrogen bonding states of cellular level water in apple tissues by confocal Raman microscopy." *Analyst* (**2020**).

438. Tsai, Chong-Bin, Wei-Yu Hung, and Wei-Yen Hsu. "A Fast and Effective System for Analysis of Optokinetic Waveforms with a Low-Cost Eye Tracking Device." *Healthcare*. Vol. 9. No. 1. Multidisciplinary Digital Publishing Institute, **2020**.

439. Elsayad, Kareem. "Spectral phasor analysis for Brillouin microspectroscopy." *Frontiers in Physics* 7 (**2019**): 62.

440. Takis, Panteleimon G., et al. "SMolESY: An Efficient and Quantitative Alternative to On-Instrument Macromolecular 1 H-NMR Signal Suppression." *Chemical Science* (**2020**).

441. Pipathanapoompron, Thalerngsak, et al. "Magnetic reader testing for asymmetric oscillation noise." *Journal of Magnetism and Magnetic Materials* (**2020**): 167064.

442. Ito, Motohiro, et al. "Evaluation of cone-beam computed tomography over a small field of view in a water bath based on the modulation transfer function with repeating-edge oversampling." *Journal of Oral Science* (**2020**): 20-0479.

443. Li, Tianjun, Long Chen, and Xiliang Lu. "An Alternating Direction Minimization based denoising method for extracted ion chromatogram." *Chemometrics and Intelligent Laboratory Systems* 206 (**2020**): 104138.

444. Zhang, Jing, Shuai Chen, and Guoxiang Sun. "Spectral and chromatographic overall analysis: An insight into chemical equivalence assessment of traditional Chinese medicine." *Journal of Chromatography A* 1610 (**2020**): 460556.

445. McPherson, David L., Richard Harris, and David Sorensen. "Functional Neuroimaging of the Central Auditory System." *Advances in Audiology and Hearing Science: Volume 1: Clinical Protocols and Hearing Devices* (**2020**): 327.

446. Hebert, Michael J., and David H. Russell. "Tracking the Structural Evolution of 4-Aminobenzoic Acid in the Transition from Solution to the Gas Phase." *The Journal of Physical Chemistry B* 124.11 (**2020**): 2081-2087.

447. Tang, Hui, et al. "On 2D-3D Image Feature Detections for Image-To-Geometry Registration in Virtual Dental Model." *2020 IEEE International Conference on Visual Communications and Image Processing (VCIP)*. IEEE, **2020**.

448. Bhatia, Siddharth, and Karthikeyan Vasudevan. "Comparative proteomics of geographically distinct saw-scaled viper (Echis carinatus) venoms from India." *Toxicon: X* 7 (**2020**): 100048.

449. Resentini, Alberto, et al. "Zircon as a provenance tracer: Coupling Raman spectroscopy and UPb geochronology in source-to-sink studies." *Chemical Geology* 555 (**2020**): 119828.

450. Ajemigbitse, Moses A., Fred S. Cannon, and Nathaniel R. Warner. "A rapid method to determine 226Ra concentrations in Marcellus Shale produced waters using liquid scintillation counting." *Journal of Environmental Radioactivity* 220 (**2020**): 106300.

451. Muirhead, D. K., et al. "Raman spectroscopy: an effective thermal marker in low temperature carbonaceous fold–thrust belts." *Geological Society, London, Special Publications* 490.1 (**2020**): 135-151.

452. Quilfen, Y., and B. Chapron. "On denoising satellite altimeter measurements for high-resolution geophysical signal analysis." *Advances in Space Research* (**2020**).

453. Wu, Dan, Pol Mac Aonghusa, and Donal O'Shea. "Correlation of National and Healthcare Workers COVID-19 Infection Data; Implications for Large-scale Viral Testing Programs." *medRxiv* (**2020**).

454. Battini, Davide, et al. "Modeling Approach and Finite Element Analyses of a Shape Memory Epoxy-Based Material." *Conference of the Italian Association of Theoretical and Applied Mechanics*. Springer, Cham, **2019**.

455. Wu, Billy, et al. "An Energy Storage Device Monitoring Technique." U.S. Patent Application No. 16/088,016, **2020**.

456. Ge, X., et al. "Electrical and structural characterization of nano-carbon–aluminum composites fabricated by electro-charging-assisted process." *Carbon* 173: 115-125, **2021**.

456. Hsu, Gee-Sern Jison, et al. "A deep learning framework for heart rate estimation from facial videos." *Neurocomputing* 417 (**2020**): 155-166.

457. Hammonds, James S., and Kimani A. Stancil. "Phonon effect based nanoscale temperature measurement." U.S. Patent No. 10,520,374. 31 Dec. **2019**.

458. Weaver, Jordan S., Veronica Livescu, and Nathan A. Mara. "A comparison of adiabatic shear bands in wrought and additively manufactured 316L stainless steel using nanoindentation and electron backscatter diffraction." *Journal of Materials Science* 55.4 (**2020**): 1738-1752.

459. Gallagher, John Barry, Vishnu Prahalad, and John Aalders. "Inorganic and black carbon hotspots constrain blue carbon mitigation services across tropical seagrass and temperate tidal marshes." *bioRxiv* (**2020**).

460. Wu, Wenchang, et al. "Mutual and Thermal Diffusivities as well as Fluid-Phase Equilibria of Mixtures of 1-Hexanol and Carbon Dioxide." *The Journal of Physical Chemistry B* 124.12 (**2020**): 2482-2494.

461. Busa, William, Phillip H. Coelho, and Paul Getchel. "Lateral flow immunoassay test reader and method of use." U.S. Patent No. 10,823,746. 3 Nov. **2020**.

462. Dubrovkin, Joseph. *Mathematical processing of spectral data in analytical chemistry: A guide to error analysis*. Cambridge Scholars Publishing, **2019**.

463. Bonnin-Pascual, Francisco, and Alberto Ortiz. "UWB-Based Self-Localization Strategies: A Novel ICP-Based Method and a Comparative Assessment for Noisy-Ranges-Prone Environments." *Sensors* 20.19 (**2020**): 5613.

464. Chaumel, Júlia, et al. "Co-aligned chondrocytes: Zonal morphological variation and structured arrangement of cell lacunae in tessellated cartilage." *Bone* (**2020**): 115264.

465. Bruendl, Stefan A., et al. "A New Emulation Platform for Real-time Machine Learning in Substance Use Data Streams." *2020 IEEE 21st International Conference on Information Reuse and Integration for Data Science (IRI)*. IEEE, **2020**.

466. Cuyt, Annie, and Wen-shin Lee. "Parameter spectral analysis: scale and shift." *arXiv preprint arXiv:2008.02125* (**2020**).

467. Li, Wenda, et al. "A novel processing methodology for traffic-speed road surveys using point lasers." *IEEE Transactions on Intelligent Transportation Systems* (**2019**).

468. Li, Yuanlu, Kun Li, and Qiyu Lu. "Applying segmentation and classification to improve performance of smoothing." *Digital Signal Processing* 109: 102913, **2021**.

469. Rutt, Daryl, et al. "Importance of Accurate and Detailed Data Processing of Laser Mapping in Coke Drum." *Pressure Vessels and Piping Conference*. Vol. 58943. American Society of Mechanical Engineers, **2019**.

470. Al-Mbaideen, Amneh A. "Application of moving average filter for the quantitative analysis of the NIR spectra." *Journal of Analytical Chemistry* 74.7 (**2019**): 686-692.

471. Vitali, L., et al. "Infrared image processing for local convective heat transfer measurements in rib-enhanced channels." *Journal of Physics: Conference Series*. Vol. 1599. No. 1. IOP Publishing, **2020**.

472. Vergel, Ángelo Joseph Soto, Luis Enrique Mendoza, and Byron Medina Delgado. "Analysis of energy and major components in chromatographic signals for the diagnosis of prostate cancer." *Respuestas* 24.1 (**2019**): 76-85.

473. Zhang, Genwei, et al. "Multiscale orthogonal matching pursuit algorithm combined with peak model for interpreting ion mobility spectra and achieving quantitative analysis." *Analytica Chimica Acta* (**2020**).

474. Zhang, Lei, et al. "WiDIGR: Direction-Independent Gait Recognition System Using Commercial Wi-Fi Devices." *IEEE Internet of Things Journal* 7.2 (**2019**): 1178-1191.

475. Wiegand, Patrick. "Raman signal position correction using relative integration parameters." U.S. Patent No. 10,627,289. 21 Apr. **2020**.

476. Botezatu, Irina V., et al. "Asymmetric mutant-enriched polymerase chain reaction and quantitative DNA melting analysis of KRAS mutation in colorectal cancer." *Analytical Biochemistry* 590 (**2020**): 113517.

477. Joubaud, Thomas, and Grégory Pallone. "Electroacoustic method for the calibration of a heterogeneous distributed speaker system." ***2020** 28th European Signal Processing Conference (EUSIPCO)*. IEEE.

478. Xiang, YuChen, et al. "Background-free fibre optic Brillouin probe for remote mapping of micromechanics." *arXiv preprint arXiv:2005.12266* (**2020**).

479. Huang, Ronggang, et al. "Optical frequency and phase information-based fusion approach for image rotation symmetry detection." *Optics Express* 28.13 (**2020**): 18577-18595.

480. Sosin, M., et al. "Impact of vibrations and reflector movements on the measurement uncertainty of Fourier-based frequency sweeping interferometry." *Photonic Instrumentation Engineering VII*. Vol. 11287. International Society for Optics and Photonics, **2020**.

481. Chakraborty, Saikat, and Anup Nandy. "Automatic Diagnosis of Cerebral Palsy Gait Using Computational Intelligence Techniques: A Low-Cost Multi-Sensor Approach." *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 28.11 (**2020**): 2488-2496.

482. Kim, Najin, et al. "Hygroscopicity of urban aerosols and its link to size-resolved chemical composition during spring and summer in Seoul, Korea." *Atmospheric Chemistry and Physics* 20.19 (**2020**): 11245-11262.

483. Ma, Te, et al. "Rapid and nondestructive evaluation of hygroscopic behavior changes of thermally modified softwood and hardwood samples using near-infrared hyperspectral imaging (NIR-HSI)." *Holzforschung* 1.ahead-of-print (**2020**).

484. Laskaris, Nick, et al. "EVIDENCE OF AU-HG GILDING PROCESS IN POST BYZANTINE ECCLESIASTICAL SILVERWARES (CHALICES) OF EASTERN THESSALY BY PXRF." *Mediterranean Archaeology & Archaeometry* 13.1 (**2020**).

485. Obaydo, Reem H., and Amir Alhaj Sakur. "Spectrophotometric strategies for the analysis of binary combinations with minor component based on isoabsorptive point's leveling effect: An application on ciprofloxacin and fluocinolone acetonide in their recently delivered co-formulation." *Spectrochimica Acta Part A: Molecular and Biomolecular Spectroscopy* 219 (**2019**): 186-194.

486. Ma, Liya, and Peter Schegner. "State duration based event detection for domestic power disaggregation." *2019 IEEE Milan PowerTech*. IEEE, **2019**.

497. Коломиец, О. О., and С. В. Глушен. "Суточный ритм роста листьев и пролиферации клеток у перца стручкового (Capsicum annuum L.)." *Известия Национальной академии наук Беларуси. Серия*

биологических наук 64.4 (**2019**): 448-455.

498. Reynes, Julien, Pierre Lanari, and Jörg Hermann. "A mapping approach for the investigation of Ti–OH relationships in metamorphic garnet." *Contributions to Mineralogy and Petrology* 175 (**2020**): 1-17.

499. Shumeyko, Christopher M., et al. "Tunable mechanical behavior of graphene nanoribbon-metal composites fabricated through an electrocharge-assisted process." *Materials Science and Engineering: A* 800 (**2020**): 140289.

500. Psyrras, N., et al. "Physical Modeling of the Seismic Response of Gas Pipelines in Laterally Inhomogeneous Soil." *Journal of Geotechnical and Geoenvironmental Engineering* 146.5 (**2020**): 04020031.

501. Chen, Hong-Jia, et al. "Self-potential ambient noise and spectral relationship with urbanization, seismicity, and strain rate revealed via the Taiwan Geoelectric Monitoring Network." *Journal of Geophysical Research: Solid Earth* 125.1 (**2020**): e2019JB018196.

502. Mustafa, M. A., et al. "Amplification and structure of streamwise-velocity fluctuations in compression-corner shock-wave/turbulent boundary-layer interactions." *Journal of Fluid Mechanics* 863 (**2019**): 1091-1122.

503. Mekonnen, Alemu, et al. "Improved Biomass Cookstove Use in the Longer Run: Results from a Field Experiment in Rural Ethiopia." *World Bank Policy Research Working Paper* 9272 (**2020).**

504. Bradshaw, Peter R., et al. "Kinetic modelling of acyl glucuronide and glucoside reactivity and development of structure–property relationships." *Organic & Biomolecular Chemistry* 18.7 (**2020**): 1389-1401.

505. Bluffstone, Randall, et al. "Does providing improved biomass cooking stoves free-of-charge reduce regular usage? Do use incentives promote habits?." *LAND ECONOMICS* (**2020**).

506. Guo, Qimei, et al. "Mediterranean Outflow Water dynamics across the middle Pleistocene transition based on a 1.3 million-year benthic foraminiferal record off the Portuguese margin." *Quaternary Science Reviews* 247 (**2020**): 106567.

507. Mustafa, Muhammad A., David Shekhtman, and Nick J. Parziale. "Single-Laser Krypton Tagging Velocimetry (KTV) Investigation of Air and N2 Boundary-Layer Flows Over a Hollow Cylinder in the Stevens Shock Tube." *AIAA Scitech 2019 Forum*. **2019**.

508. Wang, M., et al. "Evolution of dislocation and twin densities in a Mg alloy at quasi-static and high strain rates." *Acta Materialia* 201 (**2020**): 102-113.

509. AlOmar, AbdulAzeez S. "Accurate Chebyshev Approximations for the Width of the Voigt Profile, Differential Peaks, and Deconvolution of the Lorentzian Width." *Optik* 225: 165533, **2021**.

510. Hoyer, Jorgen, et al. "Mapping calcium dynamics in a developing tubular structure." *bioRxiv* (**2020**).

511. Hansen, Lars N., et al. "Low-Temperature Plasticity in Olivine: Grain Size, Strain Hardening, and the Strength of the Lithosphere." *Journal of Geophysical Research. Solid Earth* 124.6 (**2019**).

512. Du, Siqi, et al. "Complete identification of all 20 relevant epimeric peptides in β-amyloid: a new HPLC-MS based analytical strategy for Alzheimer's research." *Chemical Communications* 56.10 (**2020**): 1537-1540.

513. Hebden, Jeremy C. "Exploring the feasibility of wavelength modulated near-infrared spectroscopy." *Journal of Biomedical Optics* 25.11 (**2020**): 110501.

514. Aikin, Timothy J., et al. "MAPK activity dynamics regulate non-cell autonomous effects of oncogene expression." *Elife* 9 (**2020**): e60541.

515. Pepermans, Vincent, et al. "Column-in-Valve Designs to Minimize Extra-Column Volumes." *Journal of Chromatography A* (**2020**): 461779.

516. Chua, Emily J., et al. "A mass spectrometer-based pore-water sampling system for sandy sediments." *Limnology and Oceanography: Methods* 19.11 (**2021**): 769-784.

517. Sanchini, Andrea, and Martin Grosjean. "Quantification of chlorophyll a, chlorophyll b and pheopigments a in lake sediments through deconvolution of bulk UV–VIS absorption spectra." *Journal of paleolimnology* 64 (**2020**): 243-256.

518. Yuen, Clement, et al. "Towards malaria field diagnosis based on surface-enhanced Raman scattering with on-chip sample preparation and near-analyte nanoparticle synthesis." *Sensors and Actuators B: Chemical* (**2021**): 130162.

519. Pal, Arpan, et al. "Instant Adaptive Learning: An Adaptive Filter Based Fast Learning Model Construction for Sensor Signal Time Series Classification on Edge Devices." *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, **2020**.

520. Razzini, Jhonathan, Marcelo Vandresen, and Luciano Amaury dos Santos. *Comparative of the Mathematical Smoothing Model for Inertial Dynamometer Software*. No. 2020-36-0151. SAE Technical Paper, **2021**.

521. Wang, Yaheng, et al. "High-speed 600-GHz-Band Terahertz Imaging System Using Polygon Mirror." ***2021** International Topical Meeting on Microwave Photonics (MWP)*. IEEE, **2021**.

522. Shanmugarajah, Sujeevan, Janani Tharmaseelan, and Luckman Sivagnanam. "AI Approach In Monitoring The Physical And Psychological State Of Car Drivers And Remedial Action For Safe Driving." ***2020** 2nd International Conference on Advancements in Computing (ICAC)*. Vol. 1. IEEE, **2020**.

523. Likhachev, D. V. "On the optimization of knot allocation for B-spline parameterization of the dielectric function in spectroscopic ellipsometry data analysis." *Journal of Applied Physics* 129.3 (**2021**): 034903.

524. Li, Yuanlu, Kun Li, and Qiyu Lu. "Applying segmentation and classification to improve performance of smoothing." *Digital Signal Processing* 109 (**2021**): 102913.

525. C. Dela Cruz, Jennifer, et al. "Deriving Heart Rate and Respiratory Rate from ECG Using Wavelet Transform." *2021 11th International Conference on Biomedical Engineering and Technology*. **2021**.

526. Zhang, Genwei, et al. "Coulombic effects on resolution of ion mobility spectrometry and its application in online qualitative analysis." *Analytica Chimica Acta* 1183 (**2021**): 338969.

527. Ramakrishnan, Saminathan, et al. "Dependence of phase transition uniformity on crystal sizes characterized using birefringence." *Structural Dynamics* 8.3 (**2021**): 034301.

528. Gupta, Smith. "Clustering based method for finding spikes in insect neurons." *arXiv preprint arXiv:2111.11152* (2021).

529. Kocevska, Stefani, et al. "Spectroscopic Quantification of Target Species in a Complex Mixture Using Blind Source Separation and Partial Least-Squares Regression: A Case Study on Hanford Waste." *Industrial & Engineering Chemistry Research* 60.27 (**2021**): 9885-9896.

530. McAvan, Bethan S., et al. "Raman Spectroscopy to Monitor Post-translational Modifications and Degradation in mAb Therapeutics." http://www.biospec.net/pubs/pdfs/McAvan-AnalChem2020SI.pdf

531. Torres-Contreras, Ignacio, et al. "Effects of Phase Shift Errors in Recurrence Plot for Rotating Machinery Fault Diagnosis." *Applied Sciences* 11.2 (**2021**): 873.

532. Renney, Harri, Benedict R. Gaster, and Thomas J. Mitchell. "There and Back Again: The Practicality of GPU Accelerated Digital Audio." PDF link

533. Feukeu, Etienne Alain, and Simon Winberg. "Photoplethysmography Heart Rate Monitoring: State-of-the-Art Design." *International Journal of E-Health and Medical Communications (IJEHMC)* 12.3 (**2021**): 17-37.

534. Munier, Pierre, et al. "Assembly of cellulose nanocrystals and clay nanoplatelets studied by time-resolved X-ray scattering." *Soft Matter* 17.23 (**2021**): 5747-5755.

535. Izima, Obinna, Ruairí de Fréin, and Mark Davis. "Predicting quality of delivery metrics for adaptive video

codec sessions." *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEEE, **2020**.

536. Sun, Yuanchang, and Jack Xin. "Lorentzian peak sharpening and sparse blind source separation for NMR spectroscopy." *Signal, Image and Video Processing* (**2021**): 1-9.

537. Tan, Jiajie, et al. "Implicit Multimodal Crowdsourcing for Joint RF and Geomagnetic Fingerprinting." *IEEE Transactions on Mobile Computing* (**2021**).

538. Guo, Zhenyu, et al. "Anthropometric-based clustering of pinnae and its application in personalizing HRTFs." *International Journal of Industrial Ergonomics* 81 (**2021**): 103076.

539. Xu, Susan Shuhong, et al. "Comparison of ISO work of breathing and NIOSH breathing resistance measurements for air-purifying respirators." *Journal of occupational and environmental hygiene* 18.8 (**2021**): 369-377.

540. Hovareshti, Pedram, et al. "VestAid: A Tablet-Based Technology for Objective Exercise Monitoring in Vestibular Rehabilitation." *Sensors* 21.24 (**2021**): 8388.

541. Ramakrishnan, Saminathan, et al. "A combined approach to characterize ligand-induced solid–solid phase transitions in biomacromolecular crystals." *Journal of Applied Crystallography* 54.3 (**2021**).

542. Dioumaev, Andrei K., et al. "Determining material parameters with resonant acoustic spectroscopy." *Applied Optical Metrology IV*. Vol. 11817. International Society for Optics and Photonics, **2021**.

543. Lu, Min, et al. "Accurate construction of 3-D numerical breast models with anatomical information through MRI scans." *Computers in Biology and Medicine* 130 (**2021**): 104205.

544. Pasquali, Mattia, et al. "Nanomechanical Characterization of Organic Surface Passivation Films on 50 nm Patterns during Area-Selective Deposition." *ACS Applied Electronic Materials* (**2021**).

545. Kalambet, Yuri. "Data acquisition and integration." *Gas Chromatography*. Elsevier, **2021**. 505-524.

546. Ramakrishnan, Saminathan, et al. "Synchronous RNA conformational changes trigger ordered phase transitions in crystals." *Nature communications* 12.1 (**2021**): 1-10.

547. Ke, Jie, et al. "Self-Optimization of Continuous Flow Electrochemical Synthesis Using Fourier Transform Infrared and Gas Chromatography." *Applied Spectroscopy* (**2021**): 00037028211059848.

548. Kim, Namgyun, Jinwoo Kim, and Changbum R. Ahn. "Predicting workers' inattentiveness to struck-by hazards by monitoring biosignals during a construction task: A virtual reality experiment." *Advanced Engineering Informatics* 49 (**2021**): 101359.

549. Jonker, D., et al. "A wafer-scale fabrication method for three-dimensional plasmonic hollow nanopillars." *Nanoscale advances* 3.17 (**2021**): 4926-4939.

550. Shekhtman, D., et al. "Freestream velocity-profile measurement in a large-scale, high-enthalpy reflected-shock tunnel." *Experiments in Fluids* 62.5 (**2021**): 1-13.

551. Rezaee, Mohammad, Iulian Iordachita, and John W. Wong. "Ultrahigh dose-rate (FLASH) x-ray irradiator for pre-clinical laboratory research." *Physics in Medicine & Biology* 66.9 (**2021**): 095006.

552. Chang, Ji Woong, Antonios Armaou, and Robert M. Rioux. "Continuous Injection Isothermal Titration Calorimetry for In Situ Evaluation of Thermodynamic Binding Properties of Ligand–Receptor Binding Models." *The Journal of Physical Chemistry B* 125.29 (**2021**): 8075-8087.

553. Bärmann, Peer, et al. "Solvent Co-intercalation into Few-layered Ti3C2T x MXenes in Lithium Ion Batteries Induced by Acidic or Basic Post-treatment." *ACS nano* 15.2 (**2021**): 3295-3308.

554. Cuyt, Annie, and Wen-shin Lee. "Parametric spectral analysis: scale and shift." *arXiv preprint arXiv:2008.02125* (**2020**).

555. Shekhtman, David, Nick J. Parziale, and Muhammad A. Mustafa. "Excitation Line Optimization for

Krypton Tagging Velocimetry and Planar Laser-Induced Fluorescence in 200-220 nm Range." *AIAA Scitech 2021 Forum*. **2021**.

556. Brasiliense, Vitor, et al. "Nanopipette-based electrochemical SERS platforms: Using electrodeposition to produce versatile and adaptable plasmonic substrates." *Journal of Raman Spectroscopy* 52.2 (**2021**): 339-347.

557. Qian, Yiwen, et al. "Crystallization of nanoparticles induced by precipitation of trace polymeric additives." *Nature communications* 12.1 (**2021**): 1-8.

558. Raman, Narayanan, et al. "GaPt Supported Catalytically Active Liquid Metal Solution Catalysis for Propane Dehydrogenation–Support Influence and Coking Studies." *ACS catalysis* 11.21 (**2021**): 13423-13433.

559. Phounglamcheik, Aekjuthon, et al. "CO2 Gasification Reactivity of Char from High-Ash Biomass." *ACS Omega* (**2021**).

560. Leaston, Joshua, et al. "Neurovascular imaging with QUTE-CE MRI in APOE4 rats reveals early vascular abnormalities." *PLoS One* 16.8 (**2021**): e0256749.

561. Asmala, Eero, Philippe Massicotte, and Jacob Carstensen. "Identification of dissolved organic matter size components in freshwater and marine environments." *Limnology and Oceanography* 66.4 (**2021**): 1381-1393.

562. Hu, Jennifer F., et al. "Quantitative mapping of the cellular small RNA landscape with AQRNA-seq." *Nature Biotechnology* (**2021**): 1-11.

563. Tomlinson, Lauren J., et al. "Exploring the conformational landscape and stability of Aurora A using ion-mobility mass spectrometry and molecular modelling." *bioRxiv* (**2021**).

564. Zhang, Qin, and Benjamin M. Tutolo. "Geochemical evaluation of glauconite carbonation during sedimentary diagenesis." *Geochimica et Cosmochimica Acta* 306 (**2021**): 226-244.

565. Parigger, Christian G., Christopher M. Helstern, and Ghaneshwar Gautam. "Hypersonic imaging and emission spectroscopy of hydrogen and cyanide following laser-induced optical breakdown." *Symmetry* 12.12 (**2020**): 2116.

566. Parigger, Christian G. "Laser-plasma and stellar astrophysics spectroscopy." *Contrib. Astron. Obs. Skalnaté Pleso* 50 (**2020**): 15-31.

567. Wolf, Moritz, et al. "Coke formation during propane dehydrogenation over Ga− Rh supported catalytically active liquid metal solutions." *ChemCatChem* 12.4 (**2020**): 1085.

568. Chen, Weiqi, et al. "Experimental data-driven reaction network identification and uncertainty quantification of CO2-assisted ethane dehydrogenation over Ga2O3/Al2O3." *Chemical Engineering Science* 237 (**2021**): 116534.

569. Roy, Sujan Kumar, and Kuldip K. Paliwal. "A noise PSD estimation algorithm using derivative-based high-pass filter in non-stationary noise conditions." *EURASIP Journal on Audio, Speech, and Music Processing* 2021.1 (**2021**): 1-18.

570. Fannes Claverol, Jean Paul. *Feasibility study of an ADAS using RADAR for In-Cabin pilot health parameters monitoring*. MS thesis. Universitat Politècnica de Catalunya, **2021**.

571. Thu, Nguyen Anh. "Quantification of acetaminophen, caffeine and ibuprofen in solid dosage forms by uv spectroscopy coupled with multivariate analysis." *Asian Journal of Pharmaceutical Analysis* 11.2 (**2021**): 127-132.

572. Mutebi, John-Paul, et al. "Diel Activity Patterns of Two Distinct Populations of Aedes Aegypti in Miami, FL and Brownsville, TX." (**2021**).

573. Hobson, Eric C., et al. "Resonant acoustic rheometry for non-contact characterization of viscoelastic biomaterials." *Biomaterials* 269 (**2021**): 120676.

574. Kurttila, Moona, et al. "Site-by-site tracking of signal transduction in an azidophenylalanine-labeled

bacteriophytochrome with step-scan FTIR spectroscopy." *Physical Chemistry Chemical Physics* 23.9 (**2021**): 5615-5628.

575. Wang, Hao, et al. "Rapid SERS quantification of trace fentanyl laced in recreational drugs with a portable Raman module." *Analytical chemistry* 93.27 (**2021**): 9373-9382.

576. Smith, Alexander J., et al. "Expanded in situ aging indicators for lithium-ion batteries with a blended NMC-LMO electrode cycled at sub-ambient temperature." *Journal of The Electrochemical Society* 168.11 (**2021**): 110530.

577. Jenkins, Lauren M., et al. "Quantification of Acyl-Acyl Carrier Proteins for Fatty Acid Synthesis Using LC-MS/MS." *Plant Lipids*. Humana, New York, NY, **2021**. 219-247.

578. Heck, Anisa, et al. "Volume Fraction Measurement of Soft (Dairy) Microgels by Standard Addition and Static Light Scattering." *Food Biophysics* 16.2 (**2021**): 237-253.

579. Teixeira, Paulo Sérgio, et. Al.. "Avaliação das respostas em frequências naturais de um violão pelo método de excitação por impulso e deconvolução de sinais (Evaluation of the natural frequency responses of a guitar by the method of impulse excitation and signal deconvolution)." *Research, Society and Development* 10.1 (**2021**)

580. Burg, David, and Jesse H. Ausubel. "Moore's Law revisited through Intel chip density." *PloS one* 16.8 (**2021**): e0256245.

581. Poorna, S. S., et al. "A transfer learning approach for drowsiness detection from EEG signals." *Innovations in Computational Intelligence and Computer Vision*. Springer, Singapore, 2021. 369-375.

582. Yang, Guofeng, et al. "Injection profile surveillance using impulse oxygen activation logging based on optimization theory." *Journal of Petroleum Science and Engineering* 196 (**2021**): 107701.

583. Navarro-Huerta, Jose Antonio, et al. "Ultra-short ion-exchange columns for fast charge variants analysis of therapeutic proteins." *Journal of Chromatography A* 1657 (**2021**): 462568.

584. Wahab, M. Farooq, Daipayan Roy, and Daniel W. Armstrong. "The theory and practice of ultrafast liquid chromatography: A tutorial." *Analytica Chimica Acta* 1151 (**2021**): 238170.

585. Samokhvalov, Alexander. "Understanding the structure, bonding and reactions of nanocrystalline semiconductors: a novel high-resolution instrumental method of solid-state synchronous luminescence spectroscopy." *Physical Chemistry Chemical Physics* 23.12 (**2021**): 7022-7036.

586. Ghadimloozadeh, Shaghayegh, Mahmoud Reza Sohrabi, and Hassan Kabiri Fard. "Development of rapid and simple spectrophotometric method for the simultaneous determination of anti-parkinson drugs in combined dosage form using continuous wavelet transform and radial basis function neural network." *Optik* 242 (**2021**): 167088.

587. de Falco, Giacomo, et al. "Proposing an unbiased oxygen reduction reaction onset potential determination by using a Savitzky-Golay differentiation procedure." *Journal of Colloid and Interface Science* 586 (**2021**): 597-600.

588. Readel, Elizabeth R., Michael Wey, and Daniel W. Armstrong. "Rapid and selective separation of amyloid beta from its stereoisomeric point mutations implicated in neurodegenerative Alzheimer's disease." *Analytica Chimica Acta* 1163 (**2021**): 338506.

589. Tatarinov, Danila A., Sofia R. Sokolnikova, and Natalia A. Myslitskaya. "Applying of Chitosan-TiO2 Nanocomposites for Photocatalytic Degradation of Anthracene and Pyrene." *Journal of Biomedical Photonics & Engineering* 7.1 (**2021**): 010301.

590. Kim, Min-Yeong, et al. "Highly stable potentiometric sensor with reduced graphene oxide aerogel as a solid contact for detection of nitrate and calcium ions." *Journal of Electroanalytical Chemistry* 897 (**2021**): 115553.

591. Karongo, Ryan, et al. "Rapid enantioselective amino acid analysis by ultra-high performance liquid

chromatography-mass spectrometry combining 6-aminoquinolyl-N-hydroxysuccinimidyl carbamate derivatization with core-shell quinine carbamate anion exchanger separation." *Journal of Chromatography Open* 1 (**2021**): 100004.

592. Gritti, Fabrice, and Farooq Wahab. "Extraction of intrinsic column peak profiles of narrow-bore and microbore columns by peak deconvolution methods." *Analytica Chimica Acta* 1180 (**2021**): 338851.

593. Russell, B., et al. "Investigating the potential of tandem inductively coupled plasma mass spectrometry (ICP-MS/MS) for 41 Ca determination in concrete." *Journal of Analytical Atomic Spectrometry* 36.4 (**2021**): 845-855.

594. Tanács, Dániel, et al. "Enantioseparation of ß2-amino acids by liquid chromatography using core-shell chiral stationary phases based on teicoplanin and teicoplanin aglycone." *Journal of Chromatography A* 1653 (**2021**): 462383.

595. Ewusi-Annan, Ebo, and Noureddine Melikechi. "Unsupervised fitting of emission lines generated from laser-induced breakdown spectroscopy." *Spectrochimica Acta Part B: Atomic Spectroscopy* 177 (**2021**): 106109.

596. Niezen, Leon E., Peter J. Schoenmakers, and Bob WJ Pirok. "Critical comparison of background correction algorithms used in chromatography." *Analytica Chimica Acta* 1201 (**2022**): 339605.

597. Kendir, Gülsen, Ayşegül Köroğlu, and Erdal Dinç. "SIMULTANEOUS SPECTROPHOTOMETRIC QUANTITATION OF RUTIN AND CHLOROGENIC ACID IN LEAVES OF Ribes uva-crispa L. BY ONE-DIMENSIONAL CONTINUOUS WAVELET TRANSFORMS." *Journal of the Chilean Chemical Society* 66.1 (**2021**): 5041-5046.

598. Nosal, Daniel G., Douglas L. Feinstein, and Richard B. van Breemen. "Chiral liquid chromatography-tandem mass spectrometry analysis of superwarfarin rodenticide stereoisomers–Bromadiolone, difenacoum and brodifacoum–In human plasma." *Journal of Chromatography B* 1165 (**2021**): 122529.

599. Wang, Zhengshuo, et al. "Recent progress in organic color-tunable phosphorescent materials." *Journal of Materials Science & Technology* 101 (**2022**): 264-284.

600. Kensert, Alexander, et al. "Deep convolutional autoencoder for the simultaneous removal of baseline noise and baseline drift in chromatograms." *Journal of Chromatography A* 1646 (**2021**): 462093.

601. Zaynidinov, H. N., et al. "Algorithms and Service for Digital Processing of Two-Dimensional Geophysical Fields Using Octave Method." *International Conference on Intelligent Human Computer Interaction*. Springer, Cham, **2021**.

602. Singh, Ritu, Navin Rajpal, and Rajesh Mehta. "Non-invasive Single Channel integration model for fetal ECG extraction and sustainable fetal healthcare using wavelet framework." *Multimedia Tools and Applications* (**2022**): 1-27.

603. Luo, Canhuang, et al. "Nudging the N170 forward with prior stimulation—Bridging the gap between N170 and recognition potential." *Human brain mapping* 43.4 (**2022**): 1214-1230.

604. Kurapati, Hemasai. "How does the period of oscillation of a cantilever relate with its mass?." (**2022**). PDF file

605. Edelman, Joel. "Overtone Invariants are the Framework for Musical Consonance; An Illustrated Overview." Humanities Commons, PDF file. (**2022**).

606. Hsu, W. Y., Y. W. Cheng, and C. B. Tsai. "An Effective Algorithm to Analyze the Optokinetic Nystagmus Waveforms from a Low-Cost Eye Tracker. *Healthcare 2022*, 10, 1281." (**2022**).

607. Ghosh, Koushik. "Smart filter and smoothing: A new approach of data denoising." Noise Filtering for Big Data *Analytics* 12 (**2022**): 139.

608. Maggiorea, F., et al. "Process optimization by real time analysis of liquids' composition in Metal & Mining." *TOS forum*. No. 1. IM Publications Open, **2022**.

609. Mehic, Amela. "Development of a computational method for determining gamma energy escape from calorimeters at CLAB." (**2022**).

610. Xue, Qingsheng, et al. "Detection of microplastics based on spatial heterodyne Raman spectroscopy." *Spectrochimica Acta Part A*: Molecular and Biomolecular Spectroscopy 283 (**2022**): 121712.

611. Zhao, Nie, et al. "SGTools: a suite of tools for processing and analyzing large data sets from in situ X-ray scattering experiments." *Journal of Applied Crystallography* 55.1 (**2022**).

612. Jeong, Mok Kun, Min Joo Choi, and Sung Jae Kwon. "High-spatial-resolution, instantaneous passive cavitation imaging with temporal resolution in histotripsy: a simulation study." *Ultrasonography* 41.3 (**2022**): 566.

613. Li, X., et al. "Process-Oriented Estimation of Chlorophyll-a Vertical Profile in the Mediterranean Sea Using MODIS and Oceanographic Float Products. Front." *Mar. Sci* 9 (**2022**): 933680.

614. Riddick, Stuart N., et al. "Estimating Regional Methane Emission Factors from Energy and Agricultural Sector Sources Using a Portable Measurement System: Case Study of the Denver–Julesburg Basin." *Sensors* 22.19 (**2022**): 7410.

615. Yang, Fanlin, et al. "An airborne LiDAR bathymetric waveform decomposition method in very shallow water: A case study around Yuanzhi Island in the South China Sea." *International Journal of Applied Earth Observation and Geoinformation* 109 (**2022**): 102788.

616. Utt, Kainen L., et al. "Spatially-Resolved Mid-Infrared Spectral Evidence of Space Weathering." *Authorea Preprints* (**2022**).

617. Termsuk, C., S. J. Sweeney, and C. Shenton-Taylor. "Thermoluminescence glow curve study of beta irradiated germanium doped core fibre with different dopant concentrations." *Radiation Physics and Chemistry* 193 (**2022**): 109974.

618. Alomar, Abdulazeez S. "Impact of Faddeeva–Voigt broadening on line-shape analysis at critical points of dielectric functions." *AIP Advances* 12.6 (**2022**): 065127.

619. Likhachev, D. V. "A method of optimal dielectric function modeling by B-splines for spectroscopic ellipsometry analysis.", researchgate.net, 07/31/**2022**

620. Li, Dongmei, Zhiwei Zhu, and Da-Wen Sun. "Visualization and quantification of content and hydrogen bonding state of water in apple and potato cells by confocal Raman microscopy: A comparison study." *Food Chemistry* 385 (**2022**): 132679.

621. Zhong, Yu, et al. "Summary Report of CALPHAD GLOBAL, 2021." Available at SSRN 4229474.

622. Xuan, Doan Thanh, and Vu Dang Hoang. "Application of Fourier transform-based algorithms to resolve spectral overlapping for UV spectrophotometric co-assay of spiramycin and metronidazole in tablets." *Spectrochimica Acta Part A: Molecular and Biomolecular Spectroscopy* 277 (**2022**): 121253.

624. Lâm, Vũ Tùng, et al. "Simultaneous quantification and solubility test of tenofovir disoproxil fumarate and emtricitabine in tablets by ultraviolet spectrum derivation using fast Fourier transform algorithm." Link: hup.edu.vn

625. Pasquali, Mattia, et al. "Understanding Selectivity Loss Mechanisms in Selective Material Deposition by Area Deactivation on 10 nm Cu/SiO2 Patterns." *ACS Applied Electronic Materials* 4.4 (**2022**): 1703-1714.

625. Zlabinger, Johannes. *Development of a peak deconvolution software for X-ray fluorescence spectra*. Diss. Wien, **2022**.

626. García-Figueroa, Jesús M. *Depiction of the Ambipolar-State of the Gas-Substrate Interphase of the Electron Cyclotron Resonance–Microwave–Chemical Vapor Deposition (Ecr-Mw-Cvd) Method and Its Influence over the Properties of Vapor-Deposited Hydrocarbon Films*. Diss. University of Rochester, **2022**.

627. Schulze, H. Georg, et al. "Critical Evaluation of Spectral Resolution Enhancement Methods for Raman Hyperspectra." *Applied spectrosco*py 76.1 (**2022**): 61-80.

628. Danilenko, S. D. "Research of methods of digital-analog conversion for the implementation of mechanical co-living for the support of the software and hardware complex." (**2022**).

629. Vasquez, Krystal TonyBeth. *Isomer separation of multifunctional atmospheric compounds using gas chromatography and chemical ionization mass spectrometry*. Diss. California Institute of Technology, **2022**.

630. Hesgrove, Cherie S., et al. "Tardigrade CAHS Proteins Act as Molecular Swiss Army Knives to Mediate Desiccation Tolerance through Multiple Mechanisms." bioRxiv (**2021**): 2021-08.

631. Shareef, Abdulwahhab F., and Riyadh Z. Mahmoud. "Study on Turning Arabic Text into Spoken Words." AL-Rafidain *Journal of Computer Sciences and Mathematics* 15.1 (**2021**): 197-209.

632. Teixeira, Paulo Sérgio, Alexandre Furtado Ferreira, and José Flávio Silveira Feiteira. "Simulação matemática dos sinais sonoros do violão através da convolução." *Cadernos UniFOA* 16.46 (**2021**).

633. Moss, Frank R., et al. "Brominated Lipid Probes Expose Structural Asymmetries in Constricted Membranes." *bioRxiv* (**2021**).

634. Barmann, Peer, et al. "Solvent co-intercalation into few-layered Ti3C2T x MXenes in lithium ion batteries induced by acidic or basic post-treatment." *ACS nano* 15.2 (**2021**): 3295-3308.

635. Guo, Shanzeng, Salman Akhtar, and Anthony Mella. "A method for radar model identification using time-domain transient signals." *IEEE Transactions on Aerospace and Electronic Systems* 57.5 (**2021**): 3132-3149.

636. Dasappa, Shruthi. *Fundamental Studies of Gas-to-Particle Conversion for Nanoparticle Synthesis in Fames*. Diss. University of California, San Diego, **2021**.

637. Raab, Meaghan T., Alexandra K. Prýmek, and Andrea N. Giordano. "ESTIMATION OF THE GROUND AND EXCITED STATE DIPOLE MOMENTS FOR IBUPROFEN AND NAPROXEN SODIUM USING THE SOLVATOCHROMIC SHIFT METHOD." *Journal of Undergraduate Chemistry Research* 20.4 (**2021**): 68.

638. Hu, Jennifer F., et al. "Quantitative mapping of the cellular small RNA landscape with AQRNA-seq." *Nature biotechnology* 39.8 (**2021**): 978-988.

639. Lee, Sung Hoon, et al. "A molecular clock controls periodically driven cell migration in confined spaces." *Cell Systems* (**2022**).

640. Balewski, Zuzanna Z., Eric B. Knudsen, and Joni D. Wallis. "Fast and slow contributions to decision-making in corticostriatal circuits." *Neuron* (**2022**).

641. Firrone, Christian Maria, Antonio Giuseppe D'Ettole, and Matteo Nicita. "Extraction of relevant data from experimental test and matching with FEA." PDF file link. **2022**

642. Baroudi, Ahmad Jamal. "A Tool for Biometric Interpretation of Forensic STR DNA Profiles." (**2022**). PDF file link.

643. Senecaut, Nicolas. Nouvelles approches de quantification des variations du protéome au niveau des protéines intactes: analyses expérimentales et computationnelles. Diss. Universite Paris Cité, **2022**. PDF file link.

644. Zhang, Linshan, et al. "Colorimetric detection for uranyl ions in water using vinylphosphonic acid functionalized gold nanoparticles based on smartphone." *Spectrochimica Acta Part A: Molecular and Biomolecular Spectroscopy* 269 (**2022**): 120748.

645. Ji, Songbai, Shaoju Wu, and Wei Zhao. "Dynamic characteristics of impact-induced brain strain in the corpus callosum." *Brain Multiphysics* 3 (**2022**): 100046.

646. Sabr, Muhammad W., and Diyar S. Ali. "H-point standard addition method for simultaneous determination of phenylephrine hydrochloride, chlorpheniramine maleate, and paracetamol as a ternary mixture in pharmaceutical formulations." *Journal of the Indian Chemical Society* (**2022**): 100526.

647. Takihata, Yasuhiro, et al. "In vivo diffuse reflectance spectroscopic analysis of fatty liver with inflammation in mice." *Surgery Open Science* 6 (**2021**): 21-28.

648. Liang, Chen, et al. "Meter scale and sub-second resolution coherent Doppler wind lidar and hyperfine." *Optics Letters Vol. 47,* 10 (**2022**)

649. Shang, Qiufeng, et al. "Pink noise removal and spectral distortion correction based fiber Bragg grating demodulation algorithm." *Optics Express* 30.2 (**2022**): 1066-1080.

650. Liu, Luzheng, et al. "Selective Detection of Mixtures via a Single Nonselective Sensor—Making the Unworkable Sensor Workable by Machine Learning." *Advanced Intelligent Systems* (**2022**): 2200136.

651. Li, Ping, et al. "Discrimination of raw and sulfur-fumigated ginseng based on Fourier transform infrared spectroscopy coupled with chemometrics." *Microchemical Journal* 181 (**2022**): 107767.

652. Lum, Jordan S., et al. "In Situ Optical Detection for Ultrasonic Characterization of Materials in a Mach 10 Hypersonic Wind Tunnel." *Physical Review Applied* 18.4 (**2022**): 044062.

653. de CASTRO, PEDRO AA, et al. "Assessment of bone dose response using ATR-FTIR spectroscopy." *Spectrochimica Acta Part A: Molecular and Biomolecular Spectroscopy* (**2022**).

654. Zanotto, S., et al. "Optomechanical Modulation Spectroscopy of Bound States in the Continuum in a Dielectric Metasurface." *Physical Review Applied* 17.4 (**2022**): 044033.

655. Wang, Kristen. *Quantification of Resveratrol in Red Wine using Liquid Chromatography—Surface-Enhanced Raman Spectroscopy (LC-SERS)*. Diss. The Ohio State University, **2022**.

656. Belal, Fathalla, et al. "Multi‐spectroscopic, thermodynamic and molecular docking studies to investigate the interaction of eplerenone with human serum albumin." *Luminescence* 37.7 (**2022**): 1162-1173.

657. Kim, Myung-Hoon. "Advances in Derivative Voltammetry-A Search for Diagnostic Criteria of Several Electrochemical Reaction Mechanisms." *Analytical Chemistry-Advancement, Perspectives and Applications*. IntechOpen, **2021**.

658. Zhou, Yongjie, et al. "An improved algorithm for peak detection based on weighted continuous wavelet transform." *IEEE Access* (**2022**).

659. Al-Samarrai, Mumin F. Hamad, Emad T. Hanon, and Ali I. Khaleel Khaleel. "Development of derivative of subtracting spectra method for the simultaneous determination of some decongestant drugs." *Samarra Journal of Pure and Applied Science* 3.3 (**2021**): 18-30.

660. Bishop, Logan DC, Anastasiia Misiura, and Christy F. Landes. "A new metric for relating macroscopic chromatograms to microscopic surface dynamics: the distribution function ratio (DFR)." *Analyst* 146.13 (**2021**): 4268-4279.

661. de Castro, Pedro Arthur Augusto, et al. "Assessment of bone dose response using ATR-FTIR spectroscopy: A potential method for biodosimetry." *Spectrochimica Acta Part A: Molecular and Biomolecular Spectroscopy* 273 (**2022**): 120900.

662. Luo, Hao, et al. "In Situ Nanofluid Dispersion Monitoring by Liquid–Solid Triboelectric Nanogenerator Based on Tuning the Structure of the Electric Double Layer." *Advanced Functional Materials* (**2022**): 2200862.

663. Leal, Ana L., et al. "Data driven models exploring the combination of NIR and 1H NMR spectroscopies in the determination of gasoline properties." *Microchemical Journal* 175 (**2022**): 107217.

664. Liang, Haibo, and Gang Liu. "Research on quantitative analysis method of PLS hydrocarbon gas infrared spectroscopy based on net signal analysis and density peak clustering." *Measurement* 188 (**2022**): 110392.

665. Hassanzadeh, Amirhossein. *On the Use of Imaging Spectroscopy from Unmanned Aerial Systems (UAS) to Model Yield and Assess Growth Stages of a Broadacre Crop*. Diss. Rochester Institute of Technology, **2022**.

666. Rojas, Myriam, et al. "Kinetic Studies on Cocoa Roasting Including Volatile Characterization." *ACS Food Science & Technology* (**2022**).

667. Chen, Yi‐Hsuan, et al. "Green Polymer Electrolytes Based on Polycaprolactones for Solid‐State High‐Voltage Lithium Metal Batteries." *Macromolecular Rapid Communications* 43.20 (**2022**): 2200335.

668. Manasi, Iva, et al. "Surfactant effects on the synthesis of porous cerium oxide from a type IV deep eutectic solvent." *Journal of Materials Chemistry A* 10.35 (**2022**): 18422-18430.

669. Jander, Julius H., et al. "Determination of hydrogen loading in the carrier system diphenylmethane/dicyclohexylmethane by depolarized Raman spectroscopy." *International Journal of Hydrogen Energy* 47.15 (**2022**): 9331-9345.

670. Dunham-Cheatham, Sarrah M., Seth Lyman, and Mae Sexauer Gustin. "Comparison and calibration of methods for ambient reactive mercury quantification." *Science of The Total Environment* 856 (**2023**): 159219.

671. Martin, Jeremy E., et al. "The stability of dinosaur communities before the K− Pg boundary: A perspective from southern Alberta using calcium isotopes as a dietary proxy." *GSA Bulletin* (**2022**).

672. Scarpitti, Brian T., et al. "In Vitro Imaging of Lycopene Delivery to Prostate Cancer Cells." *Analytical Chemistry* 94.12 (**2022**): 5106-5112.

673. Schuster, Miriam. "Investigation of the semicrystalline structure of EVA and ionoplastic interlayers." *Characterization of Laminated Safety Glass Interlayers.* Springer Vieweg, Wiesbaden, **2023**. 117-162.

674. Cuss, Chad W., and Celine Gueguen. "The contribution of endmembers to mixtures of leaf leachates and riverine DOM can be determined by measuring their size and fluorescence properties." *Frontiers in Environmental Chemistry* (**2022**): 14.

675. Liu, Donghong, and Chuanjiang He. "Peak-aware guided filtering for spectrum signal denoising." *Chemometrics and Intelligent Laboratory Systems* 222 (**2022**): 104508.

676. Gupta, S., Singh, A., Sharma, A., & Tripathy, R. K. (**2022**). Higher Order Derivative-Based Integrated Model for Cuff-Less Blood Pressure Estimation and Stratification Using PPG Signals. *IEEE Sensors Journal*, 22(22), 22030-22039.

677. Hua, Zhen-Ming, et al. "Large hardening response mediated by room-temperature dynamic solute clustering

behavior in a dilute Mg-Zn-Ca-Sn-Mn alloy." *Acta Materialia* 240 (**2022**): 118308.

678. Bhalode, Pooja, et al. "Statistical Data Pre-Treatment for Residence Time Distribution Studies in Pharmaceutical Manufacturing." Available at SSRN 4249747 (**2022**).

679. Kukk, Anatoly Fedorov, Elias Blumenröther, and Bernhard Roth. "Self-made transparent optoacoustic detector for measurement of skin lesion thickness in vivo." *Biomedical Physics & Engineering Express* 8.3 (**2022**): 035029.

680. Wang, Yaheng, et al. "High-Speed 600 GHz-Band Terahertz Imaging Scanner System with Enhanced Focal Depth." *Photonics*. Vol. 9. No. 12. MDPI, **2022**.

681. Likhachev, D. V. "Optimization of the dielectric-function modeling by B-splines in spectroscopic ellipsometry analysis: A hybrid approach." *Thin Solid Films* 762 (**2022**): 139545.

682. Ai, Xupeng, et al. "Phase Segmentation and Percentage Prediction of Trunk Movement Cycle." *2022 9th IEEE RAS/EMBS International Conference for Biomedical Robotics and Biomechatronics* (BioRob). IEEE, **2022**.

683. Fan, Han, Erik Schaffernicht, and Achim J. Lilienthal. "Ensemble Learning-Based Approach for Gas Detection Using an Electronic Nose in Robotic Applications." *Frontiers in chemistry* 10 (**2022**).

684. Chuang, Chun-Hsiang, et al. "IC-U-Net: a U-Net-based denoising autoencoder using mixtures of independent components for automatic EEG artifact removal." *NeuroImage* 263 (**2022**): 119586.

685. Du, Mao, et al. "Combined application of online FIGAERO-CIMS and offline LC-Orbitrap mass spectrometry (MS) to characterize the chemical composition of secondary organic aerosol (SOA) in smog chamber studies." *Atmospheric Measurement Techniques* (**2022**): 4385-4406.

686. Schito, Andrea, et al. "Calibrating Carbonization Temperatures of Wood Fragments Embedded within Pyroclastic Density Currents through Raman Spectroscopy." *Minerals* 12.2 (**2022**): 203.

687. Sosa, Jonathan. Advanced Optical Diagnostics in Hypersonic Flows. NAVAL RESEARCH LAB WASHINGTON DC, **2022**.

688. Li, Chunyan. "Global shockwaves of the Hunga Tonga-Hunga Ha'apai volcano eruption measured at ground stations." *Iscience* 25.11 (**2022**): 105356.

689. Mekonnen, Alemu, et al. "Do improved biomass cookstoves reduce fuelwood consumption and carbon emissions? Evidence from a field experiment in rural Ethiopia." *Ecological Economics* 198 (**2022**): 107467.

690. da Silva, Igor JG, Ivo M. Raimundo, and Boris Mizaikoff. "Analysis of sugars and sweeteners via terahertz time-domain spectroscopy." *Analytical Methods* 14.27 (**2022**): 2657-2664.

691. Lee, Sea On. RNA-TECHNOLOGIES TO FACILITATE THE SYNTHESIS AND PROCESSING OF BIOMATERIALS WITH REPETITIVE AMINO ACID SEQUENCES. Diss. Johns Hopkins University, **2022**.

692. Pištek, Peter, Simon Harvan, and Michal Valicek. "Automated People Counting in Public Transport." *Industry 4.0 Challenges in Smart Cities* (**2022**): 75.

693. Ke, Jie, et al. "Self-Optimization of Continuous Flow Electrochemical Synthesis Using Fourier Transform Infrared Spectroscopy and Gas Chromatography." *Applied Spectroscopy* 76.1 (**2022**): 38-50.

694. Schultz, Jeremy F., et al. "Chemically imaging nanostructures formed by the covalent assembly of molecular building blocks on a surface with ultrahigh vacuum tip-enhanced Raman spectroscopy." *Journal of Physics: Condensed Matter* 34.20 (**2022**): 204008.

695. Valero, Maria, et al. "Vibration sensing-based human and infrastructure safety/health monitoring: A survey." *Digital Signal Processing* 125 (**2022**): 103572.

696. Oussalah, Abderrahim, David-Alexandre Trégouët, and Jean-Louis Guéant. "The Smoothing Method for DNA Methylome Analysis Identifies Highly Accurate Epigenomic Signatures in Epigenome-Wide Association Studies." (**2022**).

697. Chunyan Li, "Analysis of globally propagating spherical shockwaves from great volcano eruptions using barometric pressure data". STAR Protocols, Volume 4, Issue 2, **2023**, ISSN 2666 1667, https://doi.org/10.1016/j.xpro.2023.102207. (https://www.sciencedirect.com/science/article/pii/S266616672300165X)

698. Zhao, Ziwen, et al. "Automated Processing of Nano-Impact Electrochemistry Signals Using Data-Driven Template Matching." *Electrochemical Society Meeting Abstracts 244*. No. 57. The Electrochemical Society, Inc., **2023**.

699. Singh, Ritu, Navin Rajpal, and Rajesh Mehta. "Non-invasive Single Channel integration model for fetal ECG extraction and sustainable fetal healthcare using wavelet framework." *Multimedia Tools and Applications* 82.25 (**2023**): 39669-39695.

700. Kurapati, Hemasai. "How does the period of oscillation of a cantilever relate with its mass?." *Authorea Preprints* (**2023**).

701. Ma, Jielin, et al. "Effect of High Relative Humidity on the Thermal Aging of Composite Epoxy Insulation Materials." *2023 International Symposium on Electrical Insulating Materials (ISEIM)*. IEEE, **2023**.

702. Moss III, Frank R., et al. "Brominated lipid probes expose structural asymmetries in constricted membranes." *Nature Structural & Molecular Biology* 30.2 (**2023**): 167-175.

703. Mouton, J. W. A., et al. "Development and validation of a bioanalytical assay for the measurement of total and unbound teicoplanin in human serum." *Journal of Antimicrobial Chemotherapy* 78.11 (**2023**): 2723-2730.

704. Durán, Antonio Andrés Figueroa, and Efren Fernandez Grande. "Navigable Reconstruction of Reverberant Sound Fields Using Distributed Microphone Arrays." *2nd International Conference on Immersive and 3D Audio*. IEEE, **2023**.

705. Pace, DM Díaz. "τ-algorithm for gathering spectroscopic information by modeling emission line shapes: application to laser-induced plasmas." *JOSA B* 40.4 (**2023**): C1-C7.

706. Dummitt, Richard. *Chemical Effects in Protein Analysis: A Systematic Investigation of Amino Acid Spontaneous Raman and SERS Responses*. Diss. The Ohio State University, **2023**.

707. Zhong, Yu, et al. "Summary report of CALPHAD GLOBAL, 2021." *Calphad* 81 (**2023**): 102527.

708. Lueder, Mona, Renée Tamblyn, and Jörg Hermann. "A framework for quantitative in situ evaluation of coupled substitutions between H+ and trace elements in natural rutile." *European Journal of Mineralogy* 35.2 (**2023**): 243-265.

709. Yang, Jerry A., et al. "Biaxial Tensile Strain Enhances Electron Mobility of Monolayer Transition Metal Dichalcogenides." *arXiv e-prints* (**2023**): arXiv-2309.

710. Mahajan, Vishal, et al. "Treating Noise and Anomalies in Vehicle Trajectories From an Experiment With a Swarm of Drones." *IEEE Transactions on Intelligent Transportation Systems* (**2023**).

711. Figueroa-Duran, Antonio, and Efren Fernandez-Grande. "Navigable Reconstruction of Reverberant Sound Fields Using Distributed Microphone Arrays." *2023 Immersive and 3D Audio: from Architecture to Automotive (I3DA)*. IEEE, **2023**.

712. Duarte, Gabriela, et al. "Raman microspectroscopy as a tool for identifying biosignatures in speleothem microbialites." *Annals XVI Latin American Congress on Organic Geochemistry*. 2023.

713. "表面增強拉曼散射探針及微流道系統 用於癌症細胞及外泌體表面生物標記物的多重檢測" (Surface-enhanced Raman scattering probes and microfluidics systems are used for multiplex detection of biomarkers on the surface of cancer cells and exosomes), PhD diss., National Central University, **2023**.

714. Handlovic, Troy T., et al. "Automated Regularized Deconvolution for Eliminating Extra-Column Effects in Fast High-Efficiency Separations." *Analytical Chemistry* 95.29 (**2023**): 11028-11036.

715. Akinci, Tahir Cetin, et al. "High Order Spectral Analysis of Ferroresonance Phenomena in Electric Power Systems." *IEEE Access* (**2023**).

716. de Araújo Gomes, Adriano, et al. "Pattern recognition techniques in food quality and authenticity: A guide on how to process multivariate data in food analysis." *TrAC Trends in Analytical Chemistry* (**2023**): 117105.

717. Fekete, Szabolcs, and Davy Guillarme. "Ultra-short columns for the chromatographic analysis of large molecules." *Journal of Chromatography A* 1706 (**2023**): 464285.

718. Hollands, Patrick Mark. *Estimating the Effect of Pore Water and Ice on Martian Rock Analogues Using Ultrasonic and Resonant Ultrasound Spectroscopy Methods*. Diss. ResearchSpace@ Auckland, **2023**.

719. Liu, Chongshan, et al. "Microbiome-induced Increases and Decreases in Bone Tissue Strength can be Initiated After Skeletal Maturity." *bioRxiv* (**2024**): 2024-01.

720. Cuyt, Annie, and Wen-shin Lee. "Multiscale matrix pencils for separable reconstruction problems." *Numerical Algorithms* 95.1 (**2024**): 31-72.

721. Allan, Matthew C., et al. "Baked sweetpotato textures and sweetness: An investigation into relationships between physicochemical and cooked attributes." *Food Chemistry: X* 21 (**2024**): 101072.

722. Yang, Guofeng, et al. "An improved radioactive tracer response analysis and injection profile quantification method based on intelligent algorithms." *Journal of Radioanalytical and Nuclear Chemistry* (**2024**): 1-15.

723. Dong, Wen, et al. "Study of UV–Vis molar absorptivity variation and quantitation of anthocyanins using molar relative response factor." *Food Chemistry* (**2024**): 138653.

724. Chen, Huo, et al. "Adaptive variational simulation for open quantum systems." *Quantum* 8 (**2024**): 1252.

725. Yu, Lihuan, et. al., Derivative Spectroscopy and its Application at Detecting the Weak Emission/ Absorption Lines, *Research in Astronomy and Astrophysics* (**2024**).

726. R Senthilkuma, S.Shek Dhavud, RealTime AC Speech Denoising Analog Digital Filters, *National*

Conference on Emerging Trends and Technology, ISBN: 978-81959812-3-6, June **2023**

727. Matteo BruschiFederico GallinaFederico GallinaBarbara FreschBarbara Fresch, A Quantum Algorithm from Response Theory: Digital Quantum Simulation of Two-Dimensional Electronic Spectroscopy, *The Journal of Physical Chemistry Letters* 15(5):1484-1492. DOI: 10.1021/acs.jpclett.3c03499, January **2024**

728. Fatma R.M. Abda Rehmann lati1, Salah I.S. Tnatin. Effect the window type in design of FIR Filter to reduce the noise in the signal, *ICRSE 2021: The 1st International Conference on Renewable and Sustainable Energy*, VOLUME 5 No(1) October 10-13, **2021**

729. Krzysztof Sozanski, Overview of Signal Processing Problems in Power Electronic Control Circuits, *Energies* 16(12):4774, DOI: 10.3390/en16124774, June **2023**

730. Dhanarasi Gowtham B.T.Krishna, Design and Applications of Digital Differentiators Using Model Order Reduction Techniques, *Journal of Propulsion Technology*, October **2023**

731. Feng, Vincent. "Spectroscopy of Chamber Plasma and Plume of a 2.4-GHz Microwave Electrothermal Thruster." (**2023**).

732. Zhang, Le, et al. "Absolute thermometry of human brown adipose tissue by magnetic resonance with laser polarized 129Xe." *Communications Medicine* 3.1 (2023): 147.

733. Andreani, Muriel, et al. "The rocky road to organics needs drying." *Nature Communications* 14.1 (**2023**): 347.

734.Schorr, Hannah C., and Zachary D. Schultz. "Chemical conjugation to differentiate monosaccharides by Raman and surface enhanced Raman spectroscopy." *Analyst* 148.9 (**2023**): 2035-2044.

735. Chowdhury, Nildari Roy, et al. "Influence of state of charge window on the degradation of Tesla lithium-ion battery cells." *Journal of Energy Storage* 76 (**2024**): 110001.

736. Sanchez-Martinez, Silvia, et al. "Labile assembly of a tardigrade protein induces biostasis." *bioRxiv* (**2023**): 2023-06.

737. Lokini, Parneeth, et al. "Plasma Parameters of Laser Irradiated Hydrocarbon Droplets in Air." *AIAA SCITECH 2024 Forum*. **2024**.

738. Zheng, Shuailin, et al. "Rapid detection of phosphorus in water using silicon attenuated total reflectance infrared spectroscopy coupled with the algorithms of deconvolution and partial least squares regression." *Sensors and Actuators B: Chemical* 380 (**2023**): 133372.

739. Lee, Se Hun, et al. "Synthesis of conducting polymer intercalated sodium vanadate nanofiber composites as active materials for aqueous zinc-ion batteries and NH3 gas sensors at room temperature." *Composites Part B: Engineering* (**2024**): 111305.

740. Morder, Courtney, and Zachary D. Schultz. "A 3D printed sheath flow interface for surface enhanced Raman spectroscopy (SERS) detection in flow." *Analyst* (**2024**).

741. Deal, Alexandra M., et al. "Infrared Reflection–Absorption Spectroscopy of α-Keto Acids at the Air–Water Interface: Effects of Chain Length and Headgroup on Environmentally Relevant Surfactant Films." *The Journal of Physical Chemistry A* 127.18 (**2023**): 4137-4151.

742. Rehmann, Kelsi MS, John Klier, and Jessica D. Schiffman. "Anionic polymerization and transport of

diethyl methylidene malonate on polyolefin copolymer surfaces." *Polymer Chemistry* 14.32 (**2023**): 3695-3706.

743. Sargent, Alexander T., et al. "Reclamation and reuse of graphite from electric vehicle lithium-ion battery anodes via water delamination." *Journal of Materials Chemistry A* 11.17 (**2023**): 9579-9596.

744. Guilherme da Fonseca, Bruno. *Application of Raman Spectroscopy for the characterization of carbon materials*. Diss. **2023**.

745. Lima, Sofía, et al. "An allosteric switch ensures efficient unidirectional information transmission by the histidine kinase DesK from Bacillus subtilis." *Science Signaling* 16.769 (**2023**): eabo7588.

746. GODINHO, Madame Maria Helena, et al. "Bioinspired photonic cellulose films", **2021**. (PDF file link)

747. Chen, Huo, et al. "Adaptive variational simulation for open quantum systems." *Quantum* 8 (**2024**): 1252.

748. Piskors, Nico, et al. "High Protein—Low Viscosity? How to Tailor Rheological Properties of Fermented Concentrated Milk Products." *Dairy* 4.4 (**2023**): 594-605.

749. Boukra, Amine, et al. "Sampling terrigenous diffuse sources in watercourse: Influence of land use and hydrological conditions on dissolved organic matter characteristics." *Science of The Total Environment* 872 (**2023**): 162104.

750. Farmer, Jessica, and Adam J. Jackson. "A fast approximate method for variable-width broadening of spectra." *arXiv preprint arXiv:2309.12135* (**2023**).

751. Nieto Benito, Guillermo. "Predicción de series temporales mediante técnicas de aprendizaje automático. Automatización del proceso." (**2023**). PDF link.

752. Le, Duc, et al. "sCL-ST: "Supervised Contrastive Learning with Semantic Transformations for Multiple Lead ECG Arrhythmia Classification." *IEEE journal of biomedical and health informatics* (**2023**).

753. Mikheenkova, Anastasiia, et al. "Ageing of High Energy Density Automotive Li-ion Batteries: The Effect of Temperature and State-of-Charge." (**2023**).

754. Chaiamarit, Tai, et al. "Mutant Prion Protein Endoggresomes are Hubs for Local Axonal Organelle-Cytoskeletal Remodeling." *bioRxiv* (**2023**): 2023-03.

755. José María Martínez-Blanes Galo Romero-García Daniel Sánchez-Gómez, "Crafting illusions: Human-made composite coating used to simulate amber beads in prehistoric Iberia", *Journal of Archaeological Science*, June **2024**, DOI: 10.1016/j.jas.2024.106011

756. Ekram A GhozzyNahed M El-EnanyManar M TolbaSamah Abo El Abass, Eco-friendly simultaneous estimation of atenolol and losartan potassium in spiked human plasma via synchronous fluorescence with sustainability assessment, *Luminescence* 39(7):e4839, July **2024** DOI: 10.1002/bio.4839

757. Tao, Linmi, et al. "Applications of Tao General Difference in Discrete Domain." *arXiv preprint arXiv:2401.15287* (**2024**). 2401.15287.pdf (arxiv.org)

758. Lennart Reuter, et. al., Temperature-Dependent Gassing Analysis By on-Line Electrochemical Mass Spectrometry of Lithium-Ion Battery Cells with Commercial Electrolytes, ECS Meeting Abstracts, August **2023**, DOI: 10.1149/MA2023-012634mtgabs

759. Xinying Zhao and Jingqi Lu, Automated acquisition of spectral data and development of peak detection algorithms, Optoelectronic Devices and Integration XIII Conference, Nov **2024**, DOI: 10.1117/12.3037246

760. Thota, Niranjan, et al. "Lowering added sugar and increasing fibre in croissants using short-chain fructooligosaccharides (sc-FOS)." Food Hydrocolloids 150 (**2024**): 109761.

761. Li, Zekun, et al. "Classification of coal gangue and identification of coal type based on first-derivative of

mid-infrared spectrum." *Infrared Physics & Technology* 142 (**2024**): 105537.

762. Cobas, Carlos, et al. "A New qNMR Compliant Savitzky–Golay Apodization Function for Resolution Enhancement." *Magnetic Resonance in Chemistry* (**2024**).

763. Wu, Jie, et al. "Water-soluble near-infrared AgInS2 quantum dots for Ca2+ detection and bioimaging." *Spectrochimica Acta Part A: Molecular and Biomolecular Spectroscopy* 322 (**2024**): 124859.

764. Mokaribolhassan, Amin, et al. "Feeder-level PV estimation using signal processing techniques in DNs with limited measurements." *2024 IEEE Power & Energy Society General Meeting (PESGM)*. IEEE, **2024**.

765. Fröhlich, Markus, et al. "Colloidal 2D Mo 1− x W x S 2 nanosheets: an atomic-to ensemble-level spectroscopic study." Physical Chemistry Chemical Physics 26.17 (**2024**): 13271-13278.

766. He, Wenhe, et al. "A Novel Overlapping ME Peaks Decomposition Algorithm Based on Iterative Derivative Sharpening." *IEEE Transactions on Instrumentation and Measurement* (**2024**).

767. Noje, Dan-Cătălin, et al. "Image Processing Using Shepard Local Approximation Operators Defined in Riesz MV-Algebras." *2024 10th International Conference on Mechatronics and Robotics Engineering (ICMRE)*. IEEE, **2024**.

768. Handlovic, Troy T., et al. "Optimization of analytical method greenness scores: a case study of amino acid enantioseparations with carbonated aqueous systems." *Green Chemistry* 26.2 (**2024**): 760-770.

769. Wang, Yan-Ning, et al. "Efficient antibiotics, small organic molecules and inorganic anions detection with a fluorescent Ni (II) coordination polymer based multiple sensor system." *Journal of Molecular Structure* 1296 (**2024**): 136829.

770. Ismoilov, Muhriddin, et al. "Improving the quality of signals using an adaptive filter." *E3S Web of Conferences*. Vol. 525. EDP Sciences, **2024**.

771. Prameswari, Sonia Anjani, and Kusrini Kusrini. "Analisis Perbandingan Kinerja DWT dan SWT dalam Pengenalan Emosi Berbasis EEG Menggunakan XGBoost." *Kesatria: Jurnal Penerapan Sistem Informasi (Komputer dan Manajemen)* 5.4 (**2024**): 1569-1579.

772. Zhao, Xiangyu, et al. "Automatic peak detection algorithm based on continuous wavelet transform for complex chromatograms from multi-detector micro-scale gas chromatographs." *Journal of Chromatography A* 1714 (**2024**): 464582.

773. Liu, Shujuan, et al. "Resolving severely overlapping ion mobility peaks using enhanced Fourier self-deconvolution." *Analytical Methods* (**2025**).

774. Burk, Ryan Jacob, M. Farooq Wahab, and Daniel W. Armstrong. "Influence of theoretical and semi-empirical peak models on the efficiency calculation in chiral chromatography." *Talanta* 277 (**2024**): 126308.

775. Shang, Nianze, et al. "Evidence of abnormal hot carrier thermalization at van Hove singularity of twisted bilayer graphene." *Science Bulletin* (**2024**).

776. Milani, Nino BL, et al. "Generating realistic data through modeling and parametric probability for the numerical evaluation of data processing algorithms in two-dimensional chromatography." *Analytica Chimica Acta* 1312 (**2024**): 342724.

777. Chen, Chen, et al. "Knowledge-informed wheel wear prediction method for high-speed train using multisource signal data." *IEEE Transactions on Instrumentation and Measurement* 73 (**2024**): 1-12.

778. Chekin, Yaroslav, et al. "Time-Domain Diffuse Optical Tomography for Precision Neuroscience." *bioRxiv* (**2024**): 2024-04.

779. Liu, Chongshan, et al. "Microbiome-induced Increases and Decreases in Bone Tissue Strength can be Initiated After Skeletal Maturity." *bioRxiv* (**2024**).

780. Li, Xiaojuan, et al. "Change in water column total chlorophyll-a in the Mediterranean revealed by satellite

observation." *Science of The Total Environment* 945 (**2024**): 174076.

781. Teng, Jeffrey W., et al. "How to Consider SEEs When Designing a SiGe Low-Noise Amplifier-An Overview." *IEEE Transactions on Nuclear Science* (**2024**).

782. Ulbrich, Lisa. "Detection and elimination of strain reading anomalies in distributed strain sensing readings." *35. Forum Bauinformatik, fbi 2024*. Technische Universität Hamburg, Institut für Digitales und Autonomes Bauen, **2024**.

783. McRae, Helen M., et al. "Loss of PHF6 causes spontaneous seizures, enlarged brain ventricles and altered transcription in the cortex of a mouse model of the Börjeson–Forssman–Lehmann intellectual disability syndrome." *PLoS genetics* 20.10 (**2024**): e1011428.

784. Anbazhagan, P., and Sauvik Halder. "A novel algorithm for identifying arrival times of P and S Waves in seismic borehole surveys." *Computers & Geosciences* 194 (**2025**): 105746.

785. Dobal, Breno C. "Adaptation of a Commercially Available Galvanic Skin Response Sensor to Measure Respiration Across the Chest for Heart Rate Variability Monitoring." (**2024**).

786. Prosper, Arinaitwe, Murungi Emelta, Ogenyi Fabian Chukwudi, Asiimwe Robert, and Mohammed Dahiru Buhari. "Review of Techniques used in Speech Signal Processing." (**2024**).

787. Gies, Nils B., Pierre Lanari, and Jörg Hermann. "A workflow and software solution for spatially resolved spectroscopic and numerical data (SpecXY)." *Computers & Geosciences* (**2024**): 105626.

788. Chu, Meng-Yuan, et al. "Temporal Evolution of Vehicle Exhaust Plumes in a Congested Street Canyon Environment." *Environments* 11.3 (**2024**): 57.

789. Jayasooriya, Don Cyril Prathap Vishwanath. *An Instantaneous Frequency Based Approach to Estimate Heart Rate and Calculate Heart Rate Variability Metrics*. Diss. Virginia Tech, **2024**.

790. Jouskari, Eetu. "Characterization and application of black silicon drift detector to X-ray measurements." (**2024**).

791. Theurer, Thomas, et al. "A novel proxy for energy flux in multi-era wildfire reconstruction." *Scientific Reports* 14.1 (**2024**): 26409.

792. Gaines, Rachel N., et al. "Optimizing the Flow Electrooxidation of Glycerol Using Statistical Design of Experiments." *Journal of The Electrochemical Society* (**2024**).

793. Barua, Ayan. *Establishment and validation of a gas chromatography-mass spectrometry analytical method for targeted metabolomics: towards comprehensive profiling of metabolic signatures*. MS thesis. The University of Alabama in Huntsville, **2024**.

794. Li, Wang, et al. "High-Throughput Single-Molecule Surface-Enhanced Raman Spectroscopic Profiling of Single-Amino Acid Substitutions in Peptides by a Gold Plasmonic Nanopore." *ACS nano* 18.29 (**2024**): 19200-19207.

795. Loginova, Alexandra N., et al. "Qualitative variability of dissolved organic matter in the Baltic Sea sediments apparent from fluxes and optical properties." *Frontiers in Marine Science* 11 (**2024**): 1433199.

796. Yang, Guofeng, et al. "An Improved Radioactive Tracer Spectral Analysis and Injection Profile Quantification Method Based on Intelligent Algorithms in Complex Completions." *Available at SSRN 4735742*.

797. Zhang, Weijian, et al. "Isotopic Measurement of HTR-10 Irradiated Fuels Based on Anti-Compton Gamma Spectroscopic Method." *International Conference on Nuclear Engineering*. Vol. 88223. American Society of Mechanical Engineers, **2024**.

798. Fonseca, Jordan. *Engineering Tunable Strain in 2D Materials and Their Device Heterostructures*. Diss. University of Washington, **2024**.

799. Wang, Yu, et al. "Addressing Challenges of membrane Clogging in AF4-UV-ICPMS Analysis for Size

Determination of Trace Elements in Acidic, Organic-Rich Peat Bog Waters." *Analytical Chemistry* 96.37 (**2024**): 14953-14962.

800. Horne, Julie, et al. "Determination of Size, Size Distribution, and Concentration of Nanoparticles Using ICP-MS in the Context of SERS Substrates." *Journal of Raman Spectroscopy* (**2024**).

801. Asaf Bilgory, et. al., "Mid-infrared spectroscopy as a real-time diagnostic tool for chronic endometritis: A preliminary study", *Spectrochimica Acta Part A Molecular and Biomolecular Spectroscopy*, **2024** DOI: 10.1016/j.saa.2024.125670

802. M. Mwelango, et. al., "Spreading and Shrinking Effects of Coplanar Capacitive Sensors for Surface Defects", *Journal of Nondestructive Evaluation* 44(1), Feb **2025.** DOI: 10.1007/s10921-025-01160-6

803. Chochia, P. A. "On the Affinity of Image Smoothing and Segmentation Problems." *Journal of Communications Technology and Electronics* (**2025**): 1-11.

804. Figueroa-Duran, Antonio, and Efren Fernandez-Grande. "Reconstruction of reverberant sound fields over large spatial domains." *The Journal of the Acoustical Society of America* 157.1 (**2025**): 180-190.

805. Li, C., R. Rohli, W. Huang, J. Alejandro Kurczyn, Enhanced water transport when a tropical storm meets an atmospheric cold front, *Scientific Reports*. 2025, *15*(1), 6860. https://doi.org/10.1038/s41598-025-91097-7.

806. Li, C. Determining meteorological tidal transport through a channel on the coast. *Continental Shelf Research*, **2025**, Vol. 285: 105394. https://doi.org/10.1016/j.csr.2024.105394

807. Li, C**.** A. Sheremet, W. Huang, P. Dash, A. Katkar, M. N. Allahdadi, N. Chaichitehrani, C. M. Bachmann, V. H. Rivera-Monroy, "Dynamic impact of transiting weather systems on coastal currents in the northern Gulf of Mexico, Deep Sea Research Part II: Topical Studies in Oceanography", Volume 216, **2024**. https://doi.org/10.1016/j.dsr2.2024.105395.

808. Fabio Eliveny Rivadeneira-Bolaños, Sandra Esperanza Nope-Rodríguez, Martha Isabel Páez-Melo, "Hyperspectral analysis for automated quantification of total phosphorus in enriched soil samples", *Applied Optics*, Sept **2025**

809. Bertram Richter, Lisa Ulbrich, Max Herbers, Steffen Marx, "Experimental Vibration Analysis for Civil Engineering Structures", October **2025**, DOI: 10.1007/978-3-031-96110-6_58

810. Meaza Eyakem Gebreamlak, Meghna P Ayyar, Jenny Benois-Pineau, Jean-Pierre Salmon, "Leveraging Explainability methods in Spectral Domain for Data Augmentation and efficient training of CNN classifiers for Covid-19 Detection", 2023 Twelfth International Conference on Image Processing Theory, Tools and Applications (IPTA), October **2023**, DOI: 10.1109/IPTA59101.2023.10320061

811. B. A. Segall, T. C. Kennoy, J. C. Kokinakos, and J. D. Langhorn, "Hypersonic turbulent quantities in support of Morkovin's hypothesis", *Nature Communications*, Nov. 2025.