

Pragmatic Introduction to Signal Processing

Applications in scientific measurement

April 2019 edition

An illustrated essay with free software and spreadsheet templates to download

ISBN-13: 978-1792916595
ISBN-10: 1792916590

A retirement project by

Tom O'Haver

Professor Emeritus

Department of Chemistry and Biochemistry

University of Maryland at College Park

orcid.org/0000-0001-9045-1603

Online access

PDF format: <http://bit.ly/1TucWLF>

Web address: <http://bit.ly/1NLOILR>

Interactive 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. You can also click on the https Web addresses or on the names of downloadable software or animations in order to view or download that item.

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

Join our Facebook group: “Pragmatic Signal Processing”

Table of Contents

Introduction.....	10
Signal arithmetic	13
Signal arithmetic in SPECTRUM	15
Signal arithmetic in Spreadsheets	15
Signal arithmetic in Matlab.....	16
GNU Octave.....	20
Spreadsheet or Matlab/Octave?	20
Signals and noise.....	22
Ensemble averaging	25
Frequency distribution of random noise	26
Dependence on signal amplitude	27
Probability distribution of random noise	28
Spreadsheets.....	30
Matlab and Octave	31
The difference between scripts and functions	32
User-defined functions related to signals and noise.....	33
Smoothing	35
Smoothing algorithms	35
Noise reduction	37
Effect of frequency distribution of noise	37
End effects and the lost points problem	38
Examples of smoothing.....	38
The problem with smoothing	39
Optimization of smoothing	41
When should you smooth a signal?.....	42
When should you NOT smooth a signal?	43
Dealing with spikes and outliers.	43
Ensemble Averaging.....	44
Condensing oversampled signals	44
Smoothing in spreadsheets.....	45
Smoothing in Matlab and Octave.....	47
Smoothing performance comparison	51
Differentiation.....	53
Basic Properties of Derivative Signals.....	54

Applications of Differentiation	56
Peak detection	57
Derivative Spectroscopy	58
Trace Analysis	60
Derivatives and Noise: The Importance of Smoothing.....	62
Differentiation in Spreadsheets	65
Differentiation in Matlab and Octave	65
Peak Sharpening.....	69
Even derivative sharpening.....	69
First derivative symmetrization	71
The Power Law Method.....	73
Peak Sharpening for Excel and Calc Spreadsheets.....	75
<i>The spreadsheet.....</i>	76
Peak Sharpening for Matlab and Octave.....	76
Harmonic analysis and the Fourier Transform.....	80
Software details.....	89
Matlab and Octave	89
Observing Frequency Spectra with iSignal.....	91
Fourier Convolution	93
Simple whole-number convolution vectors	94
Software details for convolution	95
Fourier Deconvolution	96
Computer software for deconvolution	100
Matlab and Octave	100
Fourier filter	107
A simple example.....	107
Computer software for Fourier Filtering.....	108
Integration and peak area measurement.....	109
Dealing with overlapping peaks.....	111
Peak area measurement using spreadsheets.	112
Using sharpening for overlapping peak area measurements.....	113
Peak area measurement using Matlab and Octave	114
Area measurement by iterative curve fitting.....	117
Correction for background/baseline.....	118
Asymmetrical peaks and peak broadening: perpendicular drop vs curve fitting.....	120

Curve fitting A: Linear Least Squares.....	126
Examples of polynomial fits	126
Reliability of curve fitting results	131
Algebraic Propagation of errors	132
Monte Carlo simulation	133
The Bootstrap method.....	134
Comparison of error prediction methods.	135
Effect of the number of data points on least-squares fit precision.....	136
Transforming non-linear relationships.....	137
Fitting Gaussian and Lorentzian peaks	138
Math and software details	142
Spreadsheets.....	143
Application to analytical calibration and measurement.....	145
Matlab and Octave	145
Fitting Single Gaussian and Lorentzian peaks	150
Curve fitting B: Multicomponent Spectroscopy	152
Classical Least Squares (CLS) calibration.....	152
Inverse Least Squares (ILS) calibration.....	154
Computer software for multiwavelength spectroscopy	156
Spreadsheets.....	156
Matlab and Octave	158
Curve fitting C: Non-linear Iterative Curve Fitting	163
Spreadsheets and stand-alone programs	164
Matlab and Octave	167
Fitting peaks	167
Peak Fitters for Matlab and Octave.....	171
Accuracy and precision of peak parameters.....	174
a. Model errors.	174
b. Background correction.....	182
c. Random noise in the signal.	185
d. Iterative fitting errors	188
Fitting signals that are subject to exponential broadening.....	192
The Effect of Smoothing before least-squares analysis	197
Peak Finding and Measurement.....	198
Simple peak detection	199

Gaussian peak measurement	200
Optimization of peak finding	202
Finding valleys	204
Accuracy of the measurements of peaks.....	204
The accuracy of the measurements	204
Comparison of peak finding functions.....	208
Using the peak table	213
Demo scripts	214
Peak Identification	215
<i>iPeak</i> , for Matlab.....	216
<i>iPeak</i> keyboard Controls (version 7.9):	229
<i>iPeak</i> Demo functions	230
Spreadsheet Peak Finders.....	234
Hyperlinear Quantitative Absorption Spectrophotometry	237
Background	238
Spreadsheet implementation	242
Matlab/Octave implementation: The fitM.m function.....	243
Demo function for Octave or Matlab	245
Interactive demo for the Tfit method for Matlab, version 2.1	246
Statistics of methods compared (TFitStats.m, for Matlab or Octave)	247
Comparison of analytical curves (TFitCalDemo.m, for Matlab or Octave)	248
Application to a three-component mixture	250
Tutorials, Case Studies and Simulations.....	252
A. Can smoothed noise may be mistaken for an actual signal?	252
B. Signal or Noise?	252
C. Buried treasure	256
D. The Battle Rounds: a comparison of methods	259
E: Ensemble averaging patterns in a continuous signal.....	262
F: Harmonic Analysis of the Doppler Effect.....	264
G: Measuring spikes.....	266
H: Fourier deconvolution vs curve fitting (they are <i>not</i> the same)	268
I: Digitization noise - can adding noise really help?	270
J: How Low can you Go? Performance with very low signal-to-noise ratios.	272
K: Signal processing in the search for extraterrestrial intelligence.....	274
L: Why measure peak area rather than peak height?	276

N: Using macros to extend the capability of spreadsheets.....	278
O: Random walks and baseline correction.....	280
P. Modulation and synchronous detection.....	282
Q: Measuring a buried peak	285
R. Signal and Noise in the Stock Market	288
S. Measuring signal-to-noise ratio in complex signals	292
T. Dealing with wide ranging signals: segmented processing	295
U. Measurement Calibration	298
V. Numerical precision of computer software.....	302
W: Miniaturized signal processing: The Raspberry Pi.....	304
X: Batch processing	306
Y: Real-time signal processing.....	308
Z. Dealing with variable data arrays in spreadsheets.....	313
AA. Computer simulation of signals and instruments.	316
AB. Who uses this book and its associated web site, documents and software?	318
AC. The Law of Large Numbers.....	321
Signal processing software details	323
Interactive smoothing, differentiation, and signal analysis (iSignal).....	323
<i>iSignal</i> keyboard controls (Version 6.1):.....	337
<i>iFilter</i> , keyboard-operated interactive Fourier filter	339
Matlab/Octave Command-line Peak Fitters	343
Matlab/Octave command-line function: peakfit.m	343
Examples	348
How do you find the right input arguments for peakfit?.....	359
Working with the fitting results matrix "FitResults".....	359
Demonstration script for peakfit.m	359
Automatically finding and Fitting Peaks	360
Interactive Peak Fitter (ipf.m)	362
<i>ipf</i> keyboard controls (Version 13.2):.....	364
Practical examples with experimental data:	366
Operating instructions for ipf.m (version 13.2).	368
Demoipf.m	375
Execution time of peak fitting and other signal processing tasks	376
Curve Fitting Hints and Tips	377
Extracting the equations for the best-fit models	379

How to add a new peak shape to <i>peakfit.m</i> or <i>ipf.m</i>	380
Which to use? <i>peakfit</i> , <i>ipf</i> , <i>findpeaks</i> ..., <i>iPeak</i> , or <i>iSignal</i> ?	381
S.P.E.C.T.R.U.M.: Simple freeware signal processing program for Macintosh OS 8.1	383
Features	383
Machine Requirements	384
Download links	384
Worksheets for Analytical Calibration Curves.....	386
Background	386
Fill-in-the-blanks worksheets for several different calibration methods	386
Comparison of calibration methods	391
Instructions for using the calibration templates	391
Frequently Asked Questions (taken from emails and search engine queries).....	394
Catalog of signal processing functions, scripts and spreadsheet templates	400
Peak shape functions (for Matlab and Octave)	400
Signal Arithmetic	401
Signals and Noise.....	403
Smoothing	405
Differentiation and peak sharpening	407
Harmonic Analysis.....	409
Fourier convolution and deconvolution	410
Fourier Filter	411
Peak area measurement.....	412
Linear Least Squares	413
Peak Finding and Measurement.....	415
Multicomponent Spectroscopy	421
Non-linear iterative curve fitting and peak fitting	422
Keystroke-operated <i>interactive</i> signal processing tools (for Matlab only)	426
Hyperlinear Quantitative Absorption Spectrophotometry	427
MAT files (for Matlab and Octave) and Text files (.txt)	427
Spreadsheets (for Excel or OpenOffice Calc).....	428
Afterword	432
How this book came to be.....	432
Who needs this software?	432
Organization.....	432
Methodology	433

Influence of the Internet.....	433
Writing	433
Software platform selection criteria.....	434
Outcomes	435
References.....	436
Publications that cite the use of these programs and documentation.....	440

Introduction

The interfacing of measurement instrumentation to small computers for the purpose of online data acquisition has now become standard practice in the modern laboratory for the purposes of performing signal processing and data analysis and storage, using a large number of digital computer-based numerical methods that are used to transform signals into more useful forms, detect and measure peaks, reduce noise, improve the resolution of over-lapping peaks, compensate for instrumental artifacts, test hypotheses, optimize measurement strategies, diagnose measurement difficulties, 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 to appreciate the abilities, as well as the limitations, of these techniques. But in recent decades, computer storage and digital processing has 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. It's 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, supermagnets, holograms, quantum technology, nanotechnology, and more. Sensors are now smaller, cheaper, and faster than ever before; we can measure over a wider range of speeds, temperatures, pressures, and locations. There are new kinds of data that we never had before. As Erik Brynjolfsson and Andrew McAfee wrote in *The Second Machine Age* (W. W. Norton, 2014): "...many types of raw data are getting dramatically cheaper, and as data get cheaper, the bottleneck increasingly is the ability to interpret and use data".

This essay covers only basic topics related to one-dimensional time-series signals, not two-dimensional data such as images. It uses a pragmatic approach and is limited to mathematics only up to the most elementary aspects of calculus, statistics, and matrix math. For the math phobic, you should know that this essay does not dwell on the math and that it contains more than twice as many figures as equations. Data processing without math? Not really! Math is essential, just as it is for the technology of cell phones, GPS, digital photography, the Web, and computer games. However, you can get started using these tools without understanding all the underlying math and software details. Seeing it work makes it more likely that you'll want to understand how it works. Nevertheless, in the end, it's 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.

Why do I title this document "signal processing" rather than "data processing"? By "signal" I mean the continuous x,y numerical data recorded by scientific instruments as time-series, where x may be time or another quantity like energy or wavelength, as in the various forms of spectroscopy. "Data" is a more general term that includes categorical data as well. 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 come from my own areas of research in analytical chemistry, but these techniques have been used in a wide range of application areas. Over 360 journal papers, theses, and patents have cited my software, covering fields from academia, industry, environmental, medical,

engineering, earth science, space, military, financial, agriculture, communications, and even music and speech science (page 440). Suggestions and experimental data sent by hundreds of readers from their own work has helped shape my writing and software development. Much effort has gone into making this document concise and understandable; it has been [highly praised by many readers](#).

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

This book had its origin in one of the experiments in a course called "Electronics and Computer Interfacing for Chemists" that I developed and taught at the University of Maryland in the 80's and 90's. The first Web-based version went up in 1995. Subsequently I have revised and greatly expanded it based on feedback from users. It is still a work in progress and, as such, will always benefit from feedback from readers and users.

This tutorial 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 (references 14, 17, 19, 20 on page 386), and Octave, a free Matlab alternative that runs almost all of the programs and examples in this tutorial (page 19). There is a good reason why Matlab is so massively popular in science and engineering; it's powerful, fast, and relatively easy to learn; it *comes with built-in functions for doing data processing tasks* like matrix math, filtering, Fourier transforms, convolution and deconvolution, multilinear regression, and optimization; you can download powerful toolboxes and free user-contributed functions; it can interface to C, C++, Java, Fortran, and Python; and it's extensible to symbolic computing and model-based design for dynamic and embedded systems. There are many code examples in this text that you can Copy and Paste (or drag and drop) into the Matlab/Octave command line to run or to modify, which is especially convenient if you can split your screen between Matlab/Octave and the website or PDF file.

Some of the illustrations were produced on my old 90s-era freeware signal-processing application for Macintosh OS8, called S.P.E.C.T.R.U.M. (Signal Processing for Experimental Chemistry Teaching and Research / University of Maryland). See page 383.

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

You can download all of the Matlab/Octave scripts and functions, the SPECTRUM program, and the spreadsheets used here 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 path, and try again.

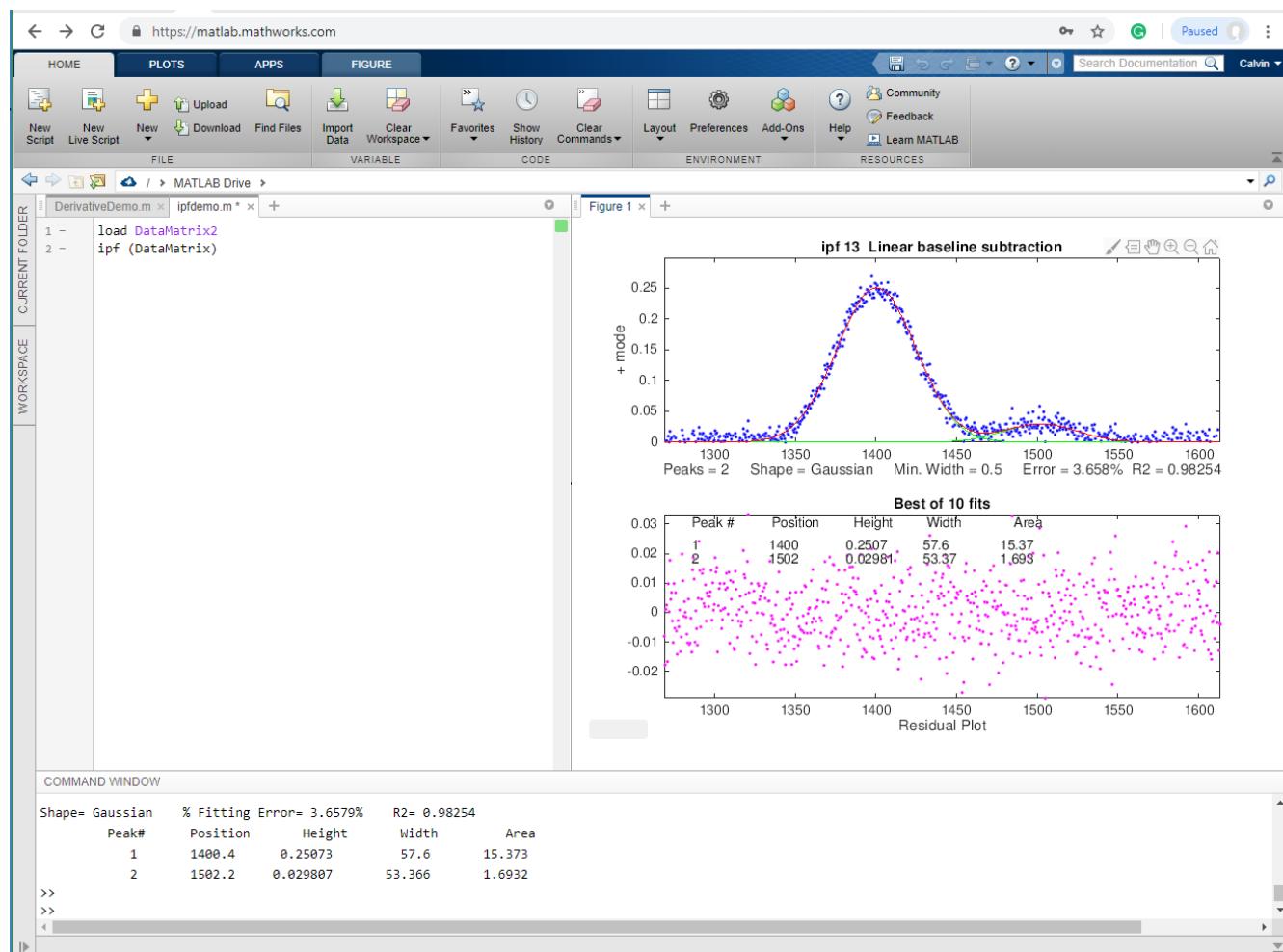
If you don't know Matlab or Octave, read page 16 and following for a quick start-up. These are not general-purpose programming languages, like C++ or Python; rather, they are specifically suited to

matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages - essentially the needs of numerical computing by scientists and engineers. Matlab and Octave are more loosely typed and are less well-structured in a formal sense than other languages, and they tend to be more favored by scientists and engineers and less well liked by computer scientists and professional programmers. To get a basic language like Python up to where Matlab *starts* requires the installation of many add-on “packages”.

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.

There are also other alternatives to MATLAB, in particular 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>.

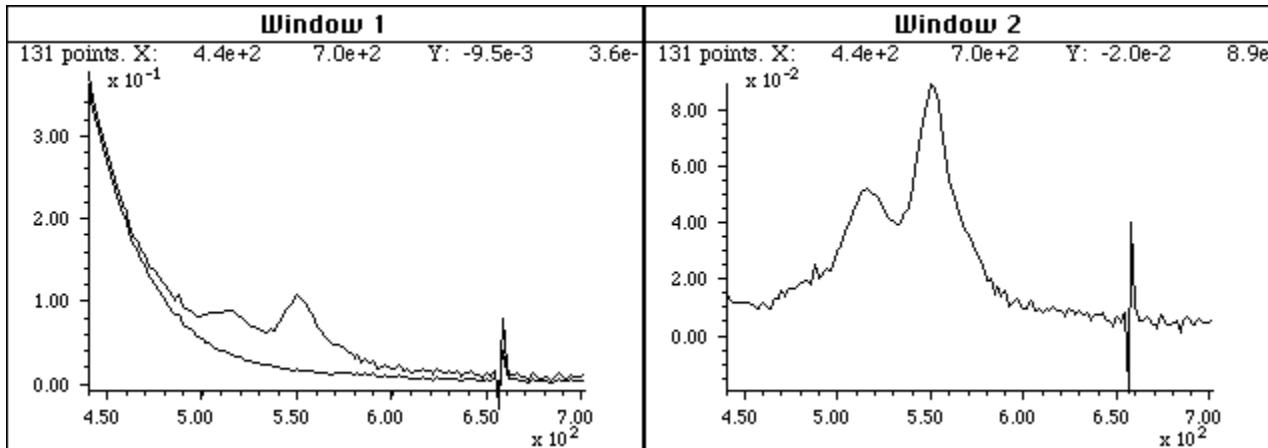
If you are reading this book online, on an Internet connected computer, you can Ctrl-Click on any of the http Web addresses or on the names of downloadable software or animations in order to view or download that item. For a complete list of all my software, see page 399 or <http://tinyurl.com/cey8rwh>.



Matlab Online running my “interactive peak fitter” (ipf.m) in the Chrome browser on a Windows PC

Signal arithmetic

The most basic signal processing functions 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 functions can be very useful. For example, in the left part of 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.



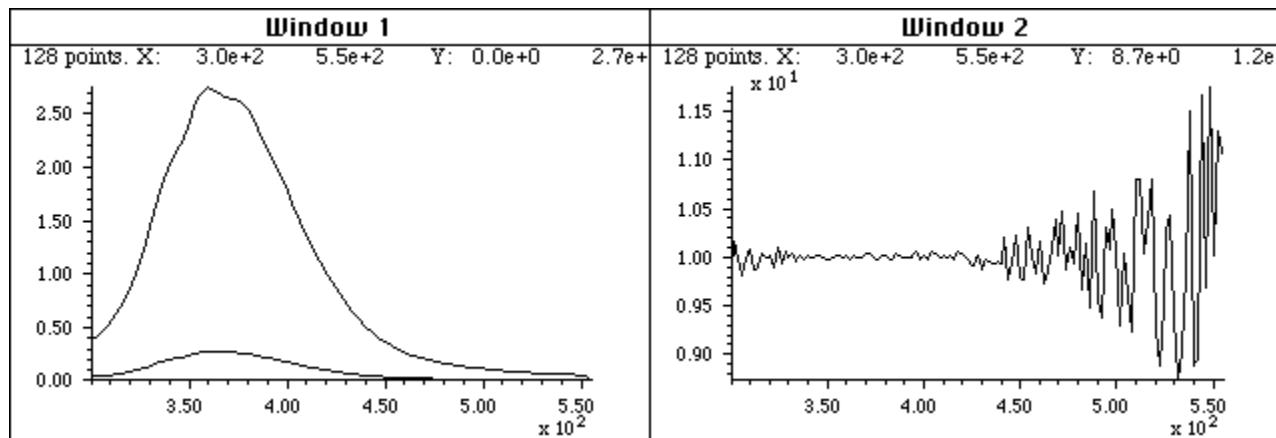
A simple point-by-point subtraction of two signals allows the background (bottom curve on the left) to be subtracted from a complex sample (top curve on the left), resulting in a clearer picture of what is really in the sample (right). (X-axis = wavelength in nm; Y-axis = absorbance).

This optical spectrum exhibits two absorption bands, at about 515 nm and 550 nm, that are due to a class of molecular fossils of chlorophyll called *porphyrins*. (Porphyrins 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 extracted from 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 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 that you can see the two bands more clearly and it is easier to measure precisely their absorbances and wavelengths. (Thanks to Prof. David Freeman for the spectra of oil shale 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 number of points or the x-axis intervals of signals; the results are only

approximate but often close enough in practice. (Interpolation is one of the functions of my *iSignal Matlab* function described on page 323).

Sometimes one needs to know whether two signals have the same shape, for example in 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 a 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? The ratio of the two signals, shown in the right part (Window 2), 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 $\pm 2\%$, over this x-axis range, and that top curve is very nearly 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 21).

A **division by zero error** will be caused by *even a single zero* in the denominator vector, but that can usually be avoided by applying a small amount of smoothing (page 34) of the denominator, by adding a small positive number, or by using the Matlab/Octave function [rmz.m](#) (*remove zeros*) 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 Web site and a smartphone app that is a computational tool and information source, including capabilities for mathematics, plotting, vector and matrix manipulations, statistics and data analysis, and many other topics. Statpages.org can perform a huge range of statistical calculations and tests. There are several Web sites that specialize in plotting data, including [Tableau](#), [Plotly](#), [Grapher](#), and [Plotter](#). 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 SPECTRUM

SPECTRUM is a 90s era freeware signal-processing application for Macintosh OS8 that includes the following signal arithmetic functions: addition and multiplication with constant; addition, subtraction, multiplication, and division of two signals, normalization, and a large number of other basic functions (common and natural log and antilog, reciprocal, square root, absolute value, standard deviation, etc.) in the **Math** menu.

It also performs many other signal processing functions described in this paper, including smoothing, differentiation, peak sharpening, interpolation, fused peak area measurement, Fourier transformation, Fourier convolution and deconvolution, and polynomial curve fitting. It runs only in Macintosh OS 8.1 and earlier and can be made to work on Windows PCs and various

specific Linux distributions using the Executor emulator. No native PC version is available or planned.

Signal arithmetic in Spreadsheets



Popular **spreadsheets**, such as *Excel* or *Open Office Calc*, have built-in functions for all common math operations, named variables, x,y plotting, text formatting, matrix math, etc. Cells can contain numerical values, text, mathematical expression, 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

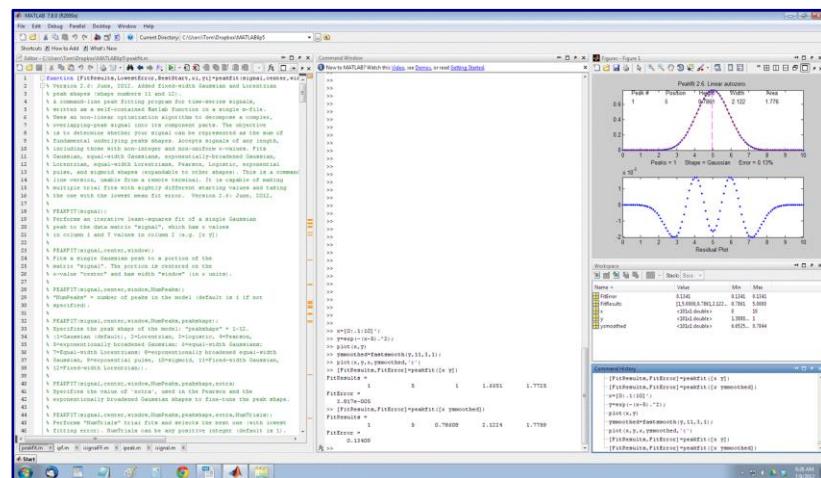
ranges of cells, and then refer to them in mathematical expression by name. You can copy mathematical expressions across a range of cells, with the cell references changing or not as desired. You can make plots of various types (including the all-important x-y or *scatter* graph) by menu selection. For 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 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* (*Excel 2013*) and *OpenOffice Calc* (3.4.1) can open and save either spreadsheet file format (.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 functions are interpreted, and for that reason I supply most of my spreadsheets in .xls (for *Excel*) and in .ods (for *Calc*) formats. See "Differences between the OpenDocument Spreadsheet (.ods) format and the Excel (.xlsx) format". Basically, *Calc* and do most 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 for some operations is faster than *Calc*. If you have access to *Excel*, I would recommend using that.

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 can do basic tasks but do not have the fancier capabilities of the desktop 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 computers, making them useful for field data entry.

Signal arithmetic in Matlab

Matlab is a "multi-paradigm numerical computing environment and fourth-generation programming language" (Wikipedia). In Matlab (and in Octave, its GNU clone), 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



axis for comparison by typing ". (Matlab automatically assigns different colors to each line, but you can control the color and line style yourself 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 **pth** section of a **m**-by-**n** grid of plots. (In the PDF version of this book, 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). Type "help plot" or "help subplot" for more plotting options.

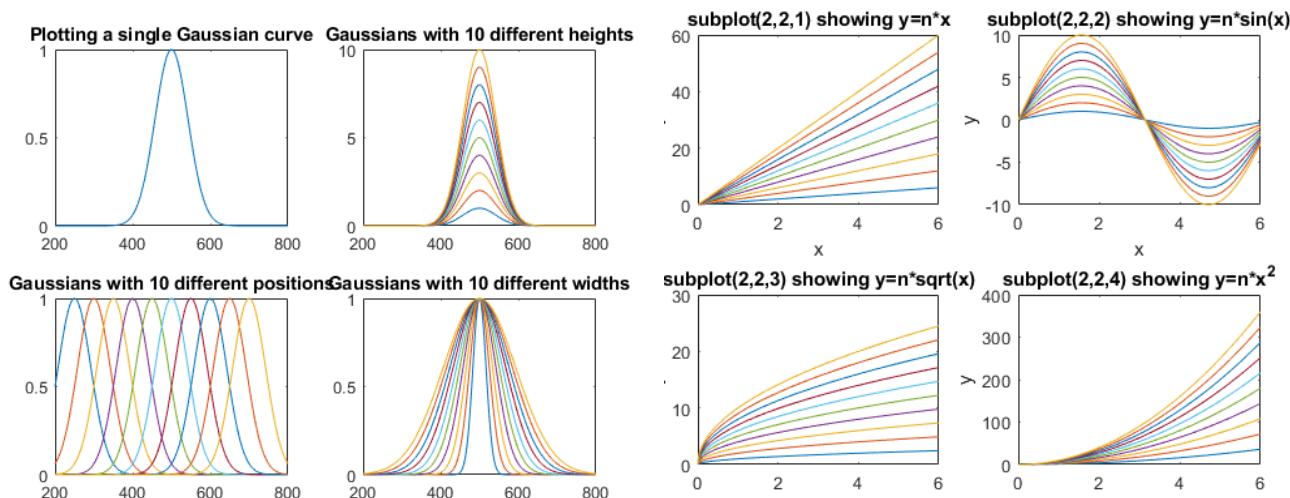
To create *publication-quality* plots within Matlab, there are many choices. For example, PlotPub is a downloadable library that is free, easy to use, and creates good looking graphs that can be exported in

operations and functions adjust to match. This greatly facilitates mathematical operations on signal waveforms. For example, 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

EPS, PDF, JPEG, PNG and TIFF with adjustable resolution. For a simple example, [click here](#).

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 the downloadable 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/Octave scripts [plotting.m](#) (left) and [plotting2.m](#) (right) show how to plot multiple signals using matrices and subplots.



The subtraction of two signals **a** and **b**, as on page 13, can be performed simply by writing **a-b**. To plot the difference, you would write "plot (a-b)". Likewise, to plot the ratio of two signals, as on page 14, you would write "plot (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, but more compact and faster than, writing a "for" loop like this:

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

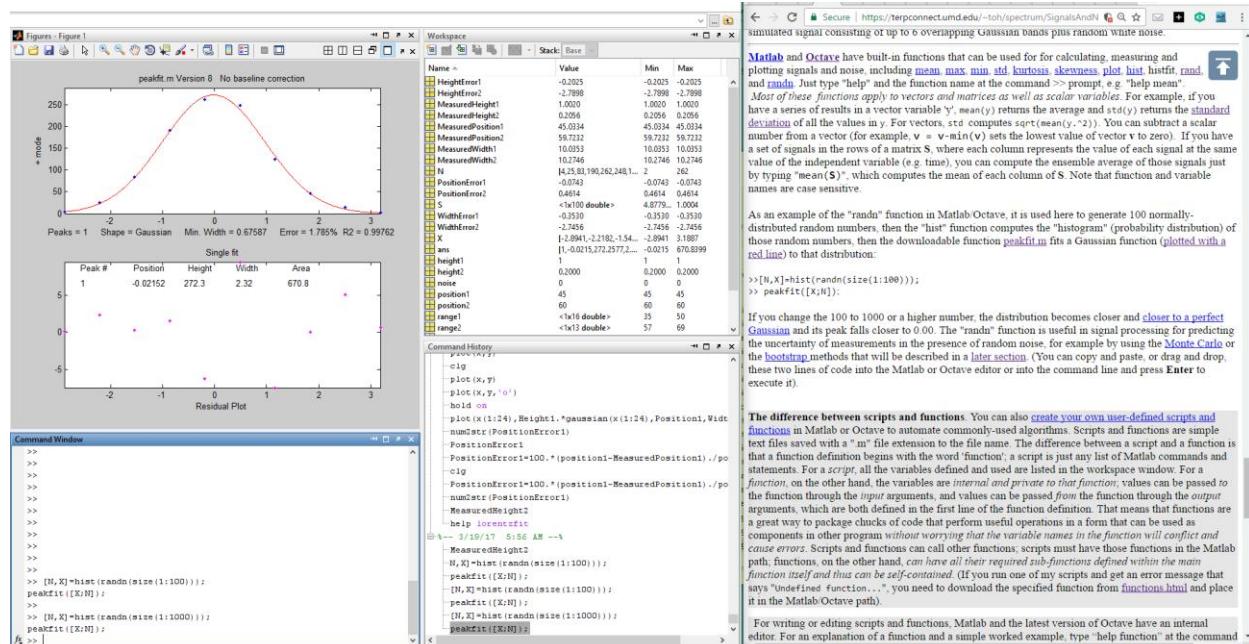
See [TimeTrial.txt](#) for details. It will help if you pre-allocate memory space for the **A** matrix by adding

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 right divide", in effect the weighted average ratio of the amplitudes of the two vectors (a type of least-squares best-fit solution), which in the example on page 11 will be a number close to 10. The point here is that *Matlab doesn't 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 your calculations accordingly. See http://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). Here's 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*.

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 here, or, even better, if you have *two* monitors hooked to your



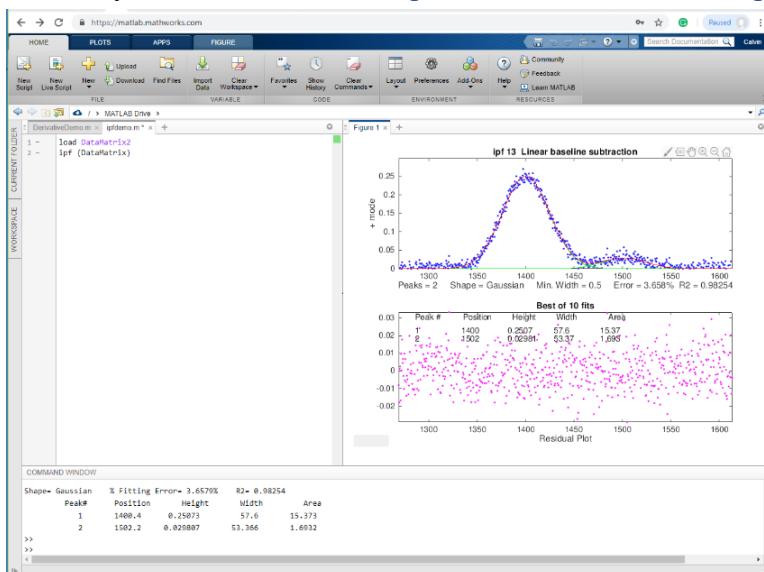
computer configured to expand the desktop horizontally. If you try to run one of my scripts or functions and it gives you a "missing function" error, look for the missing item from <http://tinyurl.com/cev8rwh>.

download it into your path, and try again

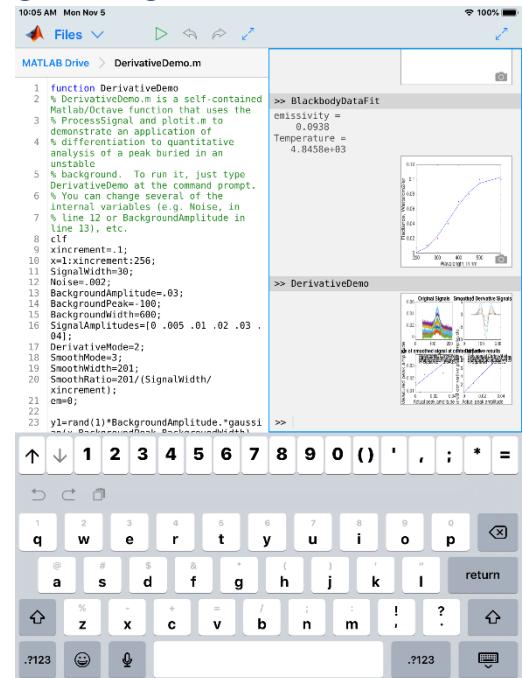
One thing that you will notice about Matlab is that the *very first time* you execute a script or function; there is a small delay before execution, while Matlab compiles the code into binary machine language. However, that only happens the *first* time; after that, the execution starts instantly. For the fastest execution, the separately available “Matlab Compiler” lets you share programs as *standalone* 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. For *real-time* plotting in Matlab/Octave, see page 308.

Getting data into Matlab/Octave. You can easily import your own data into Matlab or Octave by using the load command. You can import data from plain text files, comma-separated values (CSV), several image and sound formats, or spreadsheets. Matlab has a convenient Import Wizard (click **File > Import Data**). It is also possible to import 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 , as in *DataTheif* (sic) or *Figure Digitizer*. Matlab R2013a or newer can even read the sensors on your iPhone or Android phone via Wi-Fi. To read the outputs of older analog instruments, you need an analog-to-digital converter, an Arduino microcontroller board, or a USB voltmeter.

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 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 web browser ([below, left](#)); and the free [Matlab Mobile](#) app that runs Matlab on iPads and iPhones ([below, right](#)). This requires only a basic student license and uses functions, scripts, and data that you have previously uploaded to your account on the [Matlab cloud](#). All these versions have computational speeds within roughly a factor of 2 of each other, as shown by [TimeTrial.txt](#). See <https://www.mathworks.com/pricing-licensing.html>.

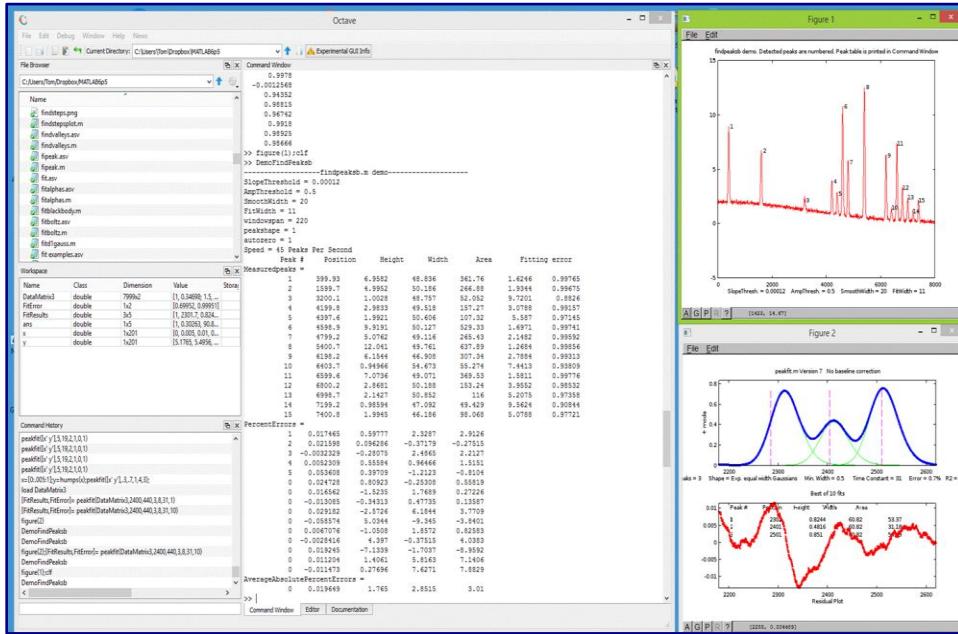


Left: *Matlab Online* running in a Web browser window.
Right: The free *Matlab Mobile* app running on an iPad.



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 of my Matlab functions, scripts, demos, and examples in this document will work in the latest version of Octave without change*. The exceptions are the keystroke-operated interactive *iPeak* (page 216), *iSignal* (page 323), and *ipf.m* (page 361). If you plan to use Octave, make sure you get the current versions; many of them were updated for Octave compatibility in 2015 and this is an ongoing project. There is a [FAQ](#) that may help in porting Matlab programs to Octave. See "[Key Differences Between Octave & Matlab](#)". There are Windows, Mac, and Unix versions of Octave. The Windows version can be downloaded from Octave Forge; be sure to install all the "packages". 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 4.2.1 has been released and is now available for [download](#). Installation of Octave is somewhat more laborious than installing a commercial package like Matlab. More seriously, Octave is also computationally 2 to 20 times slower than any of the Matlab versions, depending on the task (specific comparisons are in [TimeTrial.txt](#)). In addition, the error handling is less graceful and it's less stable and tends to crash more often (in my experience). I am working to make all my Matlab scripts and functions compatible with Octave; make sure you have the latest versions of my functions (a number of functions were made more Octave-compatible in March 2015). Bottom line: Matlab is better, but if you can't afford Matlab, Octave provides most of the functionality for 0% of the cost.

Spreadsheet or Matlab/Octave?

For signal processing, Matlab/Octave is faster and more powerful than using a spreadsheet, but it's safe to say that spreadsheets are more commonly installed on science workers' computers than Matlab or

Octave. For one thing, spreadsheets are easier to get started with and they offer flexible presentation and user interface design. Spreadsheets are better for data entry; you can easily deploy them on portable devices such as smartphones and tablets (e.g. using *iCloud Numbers* or the Excel app). *Spreadsheets are concrete and more low-level, showing every single value explicitly in a cell.* In contrast, *Matlab/ Octave* is more high level and abstract, because a single variable, punctuation, or function can do so much. An advantage of Matlab and Octave is that their function and script files (“m-files”) are just plain text files with a “.m” extension, so *those files can be opened and inspected using any text editor even on devices that do not have Matlab or Octave installed*, and would facilitate 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 can easily read Excel .xls and .xlsx files and import the rows and columns into 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 ("m-files" that begin with "function...") are the "chips", which condense complex operations into one package with documented input and output pins (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). These days, almost all electronics is done with chips, because it's easier to understand the relatively small number of inputs and outputs of a chip than to deal with the greater number of internal components.

The bottom line is that spreadsheets are easier at first, but, in my experience, eventually the Matlab/Octave approach is more productive. This point is demonstrated by the comparison of both approaches to multilinear regression in multicomponent spectroscopy covered on page 152([RegressionDemo.xls](#) vs the Matlab/Octave [CLS.m](#)). Even more dramatic is the difference between the spreadsheet and Matlab/Octave approaches to finding and measuring peaks in signals covered in the section beginning on page 198 (i.e. a 250Kbyte spreadsheet vs a 7Kbyte script that's 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 is a great way to go. It's much easier to 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 306.

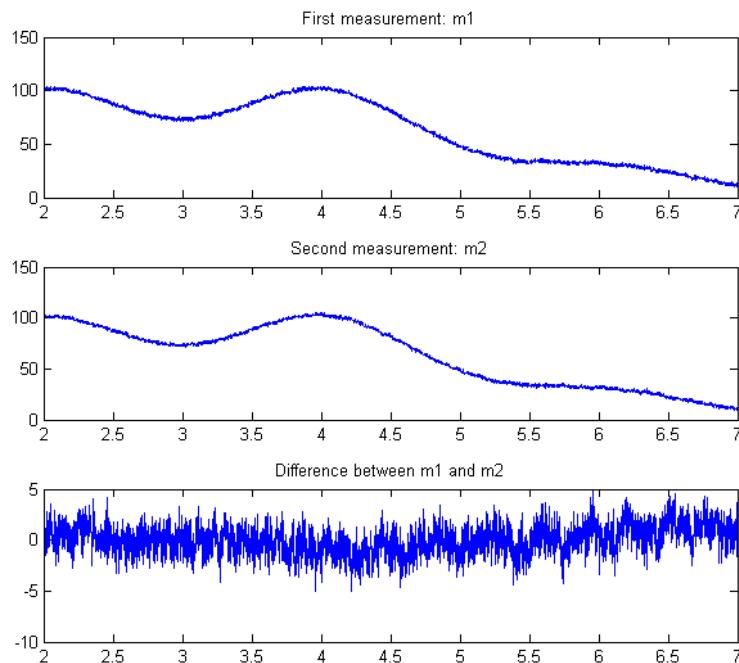
Both spreadsheets and Matlab/Octave programs have a huge advantage over commercial end-user programs and compiled freeware programs such as SPECTRUM (page 383); you can *inspect and modify them* to customize the routines for specific needs. Simple changes are easy to make with little or no knowledge of programming. For example, you could easily change the labels, titles, colors, or line style of the graphs - in Matlab or Octave programs, search for "title()", "label()" or "plot()". My code contains *comments that indicate places where you can make specific changes*: just use **Find...** to search for the word "change". *I invite you to modify my scripts and functions as you wish.* The [software license](#) imbedded within the comments of all 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, "cosmic rays" from outer space (seriously), the basic quantum nature of matter and energy itself, and digitization noise (the rounding of numbers to a fixed number of digits). 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 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 "d": `sum(d) ./ length(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, 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, this is most easily calculated by the built-in function `std(d)`. 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, as demonstrated by the series of Matlab/Octave commands [at this link](#). Try it.

The term "signal" actually has two meanings: in the more general sense, it can mean the *entire* data recording, including the noise and other artifacts, as in the "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 when you say "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 period 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 13, 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?

If you have a sample and an instrument that are 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 sample*. Then the standard deviation of the noise in the original signals is given by $\text{sqrt}((\text{std}(m1-m2)^2)/2)$, where “sqrt” is the square root and “std” is the standard deviation. This is shown by the simple derivation in the text file located at <https://terpconnect.umd.edu/~toh/spectrum/Derivation.txt>, which is based on the rules for mathematical error propagation. The Matlab/Octave script “SubtractTwoMeasurements.m” (below) demonstrates this process quantitatively and graphically.



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 previous section are absorption spectra of liquid solutions over the wavelength range of 450 nm to 700 nm (page 13), which 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 amount to 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. The important thing is that you must know enough about the measurement and the data it generates to recognize the kind of signals that is likely to generate, so you have some hope of knowing what is *signal* and what is *noise*.

It's important to appreciate that the standard deviations calculated of a small set of measurements can be much higher or much lower than the actual standard deviation 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. But if n is small, however, standard deviations will be different each time you evaluate that function; for example if $n=5$,

`randn(1,5)`, the standard deviations might vary randomly from 0.5 to 2 or even more. This is the [Law of Large Numbers](#); 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.

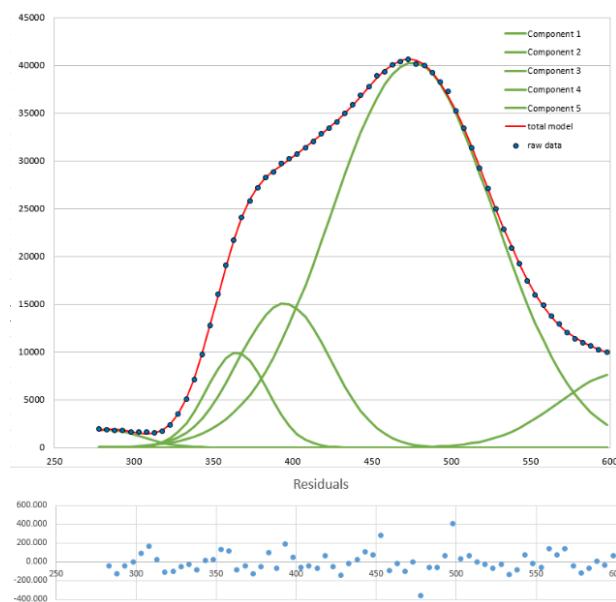
A quick but approximate way to estimate the amplitude of noise visually is the *peak-to-peak* range, which is the difference between the highest and the lowest values in a region where the signal is flat. The ratio of peak-to-peak range of $n=100$ normally-distributed random numbers to its standard deviation is approximately 5, 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 on the next page has a peak in the center with a height of about 1.0. 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's better to compute the standard deviation if possible.

In addition to the *standard deviation*, it's also possible 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: $\text{mad}=0.8*\text{std}$.

The *quality* of a signal is often expressed quantitatively as the *signal-to-noise ratio* (S/N ratio), 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 13 is about $0.08/0.001 = 80$, and the signal on page 25 has a S/N ratio of $1.0/0.2 = 5$. So we 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

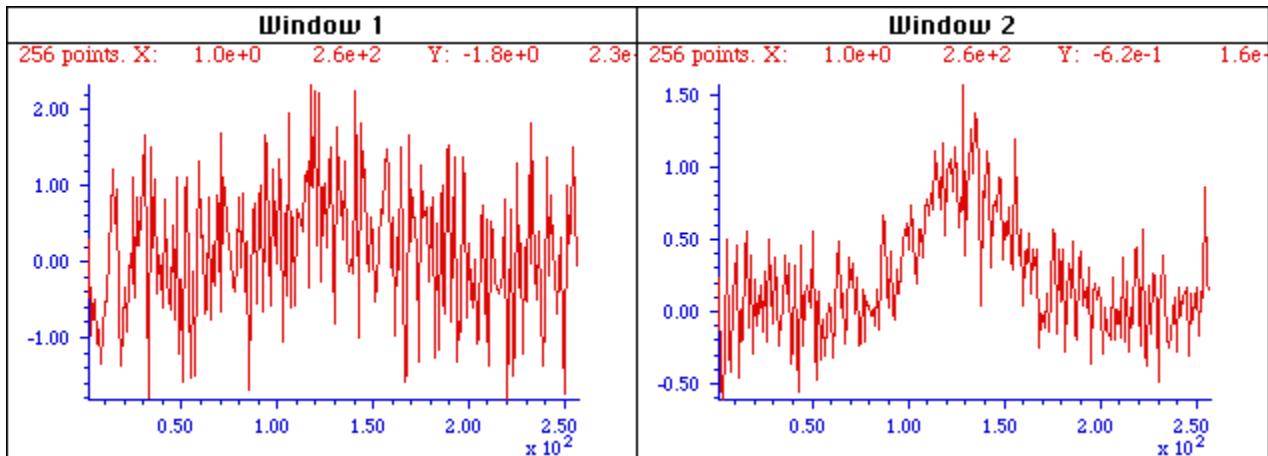
easier if the noise can be measured separately, in the absence of signal. Depending on the type of experiment, it may be possible to acquire readings of the noise alone, for example on a segment of the baseline before or after the occurrence of the signal. However, if the magnitude of the noise depends 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, where you can model the shape of the signal accurately by means of a mathematical function (such as a polynomial or the weighted sum of a number of peak shape functions), the noise may be isolated by subtracting the model from the un-smoothed experimental signal, for example by looking at the residuals in least-

squares curve fitting (page 163), as in the example shown on the left. If possible, it's usually better to determine the standard deviation of repeated measurements of the thing that you want to measure (e.g. the peak heights or areas), rather than trying to estimate the noise from a single recording of the data.



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 at least partially reproducible. So if the signal can be measured more than once, use can be made of this fact by measuring the signal over and over again, 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 this figure.



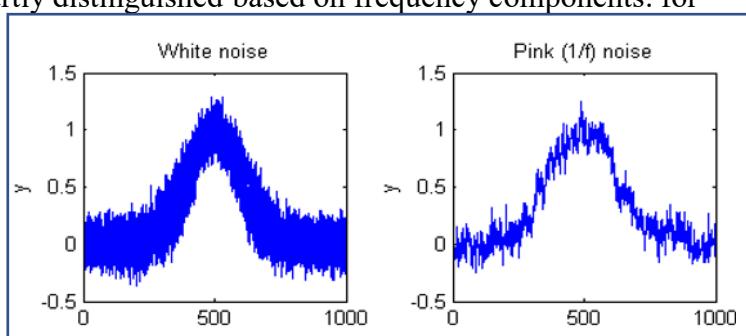
Window 1 (left) is a single measurement of a very noisy signal. There is actually a broad peak near the center of this signal, but it is difficult to measure its position, width, and height accurately because the S/N ratio is very poor. Window 2 (right) is the average of 9 repeated measurements of this 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; [click for graphic](#). Another example is shown in the video animation ([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 reduce digitization noise by ensemble averaging, *but only if small amounts of random noise are present in, or added to, the signal*; see page 270).

Visual animation of ensemble averaging. This 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, 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. Click [here](#) to download a brief video (2 MBytes) in WMV format.

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 34). In the figure 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 fairly 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.

Noise that has a more low-frequency-weighted 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 than 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 34) 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 a direct-current (DC) signal into an alternating current (AC) signal, 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 is not uncommon in experimental signal and can seriously interfere with accurate signal measurement.

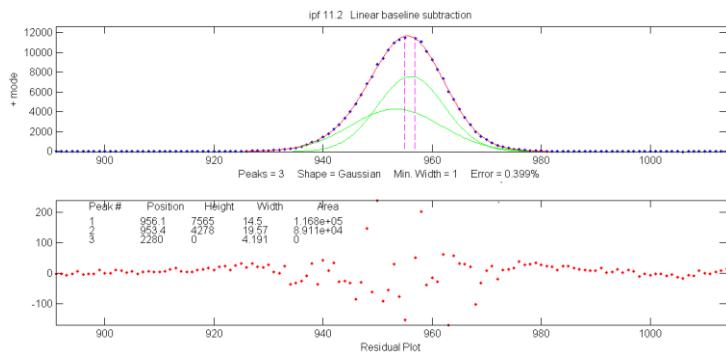
See page 280: *Random walks and baseline correction*.

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 or that have been deconvoluted from some blurring

process. Blue noise is *easier* to reduce by smoothing (page 26), and it has less effect on least-squares fits than the equivalent amount of white noise.

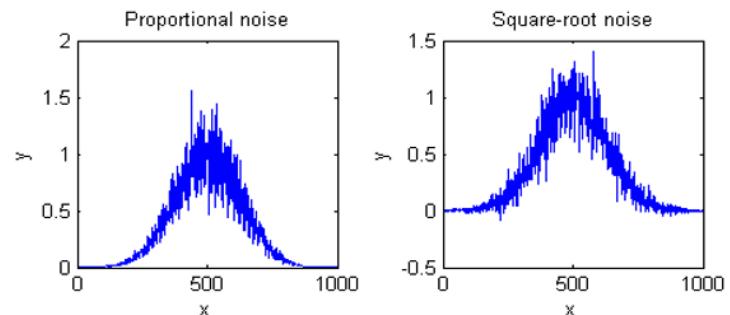
Dependence on signal amplitude

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. 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 multiple model, and observe how the residuals vary with signal amplitude. The example on the left is a real experimental signal showing the residuals from a curve-fitting operation (page 163) that reveals the noise increasing with signal amplitude. In other cases the noise is almost independent of the signal amplitude.

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 photo-multiplier detectors) is *white* and is proportional to the *square root* of light intensity, and therefore the SNR is proportional to the square root of light intensity and directly proportional to the monochromator slit-width. 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 and to the square of the monochromator slit-width. 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 slit width. 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.

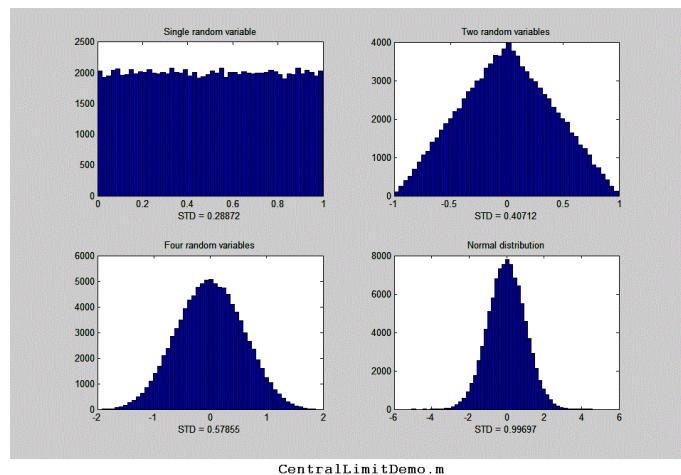


Only in a very few special cases is it possible to eliminate noise completely, so usually you must be satisfied by increasing the S/N ratio as much as possible. The key in any experimental system is to understand the possible sources of noise, break down the system into its parts and measure the noise 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. The effect of photon noise and detector noise can be reduced by increasing the light intensity at the detector or increasing the spectrometer slit width, and electronics 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. Only *photon noise* can be predicted from first principles (e.g. in these spreadsheets that simulate ultraviolet-visible spectrophotometry, fluorescence spectroscopy, and atomic emission spectroscopy).

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 *normal curve* (also called as a “bell” or “haystack” curve) and is described by a *Gaussian* function, $y=e^{-(x-\mu)^2 / (2\sigma^2)} / (\sqrt{2\pi}\sigma)$, where μ is the mean (average) value and 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 description 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*.



This is easily demonstrated by a little simulation. In the example on the left, we 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 Matlab/Octave). The graph in the upper left of the figure shows the probability distribution, called a “histogram”, of that random variable. Next, we combine two sets of such independent, uniformly-distributed random variables (changing the signs so that the average

remains 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, 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 1). Next, we combine *four* independent random variables (lower left); the resulting distribution 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 be visually close to a normal

Gaussian distribution (shown for reference in the lower right). If we combine more and more independent uniform random variables, the combined probability distribution becomes closer and closer to Gaussian (shown in the bottom right). *The Gaussian distribution that we 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 distributions of the individual events hardly matter at all.* You could modify the individual distributions in this simulation by including additional functions, such as `sqrt(rand)`, `sin(rand)`, `rand^2`, `log(rand)`, etc., to obtain other radically non-normal individual distributions. 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 are often the result of *thousands or millions* of individual microscopic events, so whatever the probability distributions of the *individual* events, the *combined* macroscopic observations approach a normal distribution essentially 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 limits, etc., are all based on the *assumption* of a normal distribution. 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 it's common to use the “[interquartile range](#)” (IQR), defined as the difference between the upper and lower quartiles, instead of the standard deviation, because *the interquartile range is not effected 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 they are roughly equal, 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 σ of some measured value, then you can predict the likelihood that your result 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 “68-95-99.7” or the [3-sigma rule](#). But the real practical problem is that *standard deviations are hard to measure accurately unless you have large numbers of samples*. See *The Law of Large Numbers* (page 321).

It important to understand that the three characteristics of noise just discussed in the paragraphs above - the frequency distribution, the amplitude distribution, and the signal dependence - are mutually independent; a noise may in principle have any combination of those properties.

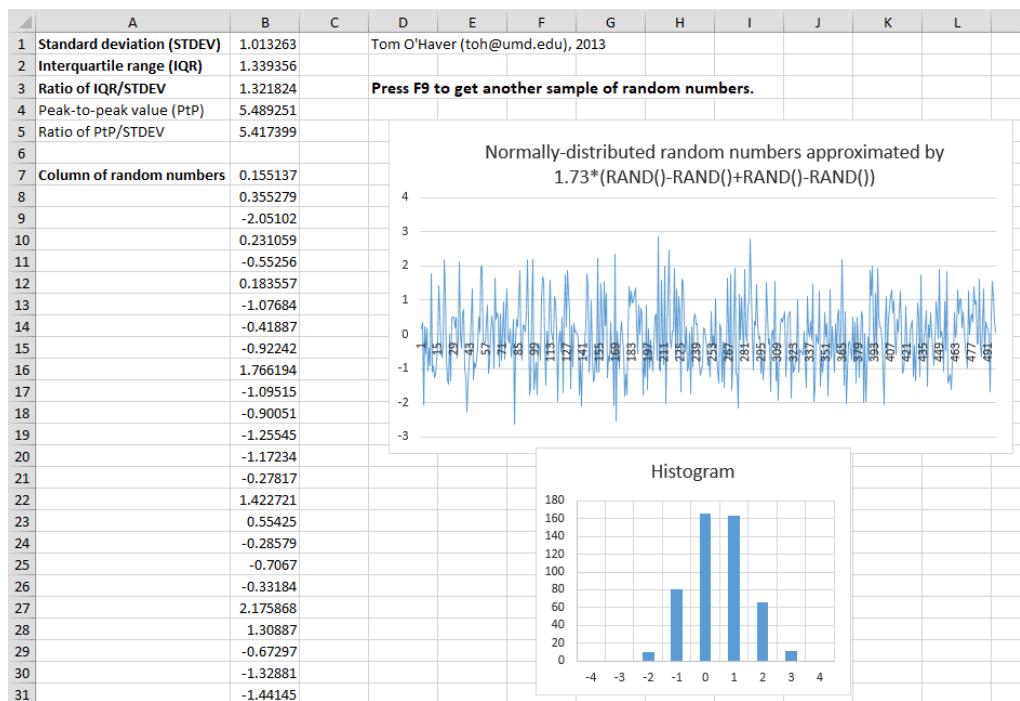
SPECTRUM, the Macintosh freeware signal-processing application, includes several functions for measuring signals and noise in the **Math** and **Window** pull-down menus, plus a signal-generator that can be used to generate artificial signals with Gaussian and Lorentzian bands, sine waves, and normally-distributed random noise in the **New** command in the **File** Menu. See page 383.

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)/(0.6005615*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, RAND, and QUARTILE.

Most spreadsheets have only a uniformly-distributed random number function (RAND) and not a normally-distributed random number function, but it's much more realistic to simulate errors that are normally distributed. In that case it's convenient to make use of the Central Limit Theorem to create approximately normally distributed random numbers by combining several RAND functions, for example, the 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(size(x))-rand(size(x))+rand(size(x))-rand(size(x))-rand(size(x)))** works similarly, but has a larger maximum range). 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](#), 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.

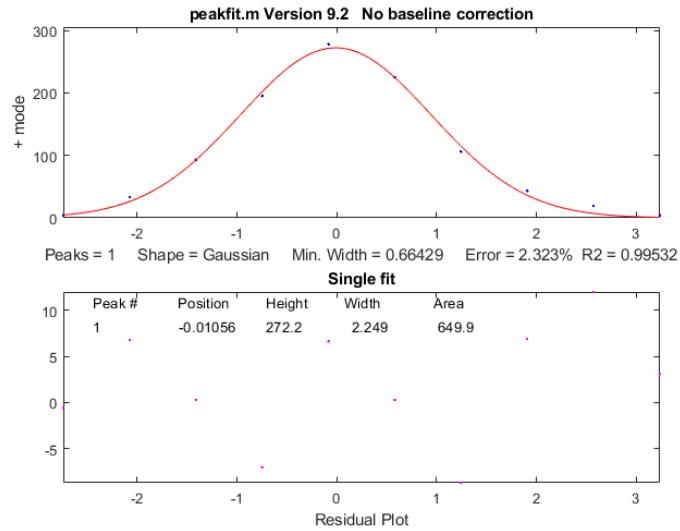


Matlab and Octave

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, hist, 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 $\sqrt{\text{mean}(y.^2)}$. You can subtract a scalar number from a vector (for example, $v = v - \text{min}(v)$ 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.

As an example of the "randn" function in Matlab/Octave, it is used here to generate 100 normally-distributed random numbers, then the "hist" function computes the "histogram" (probability distribution) of those random numbers, then the downloadable function peakfit.m fits a Gaussian function (plotted with a red line) to that distribution.

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



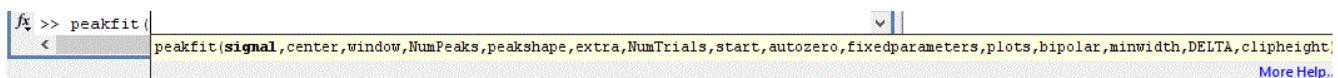
If you change the 100 to 1000 or a higher number, the distribution becomes closer and closer to a perfect Gaussian and its peak falls closer to 0.00. Here is [an MP4 animation](#) that demonstrates the gradual emergence of a Gaussian normal distribution and the number of randn samples increase from 2 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 133, 134). (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

You can also create your own user-defined scripts and functions in Matlab or Octave to automate commonly-used algorithms. Scripts and functions are simple text files saved with a ".m" file extension to the file name. 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. 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* arguments, and values can be passed *from* the function through the *output* arguments, which are both defined in the first line of the function definition. 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 program *without worrying that the variable names in the function will conflict and cause errors*. Scripts and functions can call other functions; scripts must have those functions in the Matlab path; functions, on the other hand, *can have all their required sub-functions defined within the main function itself and thus can be self-contained*. (If you run one of my scripts and get 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 path). Note: in Matlab R2016b or later, you CAN include functions within scripts (see https://www.mathworks.com/help/matlab/matlab_prog/local-functions-in-scripts.html).

For writing or editing scripts and functions, Matlab and the latest version of Octave have an internal editor. For an explanation of a function and a simple worked example, type "help function" at the command prompt. When you are writing your own functions or scripts, you should always add lots of "comment lines", beginning with the character %, that explain what is going on. *You'll be glad you did later*. The first group of comment lines, up to the first blank line that does not begin with a %, are considered to be the "help file" for that script or function. Typing "help NAME" displays those comment lines for the function or script NAME in the command window, just as it does for the built-in functions and scripts. 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.

Here's a very handy helper: 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 arguments as a reminder. *This works even for downloaded functions and for any new functions that you yourself create*. It's especially handy when there are so many possible input arguments that it's hard to remember all of them. The popup *stays on the screen as you type*, highlighting each argument in turn:



This feature is easily overlooked, but it's very handy. Clicking on "[More Help...](#)" on the right displays the help for that function in a separate window.

User-defined functions related to signals and noise

Some examples of my Matlab/Octave user-defined functions related to signals and noise that you can download and use include the following.

stdev.m, a standard deviation function that works in both Matlab and in Octave; rsd.m, the relative standard deviation;

halfwidth.m for measuring the full width at half maximum of smooth peaks; plotit.m, an easy-to-use function for plotting and fitting x,y data in matrices or in separate vectors;

Several functions for peak shapes commonly encountered in analytical chemistry such as Gaussian, Lorentzian, lognormal, Pearson 5, exponentially-broadened Gaussian, exponentially-broadened Lorentzian, exponential pulse, sigmoid, Gaussian/Lorentzian blend, bifurcated Gaussian, bifurcated Lorentzian), Voigt profile, triangular and others.

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

ShapeDemo demonstrates the 12 basic peak shapes graphically, showing the variable-shape peaks as multiple lines. (graphic on page 369)

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

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

IQRorange.m, computes the interquartile range,

rmnan.m removes "not-a-number" entries from vectors. Useful for cleaning up real data files.

val2ind.m returns the index and the value of the element of vector x that is closest to a particular value, a simple function that's more useful than you might imagine. See pages 89, 212 and 356.

These functions can be useful in modeling and simulating analytical signals and testing measurement techniques. 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 "path", you can use them just like any other built-in function. For example, you can plot a simulated 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 17, 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 of these functions will work in the latest version of Octave without change. For a complete list of downloadable functions and scripts developed for this project, see page 399 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. The Matlab/Octave script

[SubtractTwoMeasurements.m](#) (page 23) 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 323) is a downloadable Matlab function that can plot signals with pan and zoom controls, measure signal and noise amplitudes in selected regions of the signal, and compute the S/N ratio of peaks. It's operated by simple key presses. Other capabilities of iSignal include smoothing (page 34), differentiation, peak sharpening, and least-squares peak measurement, etc. The interactive keypress operation works even if you run [Matlab Online in a web browser](#), but not on [Matlab Mobile](#) or in Octave.

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 216), 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 in order 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 window 2 and saved as “EnsembleAverage.mat” in the current directory. See [iPeakEnsembleAverageDemo.m](#) for a demonstration.

See page 291: *Measuring the Signal-to-Noise Ratio of Complex Signals* for more examples of S/N ratio in Matlab/Octave.

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). As long as the true underlying signal is actually 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 over 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 algorithms

Most smoothing algorithms are based on the "*shift and multiply*" technique, in which a group of adjacent points in the original data are 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 down 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 ($m = 3$):

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

for $j = 2$ to $n-1$, where S_j the j^{th} point in the smoothed signal, Y_j the j^{th} point in the original signal, and n is the total number of points in the signal. 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*. 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 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, in order 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 "pseudo-Gaussian" or *haystack* smooth, 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 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.

Note that we are assuming here that the x-axis intervals 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 essay, and it is a very common (but not necessary) characteristic of signals that are acquired by automated and computerized equipment.

The *Savitzky-Golay* smooth is based on the least-squares fitting of polynomials to segments of the data. The algorithm is discussed in <http://www.wire.tu-bs.de/OLDWEB/mameyer/cmr/savgol.pdf>. 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 are greater than the smooth types discussed above, but with modern computers, the difference is not significant. Code in various languages is widely available online. See page 51.

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](#).

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 $D*0.8/\sqrt{m}$.

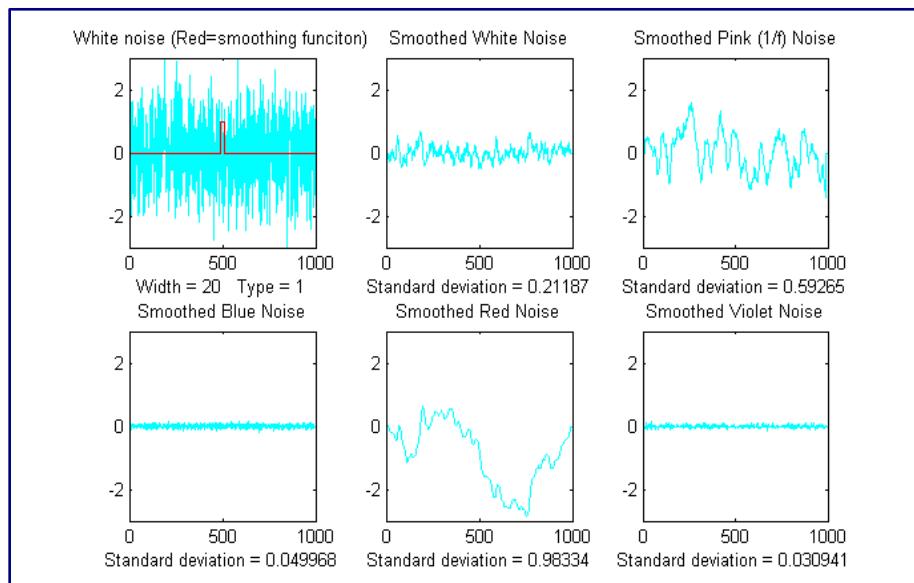
Smoothing operations can be applied more than once: that is, a previously-smoothed signal can be smoothed again. In some cases this can be useful if

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 reduces 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](#).

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

Effect of frequency distribution of noise

The frequency distribution of noise, designated by noise "color" (page 21), substantially effects 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 effects low frequency (pink and red) noise less, and effects high-frequency (blue and violet) noise more, than it does white noise.



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.

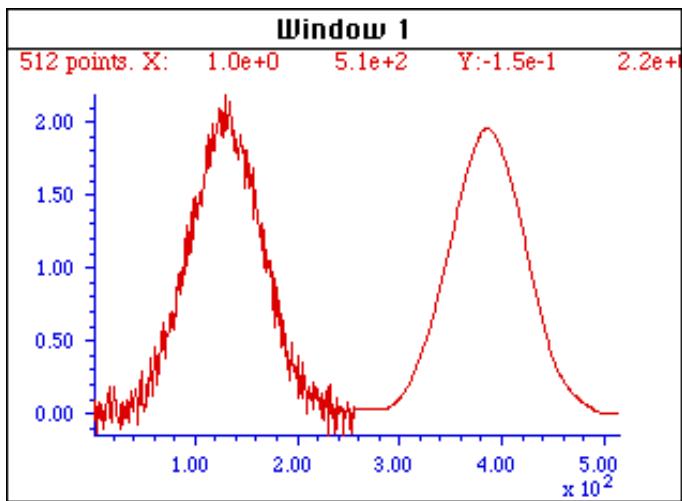
End effects and the lost points problem

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 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 paper). The other approach is to use *progressively smaller smooths* at the ends of the signal, for example to use 2, 3, 5, 7... point smooths for signal points 1, 2, 3, and 4..., and for points n , $n-1$, $n-2$, $n-3$..., respectively. The later approach may be preferable if the edges of the signal contain critical information, but it increases execution time. The Matlab/Octave *fastsmooth* function can utilize either of these two methods.

Examples of smoothing

The figure on the left 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

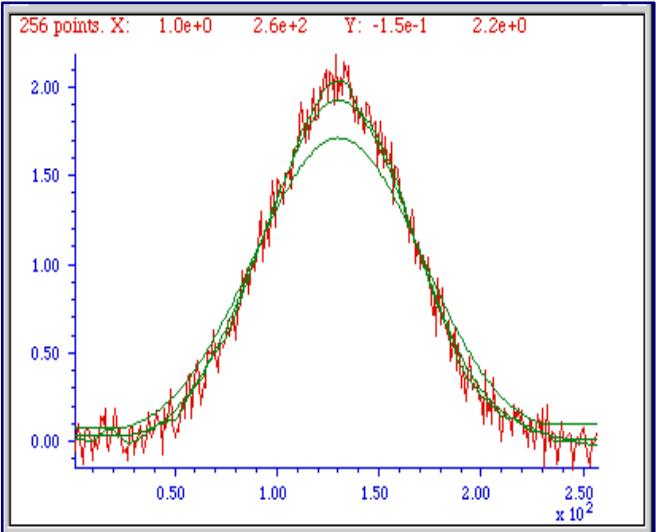
(peak position, height, width, area, etc.) to be measured more accurately by visual inspection.



*The left half of this signal is a noisy peak. The right half is the same peak after undergoing a **smoothing** algorithm. The noise is greatly reduced while the peak itself is hardly changed, making it easier to measure the peak position, height, and width directly by graphical or visual estimation (but it does not improve measurements made by least-squares methods; see below).*

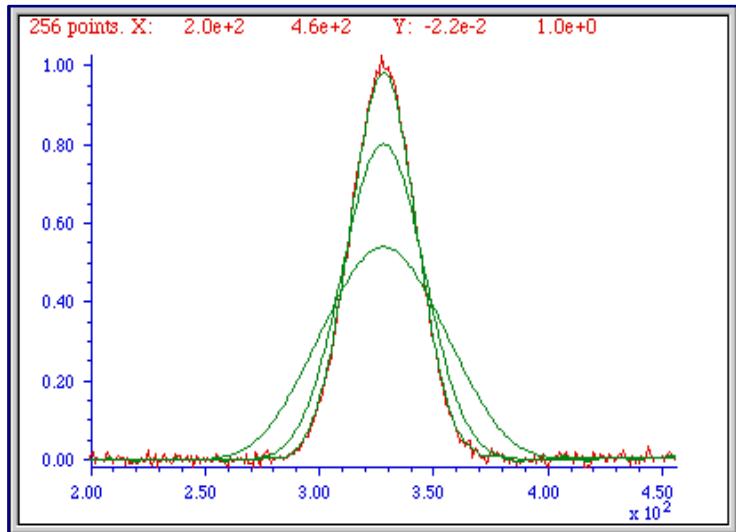
The larger the smooth width, the greater the noise reduction, but also the greater the

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 bandwidth 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

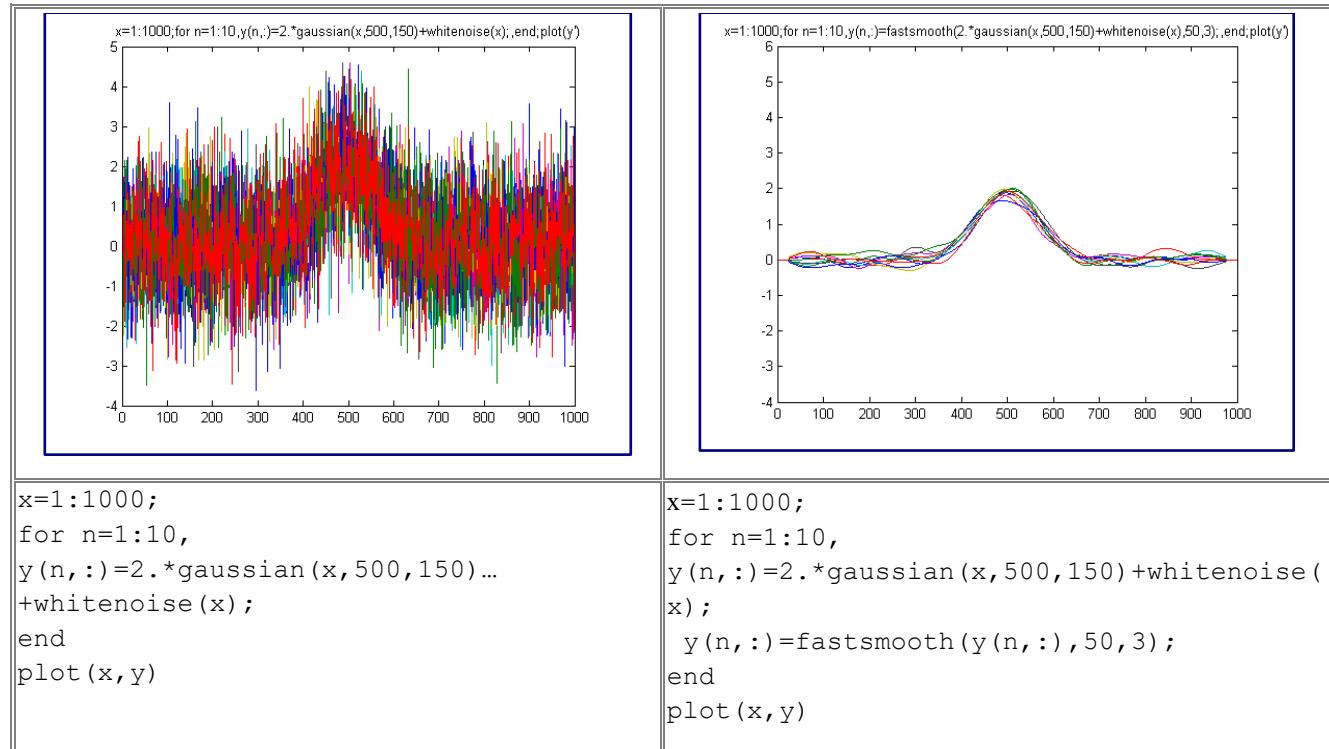
progressively reduced but the peak height also is reduced slightly. For the largest smooth, the peak *width* is noticeably increased. In the figure on the right, the original peak (in red) has a true height of 1.0 and a half-width of 33 points. (It is also less noisy than the example on the left.) The 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 effected (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 area under the peak remains unchanged. If the peak widths vary substantially, an adaptive smooth, which allows the smooth width to vary across the signal, may be used.



The problem with smoothing

Smoothing is often less beneficial than you might think. It's important to point out that smoothing results such as illustrated in the figures above may be *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 number of independent samples of a noisy signal consisting of a single peak, as illustrated in the two figures

below.



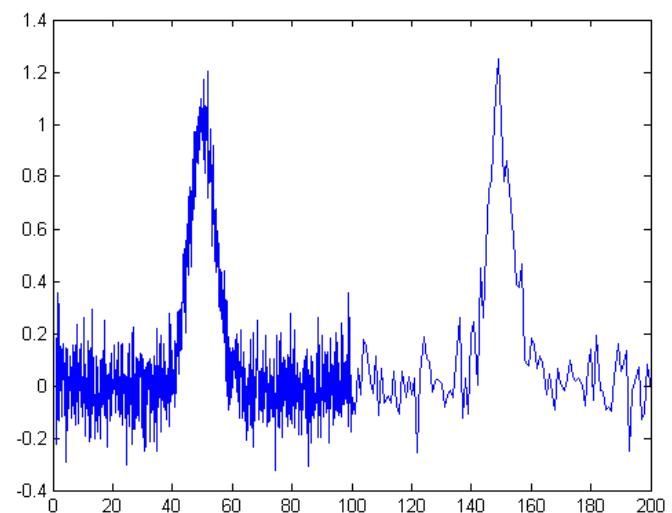
These figures show ten superimposed plots with the same peak but with independent white noise, each plotted with a different line color, unsmoothed on the left and smoothed on the right. Clearly, the noise reduction is substantial, but close inspection of the smoothed signals on the right clearly shows the variation in peak position, height, and width between the 10 samples 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 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 will still interfere with 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 *completely* eliminate noise, 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 or not that makes for a *better measurement* or not depends on the situation. And don't assume that just because a little smoothing is good that more will necessarily be better. Smoothing is like alcohol; sometimes you really need it - but you should never overdo it.

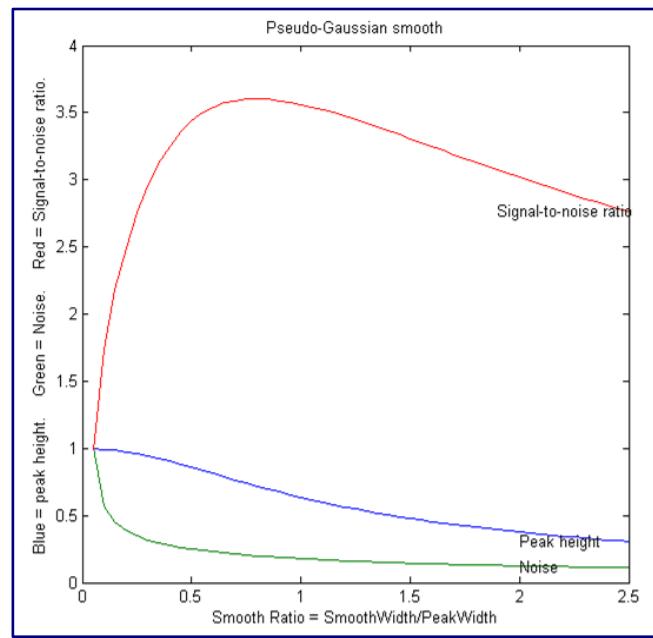
The figure on the right below is another example signal that illustrates some of these principles. The signal consists of two Gaussian peaks, one located at $x=50$ and the second at $x=150$. Both peaks have a peak height of 1.0 and a peak half-width of 10, and the same normally-distributed random white 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's 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 points* than the second peak. It may *look* like the first peak is noisier than the second, but that's 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 (it becomes 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 to be covered later, 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 in the graphic 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* 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. Peak *area* is not changed by smoothing, 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 *exactly the same* as that of the sample signals and the effect will *cancel 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). 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 *exactly* 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 51.

When should you smooth a signal?

There are three 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 37.). Generally, if a computer is available to make quantitative measurements, it's 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's 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 an 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*.

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 198), 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 of a segment of original *unsmoothed* data in the vicinity of the zero-crossing. 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.

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.

- (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 parameters errors predicted by the algebraic propagation-of-error calculations and by the bootstrap method (page 134). 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. 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 completely 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. A different approach is used by the `killspikes.m` function; it locates and eliminates the spikes by "patching over them" using linear interpolation from the signal points before and after the spike. Unlike conventional smooths, these functions can be profitably applied *prior* to least-squares fitting functions. (On the other hand, if the *spikes themselves* are actually the signal of interest, and the other components of the signal are interfering with their measurement, see page 266.)

Ensemble Averaging

Another way to reduce noise in repeatable signals, such as the set of ten unsmoothed signals on page 40, 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 163) 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 40). 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 the signal or the baseline drifts.

Condensing oversampled signals

Sometimes signals are recorded more densely (that is, with smaller x-axis intervals) than really 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 other point or every third point) or better by replacing groups of adjacent points by their *averages*. The latter approach 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 frequency distribution of the 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.

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 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 improves and the accuracy of the measurements of peak amplitude and peak width are improved. However, above a smooth width of about 40 (smooth ratio 0.2), the smoothing 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.

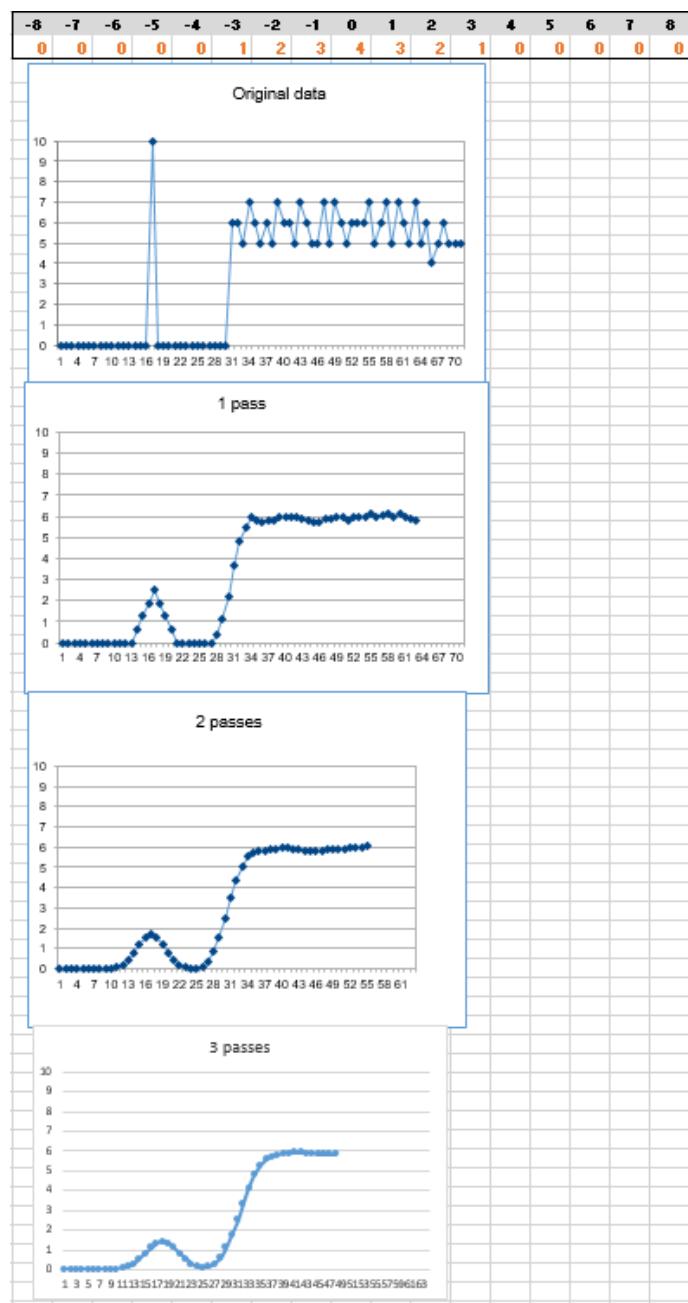
SPECTRUM, the freeware Macintosh signal-processing application, includes rectangular and triangular smoothing functions for any number of points. See page 383.

Smoothing in spreadsheets

Smoothing can be done in spreadsheets using the "shift and multiply" technique described above. In the 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's 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 rectangular, triangular, and Gaussian smooths of width 3 to 29 in both vertical (column) and horizontal (row) format. You can Copy and Paste these into your own spreadsheets.

The spreadsheets [MultipleSmoothing.xls](#) and [MultipleSmoothing.ods](#) (screen image on the left) demonstrate a more flexible 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, just insert zeros for the unused coefficients; in this example, a 7-point triangular smooth is defined in columns N - T and the rest

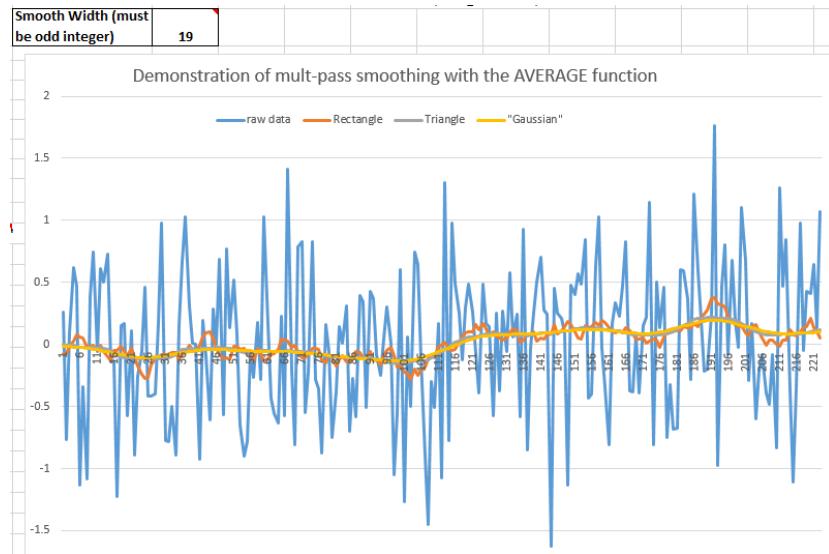


of the coefficients are zeros). In this spreadsheet, the smooth is applied *three times* in succession in columns C, E, and G, resulting in an effective maximum smooth width of $n*w-n+1 = 49$ points applied to column G. A disadvantage of the above technique for smoothing in spreadsheets is that it is 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 Gaussian shaped smooths. It is best used in conjunction with the INDIRECT function (page 313) to control a dynamic range of values, as is demonstrated in the spreadsheet [VariableSmooth.xlsx](#) (right) 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 also 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.

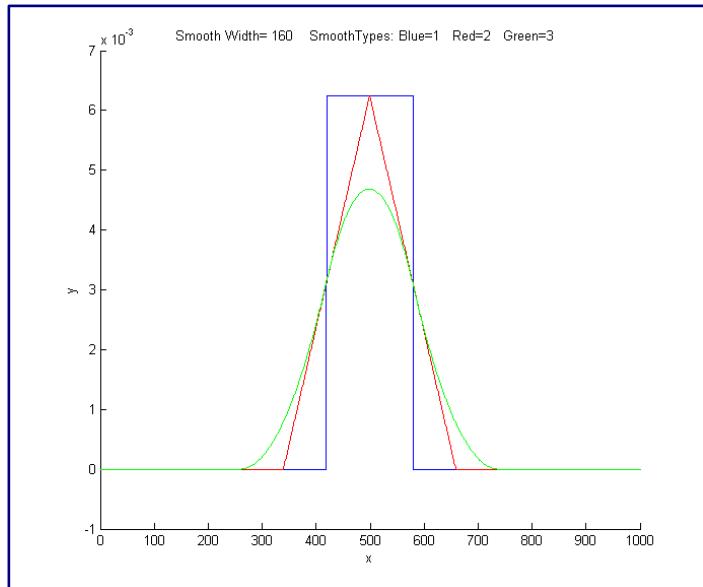
Another set of spreadsheets that uses this same AVERAGE(INDIRECT()) technique is [SegmentedSmoothTemplate.xlsx](#), a *segmented* multiple-width data smoothing spreadsheet template that can apply individually specified different smooth widths to different regions of the signal. This is especially useful if the widths 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](#)); note that the plot is conveniently lined up with the columns containing the smooth widths for each segment. A related sheet [GradientSmoothTemplate.xlsx](#) and [GradientSmoothExample2.xlsx](#) ([graphic](#)) performs a linearly increasing (or decreasing) smooth width across the entire signal, given only the starting and ending values, 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 custom function `fastsmooth` implements shift and multiply type smooths using a recursive algorithm. "Fastsmooth" 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; type=3 gives a pseudo-Gaussian smooth, equivalent to three passes of a sliding average; these shapes are compared in the figure on the left. (See page [51](#) 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, especially 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 second on a standard Windows PC. Here's a simple example of `fastsmooth` demonstrating the effect on white noise ([graphic](#)).

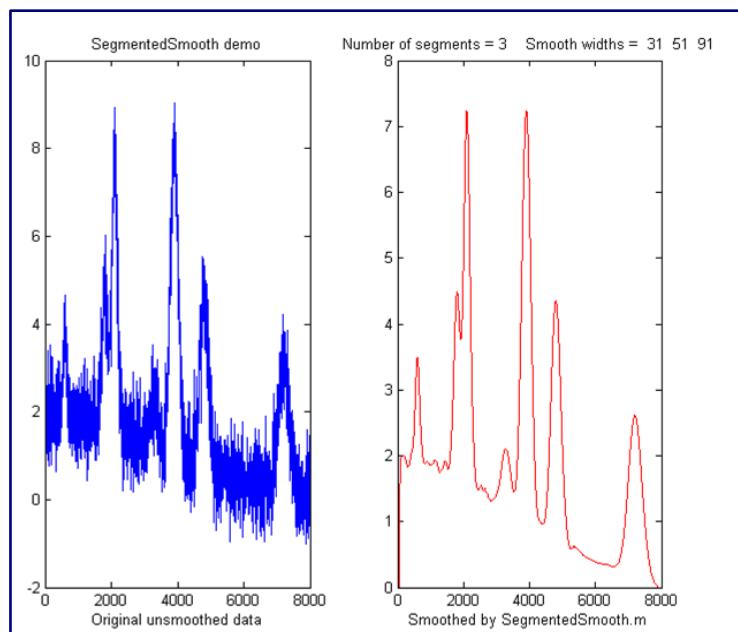
```
x=1:100;
y=randn(size(x));
plot(x,y,x, fastsmooth(y,5,3,1), 'r')
xlabel('Blue: white noise.      Red: smoothed white noise.')
```

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 a number of 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 sequence of smooth widths. Type "help SegmentedSmooth" for other examples.

The demonstration script [DemoSegmentedSmooth.m](#) shows the operation with different signals consisting of noisy variable-width peaks that get progressively wider, like the 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:

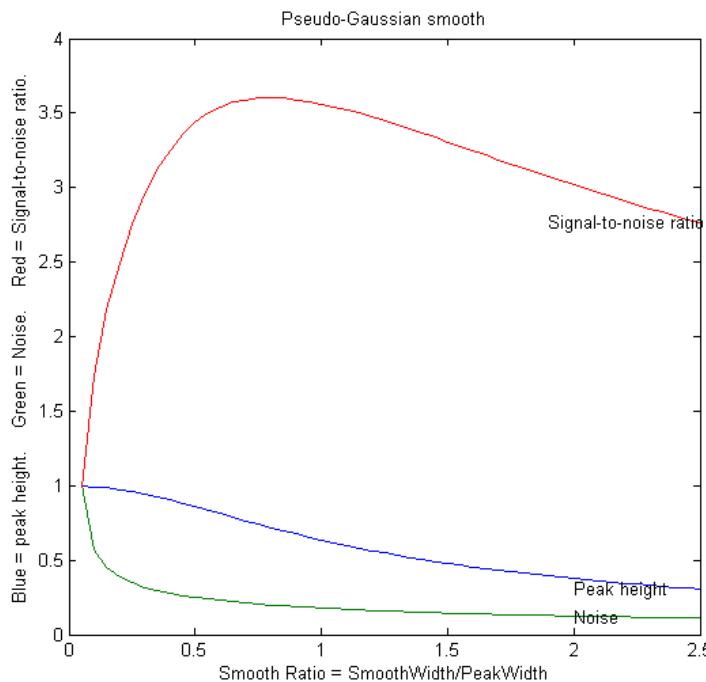
```
wstep=(endw-startw)/NumSegments;
smoothwidths=startw:wstep:endw;
```

[Diederick](#) has published a [Savitzky-Golay](#) smooth function in Matlab, which you can download from the [Matlab File Exchange](#). It's included in the [iSignal](#) function (page 323). [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 ([nanfastsmoothAngle\(Y,w,type,tol\)](#)).

[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 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 the peak height*, which could be a problem. A smooth width about *half* the width of the original unsmoothed peak produces less distortion of the peak but still achieves a reasonable noise reduction. [SmoothVsCurvefit.m](#) is a similar script, but is also compares curve fitting 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*.

(`max` and `trapz` are both built-in functions in Matlab and Octave, but you have to 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)])
```

<code>max</code>	<code>halfwidth</code>	<code>Area</code>
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 effect* on the peak area, as long as you measure the *total area* under the broadened peak. Smoothing is useful if the signal is contaminated by non-normal noise such as sharp spikes or 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 the peak parameters are measured by least-squares methods, because the least-squares results obtained on the unsmoothed data will be more accurate (see page 197).

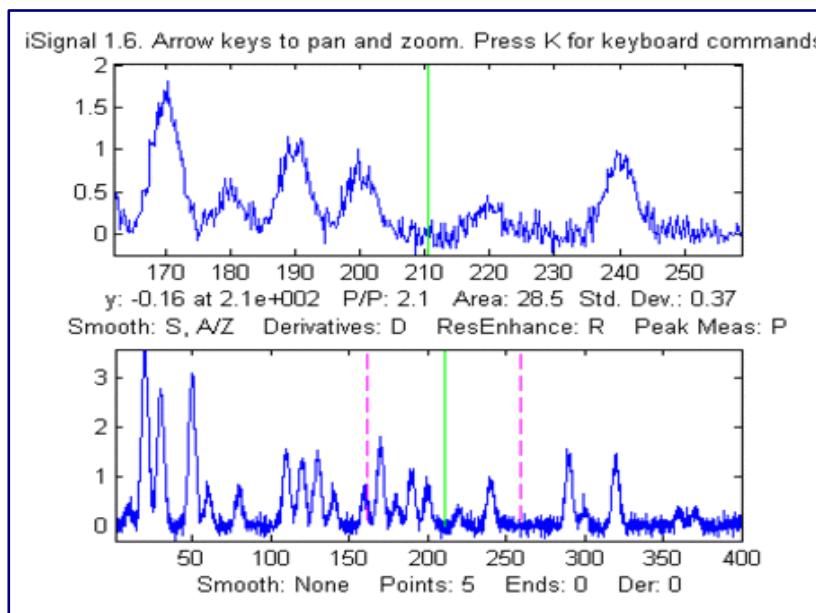
Other smoothing-related functions. The Matlab/Octave user-defined function `condense.m`, `condense(y,n)`, returns a condensed version of `y` in which each group of `n` points is replaced by its average, reducing the length of `y` by the factor `n`. (For `x,y` data sets, use this function on **both**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

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)`. See `killspikesdemo`. Type "help `killspikes`" at the command prompt.

`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. The 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, SlewRate, MedianWidth)`.

Real-time smoothing in Matlab is discussed on page 308.

iSignal (page 323) is an interactive function for Matlab that performs 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, with keystrokes that 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 the smooth width to see how the frequency response changes. View the code [here](#) or download the [ZIP file](#) with sample data for testing.



You try it: Here's an experiment you can try. 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 $x=150$, $\text{height}=1\text{e}-4$; $\text{SNR}=90$. First, download [NoisySignal.mat](#) into the Matlab path, then execute these statements:

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

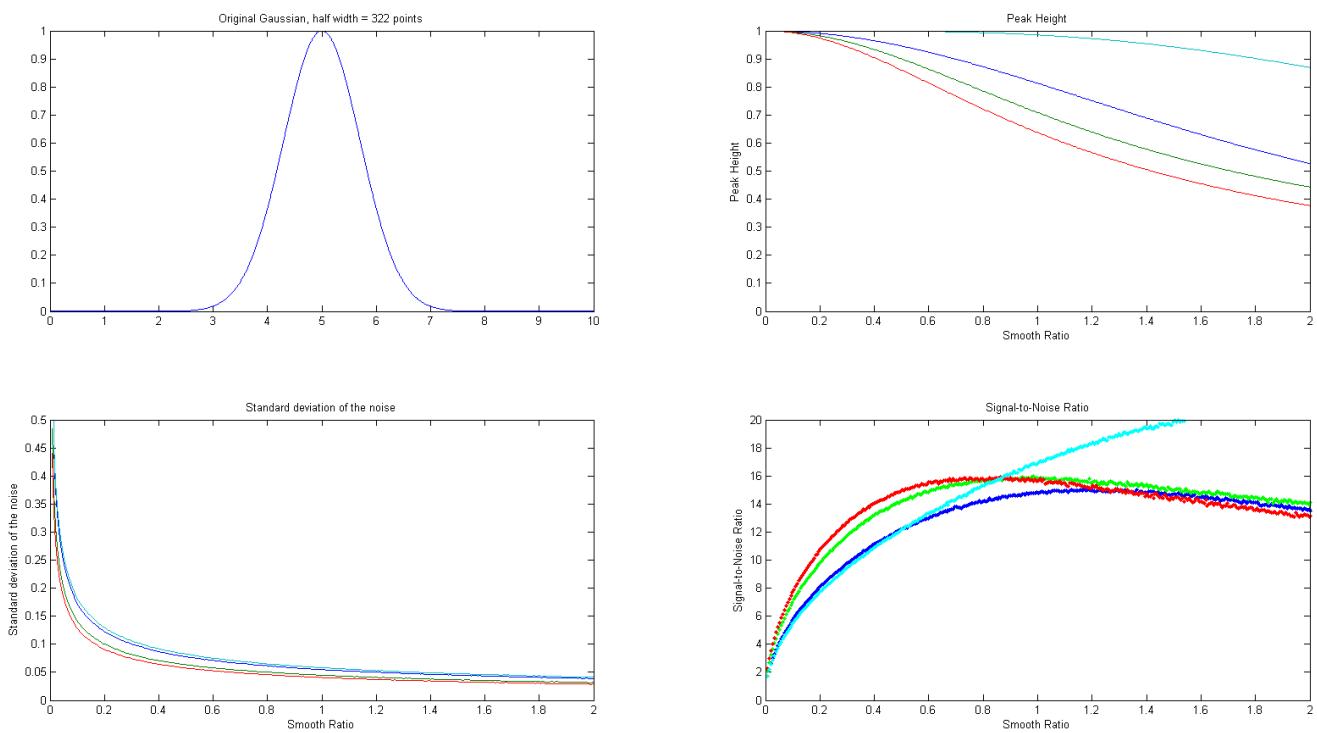
Use the **A** and **Z** keys to increase and decrease the smooth width, and the **S** key to cycle through the available smooth types. Hint: use the Gaussian smooth and keep increasing the smooth width until the peak shows. (Unfortunately, iSignal does not currently work in Octave).

Note: In the PDF version of this book, 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.

Smoothing performance comparison

The Matlab/Octave function "[smoothdemo.m](#)" is a self-contained function that compares the performance of four types of smooth operations: (1) sliding-average, (2) triangular, (3) pseudo-Gaussian (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 [iSignal](#) functions. These four smooth operations are applied to a 2000-point signal consisting of a Gaussian peak with a FWHM ([full-width at half-maximum](#)) of 322 points and to a noise array consisting of normally-distributed random white noise with a mean of zero and a standard deviation of 1.0. The peak height of the smoothed peak is taken as the 'signal'; the standard deviation of the smoothed noise is taken as the 'noise', and the resulting signal-to-noise ratio are all measured as a function of smooth width, for each smooth type. Smooth width is expressed in terms of "smooth ratio", the ratio of the width of the smooth to the width (FWHM) of the peak.

The results, when "[smoothdemo.m](#)" is run (with a noise array length of 10^7 to insure accurate sampling of the noise), are shown by the figure and text print-out below.



The four quadrants of the graph are: (upper left) the original Gaussian peak before smoothing and without noise; (upper right) the peak height of the smoothed signal as a function of smooth ratio; (lower left) the standard deviation of the noise as a function of smooth ratio; the signal-to-noise ratio (SNR) as a function of smooth ratio (lower right). The different smooth types are indicated by color: blue - sliding-average; green - triangular; red - pseudo-Gaussian, and cyan - Savitzky-Golay.

The program also calculates and prints out the elapsed time for each smooth type and the maximum in the SNR plot.

Here are the results of the four smoothing methods:

1. Sliding-average:

Elapsed Time: 0.26 Opt. SNR: 15.1 at smooth width of 1.25

2. Triangular:

Elapsed Time: 0.59 Opt. SNR: 15.8 at smooth width of 1.11

3. Pseudo-Gaussian:

Elapsed Time: 0.87 Opt. SNR: 15.6 at smooth width of 0.93

4. Savitzky-Golay:

Elapsed Time: 4.5 Opt. SNR: 20.3 at smooth width of 1.74

These results clearly show that the Savitzky-Golay smooth gives the smallest peak distortion (smallest reduction in peak height), but, on the other hand, it gives the smallest reduction in noise amplitude and the longest computation time (by far). The pseudo-Gaussian smooth gives the greatest noise reduction and, below a smooth ratio of about 1.0, the highest signal-to-noise ratio, but the Savitzky-Golay smooth gives the highest SNR above a smooth ratio of 1.0. For applications where the shape of the signal must be preserved as much as possible, the Savitzky-Golay is clearly the method of choice. In the peak detection function (page 198), on the other hand, the purpose of smoothing is to reduce the noise in the derivative signal; the retention of the shape of that derivative is less important. Therefore, the triangular or pseudo-Gaussian smooth is well suited to this purpose and has the additional advantage of much faster computation speed.

The conclusions are essentially the same for a Lorentzian peak, as demonstrated by a similar function "[smoothdemoL.m](#)", the main difference being that the peak height reduction is greater for the Lorentzian.

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 generated by this [script](#) (click for [GIF animation](#)). Assuming that the x -interval between adjacent points is constant, the simplest algorithm for computing a first derivative is:

$$Y'_j = \frac{Y_{j+1} - Y_j}{X_{j+1} - X_j} = \frac{Y_{j+1} - Y_j}{\Delta X} \quad X'_j = \frac{X_{j+1} + X_j}{2} \quad (\text{for } 1 < j < n-1).$$

where X'_j and Y'_j are the X and Y values of the j^{th} 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:

$$Y'_j = \frac{Y_{j+1} - Y_{j-1}}{2\Delta X} \quad X'_j = X_j \quad (\text{for } 2 < j < n-1).$$

This is a *central-difference* method; its advantage is that it does not involve a shift in the x -axis position of the derivative. It's 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. It turns out that this is equivalent to applying a moving-average (rectangular) smooth (page 34) 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

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

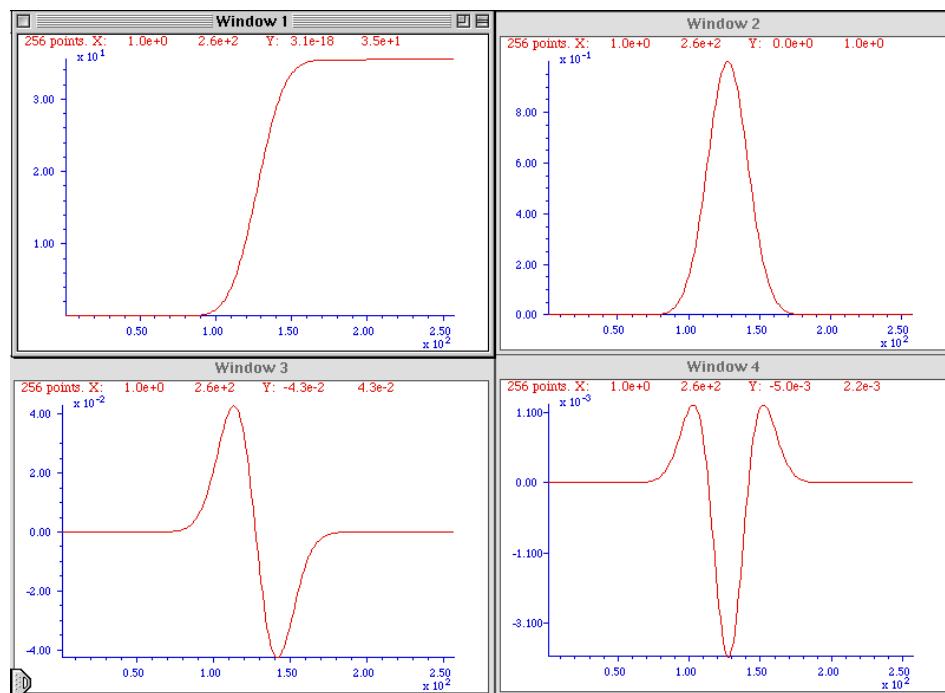
Similarly, higher derivative orders can be computed using the appropriate sequence of coefficients: for example +1, -2, +2, -1 for the third derivative and +1, -4, +6, -4, +1 for the 4th derivative, although these derivatives can also be computed simply by taking successive lower order derivatives. The first derivative we interpret as the *slope* of the original at each point, and the second derivative as the *curvature*, but beyond that, 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 34) can also be used as a differentiation algorithm with the appropriate choice of input arguments; it 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 can be a limitation only in some extreme cases (e.g. page 302).

Basic Properties of Derivative Signals

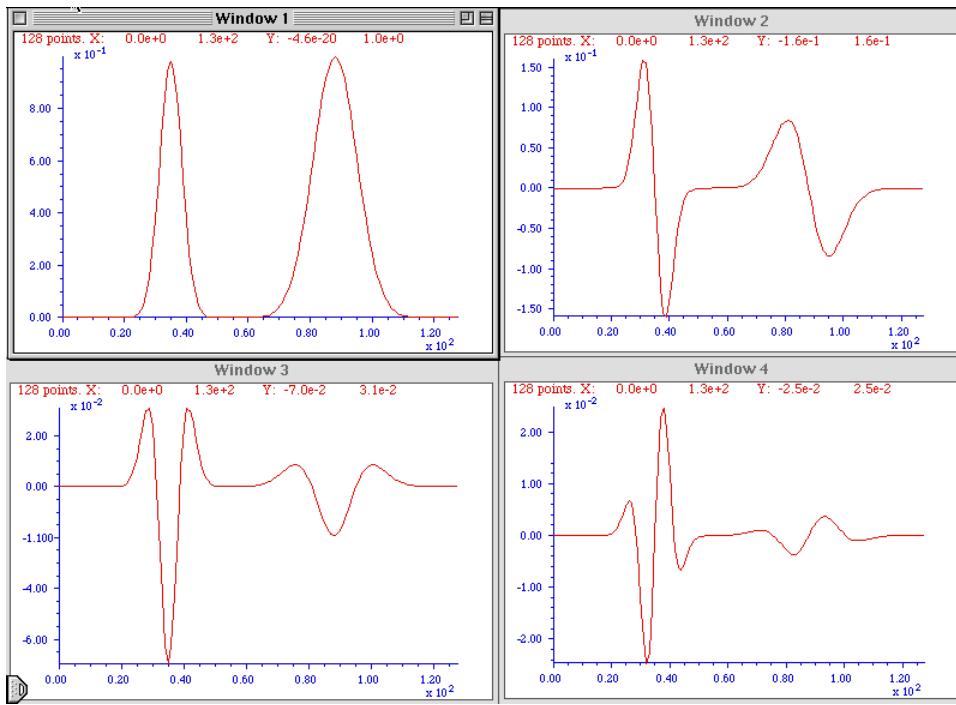


The figure on the left shows the results of the successive differentiation of a computer-generated Gaussian peak. The signal in each of the four windows is the first derivative of the one before it; that is, Window 2 is the first derivative of Window 1, Window 3 is the first derivative of Window 2, Window 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; and where a signal has zero slope, its derivative is zero. ([Matlab/Octave code](#) for this figure.)

The sigmoidal signal shown in Window 1 has an *inflection point* (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 369). 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's 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).



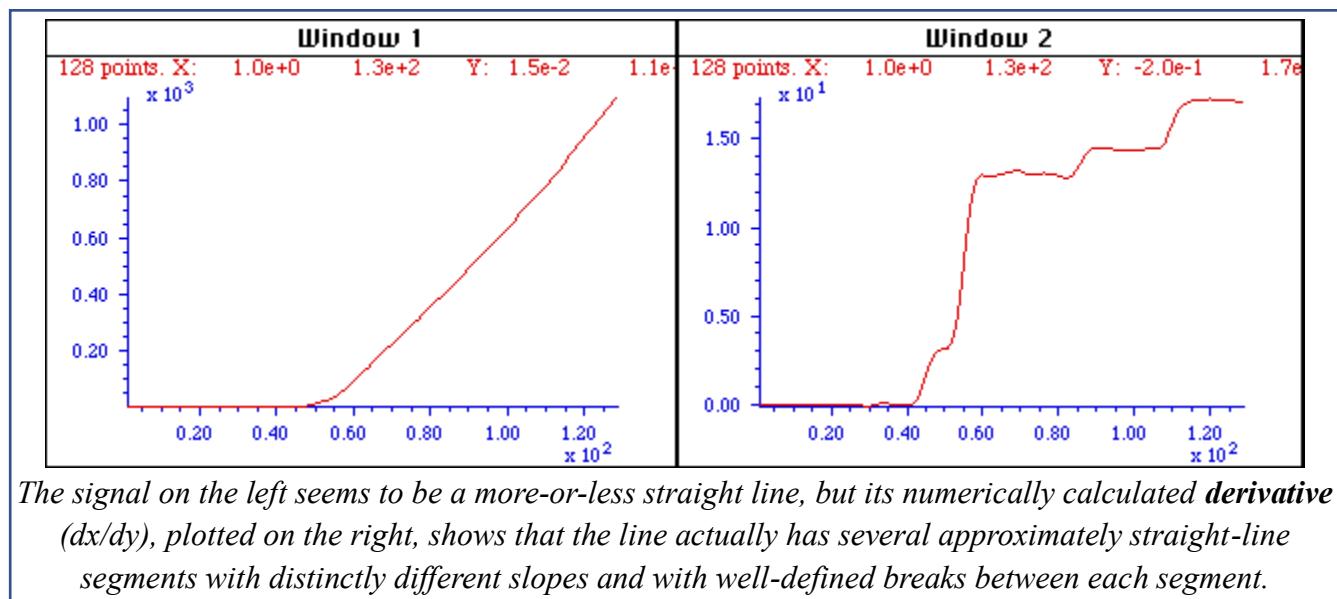
An important property of the differentiation of peak-type signals is the effect of the peak *width* on the amplitude of derivatives. The figure on the left shows the results of the successive differentiation of two computer-generated Gaussian bands. The two bands have the same amplitude (peak height) but one of them is exactly twice the width of the other. As you can see, the *wider* peak has the *smaller* derivative amplitude, and

the effect becomes more noticeable at higher derivative orders. In general, the amplitude of the n^{th} derivative of a peak is inversely proportional to the n^{th} 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 n^{th} 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 changes the shape of *peak-type* signals drastically, 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 341 for an example.

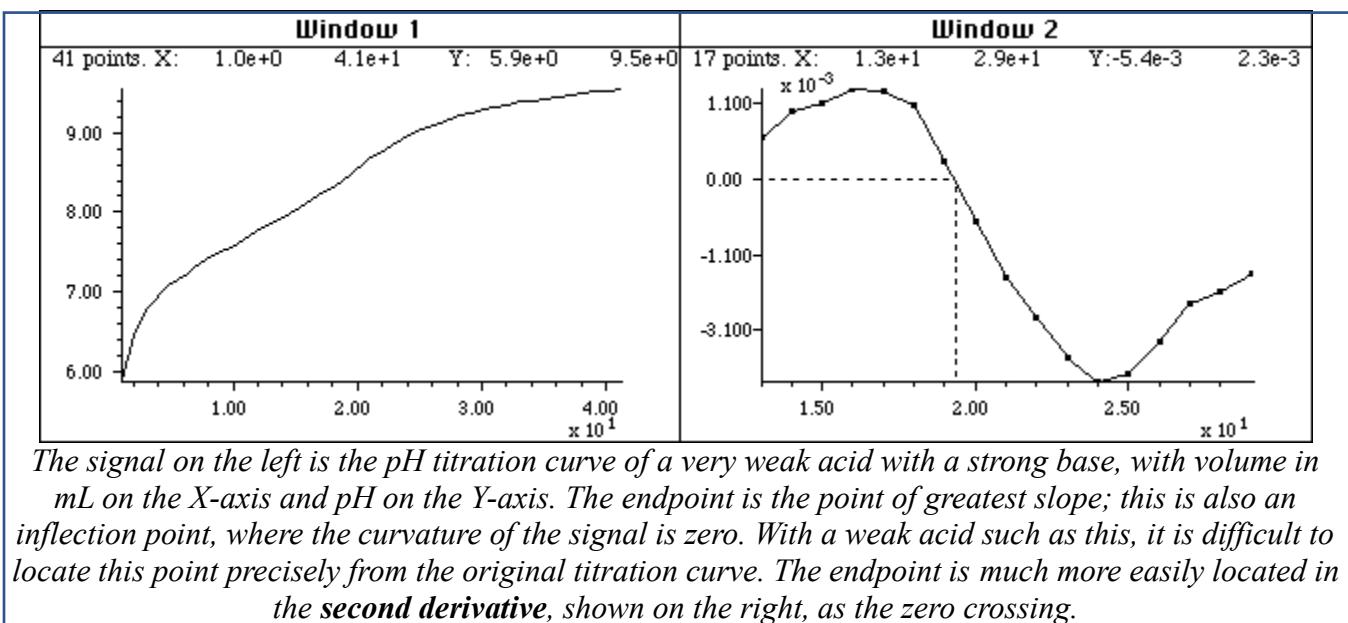
Applications of Differentiation

A simple example of the application of 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 limiting. 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.



It is commonly observed that differentiation degrades signal-to-noise ratio, unless the differentiation algorithm includes smoothing (page 34) 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 degradation (page 62).

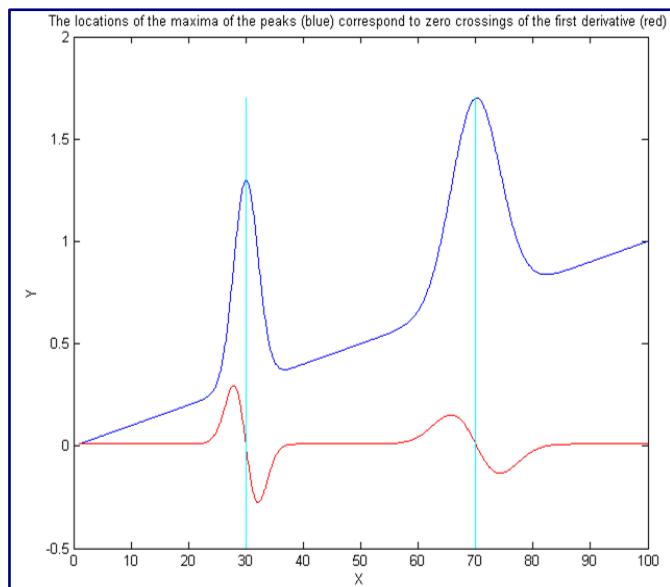
A classic use of second differentiation in chemical analysis is in the location of endpoints in potentiometric titration. In most titrations, the titration curve has a sigmoidal shape and the inflection point, the point where the slope is maximum and the curvature is zero, indicates the endpoint. The 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 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 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.

Peak detection

Another common use of differentiation is in the detection of peaks in a signal. It's 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-



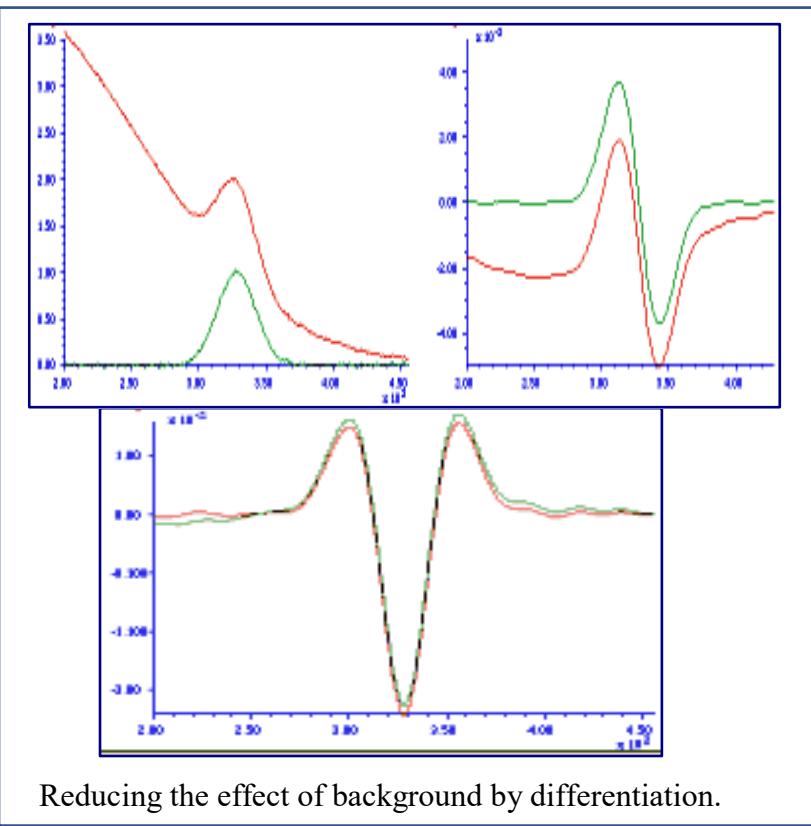
going zero-crossings, and then takes only those zero crossings whose slope exceeds a certain predetermined minimum (called the "slope threshold") at a point where the original signal amplitude exceeds a certain minimum (called the "amplitude threshold"). By carefully adjusting the smooth width, slope 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 34), 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 peaks, 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 65) and in spreadsheets (page 65).

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 as a way 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 386). Most commercial spectrophotometers now have built-in derivative capability. Some instruments are designed to measure the spectral derivatives optically, by means of dual wavelength or wavelength modulation designs.

Because of the fact that the amplitude of the n^{th} derivative of a peak-shaped signal is inversely proportional to the n^{th} 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 of this type can be reduced by differentiation.



Reducing the effect of background by differentiation.

This is illustrated by the figure above, 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 compounds 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 at the bottom. In this case, the spectra of the pure analyte and of the mixture are almost identical. In order 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 demonstration of this application.

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 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 nm². You can't compare absorbance to absorbance per nm any more than you can compare miles to miles per hour. (It's 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*.

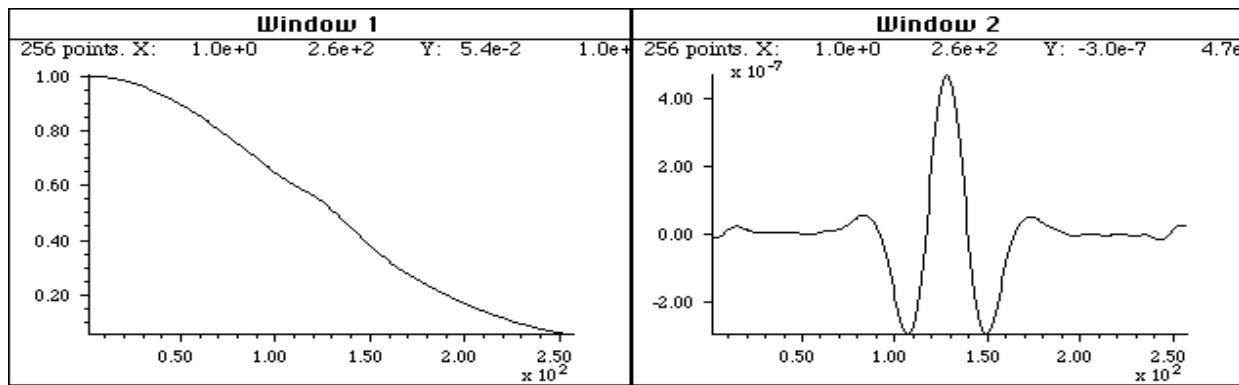
For instance, in the above example, it would be valid to say that the signal-to-background ratio is better (higher) in the derivatives.

Loosely speaking, the opposite of differentiation is integration, so if you take the first derivative of a signal, you might expect to be able regenerate the original (zeroth derivative) by integration. However, there is a catch; the constant term in 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 is used only 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 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's important to measure the standards and the unknown samples in exactly the same way. Also, because the amplitude of a derivative of a peak depends strongly on its width, it's important to control environmental factors that might change spectral peak width subtlety, such as temperature.

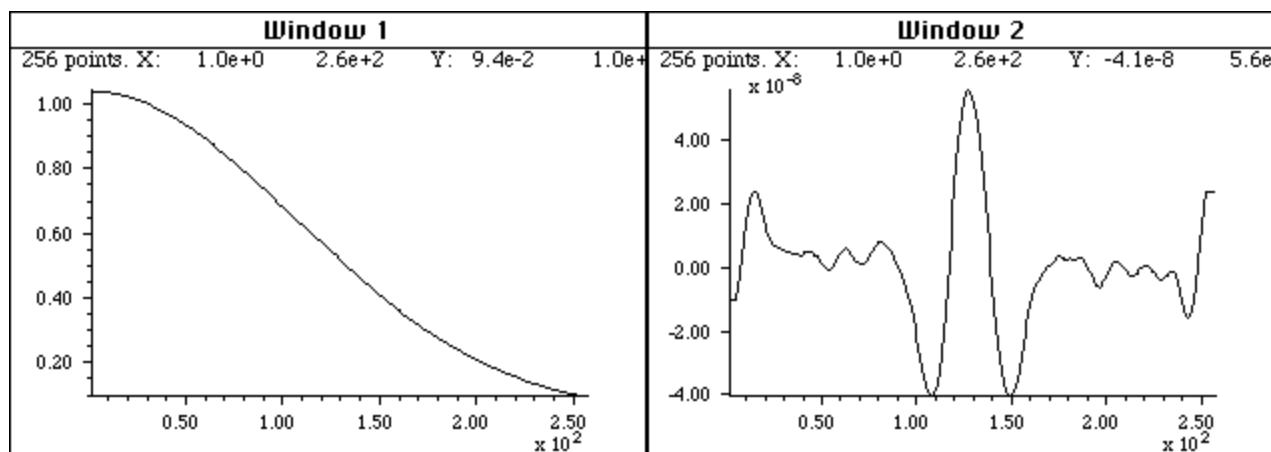
Trace Analysis

One of the widest uses of the derivative signal processing technique in practical analytical work is in the measurement of small 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 interfering 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, it 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 and 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 a pharmaceutical preparation). It is difficult to measure the intensity of this peak because it 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 due to the analyte. The signal-to-noise ratio is very good in this spectrum, but in spite of 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 clearly evident here, but nevertheless the signal-to-noise ratio is sufficiently good for a reasonable quantitative measurement.



Similar to the previous figure, but in the 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.

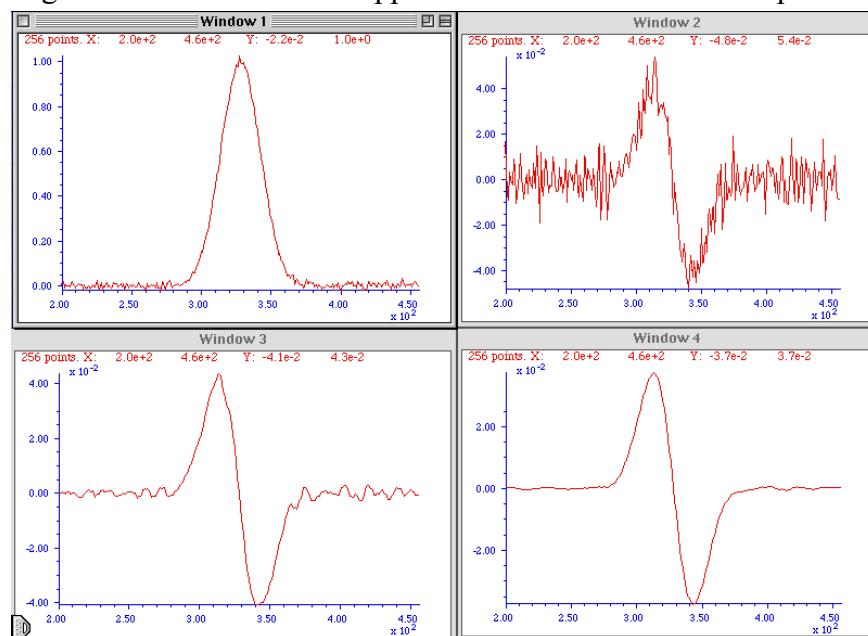
This use of signal differentiation has become widely used in quantitative spectroscopy, particularly for quality control in the pharmaceutical industry. In that application the analyte would typically be the active ingredient in a pharmaceutical preparation and the background interferences might arise 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 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 can also perform many of these quantitative measurement tasks quite well (page 259), 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 34) 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 actually be *greater* than the unsmoothed original. For the successful application of differentiation in quantitative analytical applications, it is

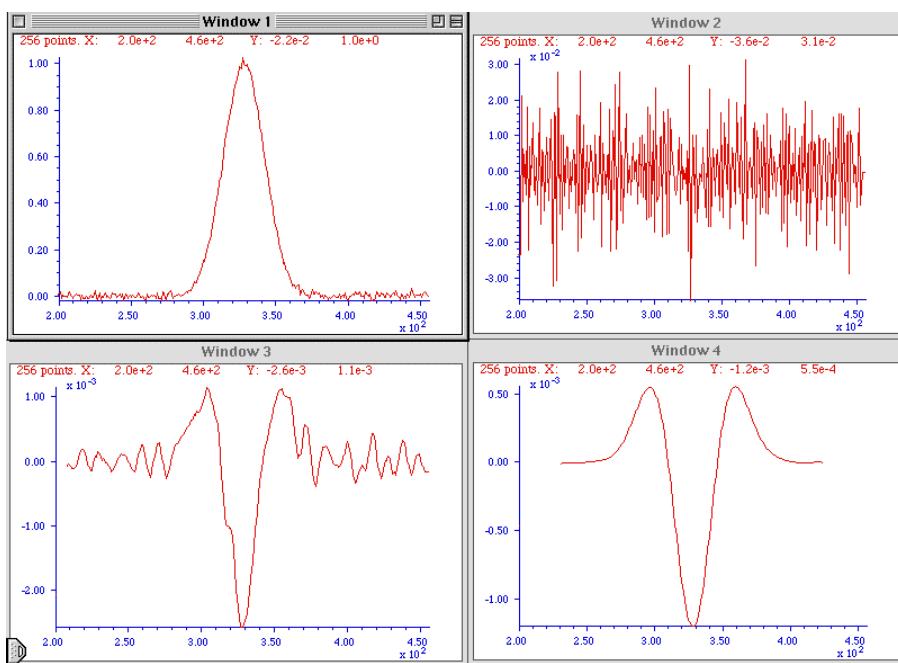
essential to use differentiation in combination with sufficient smoothing, in order to optimize the signal-to-noise ratio. This is illustrated in the figure on the left. (Matlab/Octave code for this figure.) Window 1 shows a Gaussian band with a small amount of added white noise. Windows 2, 3, and 4, 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 302).



What is particularly interesting about the noise in these derivative signals, however, is their "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 differentiated signal is easily reduced greatly by *smoothing*, as demonstrated above.

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 values of smooth ratio for derivative signals is approximately 0.5 to 1.0. For a first derivative, *two* applications of a simple rectangular 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 is adequate. The general rule is: for the n^{th} derivative, use at least $n+1$ applications of a rectangular smooth. (The Matlab signal processing program *iSignal*, discussed on page 323, automatically provides the desired type of smooth for each derivative order).

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 an adaptive segmented smooth (page 295), which makes the smooth width 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 won't be a problem in *quantitative analysis* applications, as long as 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 386). As long as 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's 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 discussed 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 actually 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.

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.

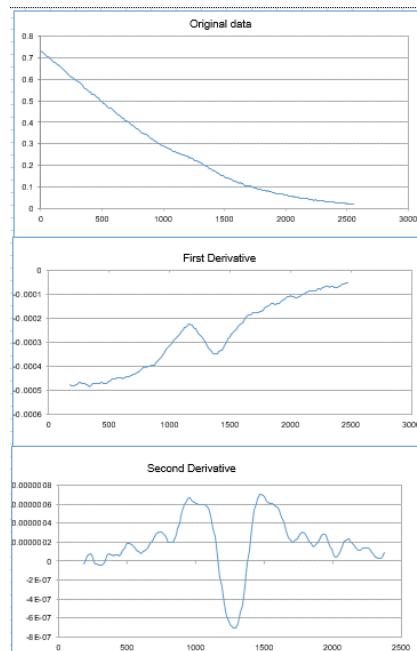
The second video, 17-second, 1.1 MByte, (DerivativeBackground2.wmv) demonstrates the measurement of a weak peak buried in a strong sloping background. At the beginning of this brief video, the amplitude (Amp) of the peak is varied between 0 and 0.14, but the background is so strong that the changes in peak amplitude, actually located at $x = 500$, is 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.

SPECTRUM, (page 383) a freeware signal-processing application for Macintosh OS8, includes first and second derivative functions, which can be applied successively to compute derivatives of any order.

Differentiation in Spreadsheets

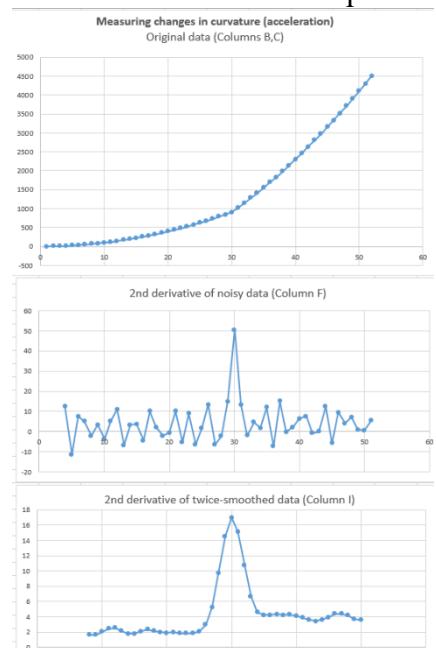
Differentiation operations such as described above can readily be performed in spreadsheets such as Excel or OpenOffice Calc. Both the derivative and the required smoothing operations can be performed by the shift-and-multiply method described in the chapter on smoothing (page 34). In principle, it is possible to combine any degree of differentiation and smoothing into one set of shift-and-multiply coefficients (as illustrated here), but it's 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 column **A** and **B** (**X** and **Y**), rows 8 to 263.



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 at the point at which the acceleration changes (at $x=30$), and it's clear that the baseline on either side of the peak are distinctly unequal, showing the change in the acceleration before and after the peak ($y=2$ and 4 , respectively).

[DerivativeSmoothingWithNoise.xlsx](#) (left) 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](#) (right), which demonstrates locating and measuring changes in the second derivative (a measure of curvature or acceleration) of a time-changing



Differentiation in Matlab and Octave

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

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). A alternative approach is to use the first derivative to find all the maxima by locating the points of 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:

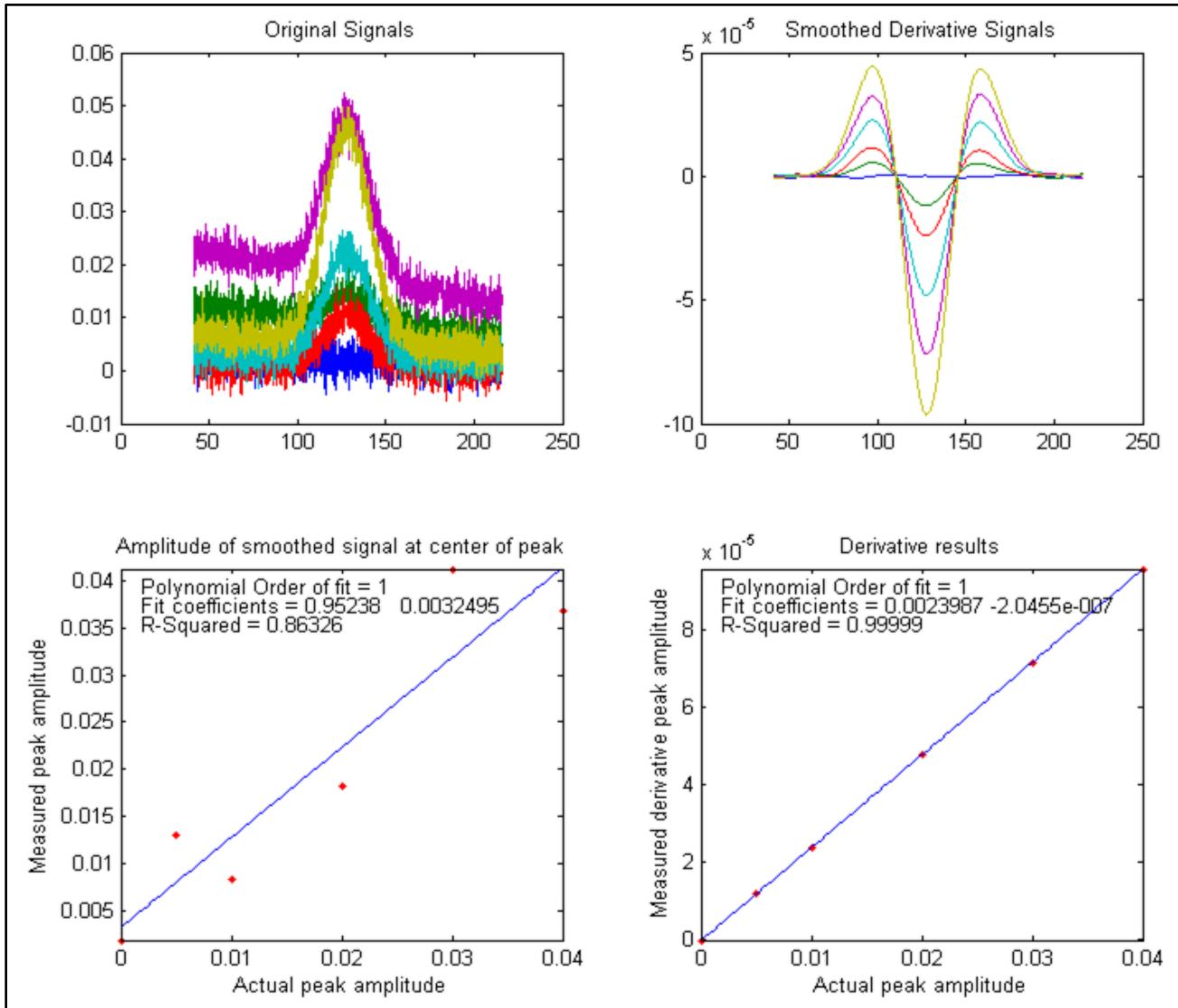
```
d=derivxy(x,y);
for j=1:length(x)-1
    if sign(d(j))>sign(d(j+1))
        disp([x(j) y(j)])
    end
end
```

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" function:

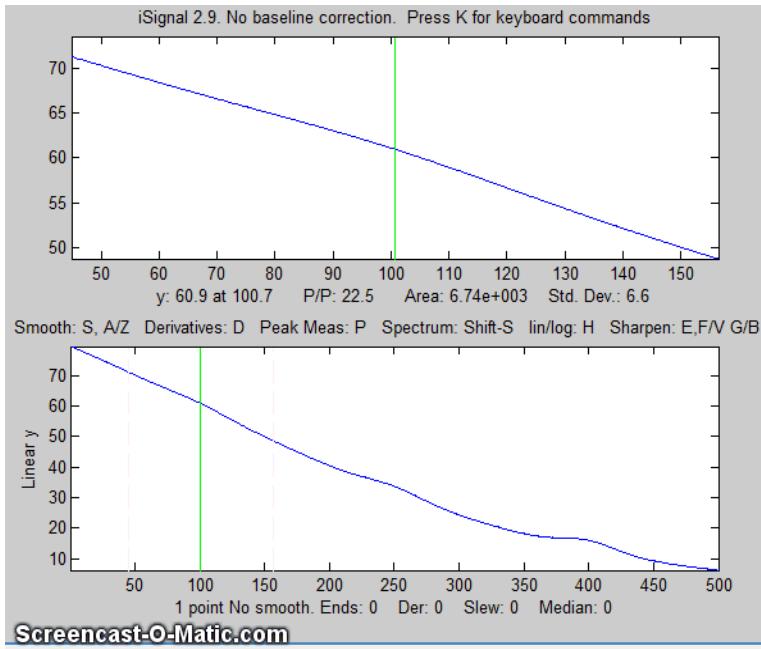
```
interp1([d(j) d(j+1)], [x(j) x(j+1)], 0)
```

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, SlewRate, MedianWidth)**

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 effected 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, despite the fact that 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 effected by the background instability.



iSignal.m (page 323), 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. (Version 2 of iSignal, November 2011, computes derivatives with respect to the x-axis vector, correcting for non-uniform x-axis intervals). The interactive keypress operation works even if you run [Matlab in a web browser](#), but not on [Matlab Mobile](#) or in Octave.

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);
```

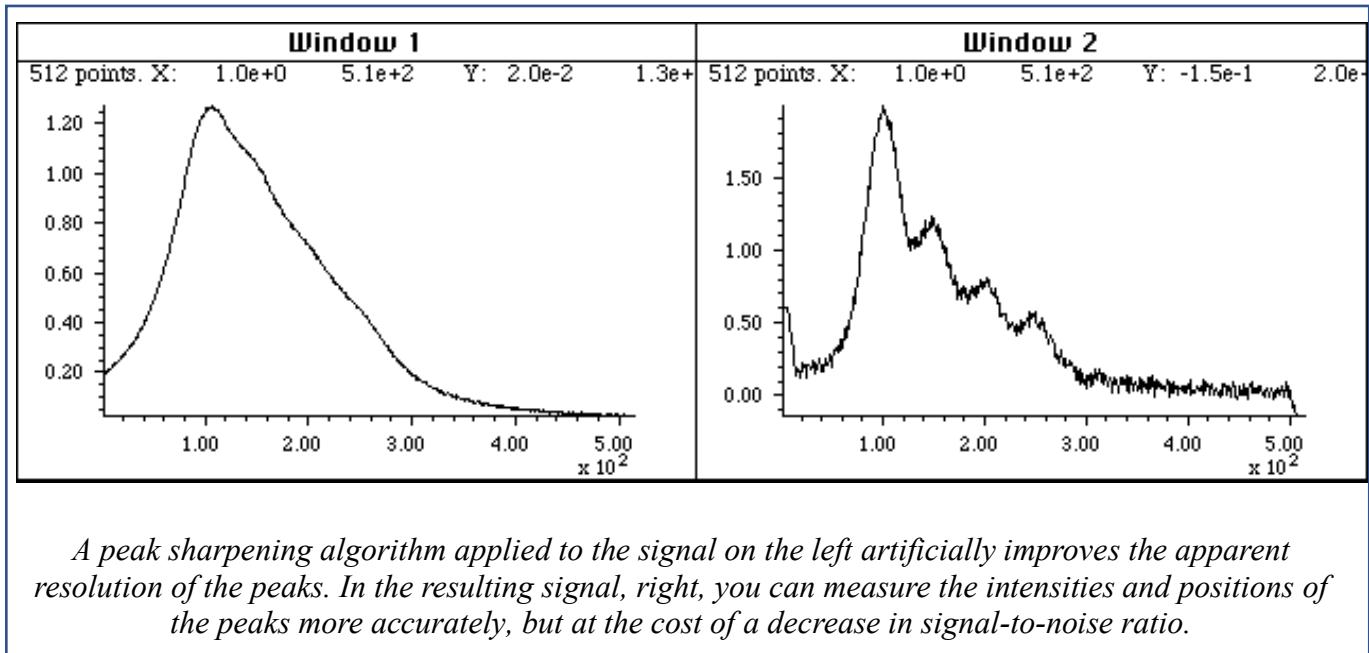
The signal is mostly blue noise (because of the differentiated white noise) unless you smooth it considerably. Use the **A** and **Z** keys to increase and decrease the smooth width and the **S** key to cycle through the available smooth types. Hint: use the Gaussian smooth and keep increasing the smooth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

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.

Real-time differentiation in Matlab is discussed in on page 308.

Peak Sharpening

The figure below shows a spectrum on the left that consists of several poorly-resolved (that is, partly overlapping) bands. The extensive overlap of the bands makes the accurate measurement of their intensities and positions impossible, even though the signal-to-noise ratio is very good. Things would be easier if the bands were more completely resolved, that is, if the bands were narrower.



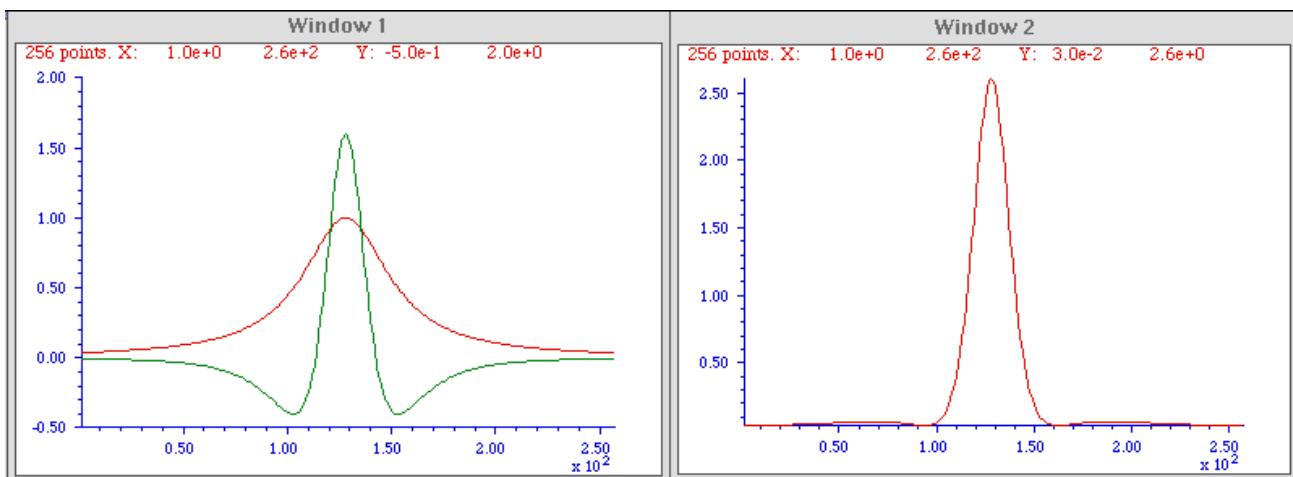
Even derivative sharpening

The technique used here, called *peak sharpening or resolution enhancement*, uses algorithms to artificially improve the apparent resolution of the peaks. One of the simplest such algorithms computes the weighted sum of the original signal and the negative of its second derivative:

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

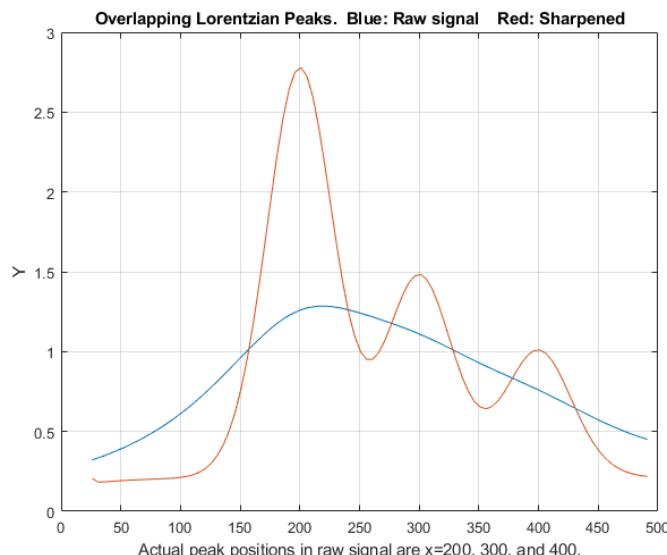
where R_j is the resolution-enhanced signal, Y is the original signal, Y'' is the second derivative of Y , and k_2 is a user-selected 2nd derivative weighting factor. It is left to the user to select the weighting factor k_2 which gives the best trade-off between the extent of sharpening, signal-to-noise degradation, 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 34), but at the expense of reducing the sharpening. Nevertheless, this technique will be *useful only if the overlap of peaks rather than the signal-to-noise ratio is the limiting factor*.

Here's 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).

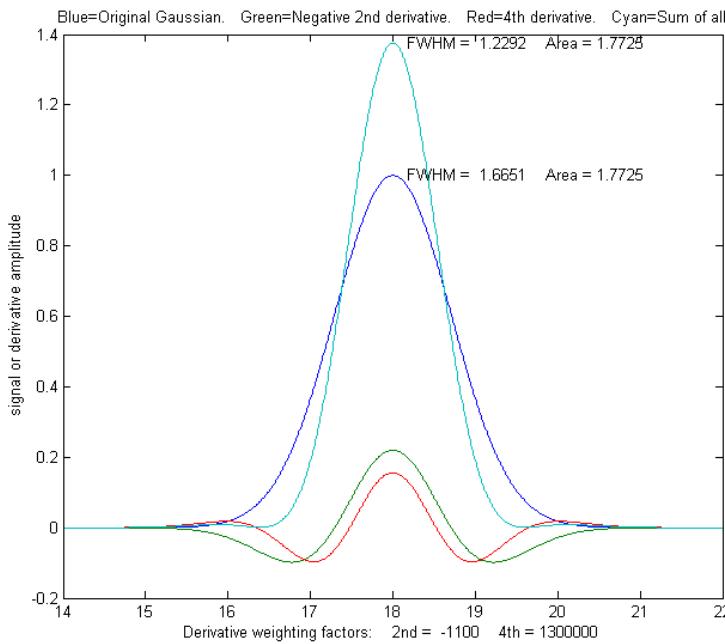


The second derivative is amplified (by multiplying it by an adjustable constant) so that the negative sides of the inverted second derivative (from approximately $X = 0$ to 100 and from $X = 150$ to 250) are a mirror image of the sides of the original peak over those regions. In this way, when the original 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, is a substantial (about 50%) reduction in the width, and a corresponding increase in height, of the peak. This effect is most dramatic with Lorentzian-shaped peaks; with Gaussian-shaped peaks, the resolution enhancement is less dramatic (only about 20 - 30%).

The reduced widths of the sharpened peaks make it easier to distinguish overlapping peaks. In the example on the right, the computer-synthesized raw signal (blue line) is actually the sum of three overlapping Lorentzian peaks at $x=200$, 300, and 400. The peaks are very wide; their halfwidths are 200, which is greater than their separation. The result is that the peaks blend together in the raw data, forming what looks like a single wide asymmetrical peak (blue line) with a maximum at $x=220$. However, the result of the even derivative sharpening algorithm (red line) clearly shows the underlying component peaks *at their correct positions*.



Note in the figure at the top of this page that the baseline of either side of the resolution-enhanced peak is not quite flat, because the cancellation of the original peak and the inverted second derivative is only approximate; you can select the adjustable weighting factor k_2 to minimize this effect. Peak sharpening will have little or no effect on the baseline, because if the baseline is linear, its derivative will be zero, and if it is gradually curved, its second derivative will be very small compared to that of the peak.



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. Somewhat better results can be obtained 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 . The result is a 26%

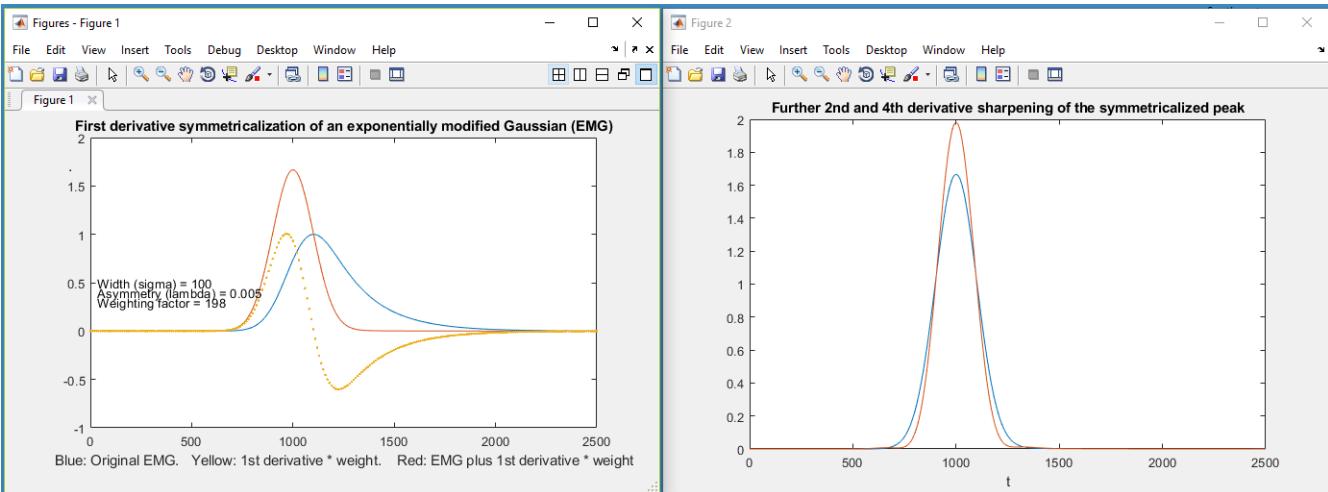
reduction in width for a Gaussian peak, as shown in the figure on the left ([Matlab/Octave script](#)), and a 60% reduction for a Lorentzian peak ([script](#)). This algorithm was used in the overlapping peak example above. (It's possible to add a sixth derivative term, but the series converges quickly and the results are only slightly improved, at the cost of increased complexity of three adjustable factors).

There is no universal optimum value for the derivative weighting factors; it depends on what you consider the best trade-off between peak sharpening and baseline flatness. However, a good place to start for a Gaussian peak are $k_2 = ((W/dx)^2)/25$ and $k_4 = ((W/dx)^4)/800$, and for Lorentzian peaks, $k_2 = ((W/dx)^2)/8$ and $k_4 = ((W/dx)^4)/700$, where W is the number of data points in the halfwidth of the peak and dx is the interval between x-axis data points. Note that the K factors for the second and fourth derivatives vary with the width raised to the 2nd and 4th power respectively, so they can vary over a very wide numerical range for peaks of different width. For this reason, if the peak widths vary substantially across the signal, it's possible to use segmented and gradient versions of this method so that the sharpening can be optimized for each region of the signal (see below).

First derivative symmetrization

The above technique works best for *symmetrical* peak shapes. 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). Take for example the "[exponentially modified Gaussian](#)" ([EMG](#)) shape. In the graphic example below, the original 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 EMG 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*.

$$S_j = Y_j + k_1 Y'$$



With the correct first derivative weighting factor, k_1 , the result is a symmetrical Gaussian with a [width](#) substantially less than that of the original EMG (orange line); in fact, it is exactly the underlying Gaussian to which the exponential convolution has been applied. The symmetrized peak S_j resulting from this 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. (Had the EMG sloped to the *left*, the *negative* of its derivative would be added). Further-more, this appears to be a general behavior and it works similarly for other peak shapes broadened by exponential convolution, such as the [Lorentzian](#). The correct first derivative weighting factor k_1 depends on the peak shape; for an exponentially modified Gaussian, it's equal to the exponential time constant τ ($1/\lambda$ in some formulations of the EMG), and for an exponentially broadened Lorentzian (EML) it's about 5 times τ . But because neither the exact peak shape nor the value of τ are known exactly in advance when working with real experimental data, in practice k_1 must be determined experimentally, just as for the even-derivative methods, which is most easily done for a single isolated peak. Put simply, if you get k_1 too low, the resulting peak will remain asymmetrical; if too high, the result will dip below the baseline. So it's easy to determine the optimal value experimentally; just increase it until the processed peak dips below the baseline after the peak, then reduce it until the baseline is flat, as shown in the GIF animation at this [link](#). For an isolated peak, measuring the ratio of the leading edge and trailing edge slopes is also a useful guide; it will be -1.000 when the peak becomes symmetrical.

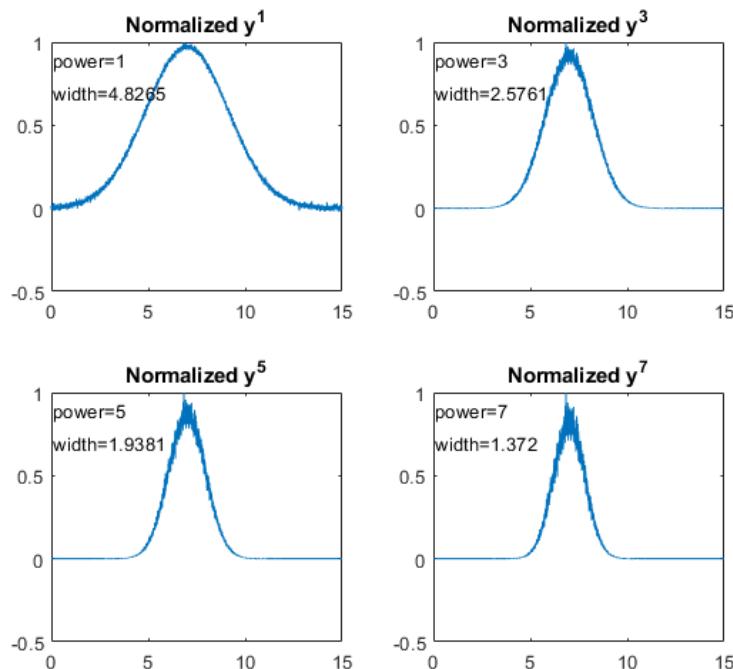
A useful property of all these derivative addition algorithms is that they do not change the *total* area under the peaks because the *total* area under the curve of *any* derivative of any peak-shaped signal is *zero* (the area under the negative lobes cancels the area under the positive lobes). Therefore, these techniques can be helpful in measuring the areas under overlapped peaks (page 109). However, a remaining problem is that the baseline on either side of the sharpened peak is *not always 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 Lorentzian peak, about 80% of the area of the peak ([graphic link](#)) is contained in the central maximum, and for a Gaussian peak, over 99% ([graphic link](#)) of the area of the peak is contained in the central maximum.

Because differentiation and smoothing are both [linear techniques](#), the [superposition principle](#) applies and the amplitude of a sharpened signal is directly proportional to the amplitude of the original signal,

which allows quantitative analysis applications employing any of the standard calibration techniques (page 386). As long as you apply the same signal-processing techniques to the standards as well as to the samples, everything works.

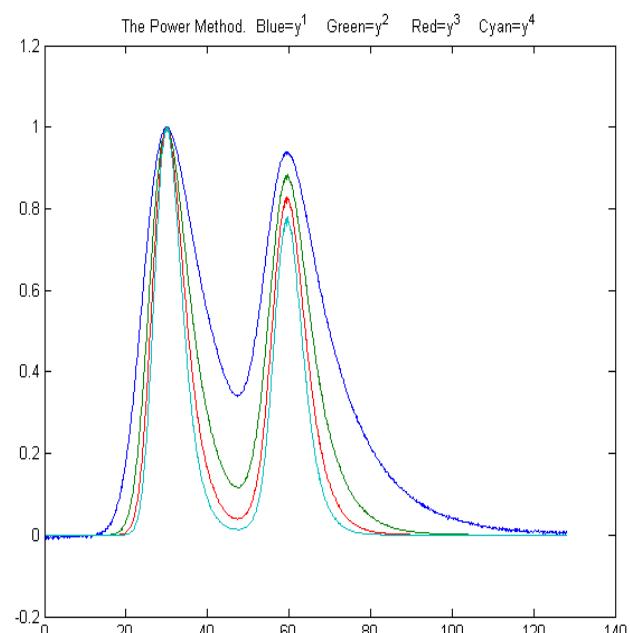
Peak sharpening can be useful in automated peak detection and measurement (page 198) to increase the ability to detect weak overlapping peaks that appear only as shoulders in the original signal. Click for [animated example](#). Peak sharpening can also be useful before [measuring the areas](#) (page 113) under overlapping peaks, because it's easier and more accurate to measure the areas of peaks that are more completely separated.

The Power Law Method



the left, which plots noisy Gaussians raised to the power $p=1$ to 7, 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 width) is increased. In the figure on the right, the blue line shows two slightly overlapping 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 peak widths, measured with the [halfwidth.m](#) function, are 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 derivative method discussed above.

An even simpler method for peak sharpening involves simply raising each data point to a power n greater than 1 ([reference 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. The result is a more nearly Gaussian peak shape (because most peak shapes are locally Gaussian near the peak maximum) and a *smaller peak width*, reducing the width of a Gaussian by the square root of the power). The technique is demonstrated by the Matlab/Octave script [PowerLawDemo.m](#), shown in the figure on



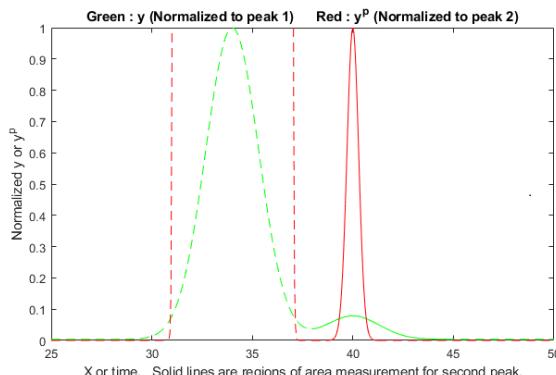
However, there are some limitations of the power law method:

- (a) It only works if the peaks of interest make a distinct maximum (it's 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 34, can help).

For Gaussian peaks, the area under the original peak can be calculated from the area under the normalized power-sharpened curve ([reference 63](#)).

Of course, this method introduces *severe non-linearity* into the signal, changing the ratios between peak heights (as is evident in the figure) and complicating further processing, especially quantitative measurement calibration. To correct 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, notably, *not the slope*) of the calibration [curves](#) used in quantitative analytical measurements (page 386). 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 n^{th} power of the original height, whereas 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 n^{th} power of the original height. This is demonstrated quantitatively by the simple Matlab/Octave script [PowerLawCalibrationDemo.m](#) ([graphic](#)) and by the self-contained Matlab/Octave function [PowerTransformCalibrationCurve.m](#) which takes the *power*, n - any positive number greater than 1 - as its single input argument (for example `PowerTransformCalibrationCurve (3)` will demonstrate the 3rd power).

The self-contained function [PowerMethodDemo.m](#) demonstrates the power method for measuring the



area of small shoulderering peak that is partly overlapped by a much stronger interfering peak ([left](#)). It shows the effect of random noise, smoothing, and any uncorrected background under the peaks.

Deconvolution. Another signal processing technique that can increase the resolution of overlapping peaks is *deconvolution* (page 96). It is applicable in the situation 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 underlying peaks shape.

SPECTRUM, page 383, the freeware signal-processing application for Mac OS8 and earlier, includes this resolution-enhancement algorithm, with adjustable weighting factor and derivative smoothing width.

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 is also [versions that has clickable ActiveX buttons](#) (detail on left) for convenient interactive

Click buttons to change K1 and K2					
K1=	8761.4	-- K1	- K1	+ K1	++ K1
K2=	1.02E+07	-- K2	- K2	+ K2	++ K2
Rs=	0.625				

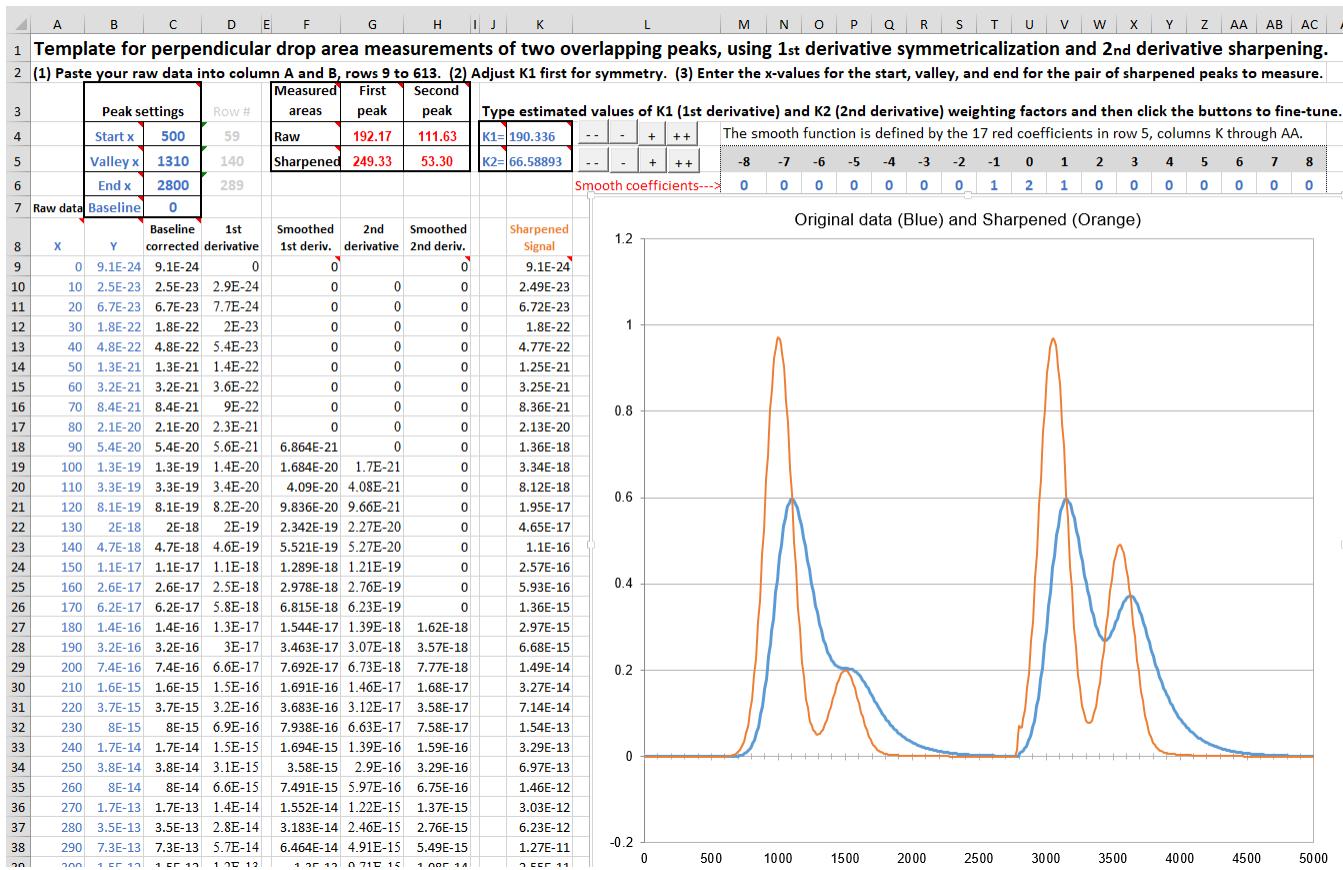
adjustment of the K1 and K2 factors by 1% or by 10% for each click. You can type in first estimates for K1 and K2 directly into cells J4 and J5 and then use the buttons to fine-tune the values. (Note: ActiveX buttons do not work in the iPad version of Excel). If the signal is noisy, adjust the smoothing using the 17 coefficients in row 5 columns **K** through **AA**, just as with the smoothing spreadsheets (page 45).

There is also a 20-segment 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 ([GradientPeakSharpeningDeriv.xlsx](#)) and an example with 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.

The template [PeakSymmetricalizationTemplate.xlsxm](#) (screen image on next page) performs symmetrization of exponentially modified Gaussians (EMG) by the weighted addition of the first derivative. An example application with sample data already typed in ([PeakSymmetricalizationExample.xlsxm](#)) 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.xlsxm](#) ([graphic](#)). These spreadsheets also allows further second derivative sharpening of the resulting symmetrical peak.

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



The spreadsheet template "PeakSymmetrizationTemplate.xlsx" is shown measuring the areas of the first of two pairs of overlapping asymmetrical peak after applying first-derivative symmetrization.

Peak Sharpening for Matlab and Octave

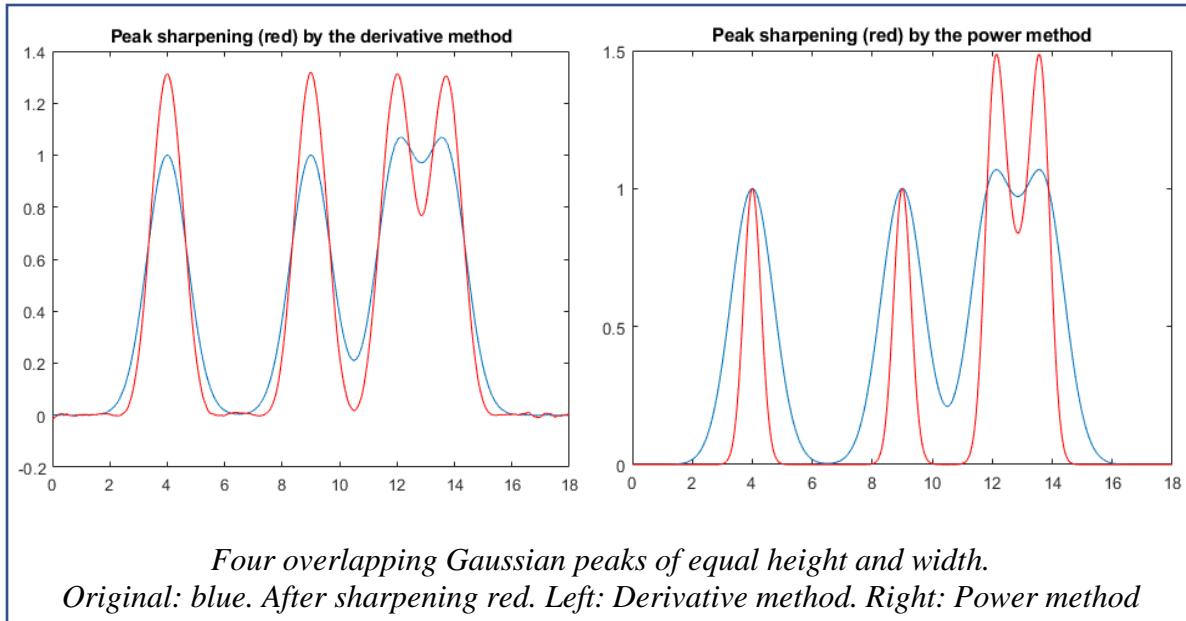
The custom Matlab/Octave function [enhance.m](#) has the form `Enhancedsignal = enhance(signal, k1, k2, SmoothWidth)`, where "signal" is the original signal vector, the arguments k_2 and k_4 are 2nd and 4th derivative weighting factors, and `SmoothWidth` is the width of the built-in `smooth`. The resolution-enhanced signal is returned in the vector "Enhancedsignal". Click on this link to inspect the code, or right-click to download for use within Matlab or Octave. The values of k_1 and k_2 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 $\text{PeakWidth}^2/25$ and for k_4 is $\text{PeakWidth}^4/800$ (or $\text{PeakWidth}^2/6$ and $\text{PeakWidth}^4/700$ for Lorentzian peaks), where `PeakWidth` is the full-width at half maximum of the peaks *expressed in number of data points*. Because sharpening methods are typically sensitive to random noise in the signal, it's usually necessary to apply smoothing: the Matlab/Octave [ProcessSignal.m](#) function allows both sharpening and smoothing to be applied in one function.

Here's a simple 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);
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')

```

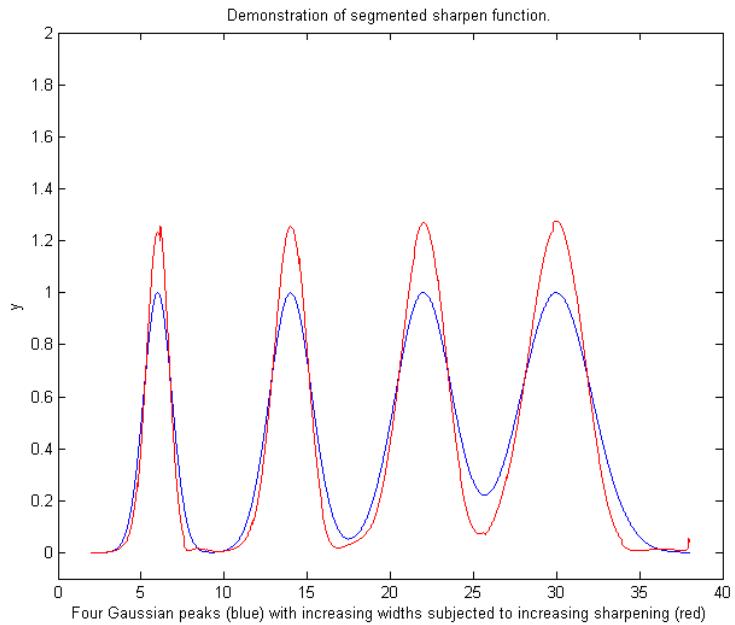


The power method (right) is effective as long as there is a valley between the overlapping peaks, but it introduces non-linearity, which must be corrected later, whereas the derivative method preserves the original peak areas and the ratio between the peak heights. [PowerLawCalibrationDemo](#) demonstrates 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 111), and then taking the 1/n power of the measured areas ([graphic](#)).

The symmetrization of asymmetric peaks by the weighted addition of the first derivative is performed by the function [symmetrize](#)(t, y, factor, smoothwidth, type, ends) and demonstrated for the exponentially modified *Gaussians* (EMG) by the self-contained Matlab/Octave function [EMGplusfirstderivative.m](#) and for an exponentially modified *Lorentzian* (EML) by [EMLplusfirstderivative.m](#). In both of these, Matlab's Figure window 1 shows the symmetrization and window 2 shows the [additional 2nd and 4th derivative sharpening](#). Both of these routines report the before and after halfwidth and area of the peak, and they measure the resulting symmetry of the processed peak two ways: (a) by the ratio of the leading edge and trailing edge slopes (ideally -1.000) and (b) by the R2 of the least-squares fit to the initial Gaussian and Lorentzian shapes peak shape

before exponential broadening respectively (ideally 1.000, see page 141).

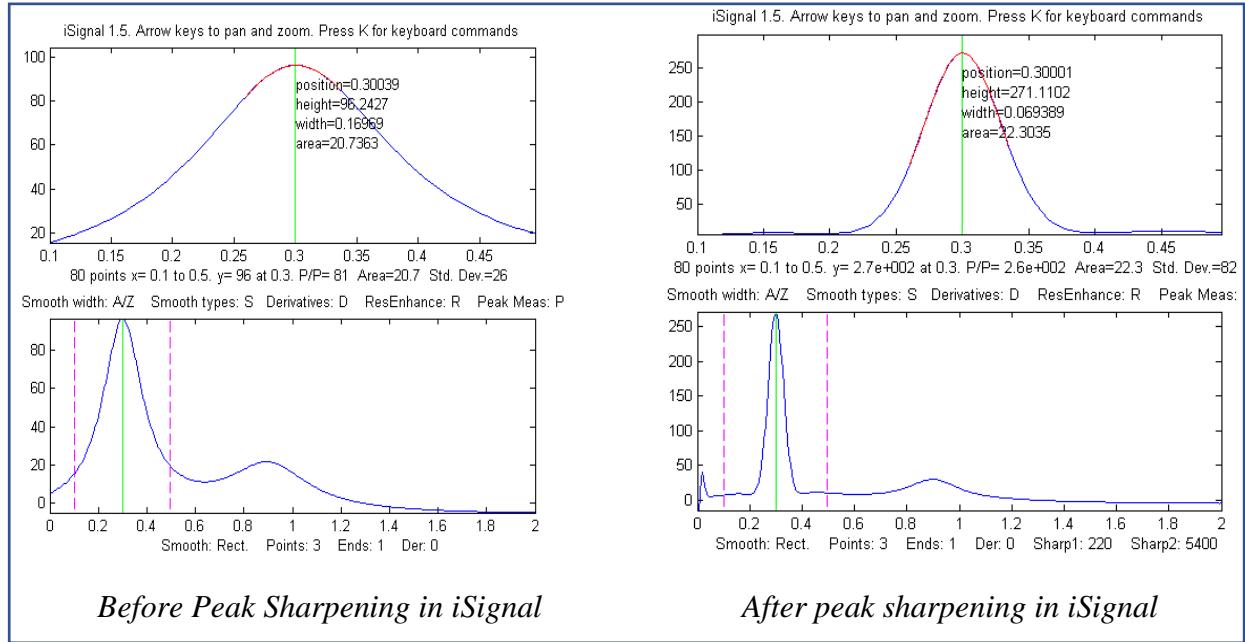
Segmented derivative peak sharpening. If the peak widths vary substantially across the signal, you can use the *segmented* version [SegmentedSharpen.m](#), for which the input arguments factor1, factor2, and SmoothWidth are *vectors*. The script [DemoSegmentedSharpen.m](#), shown on the right, uses this 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%](#) compared to the original. [DemoSegmentedSharpen2.m](#) shows four peaks of the *same* width sharpened to increasing degrees.



[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. The syntax is

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

[iSignal](#) (page 323) is a Matlab function that performs peak sharpening for time-series signals, using the above enhance function, with keystrokes that allow you to adjust the 2nd and 4th 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* calculates 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 your 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. (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. In such cases, you can use the segmented sharpen function, [SegmentedSharpen.m](#)).



iSignal 5.95 and later can also use the power transform method (press the \wedge 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 216), a Matlab interactive peak detection and measurement program, has a built-in peak sharpening mode that is based on this technique. See `ipeakdemo5` on page 230.

Real-time peak sharpening in Matlab is discussed on page 308.

Video Demonstration. This 15-second, 1.7 MByte video ([ResEnhance3.wmv](#)) demonstrates the Matlab interactive peak sharpening function [InteractiveResEnhance.m](#). The signal consists of four overlapping, poorly-resolved Lorentzian bands. First, the 2nd derivative factor (Factor 1) is adjusted, then the 4th derivative factor (Factor 2) is adjusted, then the smooth width (Smooth) is adjusted, and finally the Factor 2 is tweaked again.

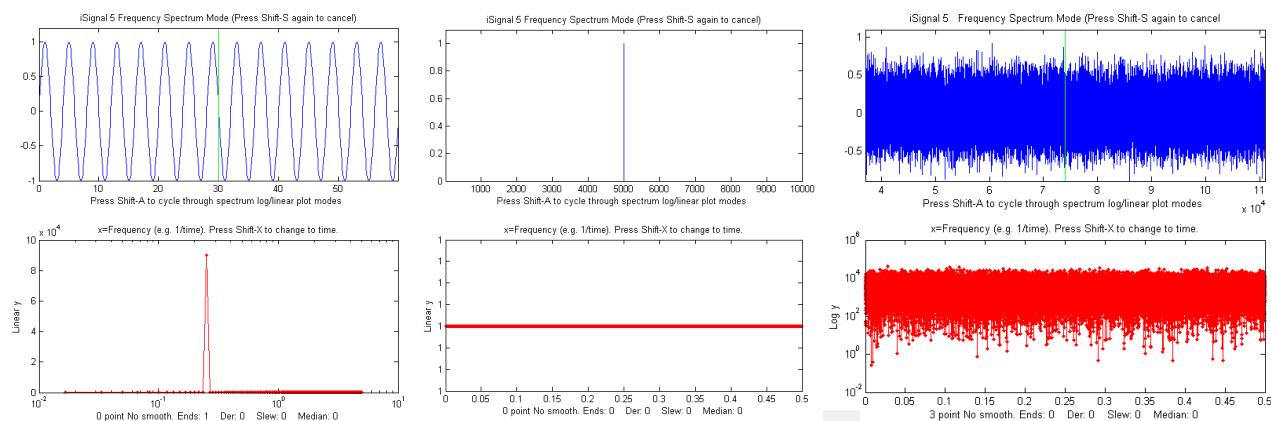
Harmonic analysis and the Fourier Transform

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. Actually, 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 $0, 1, 2, 3 \dots n/2$ 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 concept was originated by Carl Friedrich Gauss and by Jean-Baptiste Joseph Fourier in the early 19th century. The famous "[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 usually expressed in terms of "[complex numbers](#)", with "real" and "imaginary" parts that combine the sine and cosine (or amplitude and phase) information at each frequency onto a single complex number. Many computer languages will perform this operation automatically when the two quantities divided are complex.

The concept of the Fourier transform is involved in two very important instrumental methods in chemistry. 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 Michelson 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 the instrument recovers the spectrum by inverse Fourier transformation of the measured (interferogram or free induction decay) signal.

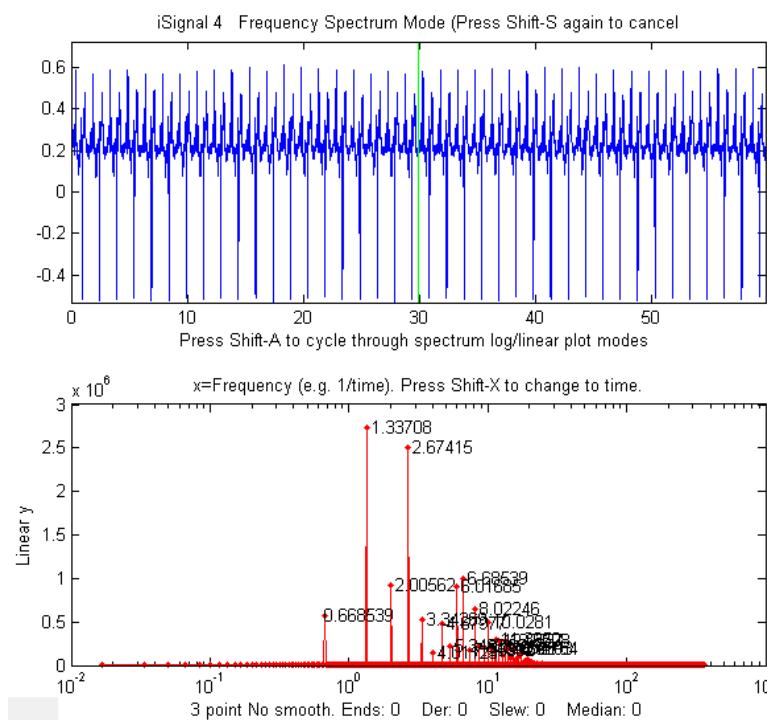
The [power spectrum](#) or *frequency spectrum* is a simple way of showing the total amplitude at each of these frequencies; it is calculated as the square root of the sum of the squares of the coefficients of the sine and cosine components. The power spectrum retains the *frequency* information but 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 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 25) the power spectra of repeated signals: this is demonstrated by the Matlab/Octave scripts [EnsembleAverageFFT.m](#) and [EnsembleAverageFTGaussian.m](#).

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. The second point corresponds to a frequency of $1/n\Delta x$ (whose period is exactly equal to the time duration of the data), the next point to $2/n\Delta x$, the next point to $3/n\Delta x$, 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 $(n/2)/n\Delta x=1/2\Delta x$, which is one-half the sampling rate. The highest frequency that can be represented in a discretely-sampled waveform is one-half the sampling frequency, which is called the [Nyquist frequency](#); frequencies above the Nyquist frequency are "folded back" to lower frequencies, severely distorting the signal. 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.

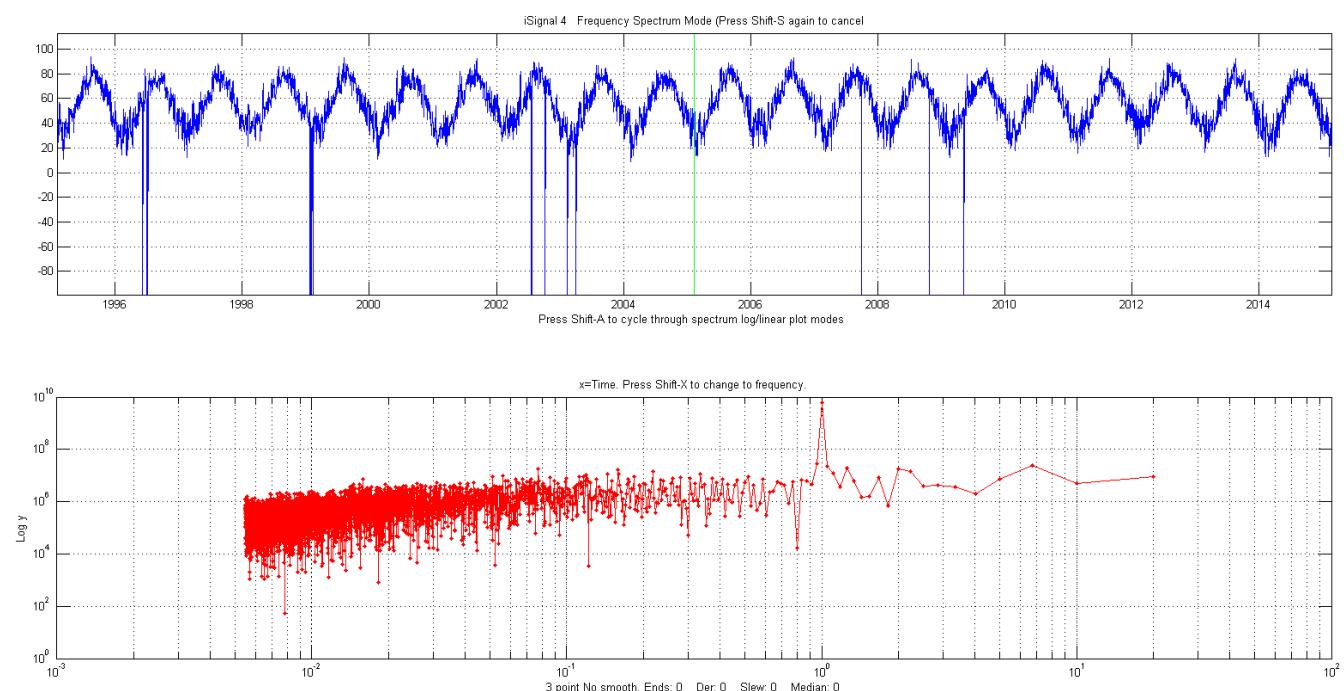


A pure sine or cosine wave that has an exactly integral number of cycles within the recorded signal will have a *single non-zero Fourier component* corresponding to its frequency (above, left). Conversely, a

signal consisting of zeros everywhere except at a single point, called a *delta function*, has *equal* Fourier components at all frequencies.(above, center), *Random noise* also has a power spectrum that is spread out over a wide frequency range. The noise amplitude depends on the *noise color* (page 26) 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). The figure on the left shows a real-data electrical recording of a heartbeat, called an [electrocardiograph](#) (ECG), which is an example of a periodic waveform that repeats over time. The smallest repeating unit of the signal is called the *period*, and



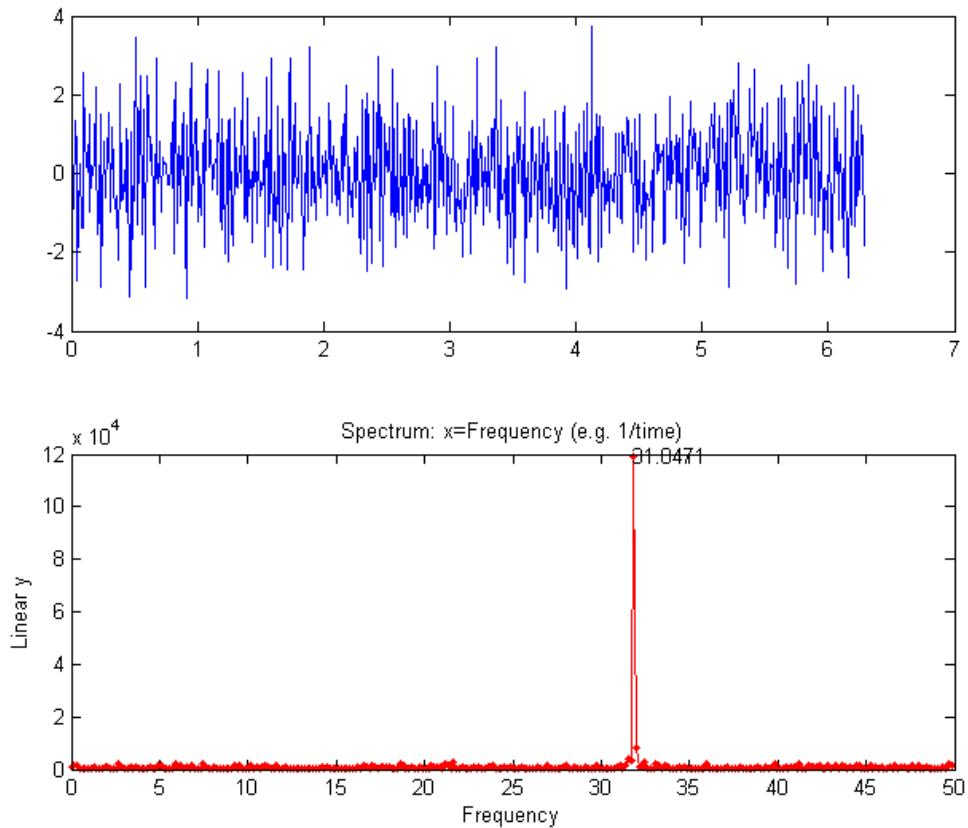
the reciprocal of that period is called the [fundamental frequency](#). Non-sinusoidal periodic waveforms like this exhibit a series of frequency components that are multiples of the fundamental frequency, which are called "harmonics". The signal shows a fundamental frequency of 0.6685 Hz with *multiple harmonics* at frequencies that are $\times 2$, $\times 3$, $\times 4$..., etc., times the fundamental frequency. The figure shows the waveform in blue in the top panel and its frequency spectrum in red in the bottom panel. The fundamental and the harmonics are sharp peaks, labeled with their frequencies. The spectrum is qualitatively similar to that for perfectly regular identical peaks ([graphic](#)). Recorded vocal sounds, especially vowels, also have a periodic waveform with harmonics ([graphic](#)). (The sharpness of the peaks in these spectra shows that the amplitude and the frequency are very constant over the recording interval in this example. 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 264.



Another familiar example of periodic change 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 above. (The negative spikes are missing data points - 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 is plotted with *time* (the reciprocal of frequency) on the x-axis (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.

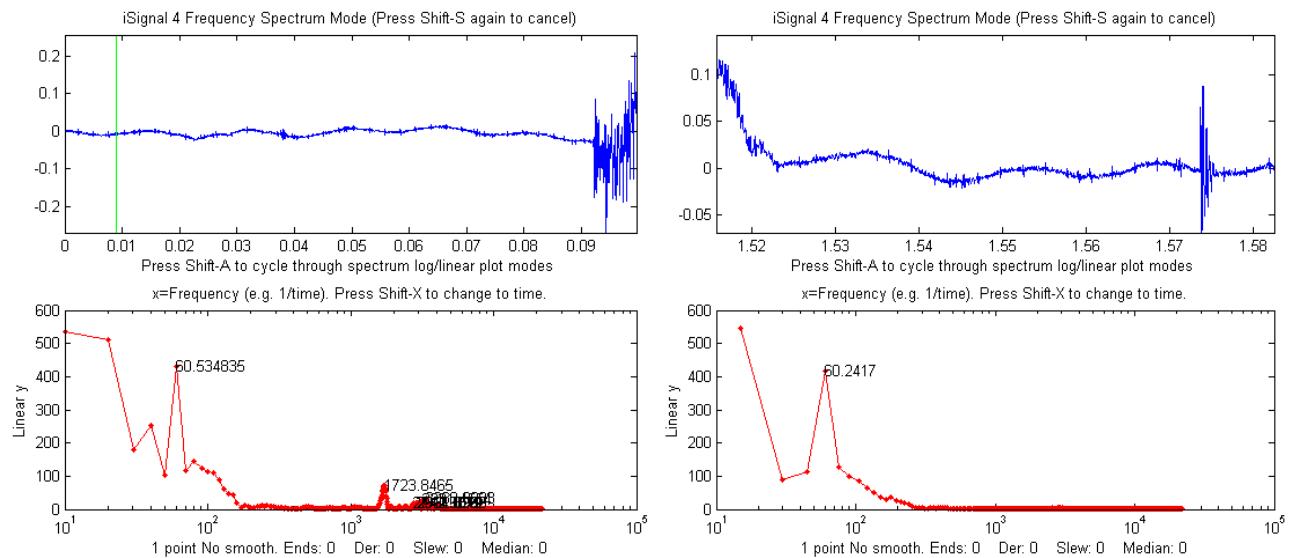
The figure below is a simulation that shows how hard it is to see a periodic component in the presence



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 the downloadable 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:

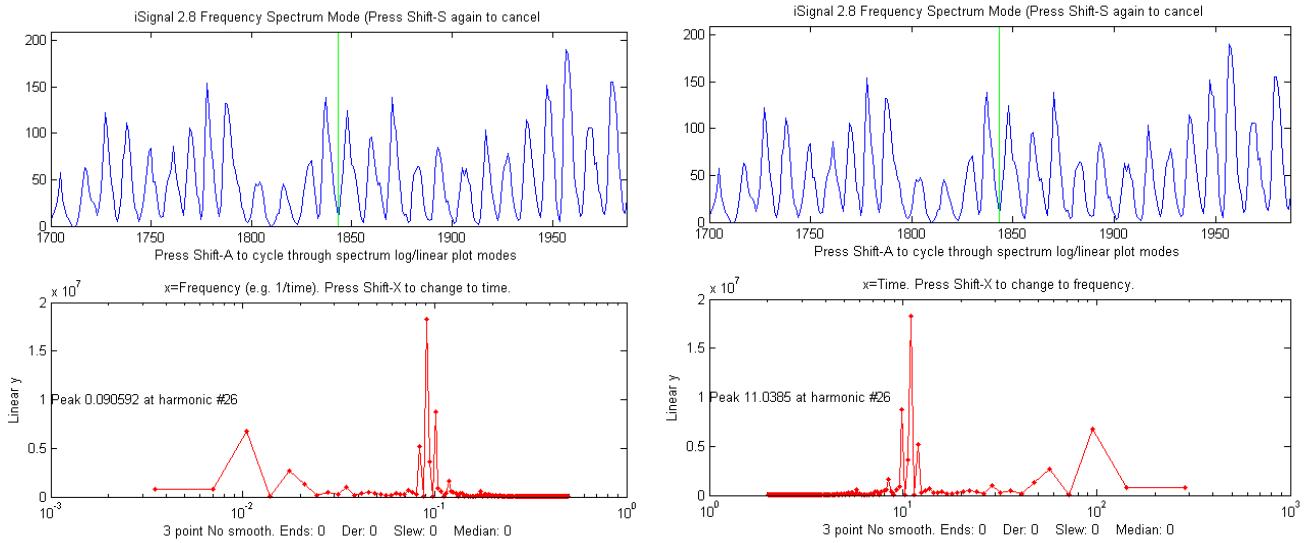
```
x=[0:.01:2*pi]';
y=sin(200*x)+randn(size(x));
subplot(2,1,1);
plot(x,y);
subplot(2,1,2);
PowerSpectrum=PlotFrequencySpectrum(x,y,1,0,1);
```

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 some 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](#).

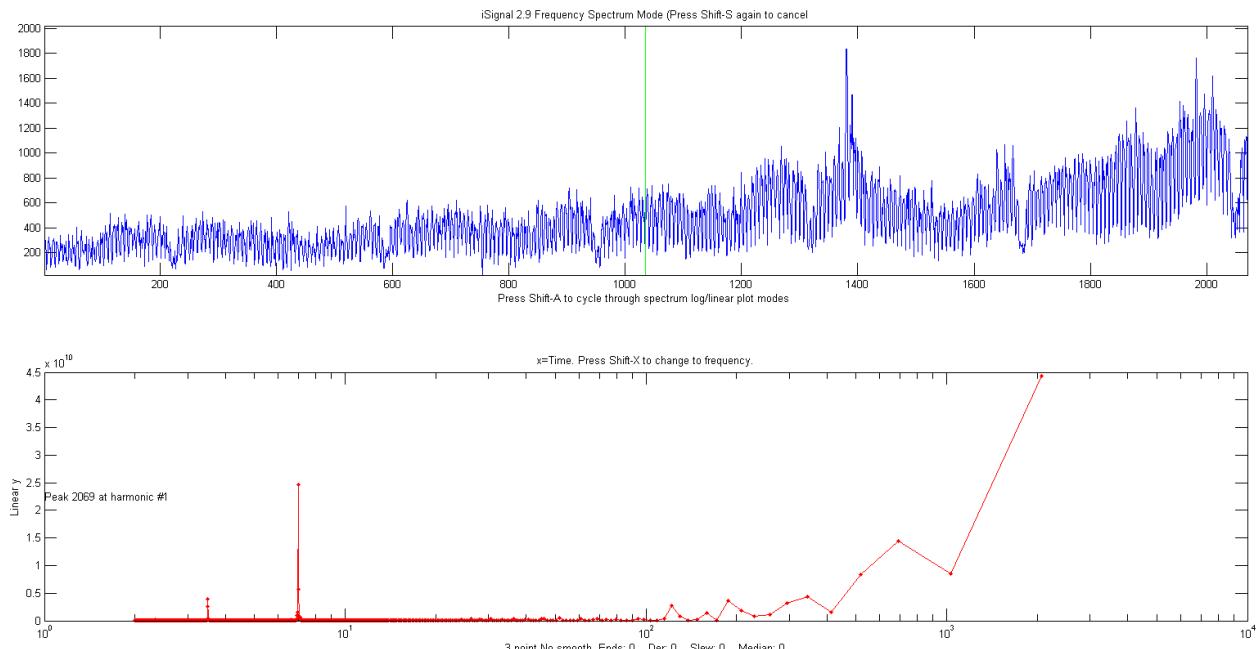


iSignal, showing data from an audio recording, zoomed in to the period immediately before (left) and after (right) the actual sound, shows a regular sinusoidal oscillation ($x = \text{time in seconds}$). In the lower panel, the power spectrum of each signal ($x = \text{frequency in Hz}$) shows a strong sharp peak very near 60 Hz, suggesting that the oscillation is caused by stray pick-up from the 60 Hz power line in the USA (it would be 50 Hz had the recording been made in Europe). 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 why 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 37).

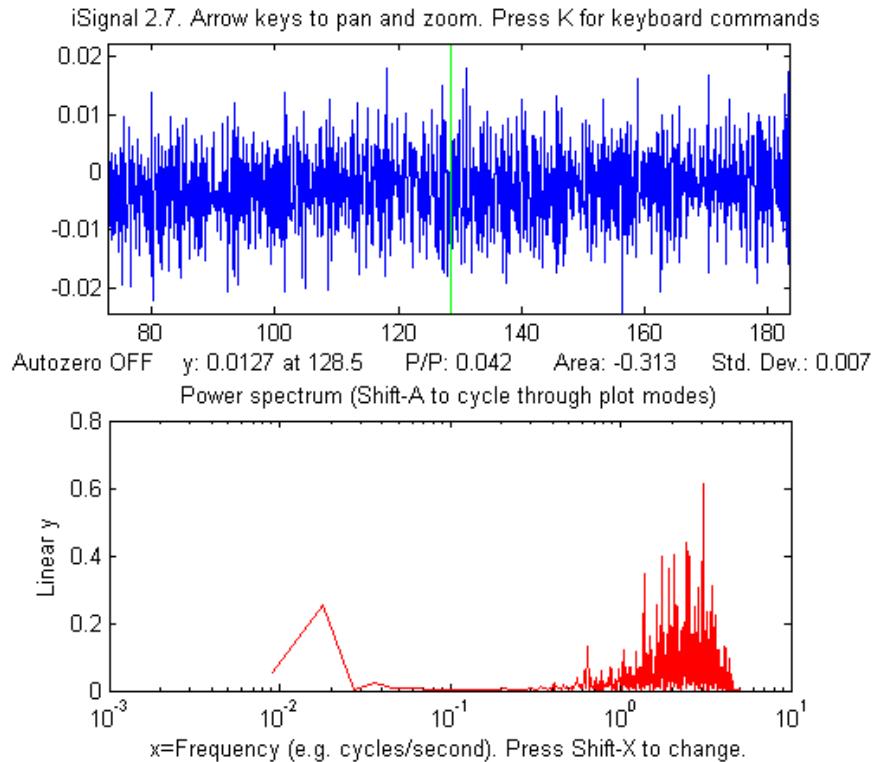


The figures above show a classic example of harmonic analysis; it shows the annual variation in the number of observed sunspots, which have been recorded since the year 1700! In this case, the time axis is in *years* (top window). A plot of the power spectrum (bottom window, left) shows a strong peak 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 the downloadable Matlab function [iSignal](#) (page 323). 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.



An example of a time series with complex multiple periodicity is the world-wide [daily page views](#) ($x=\text{days}$, $y=\text{page views}$) for [this web site](#) over a 2070-day period (about 5.5 years). In the [periodogram plot](#) (shown above) you can clearly see at 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) and at 182 days (roughly a half-year), probably 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 the entire data record, which can be thought of as a very low-frequency component whose period is much longer than the entire data record.



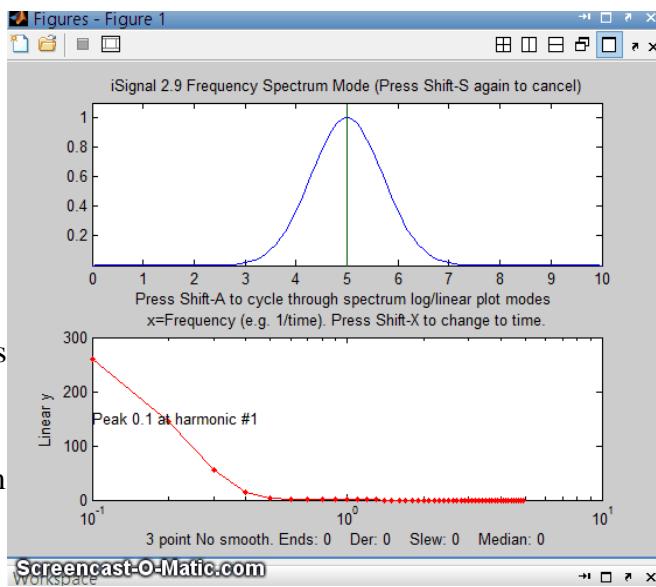
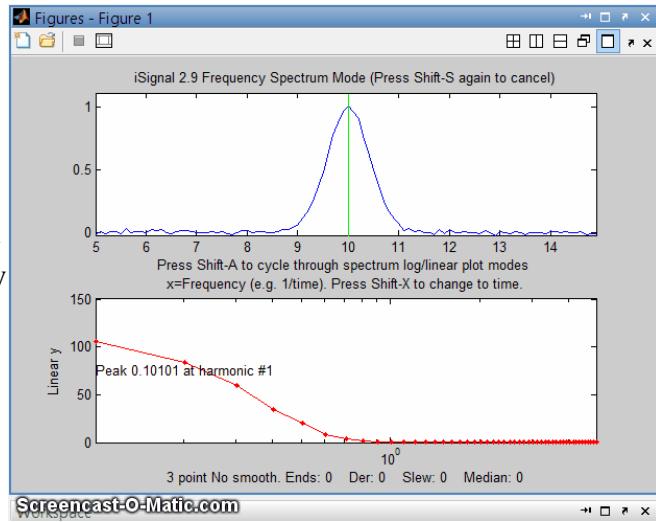
In the example shown above, the signal (in the top window) contains no visually evident periodic components; it *seems* to be just random noise. However, the frequency spectrum (in the bottom window) shows that there is much more to this signal than meets the eye. There are two major frequency components: 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 particular case, the *lower* frequency component is in fact the *signal*, and the frequency component is residual *blue noise* left over from previous signal processing operations. The two components are fortunately well separated on the frequency axis, suggesting that low-pass filtering (i.e. smoothing) will be able to remove the noise without distorting the signal.

In the examples shown above, the signals are time-series signals with *time* as the independent variable. More generally, it is also 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 you are reading this online, click this [link](#) to see this figure animated). If that signal 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 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 on the right; click for [GIF animation](#)) has most of its power concentrated in just a few low frequencies, but as successive orders of differentiation are applied, 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.

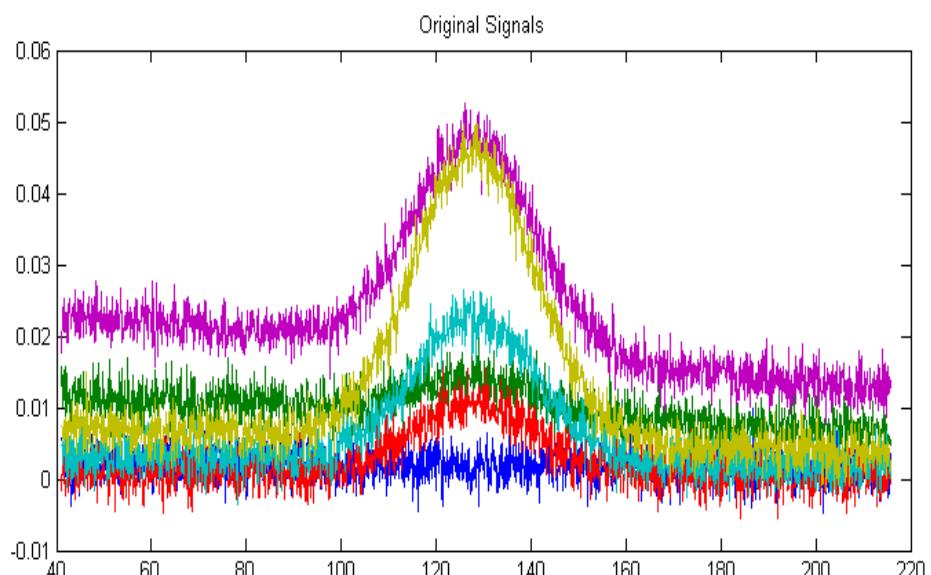


[Peak sharpening](#) (page 69) also emphasizes the high frequency components by adding a portion of the second and fourth derivatives to the original signal. You can see this clearly in the Matlab/Octave script [PeakSharpeningFrequencySpectrum.m](#), which shows the frequency spectrum of the original and sharpened version of a signal consisting of several peaks ([graphic](#)).

[SineToDelta.m](#). A demonstration animation ([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](#)).

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 65). In the set of six original signals, shown on the right, the random noise occurs mostly in a high frequency range, with *many cycles* over the x-axis range, and the baseline shift occurs mostly in 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, we could predict that a quantitative measure based on differentiation and smoothing might work well.

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 341) illustrates this idea using a brief recording of speech. Interestingly, different degrees of smoothing and differentiation will change [timbre](#) of the voice but has *little effect on the intelligibility*; 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, 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 typically terrible [telephone "hold" music](#).

Software details

In a spreadsheet or computer language, a sine wave can be described by the 'sin' function $y=\sin(2\pi f x + p)$ 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 a 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 <http://www.bowdoin.edu/~rdelevie/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 a number of dedicated FFT spectral analysis programs, including **ScopeDSP** (<https://iowegian.com/scopedsp/>) and **Audacity** (<http://sourceforge.net/projects/audacity/>).

In the PDF version of this book, you can Ctrl-Click these links to open these sites automatically.

[SPECTRUM](#), page 383, the freeware signal-processing application for Macintosh OS8, includes a power spectrum function, as well as forward and reverse Fourier transformation.

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 we compute the Fourier transform of a simple 3-element vector, we get 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 remainder are 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 of a signal vector "s" can be computed as `real(sqrt(fft(s)) .* conj(fft(s))))`. Here's a simple example where we know the answer in advance, at least

qualitatively: an 8-element vector of integers that trace out a *single cycle of a sine wave*:

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

The frequency spectrum in this case is [0 39.9 0 0.201 0 0.201 0 39.9]. 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 it the *second* element (39.8) that is the largest by far, which is just what we 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 represent one cycle, e.g. [0 +1 0 -1].

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

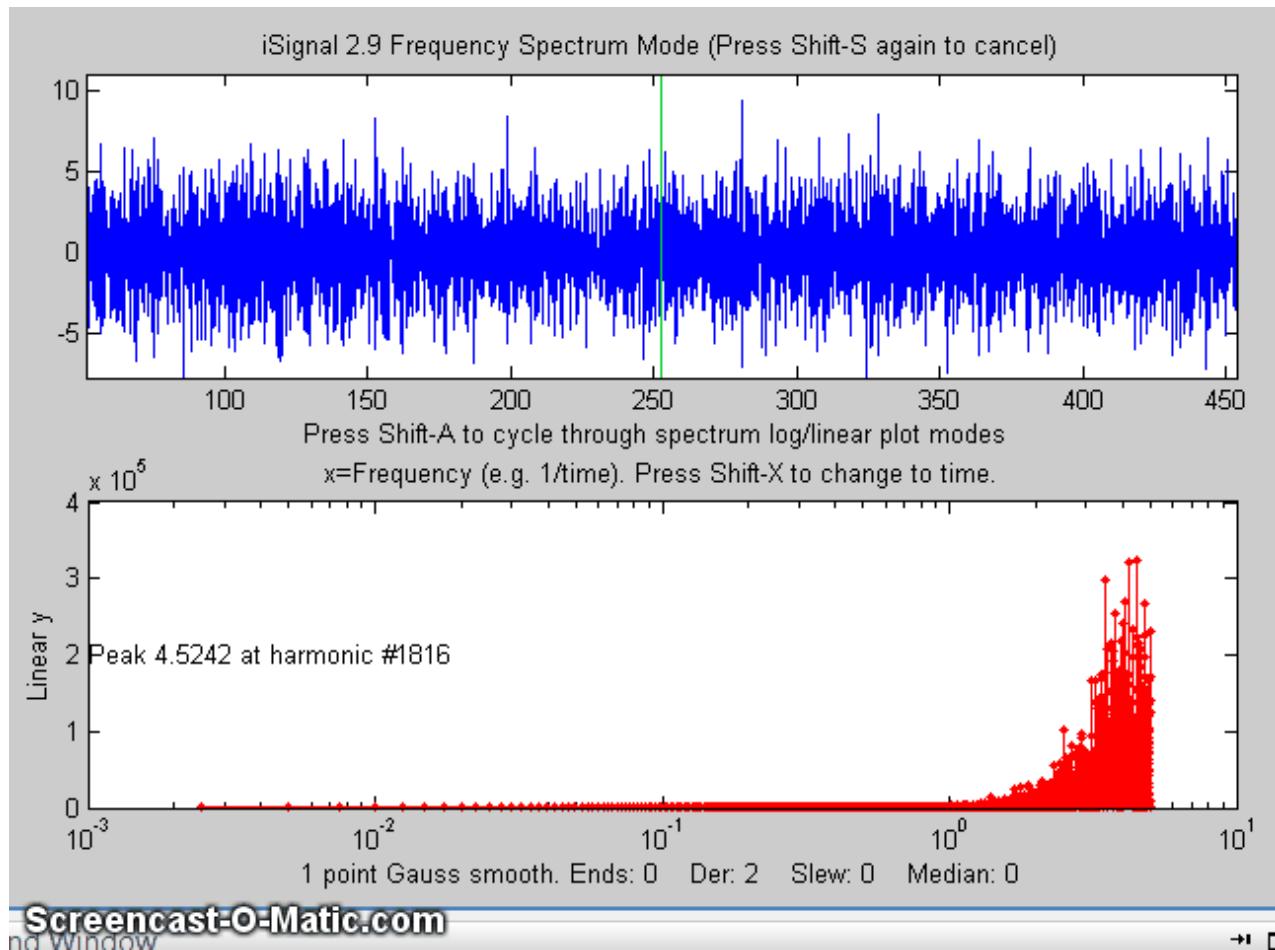
```
x=[0:.01:2*pi]';  
f=25; % Frequency  
y=sin(2*pi*f*x)+randn(size(x));  
subplot(2,1,1);  
plot(x,y);  
subplot(2,1,2);  
FS=PlotFrequencySpectrum(x,y,1,0,1);
```

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 example, see [these examples](#). A “[Slow Fourier Transform](#)” function has also been [published](#); it is *8000 times slower* with a 10,000-point data vector, as can be shown by this bit of code: `y=cos(.1:.01:100); tic; fft(y); ffttime=toc; tic; sft(y); sftime=toc; TimeRatio=sftime/ffttime`

Observing Frequency Spectra with [iSignal](#)

iSignal (page 323) is a multi-purpose interactive signal processing tool that has 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.



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 results in a clearer graph than the linear modes. You can also press **Shift-X** to toggle the x axis between *frequency* and *time*. The figure above shows the *frequency mode*, as the smooth width is varied with the **A** and **Z** keys. This shows dramatically how the frequency are both affected by smooth width. Click for [GIF animation](#).

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. This is demonstrated by the animation on the right, which show 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; with enough smoothing, the three weak peaks are clearly visible (in derivative form) and measurable. Details and instructions are on page 323. You can download a [ZIP file](#) that contains iSignal.m and some demos and sample data for testing.

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.

Demonstration that the Fourier frequency spectrum of a Gaussian is also a Gaussian:

One special thing about the *Gaussian* signal shape is that the Fourier frequency spectrum of a Gaussian is *also* a Gaussian. You can demonstrate this to yourself 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:

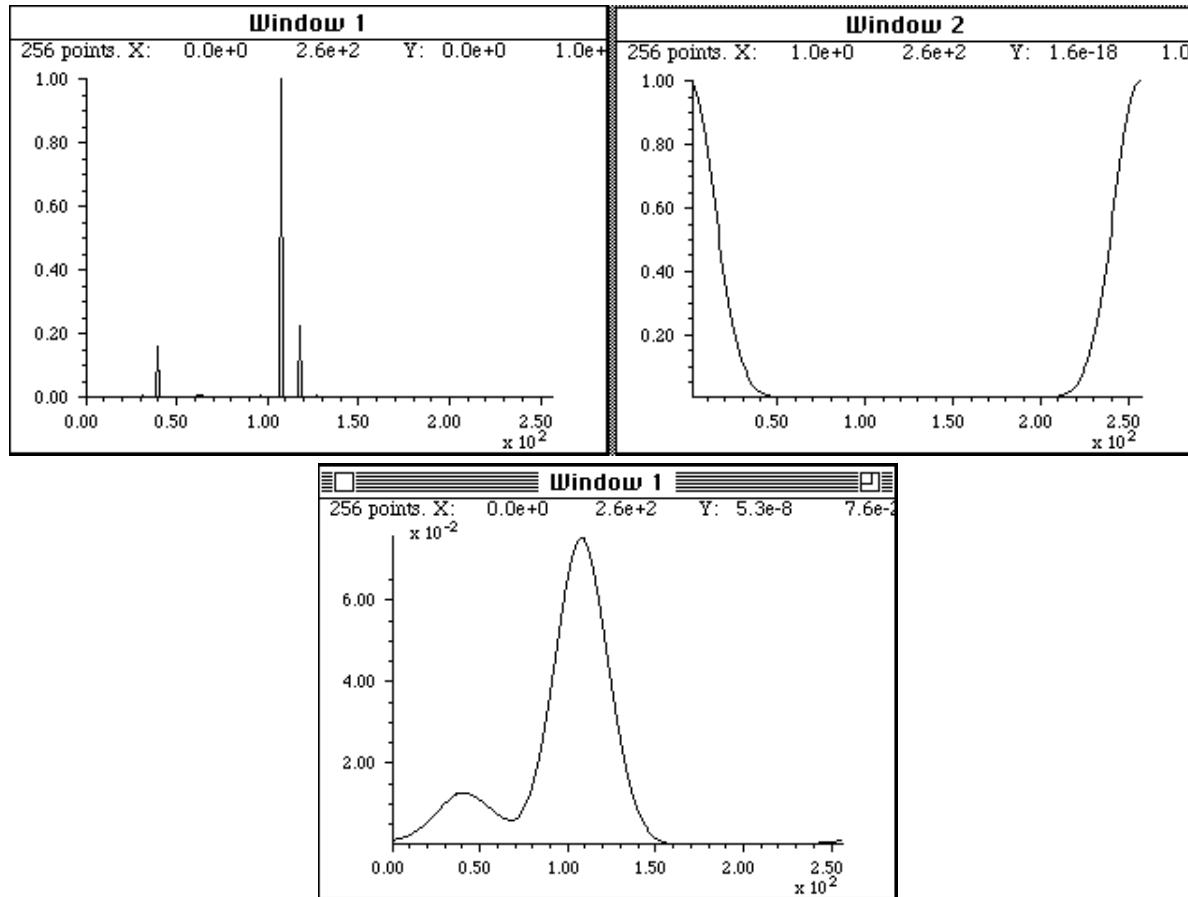
Results of least-squares fit of selected peaks to Gaussian peak model using the peakfit function:

Fitting Error = 6.4221e-007%	R2 = 1.000		
Peak# Position	Height	Width	Area
1 -6.2545e-009	113.31	0.31265	17.721

If you repeat this with Gaussians of *different widths* (e.g. width=1 or 4), you'll find that 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 the zero frequency.

Fourier Convolution

Convolution is an operation performed on two signals which 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 accurately 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 convolved 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 237).

In practice, 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; [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 are negative and the second half are positive (e.g. [-1 -2 0 2 1]).

Simple whole-number convolution vectors

Smoothing

[1 1 1]	= 3 point boxcar (sliding average) smooth
[1 1 1 1]	= 4 point boxcar (sliding average) smooth
[1 2 1]	= 3 point triangular smooth
[1 2 3 2 1]	= 5 point triangular smooth
[1 4 6 4 1]	= 5 point Gaussian smooth
[1 4 8 10 8 4 1]	= 7 point Gaussian smooth
[1 4 9 14 17 14 9 4 1]	= 9 point Gaussian smooth

Differentiation vectors:

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

Results of successive convolution by two vectors Conv1 and Conv2:

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]	Pseudo-Gaussian smooth
[-1 1]	* [-1 1]	= [1 -2 1]	2nd derivative
[-1 1]	* [1 -2 1]	= [1 -3 3 -1]	3rd 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

Rectangle * rectangle = triangle or trapezoid (depending on relative widths)

Gaussian * Gaussian = Gaussian of greater width

Gaussian * Lorentzian = Voigt profile (Something in between Gaussian and Lorentzian)

Software details for convolution

[SPECTRUM](#), page 383, the freeware signal-processing application for Mac OS8 and earlier, includes convolution and auto-correlation (self-convolution) functions.

Spreadsheets can be used to perform "shift-and-multiply" convolution (for example, [MultipleConvolution.xls](#) or [MultipleConvolution.xlsx](#) for Excel and [MultipleConvolutionOO.ods](#) for Calc), but for larger data sets the performance is much slower than Fourier convolution (which is much easier done in Matlab or Octave than in spreadsheets).

Matlab and **Octave** have a built-in function for convolution of two vectors: **conv**. This function can be used to create very general type of filters and smoothing functions, such as [sliding-average](#) and [triangular](#) smooths. For example,

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

smooths the vector y with a 5-point unweighted sliding average (boxcar) smooth, and

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

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.

Differentiation is carried out with smoothing by using a convolution vector in which the first half of the coefficients are negative and the second half are positive (e.g. [-1 0 1], [-2 -1 0 1 2], or [-3 -2 -1 0 1 2 3]) to 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 similar to 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, in order to insure 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 actually slightly faster than using the conv function.

```
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 path.

[**iSignal**](#) (page 323) 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**.

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**. Conversely if you know that **m** convoluted with **x** equals **n**, where **m** and **n** are known but **x** is unknown, then **x** equals **m** deconvoluted from **n**.

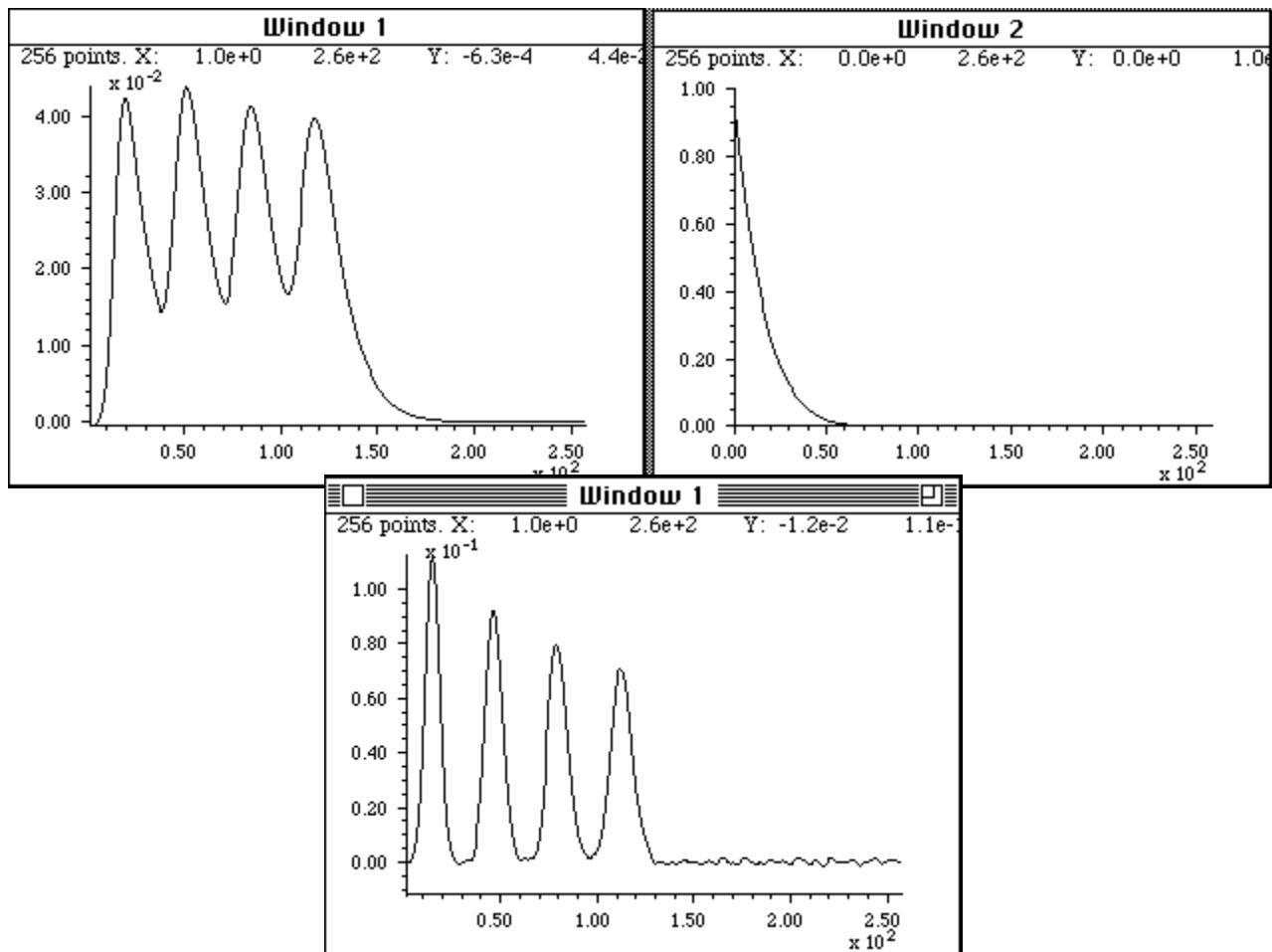
In practice, the deconvolution of one signal from another is usually performed by point-by-point *division* of the two signals in the Fourier domain, that is, dividing the Fourier transforms of the two signals point-by-point and then inverse-transforming the result. Fourier transforms are usually expressed 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 is

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

by the rules for the [division of complex numbers](#). Many computer languages (such as Fortran and Matlab) will perform this operation automatically when the two quantities divided are complex.

Note: The word "[deconvolution](#)" can have two meanings in the science literature, which can lead to confusion. The Oxford dictionary defines it 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 peaks into their separate additive components by the technique of [iterative least-squares curve fitting](#) (page 163) of a proposed peak model to the data set. However, that process is actually conceptually distinct from *Fourier* deconvolution, because in Fourier deconvolution, the underlying peak shape is unknown but the broadening function is assumed to be known; whereas in iterative least-squares curve fitting, it's 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; don't get them confused.

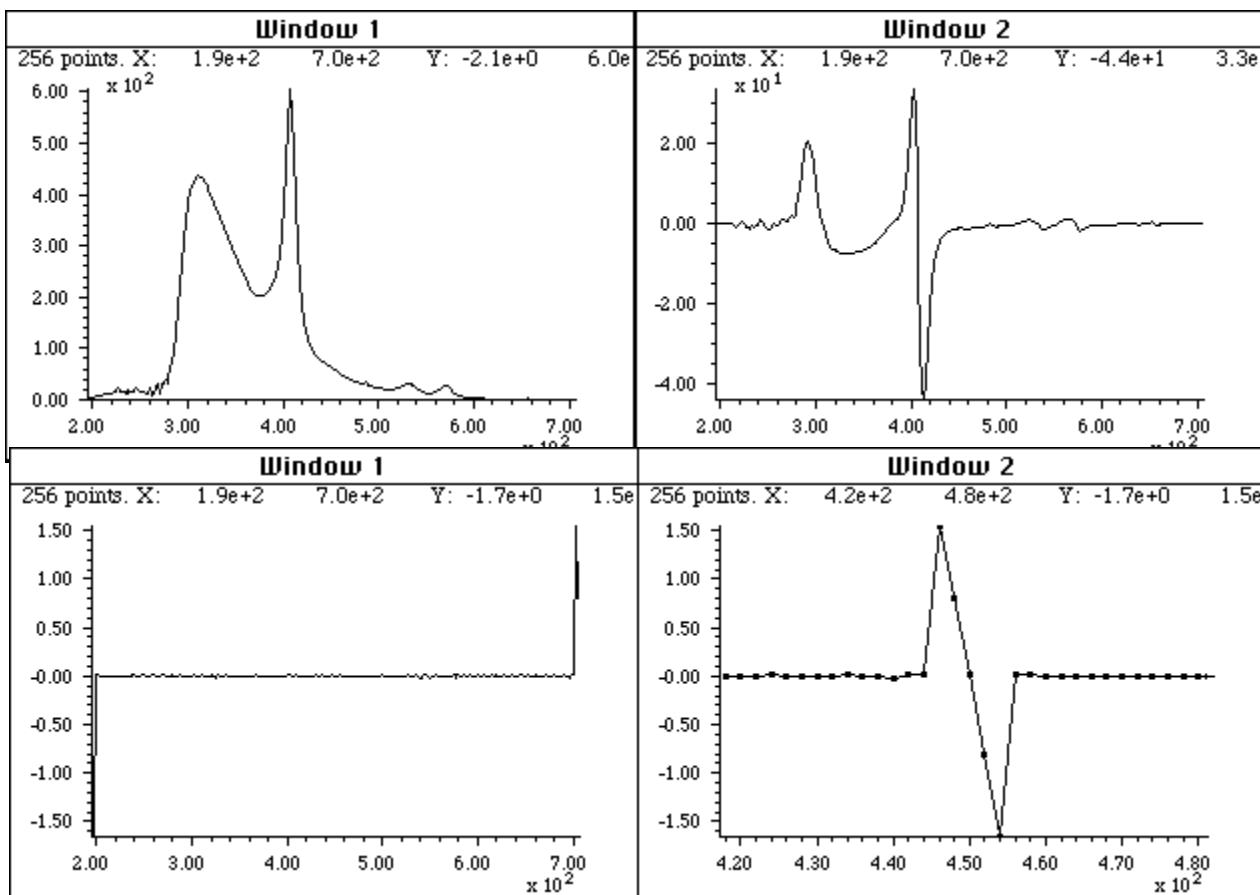
The practical significance of Fourier deconvolution in signal processing is that it is used as a computational way to reverse the result of a convolution occurring in the physical domain, for example, to reverse the signal distortion effect of an electrical filter or of the finite resolution of a spectrometer. 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 data used as a deconvolution vector. Deconvolution can also be used to determine the form of a convolution operation that has been previously applied to a signal, by deconvoluting the original and the convoluted signals. These two types of application of Fourier deconvolution are shown in the two figures below.



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 an unavoidable RC low-pass filter action in the electronics. 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 deconvolved 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.

(Click for [Matlab/Octave script](#).)

Note that this process has an effect that is visually similar to derivative peak sharpening (page 69) although the latter requires no specific knowledge of the broadening function that caused the peaks to overlap.



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 a uv-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 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. 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 seen to be +2, +1, 0, -1, -2. This simple example of "[reverse engineering](#)" would make 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 *four serious problems* that limit the utility of the method:

- (1) It is possible that 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. This occurs because the high frequency components of the broadening operator (the denominator in the division of the Fourier transforms) are typically very small, resulting in a great amplification of high frequency noise in the resulting deconvoluted signal (See the Matlab/Octave code example on page 100) ;

(4) If the denominator vector contains *even a single zero value*, the result will be a "divide-by-zero" error and the whole operation fails. The problem of low values or zeros in the denominator can be reduced by using the "remove zeros" function, [rmz.m](#), or by smoothing the data before convolution, or by constraining the Fourier deconvolution to a frequency region where the denominator is sufficiently high.

You can see the amplification of high frequency noise happening in the example in the first example above. However, 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 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.

As a method for *peak sharpening*, deconvolution can be compared to the [derivative peak sharpening method described earlier](#) or to the [power method](#), in which the raw signal is simply raised to some positive power n .

Computer software for deconvolution

[SPECTRUM](#), page 383, the freeware signal-processing application for Mac OS8 and earlier, includes a Fourier deconvolution function.

Matlab and Octave

Matlab and Octave have a built-in function for Fourier deconvolution: [deconv](#). An example of its application is shown below: the vector yc (line 6) represents a noisy rectangular pulse (y) convoluted

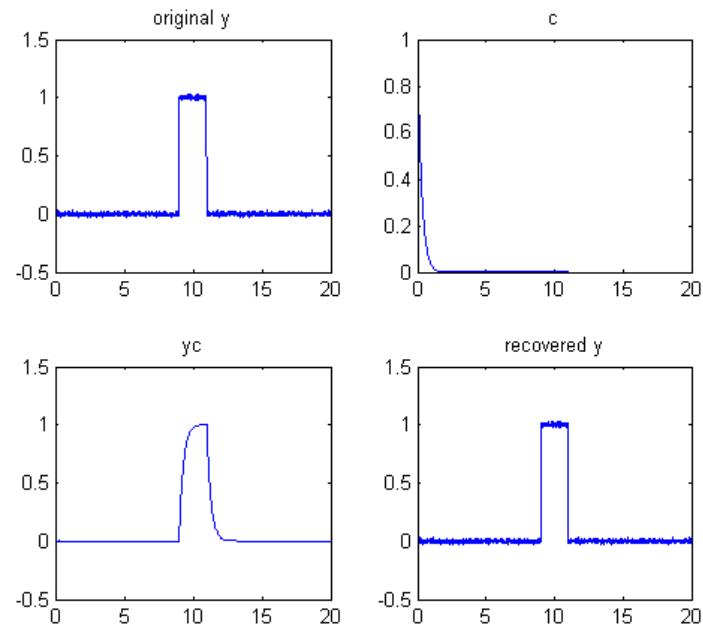
with a transfer function c before being measured. In line 7, c is deconvolved from yc , in an attempt to recover the original y . This requires that the transfer function c be known. The rectangular signal pulse is recovered in the lower right (ydc), complete with the noise that was present in the original 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 significant amplification of any noise that is added *after* the convolution by the transfer function (line 5). This script demonstrates 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 script](#).

```
x=0:.01:20; y=zeros(size(x));
y(900:1100)=1; % Create a rectangular function y,
% 200 points wide
y=y+.01.*randn(size(y)); % Noise added before the convolution
c=exp(-(1:length(y))./30); % exponential trailing 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
% The sum(c2) term is included simply to scale the amplitude of the result
% (specifically the area under the curve) to match the original y.
% 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')
```

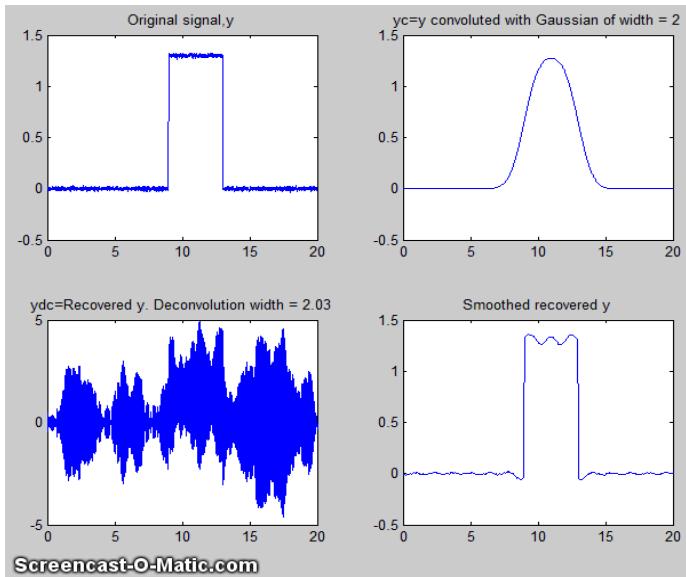
Alternatively, you could perform the Fourier deconvolution yourself *without* using the built-in Matlab/Octave "deconv" function by dividing the Fourier transforms of yc and c using the built-in Matlab/Octave "fft.m" function and inverse transform the result with the built-in Matlab/Octave "ifft.m" function. 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's actually *ten times faster* than using the deconv function:

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

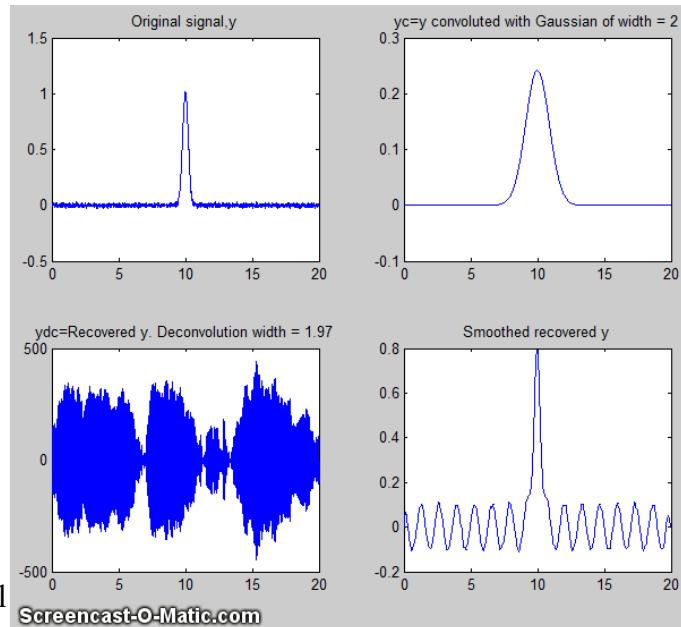


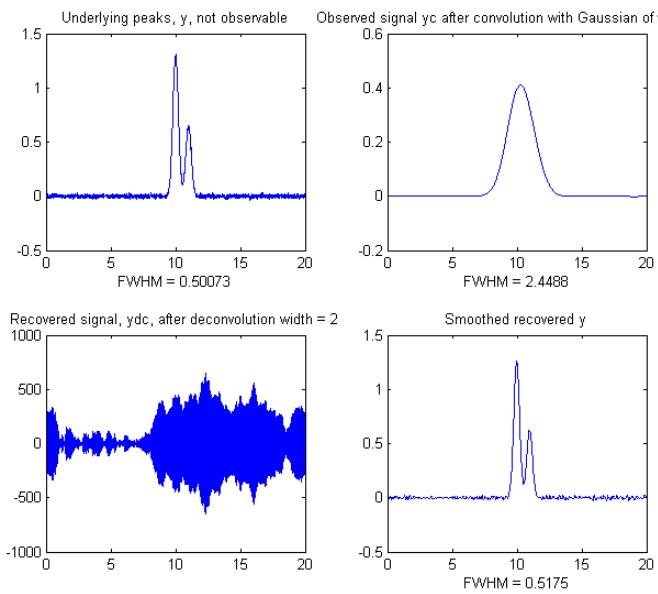
The script [DeconvDemo3.m](#) is similar to the above, except that it demonstrates *Gaussian* Fourier convolution and deconvolution of the same rectangular pulse, utilizing the fft/ifft formulation just described. The animated screen graphic on the right (click [link for animation](#)) demonstrates the effect of changing the deconvolution width. The raw deconvolved 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 34). As you can see in both of the animated examples here, deconvolution works *best* when the deconvolution width exactly matches the width of the convolution that the observed signal has been subject to; the further off you are, the worse will be the wiggles and other signal artifacts. In practice, you have to 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.

[DeconvDemo4.m](#) ([animation link](#)) shows a Gaussian deconvolved from a Gaussian function and an attempt to recover the original peak width. This is an example of "[self deconvolution](#)", so-called because the shape of the deconvolution function is the same as the shape of the peaks in the signal, in this case both *Gaussian*. Typically, this would be applied to a signal containing multiple overlapping peaks, in an attempt to sharpen the peaks to improve the resolution. Note that in this example the deconvolution width must be within 1% of the deconvolution width. In general, the wider the physical convolution width relative to the signal, the more accurately the deconvolution width must be matched the physical convolution width.



[DeconvDemo5.m](#) (shown below) 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 deconvolved and smoothed result. [DeconvDemo6.m](#) is that same except that





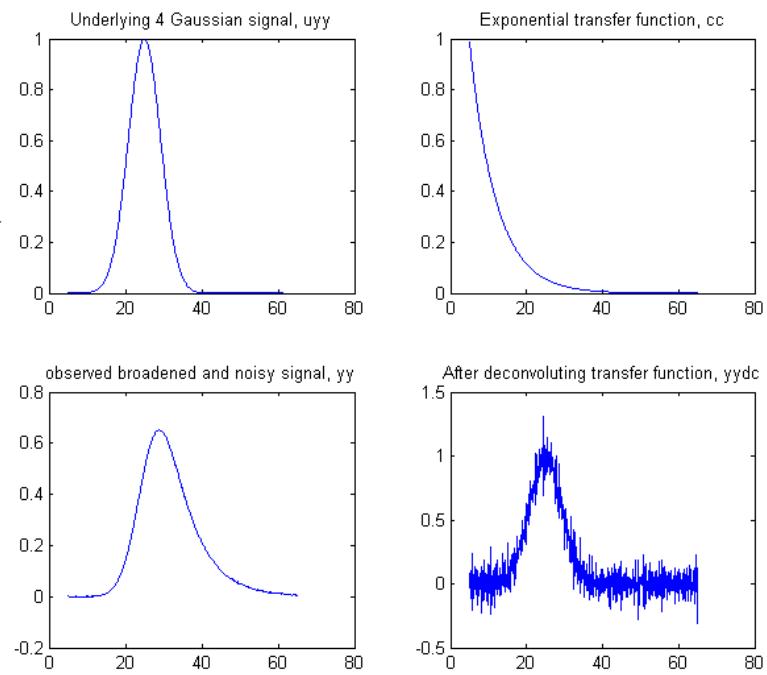
the underlying peaks are [Lorentzian](#). Note that all these scripts require functions than can be downloaded from <http://tinyurl.com/cey8rwh>.

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 right 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 over used. The image on the left shows the widths of the peaks (as full width at half maximum); the width 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.

Here is another example, shown below 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*, and then a little constant white noise is added *after* the broadening convolution (cc).

Assuming that the exponential broadening time constant (' tc ') is known, or can be guessed or measured, 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. However, the noise remaining in the deconvoluted signal is "[blue](#)" (high-frequency weighted, see page 26) and so is easily reduced by smoothing (page 37) 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 time constant ('*tc*') in line 7). To plot the recovered signal overlaid with underlying signal: `plot(xx,uyy,xx,yydc)`. To plot the observed signal overlaid with 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 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 43.

```
% Deconvolution demo 2
xx=5:.1:65;
% Underlying signal with a single peak (Gaussian) of unknown
% height, position, and width.
uyy=gaussian(xx,25,10);

% Compute observed signal yy, using the expgaussian function with time
% constant tc, adding noise added AFTER the broadening convolution (ExpG)
Noise=.001; % <<< Change the noise here
tc=70; % <<< Change the exponential time constant here
yy=expgaussian(xx,25,10,-tc)'+Noise.*randn(size(xx));

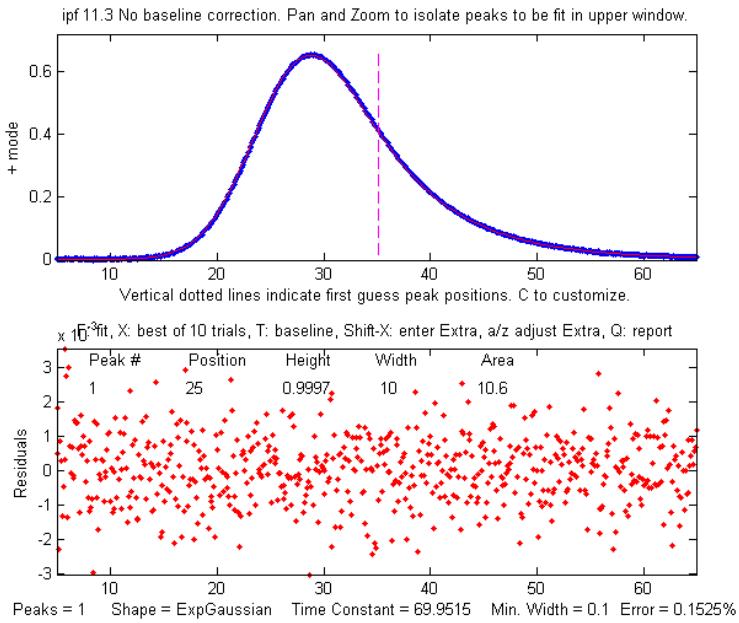
% Guess, or use prior knowledge, or curve fit one peak, to
% determine time constant (tc), then compute transfer function cc
cc=exp(-(1:length(yy))./tc);

% Use "deconv" to recover original signal uyy by deconvoluting cc
% from yy. It's necessary to zero-pad the observed signal as shown here.
yydc=deconv([yy zeros(1,length(yy)-1)],cc).*sum(cc);

% Plot the signals and results in 4 quadrants
subplot(2,2,1);
plot(xx,uyy);title('Underlying 4 Gaussian signal, uyy');
subplot(2,2,2);
plot(xx,cc);title('Exponential transfer function, cc')
subplot(2,2,3);
plot(xx,yy);title('observed broadened and noisy signal, yy');
subplot(2,2,4);
plot(xx,yydc);title('After deconvoluting transfer function, yydc')
```

An alternative to the above deconvolution approach is to use [iterative curve fitting](#) (page 163) to fit the observed signal directly with an [exponentially broadened Gaussian](#) (shape number 5):

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



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

which in this particular case gives a best fit when the exponential factor "tc" is adjusted to about 69.9 (very 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])
```

```
FitResults = 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](#) 268 for another example with four overlapping Gaussians.

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 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 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

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 361), manually adjusting the exponential factor ('extra') interactively with the A and Z keys to get the best fit:

it easier to adjust the number and values of the segments. [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. 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, “start”, and the last value, “endt”:

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

In [iSignal version 5.7](#) and later you can press **Shift-V** to display 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. It will ask you for the width or the time constant (in X units).

Fourier convolution/deconvolution menu

- 1. Convolution
- 2. Deconvolution

Select mode 1 or 2: 2

Shape of convolution/deconvolution function:

- 1. Gaussian
- 2. Exponential

Select shape 1 or 2: 2

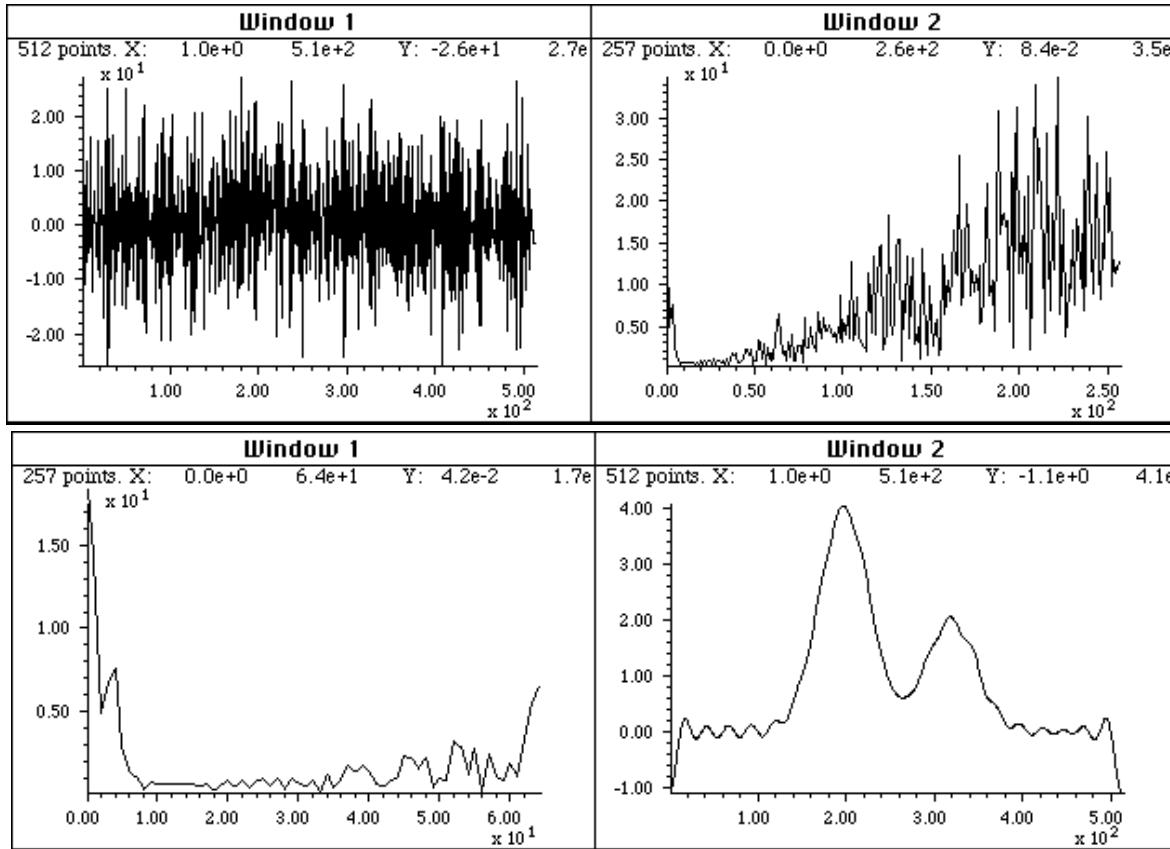
Enter the exponential time constant:

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

Fourier filter

A Fourier filter is a type of filtering function that is based on manipulation of specific [frequency components](#) of a signal. It works by taking the [Fourier transform](#) of the signal, then attenuating or amplifying specific frequencies, and finally inverse transforming the result. You must take care to use both the sine *and* cosine components of the Fourier transform (or equivalently frequency *and* phase, or real *and* imaginary). The example below is a crude low-pass, sharp cut-off filter, which simply cuts off all frequencies above a user-specified limit. As is often the case, the frequency components of the signal fall predominantly at *low* frequencies and those of the noise fall predominantly at *high* frequencies. The user tries to find a cut-off frequency that will eliminate most of the noise while not distorting the signal significantly.

A simple example

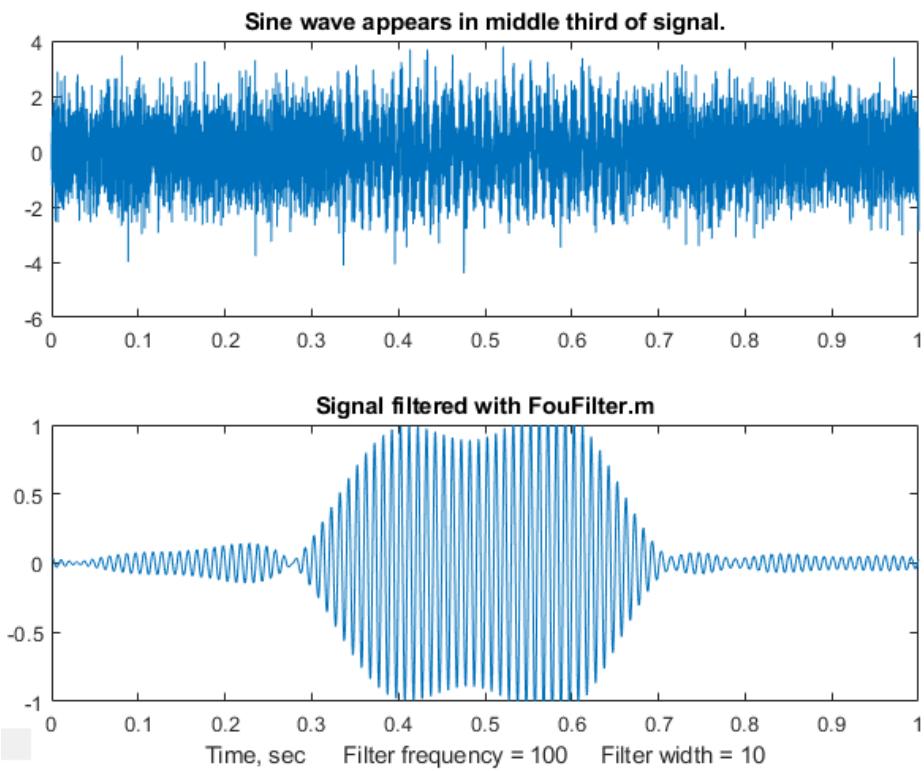


The signal at the top left seems to be only random noise, but its **power spectrum** (top right) shows that high-frequency components dominate the signal. The power spectrum is expanded in the X and Y directions (bottom left) to show more clearly the low-frequency region. Working on the hypothesis that the components above the 20th harmonic are noise, the **Fourier filter** function deletes the higher harmonics and reconstructs the signal from the first 20 harmonics. The result (bottom right) shows the signal contains two bands at about $x=200$ and $x=300$ that are totally obscured by noise in the original signal.

Computer software for Fourier Filtering

[SPECTRUM](#), page 383, the freeware signal-processing application for Mac OS8 and earlier, includes a crude Fourier low-pass filter function, with adjustable harmonic cut-off.

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. Has 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 sec, millisec, or microsec; '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 the memory in your computer. Here are two example scripts that call FouFilter.m: [TestFouFilter.m](#)



and [TestFouFilter2.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 fairly effective in extracting the signal from the noise, but that the response time is slow. [TestFouFilter2.m](#) demonstrates the 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. Further Matlab-based applications using an interactive Fourier filter *iFilter* are given on page 338. A demonstration of a *real-time* Fourier filter is discussed on page 308.

Integration and peak area measurement

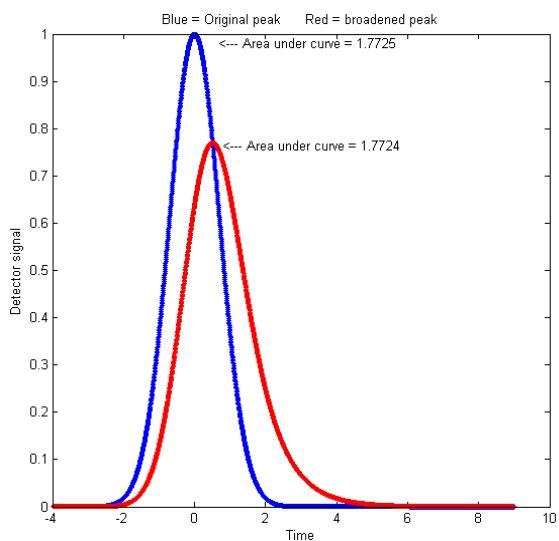
Symbolic integration of functions and calculation of definite integrals are topics that are introduced in elementary Calculus courses. The numerical integration of digitized signals finds application in analytical signal processing mainly as a method for measuring the areas under the curves of peak-type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

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 magnitude 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 area is less sensitive to the influence of 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 column. 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*, as long as 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 exactly 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). Another example with greater broadening: ([script](#) and [graphic](#)).

If the detector response is linear with respect to the concentration of the material only at *low* concentrations and becomes *non-linear at higher concentrations*, peak height measurements will suffer because the peak maximum naturally occurs at the moment when the concentration at the detector is highest. In such cases peak area measurements will still be somewhat non-linear, but less so than peak height measurements, because most of the area of a peak is measured when the concentration of material is less than maximum.

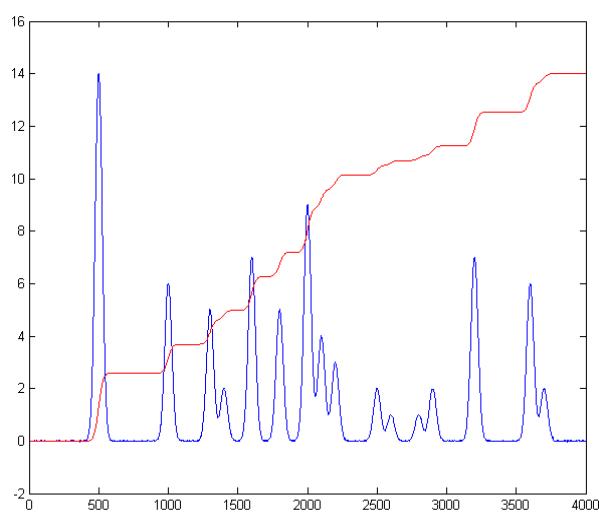


On the other hand, peak height measurements are *simpler to make* and are *less prone to interference* by neighboring, overlapping peaks. And a further disadvantage of peak area measurement is that the peak start and stop points must be determined, which may be difficult especially if the peak overlaps other peaks. In principle [curve fitting](#) 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 298).

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 276: [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 methods to compute peak areas:

- (a) plot the signal on a paper chart, cut out the peak with scissors, then weigh the cut out piece on a micro-balance compared to a square section of 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 geometry to compute the area under a [triangle constructed with its sides tangent to the sides of the peak](#), or
- (e) compute the cumulative sum of the signal magnitude and measure the heights of the resulting steps (see figure below).



Now that computing power is built into or connected to almost every measuring instrument, more accurate and convenient digital methods can be employed. However it is measured, the *units* of peak area are the *product* of the x and y units. Thus, in a chromatogram where the x is time in minutes and y is volts, the area is in volts-minute. In absorption spectrum 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 the 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).

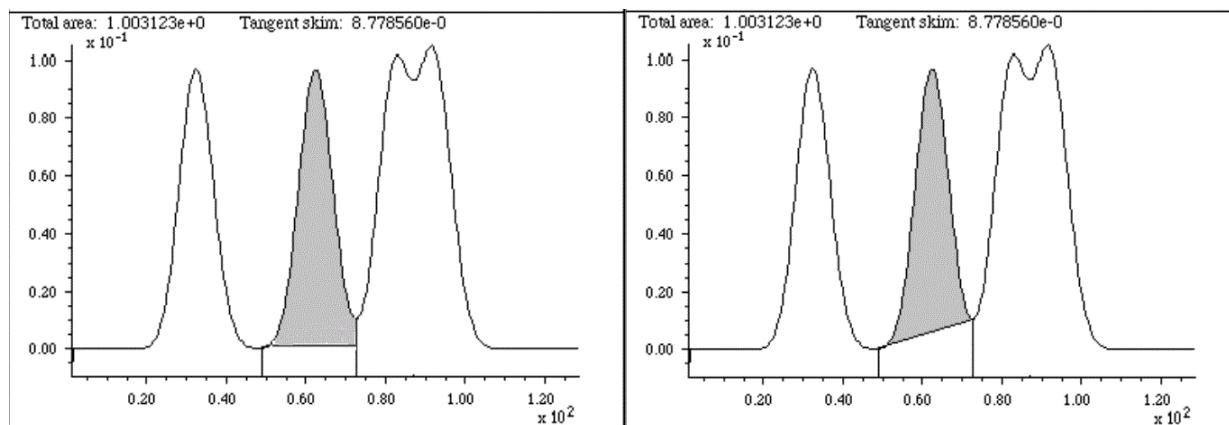
The best method for calculating the area under a peak depends whether the peak is isolated or overlapped with other peaks or superimposed on a non-zero baseline or not. The simple numeric

integration of a digital signal, for example by [Simpson's rule](#), will convert a series of peaks into a series of steps, the height of each of which is proportional to the area under that peak. But this works well only if the peaks are well separated from each other and if the baseline is zero.

This is a commonly used method in proton NMR spectroscopy, where the area under each peak or multiplet is proportional to the number of equivalent hydrogen atoms responsible for that peak.

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, below. This is often called the **perpendicular drop** method; it's an easy task for a computer, although tedious to do by hand. The left and right bounds of the peak are usually taken as the valleys (minima) between the peaks or as the point half-way between the peak center and the centers of the peaks to the left and right. The basic assumption 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 only if the peaks are 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 below to an accuracy of about 0.3% and the last two peaks to an accuracy of better than 4%. It's possible to improve the accuracy of the measured areas by applying a [peak sharpening technique](#) to narrow the peaks before the perpendicular drop measurement; see [PeakSharpeningAreaMeasurementDemo.xlsx \(screen image\)](#). 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 71).



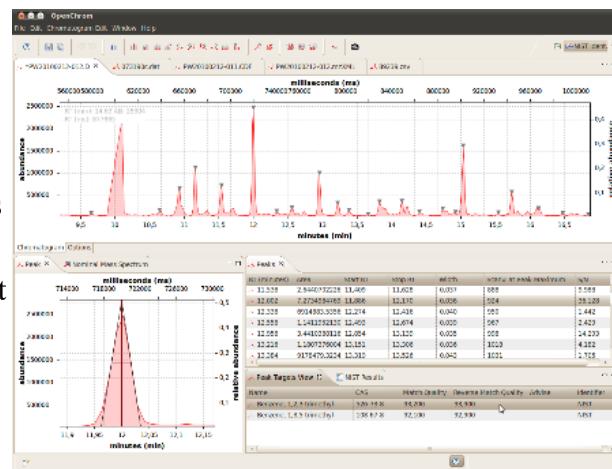
Peak area measurement for overlapping peaks, using the **perpendicular drop method** (left, shaded area) and **tangent skim method** (right, shaded area).

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 end points, 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.

If the *shape* of peaks is known, the best way to measure the areas of overlapping peaks is to use some type of least-squares curve fitting, as is discussed starting on page 138). 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 screen shot is shown on the left (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.

[SPECTRUM](#), page 383, the freeware signal-processing application for Macintosh OS8, includes an integration function, as well as peak area measurement by perpendicular drop or tangent skim methods, with mouse-controlled setting of start and stop points.

Peak area measurement using spreadsheets.

[EffectOfDx.xlsx](#) ([screen image](#)) demonstrates that the simple equation $\text{sum}(y) * \text{dx}$ 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 non-zero baseline than the same amount of random noise.

[CumulativeSum.xls](#) ([screen image](#)) illustrates 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, respectively. But neither is as versatile as using a dedicated chromatography program such as [OpenChrom](#).

Using sharpening for overlapping peak area measurements.

There is a set of downloadable spreadsheets for perpendicular drop area measurements of overlapping peaks using [2nd and 4th derivative sharpening](#). Sharpening the peaks reduces the degree of overlap and can greatly reduce the peak area measurement errors made by the perpendicular drop method. There is an empty template for you to paste your data into ([PeakSharpeningAreaMeasurementTemplate.xlsx](#)), an example version with sample data and settings already entered ([PeakSharpeningAreaMeasurementExample.xlsx](#)), and a "demo" that creates and measures *simulated data with known areas* ([PeakSharpeningAreaMeasurementDemo.xlsx](#)) 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 σ is the standard deviation of the Gaussian, then adjust to give the narrowest peaks without significant negative dips. Don't assume that increasing the K's 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 111) works perfectly with no sharpening, but if there is inequality in shapes, heights, or widths, increased K values give lower errors *up to a point*, but overdoing the sharpening can sacrifice accuracy. The two screen images [screen1](#) and [screen2](#) show that it is possible to find values of K's that give excellent accuracy even if the relative height of the smaller measured peak is quite small.

The template [PeakSymmetrizationTemplate.xlsx](#) ([graphic](#)) performs the symmetrization of exponentially broadened peaks by the weighted addition of the first derivative. See page 71. [PeakSymmetrizationExample.xlsx](#) 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 (k_1) in cell J4 and the second derivative sharpening factor k_2 (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.xlsx \(graphic\)](#).

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.

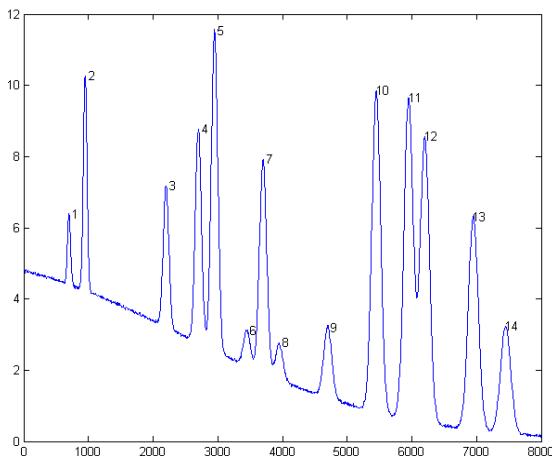
```
>> x=-5:.1:5;
>> y=exp(-(x).^2);
>> trapz(x,y)
```

These lines accurately compute the area under the curve of x,y (in this case an isolated Gaussian, whose area is theoretically known to be the [square root of pi](#), $\sqrt{\pi}$, which is 1.7725. If the interval between x values, dx , is *constant*, then the area is simply $y_i = \text{sum}(y) \cdot dx$. Alternatively, the signal can be integrated using $y_i = \text{cumsum}(y) \cdot dx$, then the area of the peak will be equal to the [height of the resulting step](#), $\max(y_i) - \min(y_i) = 1.7725$. 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 Gaussian peak with a peak height h and [full-width at half-maximum](#) w has an area of $1.0645 \cdot h \cdot w$. A Lorentzian peak has an area of $1.57 \cdot h \cdot w$. (This can be confirmed by computing the area of a peak of unit height and width: e.g. for a Lorentzian peak `dx=.001; x=0:dx:1500; y=lorentzian(x,750,1); trapz(x,y)`). For a peak of *any smooth shape on a zero baseline*, the peak width (FWHM) can be measured by the Matlab/Octave [halfwidth.m](#) function. But the peaks in real signals have some complications:

- (a) their shapes might not be known;
- (b) they may be superimposed on a baseline; and
- (c) they may be overlapped with other peaks.

These must be taken into account to measure accurate areas in experimental signals. Various Matlab/Octave functions have been developed to deal with these complications.

[**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

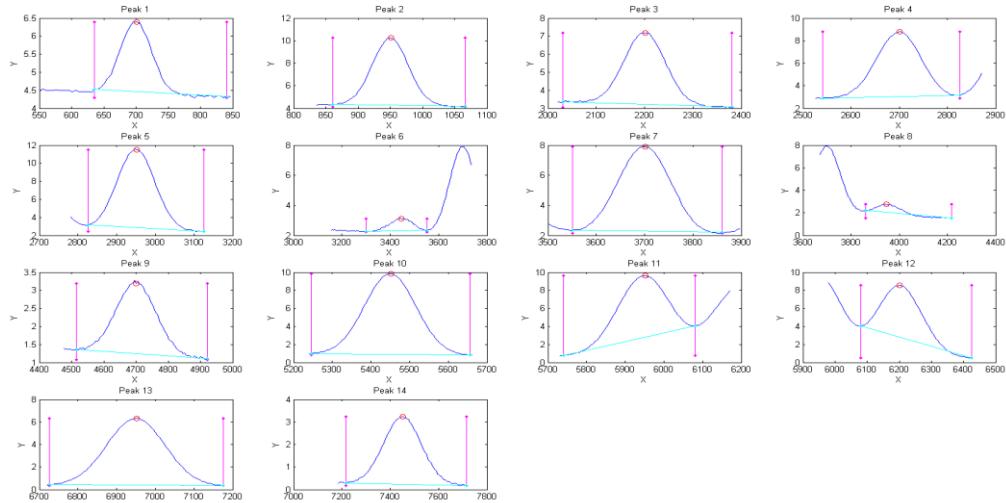


[findpeaksSG](#). It returns a [table](#) containing the peak number, peak position, absolute peak height, peak-valley difference, perpendicular drop area, and the tangent skim area of each peak it detects. If the last input argument ('plots') is set to 1, it [plots the signal](#) with numbered peaks (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, below. 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, it may be beneficial to smooth the y data yourself before calling measurepeaks.m, using any smooth function of your choice.

[\[M,A\]=autopeaks.m](#) 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 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 it 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 path.

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 R2 values, which should ideally be 1.0, and 1, if the processing has had no effect at all on peak area.

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 measurements by the triangle construction method is 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.

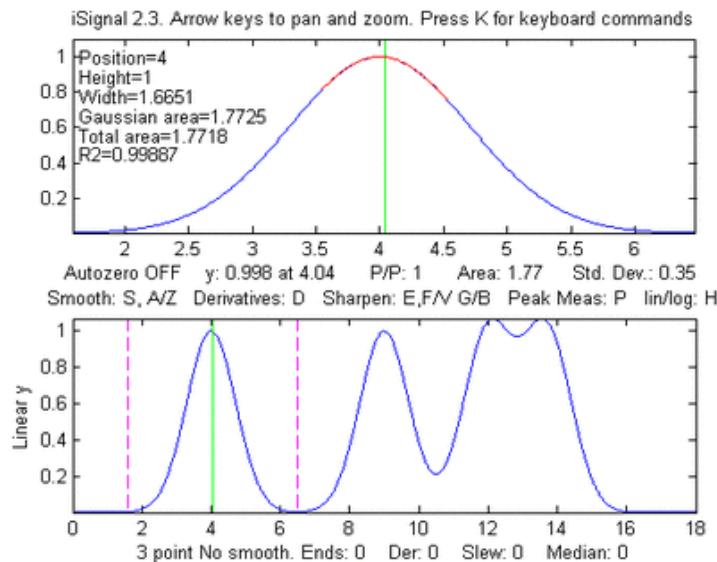
iSignal (page 323) is a downloadable user-defined Matlab function that performs various signal processing functions described in this tutorial, including measurement of peak area using Simpson's Rule and the perpendicular drop method. Click to view or right-click > Save link as... [here](#), or you can download the [ZIP file](#) with sample data

for testing. It is shown on the left 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). [Link to animation of this figure.](#)

The following Matlab/Octave code creates 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.

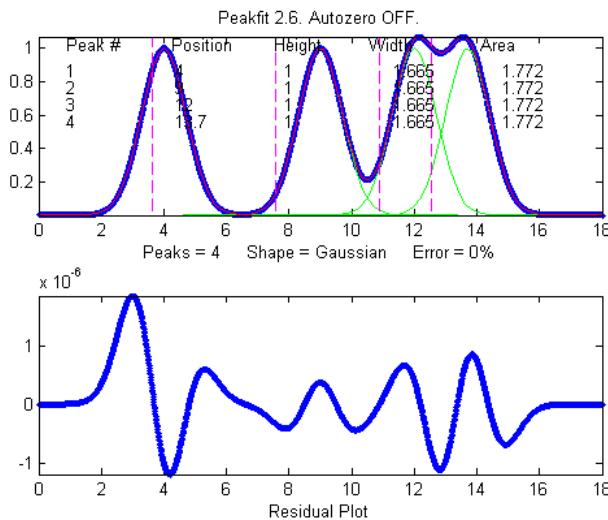
Peak #	Position	Height	Width	Area
1	4.00	1.00	1.661	1.7725
2	9.001	1.0003	1.6673	1.77
3	12.16	1.068	2.3	1.78
4	13.55	1.0685	2.21	1.79

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

iSignal version 5.9 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 mode peak. 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

[**iPeak**](#) (page 216), can also be used to estimate peak areas. It uses the same Gaussian curve fitting method as *iSignal*, but it has the advantage that it can detect and measure all the peaks in a signal in one operation. For example:

```
>> 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 peak position and width estimates as its first guesses, 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 rather ideal situation, the results are in perfect agreement with expectations.

Correction for background/baseline

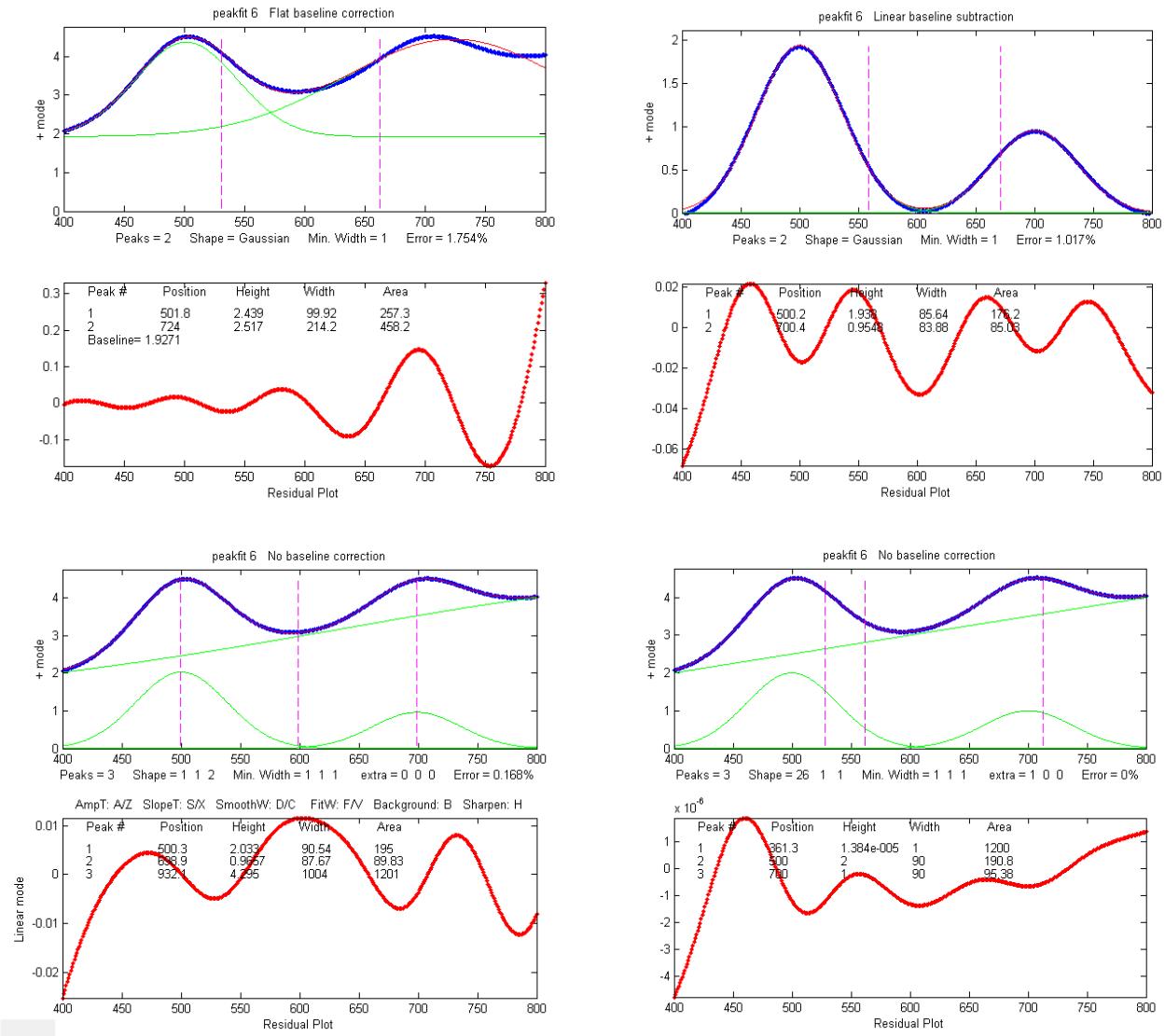
The presence of a baseline or background signal, on which the peaks are superimposed, will greatly influence the measured peak area if not corrected or compensated. *iSignal*, *iPeak*, [**measurepeaks**](#), and *peakfit* all have several different baseline correction modes, for flat, linear, and quadratic 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 actually caused by the edges of a strong overlapping adjacent peak, then it's possible to include that peak in the curve-fitting operation, as see in Example 22 on page 354.

Here's a Matlab/Octave experiment that compares several *different methods of baseline correction* in peak area measurement. The signal 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 the perpendicular drop method in baseline mode 1, 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 182). A more accurate approach is 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])
FitResults =
    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 357). 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 zero baseline, 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.

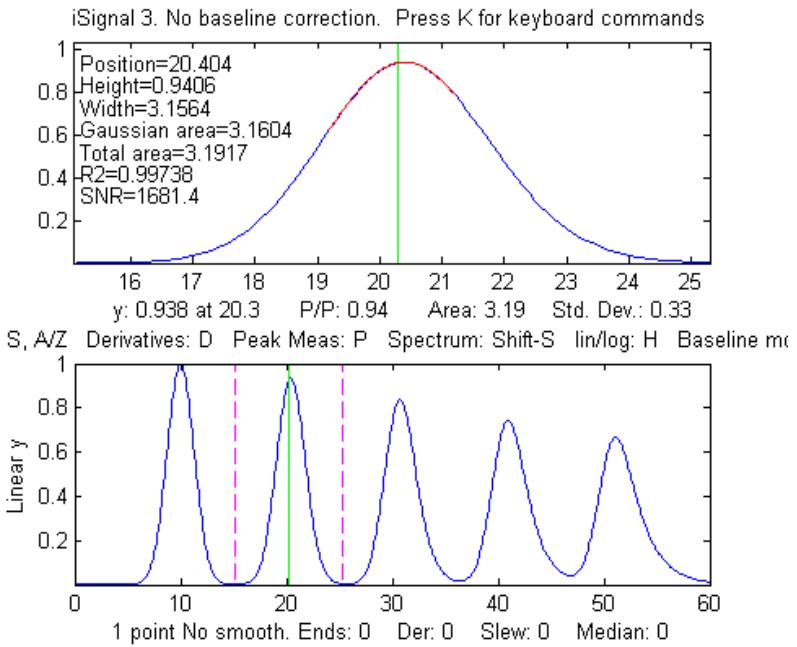
Here's a Matlab/Octave experiment that creates a signal containing five Gaussian peaks with the *same* initial peak *height* (1.0) and *width* (3.0) but which are subsequently broadened by *increasing degrees of exponential broadening*, similar to the broadening of peaks commonly encountered in chromatography:

```
>> 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*: $1.0645 \times \text{Height} \times \text{Width} = 1 \times 3 \times 1.0645 = 3.1938$. 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 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 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 323) 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 (pretty 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 we 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)
FitResults =
    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)

FitResults =
    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)
FitResults =
    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 have all the *same* width. In general, if you expect that the peaks should have equal widths, or fixed widths, then it's 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 that:

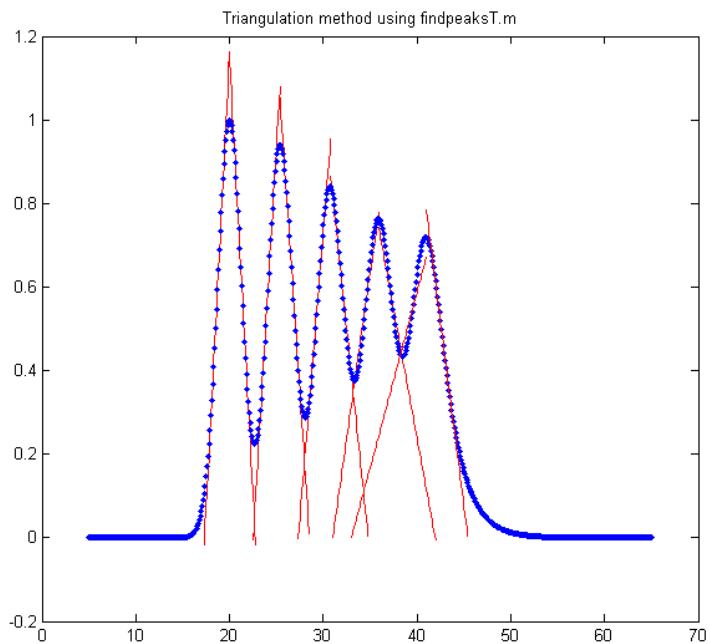
- (a) it may not be a perfect match to the actual physical broadening process;
- (b) it's slower than a simple Gaussian fit, and
- (c) it sometimes need help, in the form of a start vector or equal-widths constraints, as seen above, in order to get the best results.

Alternatively, if the objective is only to measure the peak *areas*, and not the peak positions and widths, then it's 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 (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 fit 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's 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.

By making the peaks **closer together**, we 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 3.2020 3.3958 4.1563 4.4039], seriously overestimating the areas of the last two peaks, and measurepeaks.m using the perpendicular drop method 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)
FitResults =
    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%
```

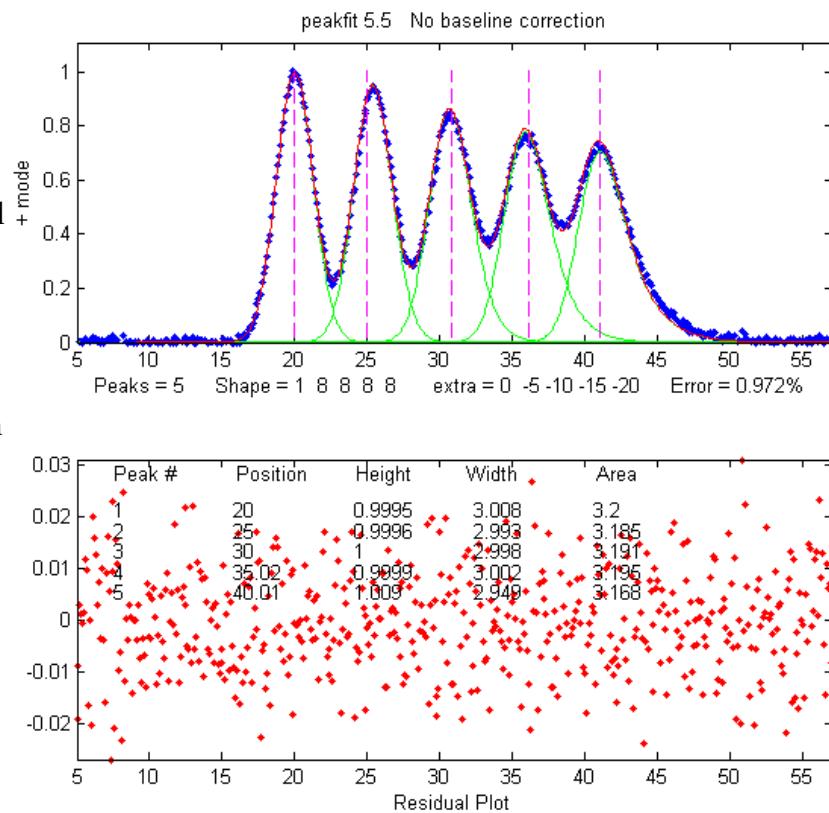
Next, we make an [even tougher challenge](#) with different peak heights (1, 2, 3, 4 and 5, respectively) and a bit of *added random noise*. The theoretical areas (Height*Width*1.0645) are 3.1938, 6.3876, 9.5814, 12.775, and 15.969.

```
>> 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)
FitResults =
    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 separate 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 ion an irrelevant background, the simpler technique of [first derivative sharpening](#) (page 71) can be useful as a method to reduce or eliminate the effects of exponential broadening. As is the case with curve fitting, it's most effective is there is also 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.



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 (usually 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 ($Y = a + bX$, where **a** is the *intercept* and **b** is the *slope*), or a quadratic ($Y = a + bX + cX^2$), or a cubic ($Y = a + bX + cX^2 + dX^3$), 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 why we call it a "linear" least-squares fit, not because the plot of X vs Y is linear.* Only for the first-order polynomial $Y = a + bX$ 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, discussed on page 163.

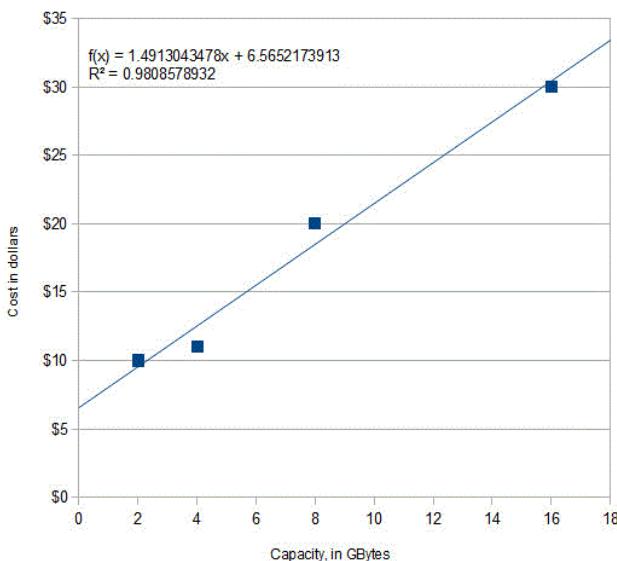
"Best fit" simply means that the differences between the actual measured Y values and the Y values predicted by the model equation are *minimized*. It does *not* mean a "perfect" fit; in most cases, a least-squares best fit *does not go through all the points* in the data set. Above all, a least-squares 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. It's not correct to say "fit data to ..." a straight line or to some other model; it's actually 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.

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'd get five slightly different answers, but if you gave the data set to five different computer programs, you'd get the exact same answer every time).

Examples of polynomial fits

Here's 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 were really the prices back in 2012).

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



What's the relationship between memory capacity and cost? Of course, we expect that the larger-capacity cards should cost more than the smaller-capacity ones, and if we plot cost vs capacity (graph on the left), we 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:

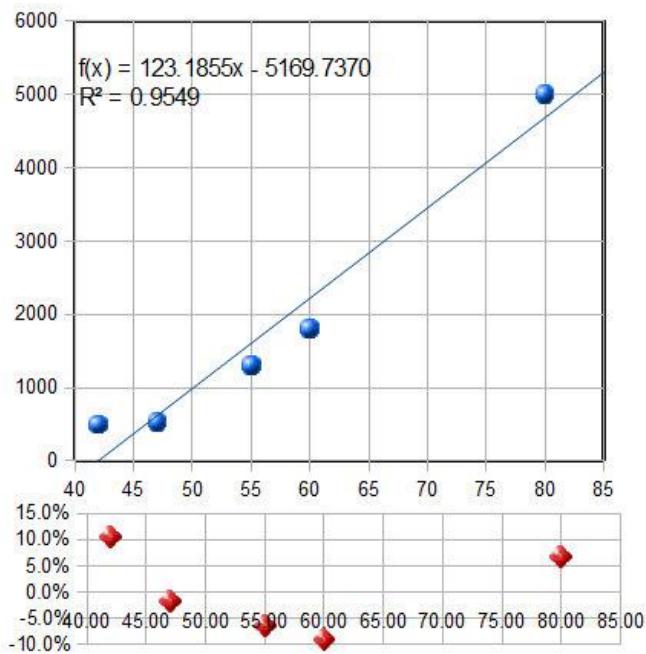
$$\text{Cost} = \$6.56 + \text{Capacity} * \$1.49$$

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 we 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 (actually 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, and have a higher chip-testing rejection rate in the production line. So in this case the slope and intercept have real physical and economic meanings.

What can we do with this information? First, we can see how closely the actual prices conform to this equation: pretty well *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 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 means a fairly good fit; 0.999 is a very good fit. (The R^2 value is calculated as shown on page 141).

The second way we 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's called "extrapolation" - and it's very risky because you don't 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, there was the prices back in 2012. Why not do a little "homework"? Look up and try fitting the *current* prices and see how they compare. Did you get a lower slope, lower intercept, or both?



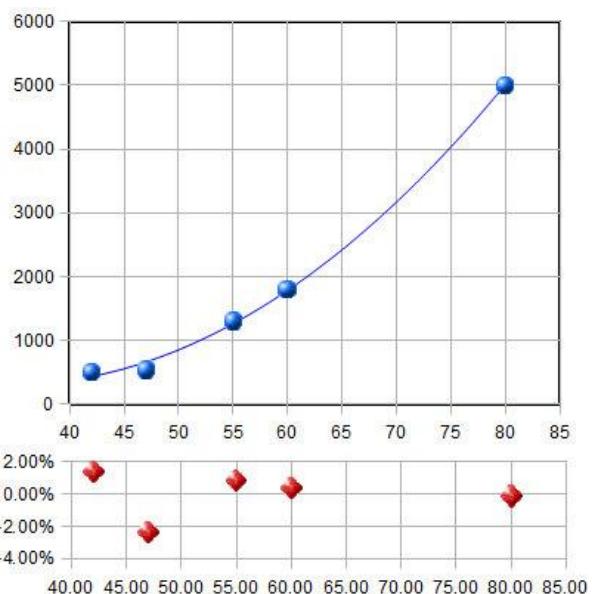
Here's another related example: the historical prices of standard high definition (not UHD) flat-screen LCD TVs as a function of screen size, as advertised on the Web in the Spring of 2012. The prices of five selected models, *similar except for screen size*, are plotted against the screen size in inches 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*! That's crazy. Would they *pay* you to take these sets? I don't think so. Clearly, something is wrong here.

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 fairly large ($\pm 10\%$), but more important, *they are not completely random*; they form a *clearly visible U-shaped curve*. This is a tip-off that the straight-line model we have used here may not be ideal and that we 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 was very careful?)

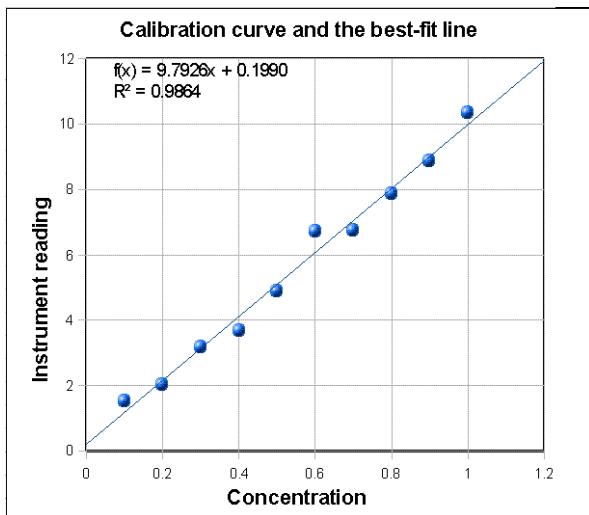
Least-squares calculations can fit not only straight-line data, but *any set of data that can be described by a polynomial*, for example a second-order (quadratic) equation ($Y = a + bX + cX^2$). Applying a second-order fit to these data, we get the graph on the right. Now the R^2 value is higher, 0.9985, indicating that the fit is better (but again not perfect), and also the residuals (the red dots at the bottom) are smaller and more random. This shouldn't really be a surprise, because 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. With this fit, 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 we 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 because it is *expected in principle* on the basis of the relationship between length and area. (Incidentally, as you might expect, prices have dropped considerably since 2012; in 2018, a 65" flat-screen HDTV 4K set was available at Costco for under \$600).

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's because a higher order model has more variable coefficients that the program can adjust to fit the data. For example, we *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 particular* 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's really 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.



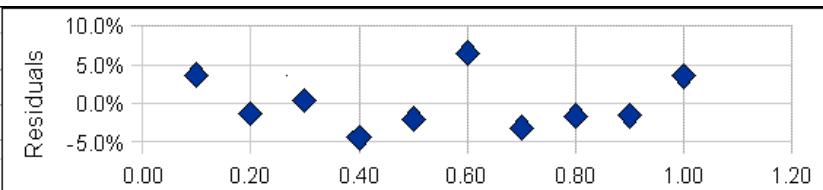
The graph on the left shows a third example, taken from analytical chemistry: a straight-line calibration



data set where X = concentration and Y = instrument reading ($Y = a + bX$). [Click to download that data](#). The blue dots are the data points. They don't 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 is 9.7926 and the intercept 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 particular calibration set. In this particular 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 particular measurement of 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's *mathematically* possible to use an even higher polynomial degree, up to one less than the number of data points, but it's 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 we 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* particular example were computer-generated to be *perfectly linear*; with normally-distributed random numbers added to simulate noise. So actually that 6th point is really 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: don't throw out data points just because they seem a little off, unless you have good reason, and don't 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. Don't be fooled. (*Full disclosure*: I obtained the above example by "[cherry-picking](#)" from among dozens of randomly generated data sets, 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 C_x is given by $C_x = (S_x - \text{intercept})/\text{slope}$, where S_x 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, but the problem of solving the calibration equation for concentration becomes [forbiddingly complex for higher order polynomial fits](#). (The concentration and the instrument readings can be recorded in any convenient units, as long as 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 model chosen; it's 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 we had a short-cut method that would let us *predict* the standard deviations of the coefficients from a *single measurement* of the signal, without actually 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](#).

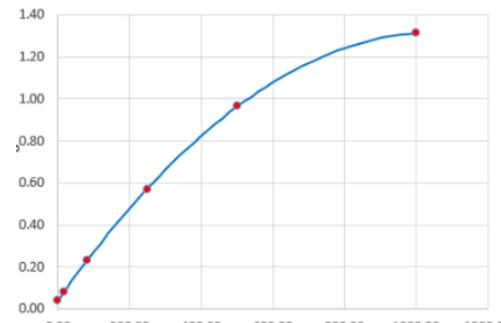
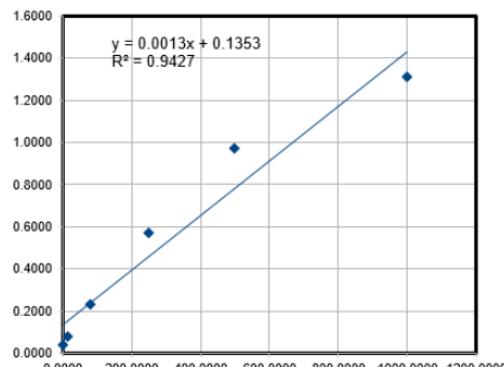
Algebraic Propagation of errors

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 result 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](#), below. 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'll 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'll 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

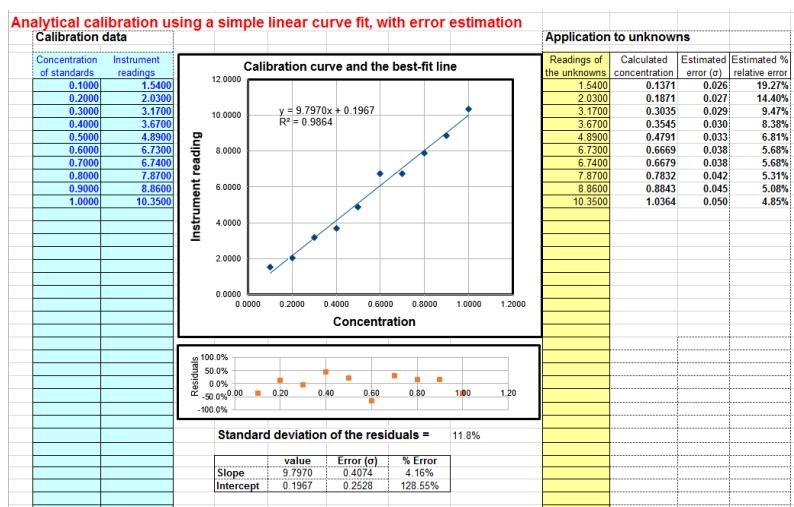
predictions are accurate only if the number of data points is large (and only if the noise is random and normally distributed). Beware: if the deviations from linearity in your data set are *systematic* and not *random* - for example, if try to fit a straight line to a smooth curved data set (left), 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 (right) will give the same slope and intercept from measurement to measurement.

In the application to analytical calibration, the concentration of the sample C_x is given by $C_x = (S_x - \text{intercept})/\text{slope}$, where S_x is the signal given by the sample solution. The uncertainty of all three terms contribute to the uncertainty of C_x . The standard deviation of C_x can be estimated from the standard deviations of slope, intercept, and S_x 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 fairly rough.



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 screen shot 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 a further discussion and some examples, see "[The Calibration Curve Method with Linear Curve Fit](#)". The downloadable 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. $Y = \text{intercept} + \text{slope} * X + \text{noise}$, where **noise** is the 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 RANDN function 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 (available online at <https://terpconnect.umd.edu/~toh/spectrum/MonteCarloAnimation.gif>), 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.

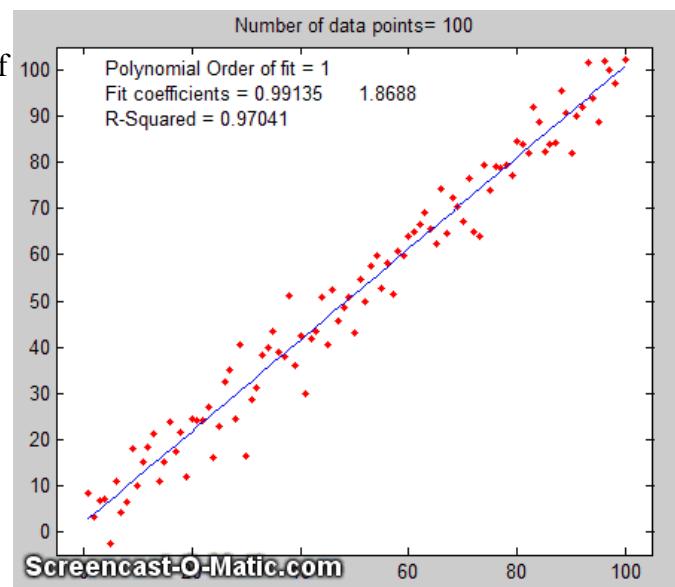
Obviously, this method involves programming a computer to compute the model and is not as convenient as evaluating a simple algebraic expression. But there are two important advantages to this method: (1) it has great generality; it can be applied to curve fitting methods that are too complicated for 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 don't 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 142).

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 more steady. 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 a given sample, and (b) the number of elements in each bootstrap sub-sample equals the number of elements in the original data set. As 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 the same number of data points, but with about a third of the data pairs skipped, a third duplicated, and a third left alone. (This is equivalent to weighting a third of the data pairs by a factor of 2, a third by 0, and a third unweighted). You would use a computer to generate hundreds or thousands of bootstrap samples like that and to apply the calculation procedure under investigation (in this case a linear least-squares) to each set.

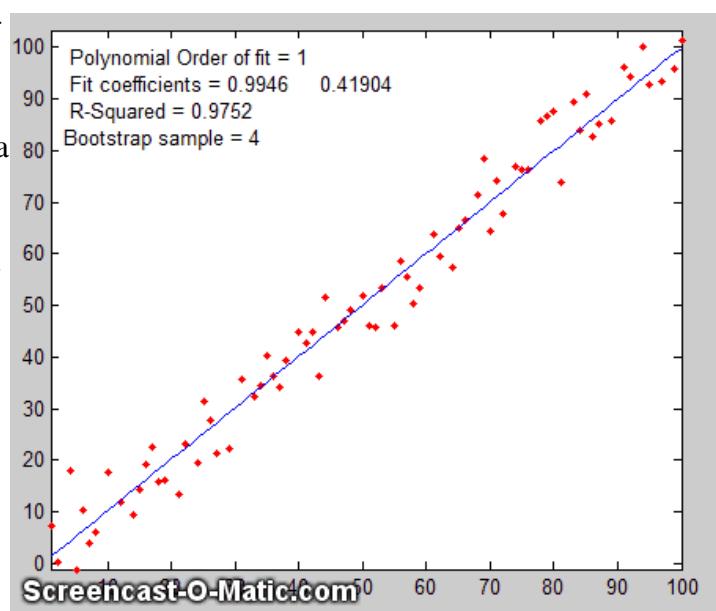


If there were *no* noise in the data set, and if the model were properly chosen, then all the points in the original data set and in all the bootstrap sub-samples would fall exactly on the model line, with the result that the least-squares results would be the *same for every sub-sample*.

However, if there *is* noise in the data set, each set 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, as the animation on the right demonstrates.

The process is illustrated by the animation on the right (available online at <https://terpconnect.umd.edu/~toh/spectrum/BootStrap.gif>), for the same 100-point straight-line data set used above.

You can see that the variation in the fit coefficients between sub-samples is the same as for the Monte Carlo simulation above. The greater the amount of random noise in the data set, the greater would be the range of results from sample in the bootstrap set. 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 is based on the assumption 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 also be done (with greater difficulty) in [spreadsheets](#).



Comparison of error prediction methods.

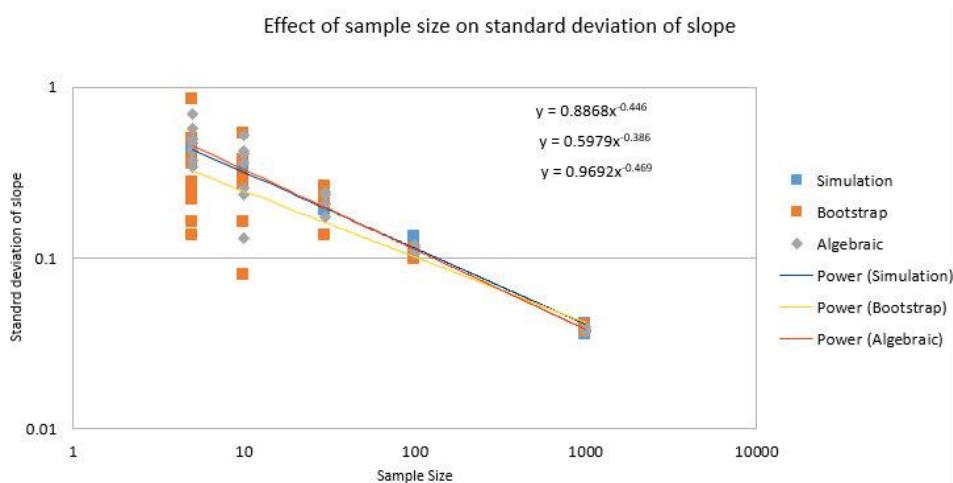
The Matlab/Octave script [TestLinearFit.m](#) compares *all three* of these methods (Monte Carlo simulation, the algebraic method, and the bootstrap method) for a 100-point first-order linear least-squares fit. Each method is repeated on different data sets with the same average slope, intercept, and 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
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 deviation ("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. Of course, the algebraic method is simpler and faster to compute than the other methods. However, an algebraic propagation of errors 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.



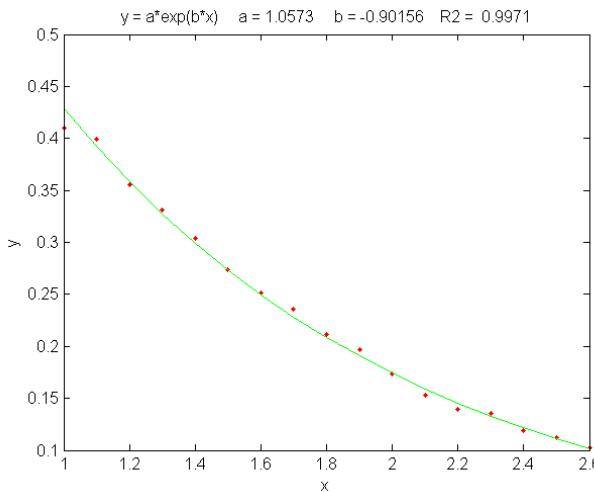
These plots really dramatize the problem of small sample sizes, but this must be balanced against the cost of obtaining more data points. For example, in analytical chemistry calibration, a larger number of calibration points could be obtained either by preparing and measuring more standard solutions 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 afterwards, 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 better to refine the laboratory techniques and instrument settings to minimize error than to attempt to compensate by taking lots of readings.

It's very important that the 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 most recent version of 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 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 least-squares method can be applied to the resulting linear equation. For example, the signal in the figure below is from a simulation of an exponential decay (X =time, Y =signal intensity) that has the mathematical form $Y = a \exp(bX)$, where 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, we obtain $\ln(Y) = \ln(a) + bX$. 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 in order to estimate $\ln(a)$ and b , from which you get a by computing $\exp(\ln(a))$. In this particular 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, in order 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 least-squares curve fitting of $\ln(Y)$ vs X .



An exponential least-squares fit (solid line) applied to a noisy data set (points) in order 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 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, a 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).

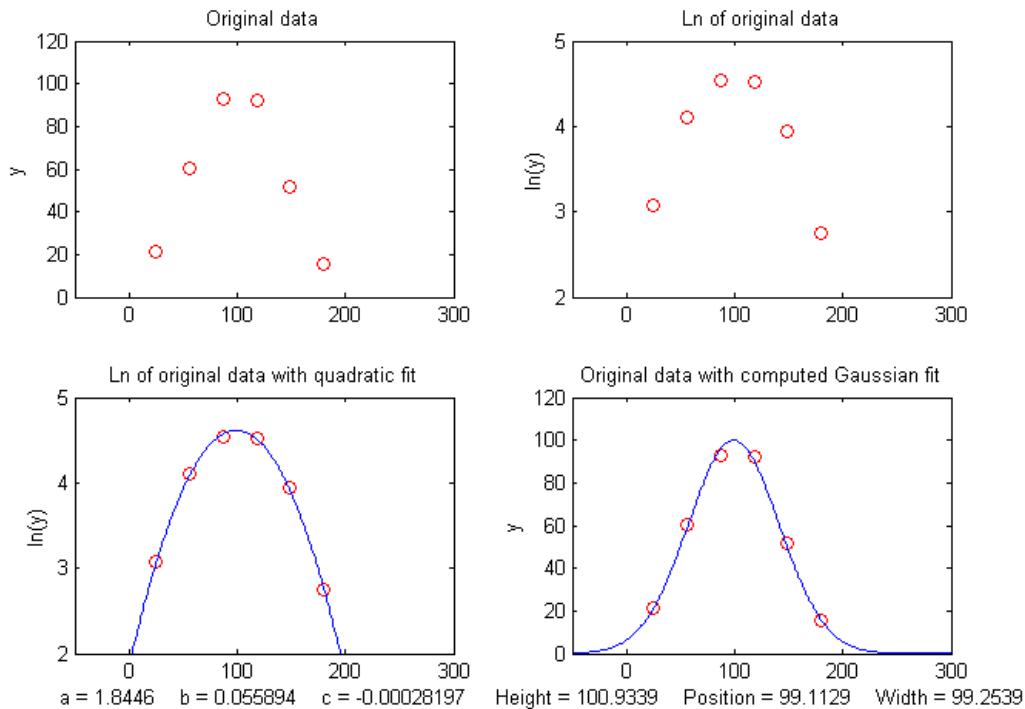
Another example of the linearization of an exponential relationship is explored on page 288: [Signal and Noise in the Stock Market](#).

Other examples of non-linear relationships that can be linearized by coordinate transformation include the logarithmic ($Y = a \ln(bX)$) and power ($Y=aX^b$) 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. (The downloadable 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](#).

Fitting Gaussian and Lorentzian peaks

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 ($y = a + bx + cx^2$). 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](#) $\ln(h) - (4 p^2 \ln(2)/w^2 + (8 p x \ln(2))/w^2 - (4 x^2 \ln(2))/w^2)$, 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**; it's a classic "3 unknowns in 3 equations" problem. 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"; see *Streamlining Digital Signal Processing: A "Tricks of the Trade" Guidebook*, Richard G. Lyons, ed., [page 298](#).

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 Gaussian peak with a true peak height of 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



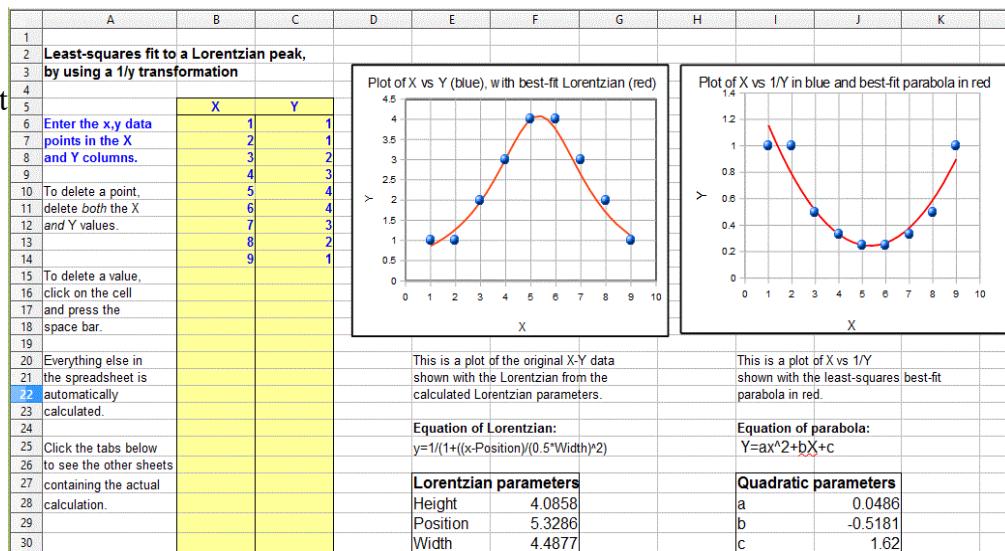
units on the x-axis. The [resulting data set](#), shown by the red points in the upper left, has only 6 data points on the peak itself. If we were to take the maximum of those 6 points (the 3rd point from the left, with $x=87$, $y=95$) as the peak maximum, we'd get only a rough approximation to the true values of peak position (100) and height (100). If we were to take the distance between the 2nd the 5th data points as the peak width, we'd get $3*31=93$, compared to the true value of 100.

However, taking the *natural log* of the data (upper right) produces a *parabola* that can be fit with a quadratic least-squares fit (shown by the blue line in the lower left). From the three coefficients of the quadratic fit, we 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). 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.

This figure 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's available in OpenOffice Calc ([Download link](#), [Screen shot](#)) and [Excel](#) formats). Note: in order 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 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).

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. (By the way, a quick way to test either of the above methods is to use this *simple 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).



In order to apply the above methods to signals containing *two or more* Gaussian or Lorentzian peaks, it's 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 198.

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 effected 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 depends not only on the amplitude of noise in the signal, but on also on how many points across the peak are taken for fitting. In particular, 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](#) 185). 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

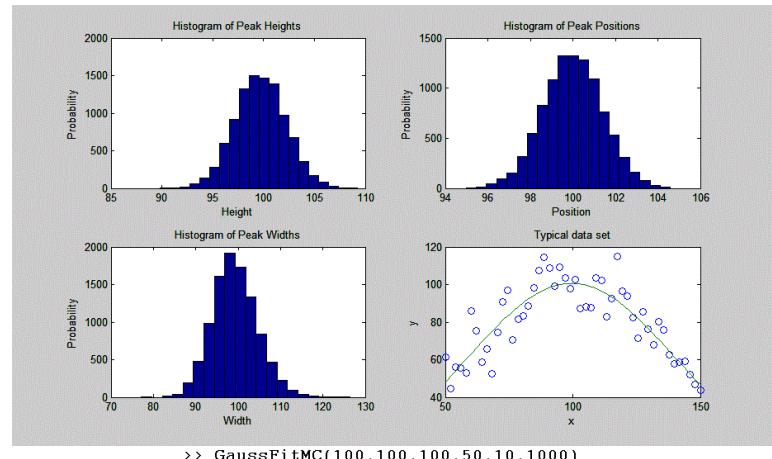
parameters are directly proportional to the noise in the data and inversely proportional to the square root of the number of data points (as expected), but that the proportionality constants differ:

relative standard deviation of the peak height = $1.73 * \text{noise}/\sqrt{N}$,

relative standard deviation of the peak position = noise/\sqrt{N} ,

relative standard deviation of the peak width = $3.62 * \text{noise}/\sqrt{N}$,

where *noise* is the standard deviation of the noise in the data and *N* in 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 134) 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 are 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

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, $y = \text{intercept} + \text{slope} \cdot x$, 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).

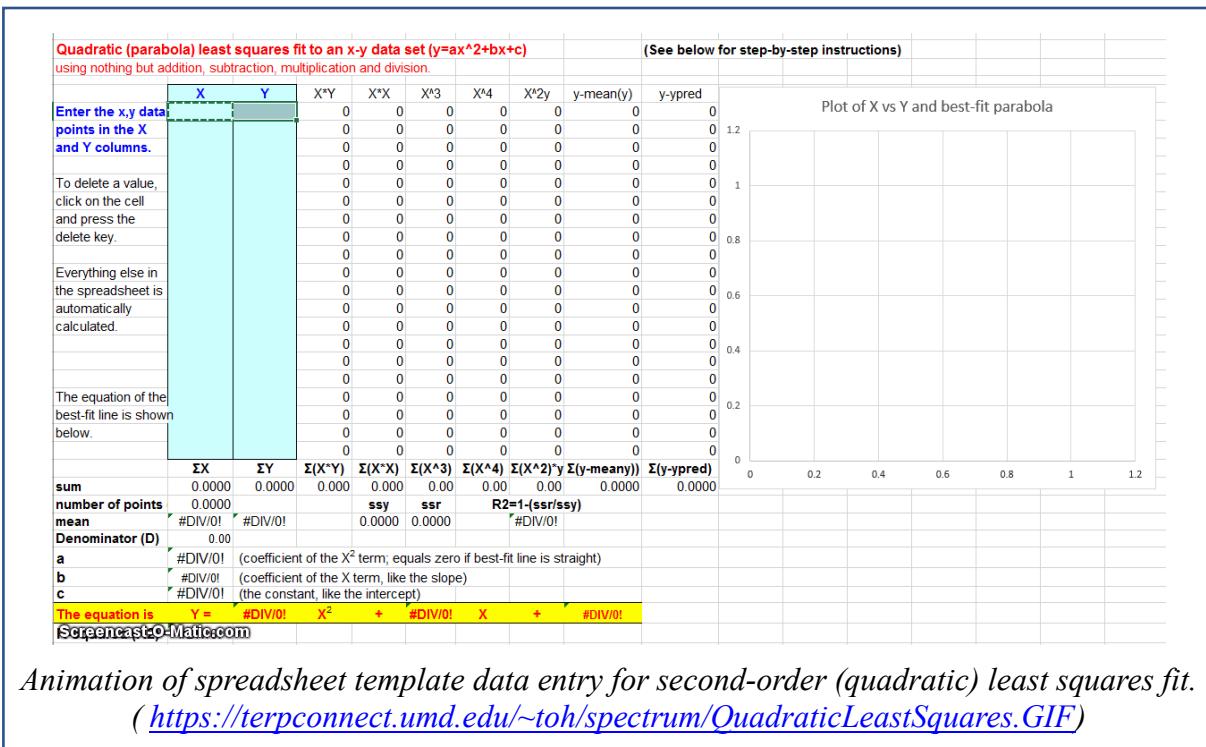
```
n = number of x,y data points  
sumx = Σx  
sumy = Σy  
sumxy = Σx*y  
sumx2 = Σx*x  
meanx = sumx / n  
meany = sumy / n  
slope = (n*sumxy - sumx*sumy) / (n*sumx2 - sumx*sumx)  
intercept = meany-(slope*meanx)  
ssy = Σ(y-meany)^2  
ssr = Σ(y-intercept-slope*x)^2  
R2 = 1-(ssr/ssy)  
Standard deviation 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 $\Sigma x*y$ 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 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 assumption of random deviations and also 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 ax^2+bx+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](#).



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 can't 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 the Macintosh, next, hold down the "apple" key and press "return." On the 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's 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 also calculate the concentrations of the unknowns (download complete set as [CalibrationSpreadsheets.zip](#)). These are described in detail starting on page 386. 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. A typical application of these spreadsheet templates to XRF (X-ray fluorescence) analysis is shown in this YouTube video: <https://www.youtube.com/watch?v=U3kzgVz4HgQ>

There is also another [set of spreadsheets](#) that 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 final 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 a 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 the peaks fits, the data may not contain zeros or negative points, and the baseline (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, you may perform the polynomial least squares calculations for the row vectors x,y *without* using the Matlab/Octave built-in polyfit function by using the [matrix method](#) with the Matlab "/" symbol, meaning "right matrix divide". The coefficients of a first order fit are given by

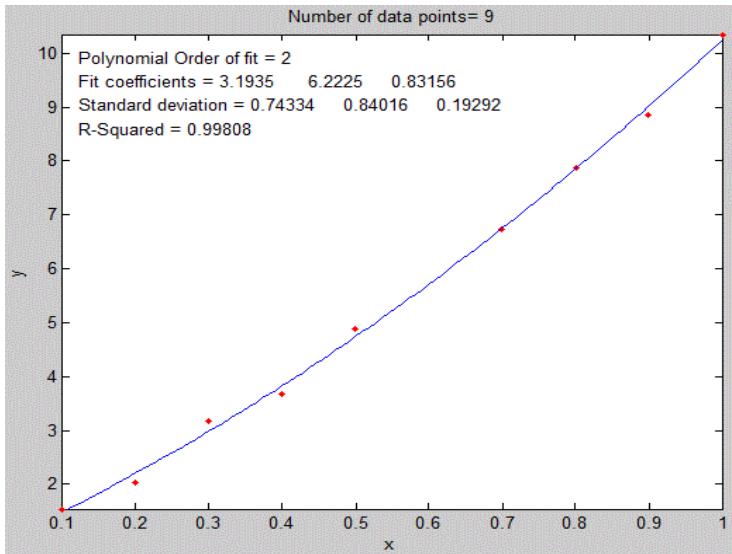
`y/[x;ones(size(y))]` and a second order (quadratic) fit by `y/[x.^2;x;ones(size(y))]`. For higher-order polynomials, just add another row to the denominator matrix, for example a third-order fit would be `y/[x.^3;x.^2;x;ones(size(y))]` and so on. The coefficients are returned in the same order as polyfit, 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
```

which shows that the *slope* and *intercept* results for the polyfit function and for the matrix method are the same. (The slope and intercept results are 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 131)).

The plotit.m function

The graph on the next page (click to see a larger size) was generated by a simple downloadable Matlab/Octave function [plotit\(data\)](#) or [plotit\(data,polyorder\)](#), that 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 $2 \times n$ or $n \times 2$ matrix with x in 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 green line and displays the fit coefficients and the goodness-of-fit measure R^2 in the upper left corner of the graph.

Here is a Matlab/Octave example of the use of plotit.m to perform the coordinate transformation described, on page 137, 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 ([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 'coef', 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. y_1 and y_2) 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 13:

```
[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 y_1 and y_2 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, $\text{coeff}(1)$, of 1.00. If the y_1 and y_2 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.

In **version 6** 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); a=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=axb : [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 288: [Signal and Noise in the Stock Market](#).

Don't forget that in Matlab/Octave, "log" means *natural log*; the *base-10* log is denoted by "log10".

Estimating the coefficient errors. The [plotit](#) function version 2 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 that data and prints out a table of bootstrap error estimates, with the slope in the first column and the intercept in the second column:

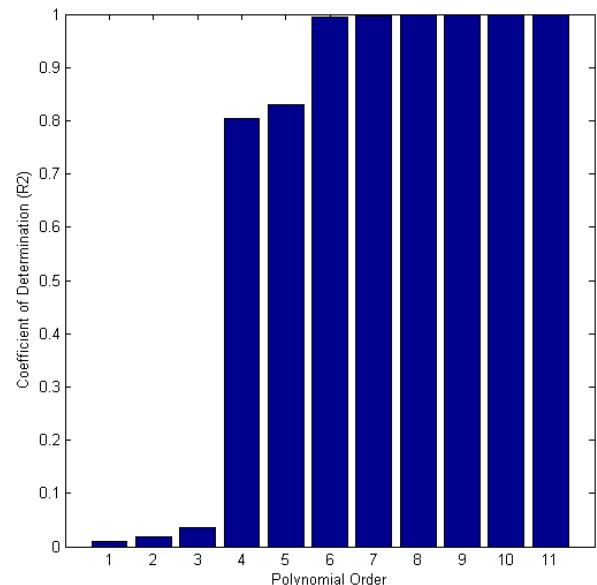
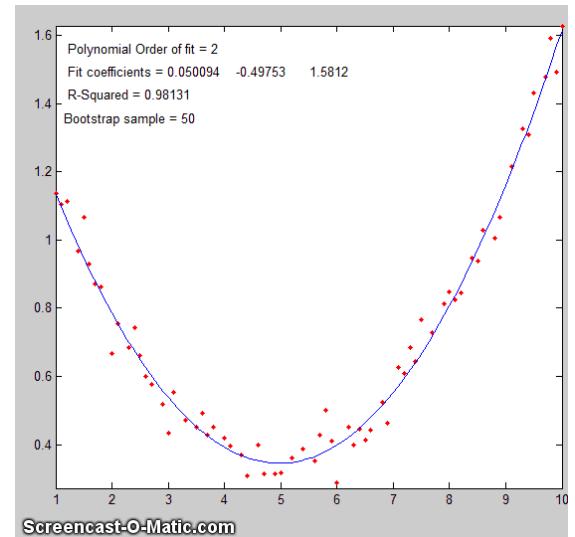
Bootstrap Results	
Mean:	100.359
STD:	0.204564
STD (IQR):	0.291484
% RSD:	0.203832
% RSD (IQR):	0.290441
	88.01638
	15.4803
	20.5882
	17.5879
	23.3914

The variation [plotfita](#) animates the bootstrap process for instructional purposes, as shown in the animation on the right a quadratic fit. You must include the output arguments, for example:

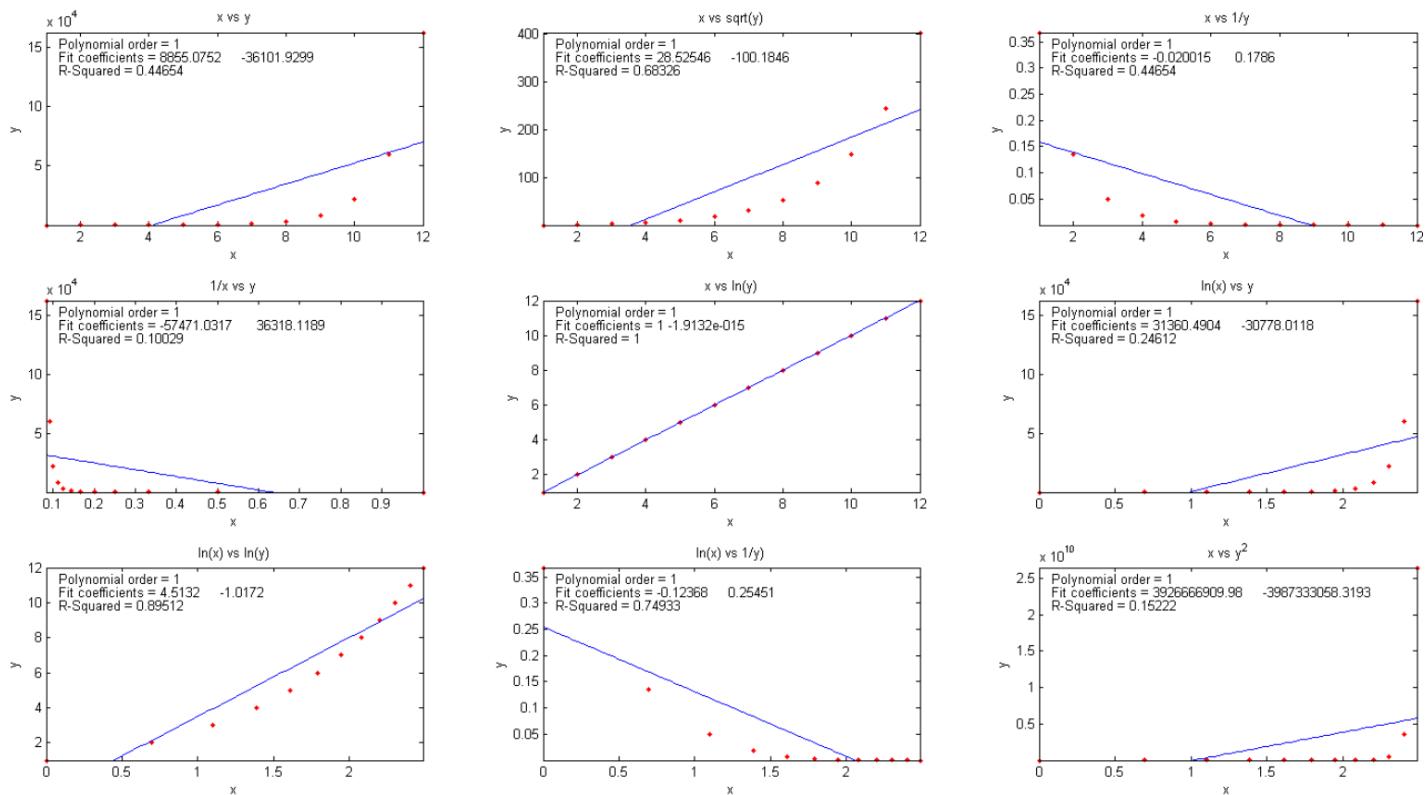
```
[coef, RSquared, BootResults]=plotfita([1 2 3 4 5 6],[1 3 4 3 2 1],2);
```

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 downloadable function [trypoly\(x,y\)](#) fits the data in x,y with a series of polynomials of degree 1 through $\text{length}(x)-1$ and returns the coefficients of determination (R2) of each fit as a vector, showing that, for *any* data, the coefficient of determination R2 approaches 1 as the polynomial order approaches $\text{length}(x)-1$. 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 R2 values in a vector. In the example below, for polyorder=1, it's the 5th one that's best – x vs ln(y).



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 as the previous example but also displays error estimates in a table 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's important that the noisy signal not be smoothed if the bootstrap error predictions are to be accurate. Smoothing the data will cause 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 demo 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 effects the accuracy. This function needs the gaussian.m, gaussfit.m, and peakfit.m functions in the path. The script also performs a measurement by the [iterative method](#) (page 163) using peakfit.m, which is [more accurate but takes about times longer to compute](#).

The downloadable Matlab-only functions [iSignal.m](#) (page 323) and [ipf.m](#) (page 361), 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. 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 spectra 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 multi-wavelength instruments 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's 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'll actually 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.

ϵ = analytical sensitivity (slope of a plot of A vs c)

A = analytical signal

M^T = matrix transpose of matrix M (rows and columns switched).

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 ϵ .

$$A = \epsilon c$$

The total signal is the sum of the signals for each component in a mixture:

$$A_{\text{total}} = A_{c1} + A_{c2} + \dots \text{ for all } n \text{ components.}$$

Classical Least Squares (CLS) calibration

This method is applicable to the quantitative analysis of a mixture of components you can measure the spectra of the individual components and where 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 ϵ at each wavelength to be determined. Then it follows that:

$$A_{w1} = \epsilon_{c1,w1} c_{c1} + \epsilon_{c2,w1} c_{c2} + \epsilon_{c3,w1} c_{c3} + \dots \text{ for all } n \text{ components.}$$

$$A_{w2} = \epsilon_{c1,w2} c_{c1} + \epsilon_{c2,w2} c_{c2} + \epsilon_{c3,w2} c_{c3} + \dots$$

and so on for all wavelengths - w3, w4, etc. It's messy 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 and so the calculations required here are actually rather simple and certainly very easy for a computer to do. So, *we really need a correspondingly simple notation* that is more compact. To do this, it's 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:

$$\mathbf{A} = \mathbf{\epsilon}\mathbf{C}$$

where **A** is the *w*-length vector of measured signals (i.e. the signal spectrum) of the mixture, **ε** is the $n \times w$ rectangular matrix of the known **ε**-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. **εC** means that **ε** "pre-multiplies" **C**; that is, *each column of ε is multiplied point-by-point by the vector C*.

If you have a sample solution containing unknown amounts of components those *n* components, you measure its spectrum **A** and seek to calculate the concentration vector of concentrations **C**. In order 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 we have a system of *n* equations in *n* unknowns which can be solved by pre-multiplying both sides of the equation by **ε**⁻¹, the [matrix inverse](#) of **ε**, and using the property that any matrix times its inverse is unity:

$$\mathbf{C} = \mathbf{\epsilon}^{-1}\mathbf{A}$$

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 **ε** matrix is a *w* × *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 $(\mathbf{\epsilon}^T\mathbf{\epsilon})^{-1}\mathbf{\epsilon}^T$:

$$(\mathbf{\epsilon}^T\mathbf{\epsilon})^{-1}\mathbf{\epsilon}^T\mathbf{A} = (\mathbf{\epsilon}^T\mathbf{\epsilon})^{-1}\mathbf{\epsilon}^T\mathbf{\epsilon}\mathbf{C} = (\mathbf{\epsilon}^T\mathbf{\epsilon})^{-1}(\mathbf{\epsilon}^T\mathbf{\epsilon})\mathbf{C}$$

where **ε**^T is the *transpose* of **ε** (rows and columns switched). But the quantity $(\mathbf{\epsilon}^T\mathbf{\epsilon})^{-1}(\mathbf{\epsilon}^T\mathbf{\epsilon})$ is a matrix times its inverse and is therefore unity. Thus, we can simplify the result to:

$$\mathbf{C} = (\mathbf{\epsilon}^T\mathbf{\epsilon})^{-1}\mathbf{\epsilon}^T\mathbf{A}$$

In this expression, **ε**^T**ε** is a square matrix of order *n*, the number of components. In most practical applications, *n* is relatively small, perhaps only 2 to 5. The vector **A** is of length *w*, the number of wavelengths. That can be quite large, perhaps several hundred; in general, the more wavelengths are used, the more effectively the random noise will be averaged out (although it won't 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 \mathbf{E} matrix.

Two extensions of the CLS method are commonly made. First, in order to account for baseline shift caused by drift, background, and light scattering, a column of 1s is added to the \mathbf{E} matrix. This has the effect of introducing into the solution an additional component with a flat spectrum; this is referred to as “background correction”. Second, in order 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:

$$\mathbf{C} = (\mathbf{E}^T \mathbf{E}^{-1} \mathbf{E})^{-1} \mathbf{E}^T \mathbf{V}^{-1} \mathbf{A}$$

where \mathbf{V} is an $w \times w$ 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 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 manual wavelength control, because of insufficient wavelength reproducibility. Specifically, many measurements are made 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 works well with automated computer-controlled scanning instruments, and is perfectly 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. The well-known [deviations from analytical curve linearity](#) in absorption spectrophotometry set a limit to the performance to this method, but that can be avoided by applying iterative curve fitting (page 163) and Fourier convolution (page 93) to the transmission spectra, an idea that idea will be developed on page 237.

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 method models the signal at each wavelength as the sum of the concentrations of the analyte times the analytical sensitivity, the inverse least squares methods use the reverse 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:

$$c_{s1} = m_{w1}A_{s1,w1} + m_{w2}A_{s1,w2} + m_{w3}A_{s1,w3} + \dots \text{ for all } w \text{ wavelengths.}$$

$$c_{s2} = m_{w1}A_{s2,w1} + m_{w2}A_{s2,w2} + m_{w3}A_{s2,w3} + \dots$$

and so on for all s samples. In matrix form

$$\mathbf{C} = \mathbf{AM}$$

where \mathbf{C} is the s -length vector of concentrations of the analyte in the s samples, \mathbf{A} is the $w \times s$ matrix of measured signals at the w wavelengths in the s samples, and \mathbf{M} is the w -length vector of calibration coefficients.

Now, suppose that you have a 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 \mathbf{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 s -length vector of concentrations \mathbf{C} . Together these data allow you to calculate the calibration vector \mathbf{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:

$$\mathbf{M} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{C}$$

(Note that $\mathbf{A}^T \mathbf{A}$ 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 but not 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 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 as long as 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

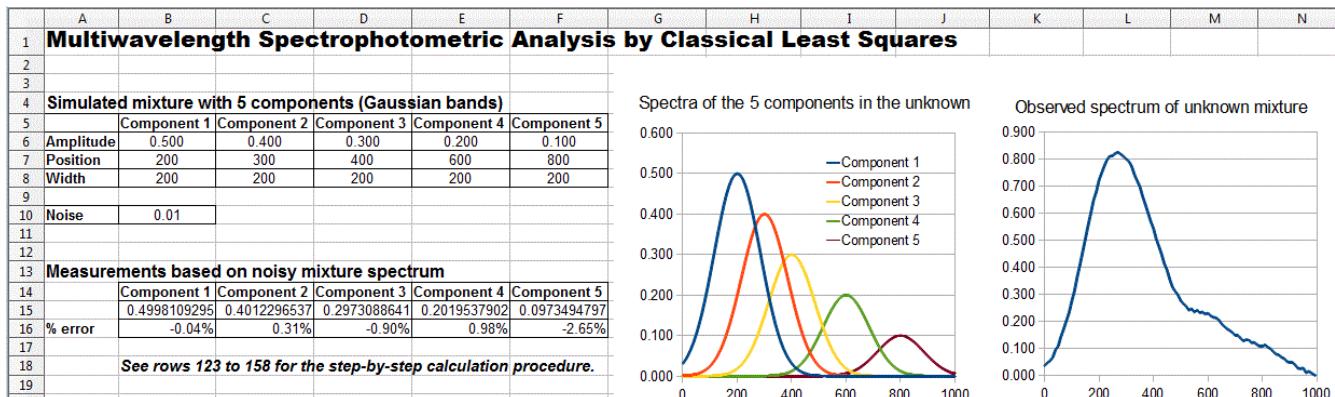
Most modern spreadsheets have basic matrix manipulation capabilities and can be used for multi-component calibration, for example [Excel](#) and [OpenOffice Calc](#). The spreadsheets [RegressionDemo.xls](#) and [RegressionDemo.ods](#) (for Excel and Calc, respectively) demonstrate the classical least squares procedure for a simulated spectrum of a 5-component mixture measured at 100 wavelengths. A screen shot is shown below. The matrix calculations described above that solves for the concentration of the components on the unknown mixture:

$$\mathbf{C} = (\mathbf{\mathbf{\mathbf{E}}^T \mathbf{E}})^{-1} \mathbf{\mathbf{\mathbf{E}}^T \mathbf{A}}$$

are 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 scary cell equation:

$$\mathbf{C} = \text{MMULT}(\text{MMULT}(\text{MINVERSE}(\text{MMULT}(\text{TRANSPOSE}(\mathbf{E});\mathbf{E}));\text{TRANSPOSE}(\mathbf{E}));\mathbf{A})$$

where \mathbf{C} is the vector of the 5 concentrations of all the components in the mixture, \mathbf{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 \mathbf{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](#)".



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

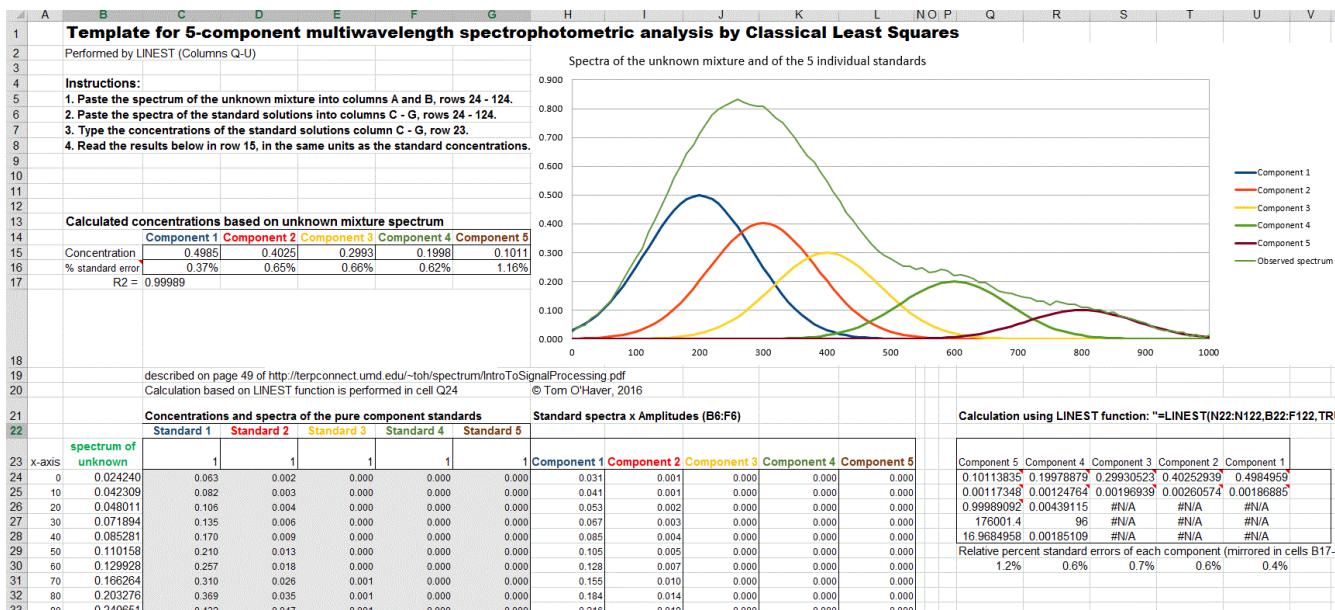
the same operation; using Matlab or Octave, that would require some extra work. (LINEST is also an array function that must also be entered by typing **Ctrl-Shift-Enter**, not just **Enter**). Note that this is the *same* LINEST function that was previously used for polynomial least-squares (page 126) 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 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 it 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 also 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 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 (page 313) in cell Q23 and in row 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'll see that it's exactly the same in both of those two example templates, even though the number of wavelengths and the number of components is different. You'll still have to adjust the graph, however, to cover the desired x-axis range. See page 313.



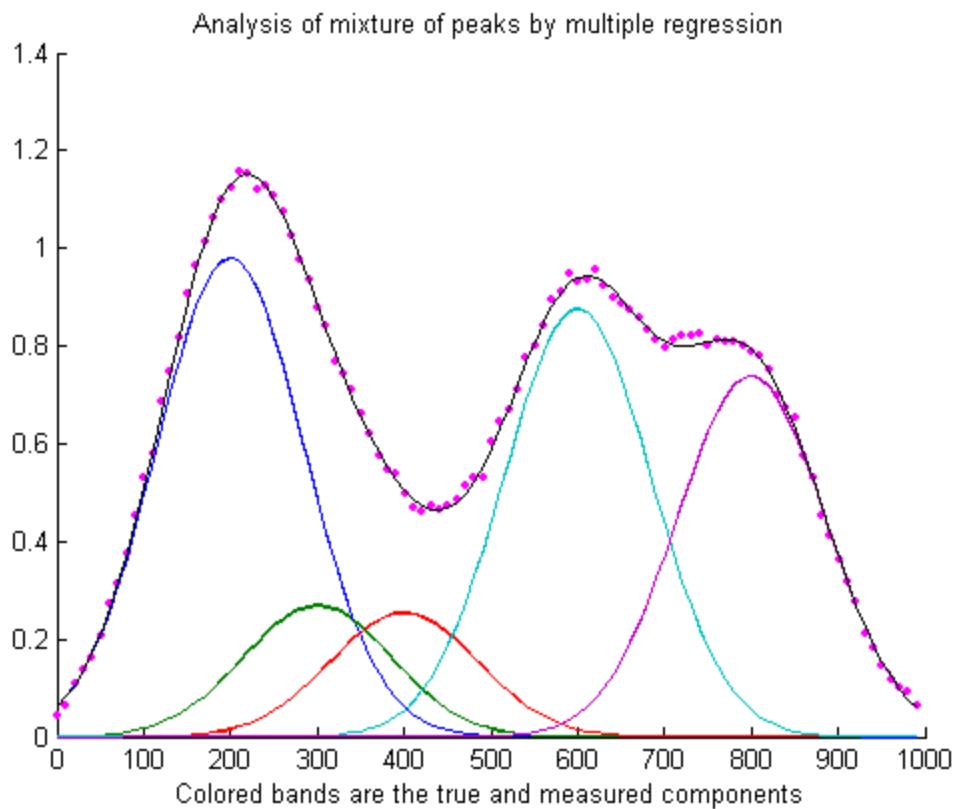
Excel template applied to the measurement of a 5-component unknown mixture at 100 wavelengths.

Matlab and Octave

Matlab and Octave are really the natural computer approaches to 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 these languages, 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

$$C = \text{inv}(E' * E) * E' * A$$

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, it's 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 in principle more accurate with respect to the numerical precision of the computer (usually negligible compared to the noise in the signal; see page 302).



The script [RegressionDemo.m](#) (for Matlab or Octave) demonstrates the classical least squares procedure for a simulated absorption spectrum of a 5-component mixture at 100 wavelengths, illustrated above. Most of this script is just signal generation and plotting; the actual least-squares regression is performed on one line:

```
MeasuredAmp = ObservedSpectrum * A' * inv(A * A')
```

where different symbols are used for the variables: "A" is a matrix containing the spectrum of each of the components in each of its rows and "ObservedSpectrum" is the observed spectrum of the unknown mixture. In this example, the dots represent the observed spectrum of the mixture (with noise) and the five colored bands represent the five components in the mixture, whose spectra are known but whose concentrations in the mixture are unknown. The black line represents the "best fit" to the observed spectrum calculated by the program. In this example, the concentrations of the five components are measured to an accuracy of about 1% relative (limited by the noise in the observed spectrum).

Comparing [RegressionDemo.m](#) to its spreadsheet equivalent, [RegressionDemo.ods](#), both running the same computer, you can see that the Matlab/Octave code computes and plots the results quicker than the spreadsheet, although both take no more than a fraction of a second for this example.

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 **T-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

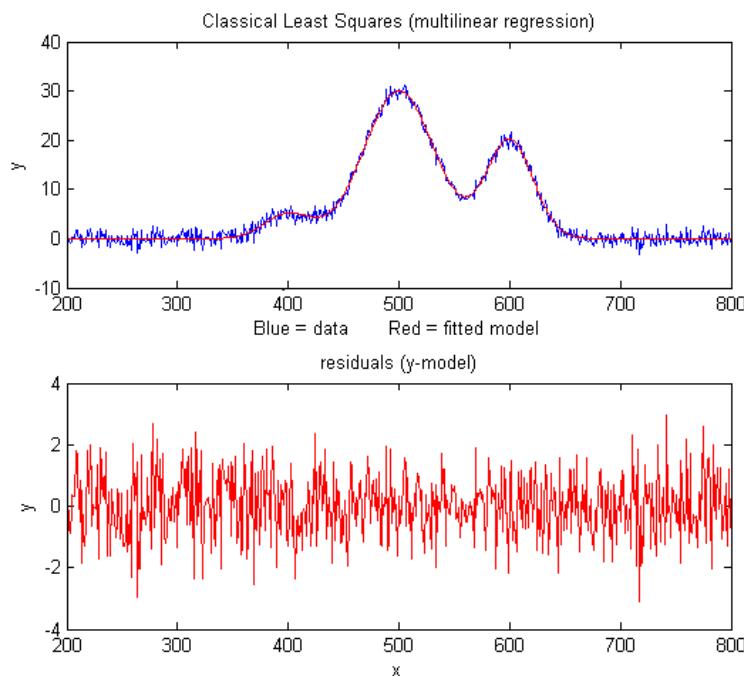
(c.f. <http://www.mathworks.com/help/techdoc/ref/mldivide.html>).

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 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 163) using the downloadable 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 faster and (usually) more accurate, especially if the peaks are highly overlapped. (This script requires `cls.m`, `modelpeaks.m`, and `peakfit.m` in the Matlab/Octave path).



```
Figure window(1) : Classical Least Squares (multilinear regression)
Elapsed time is 0.000937 seconds.
Average peak height accuracy = 0.9145%
```

```
Figure window(2) : Iterative non-linear peak fitting with peakfit.m
Elapsed time is 0.171765 seconds.
Average peak height accuracy = 1.6215%
```

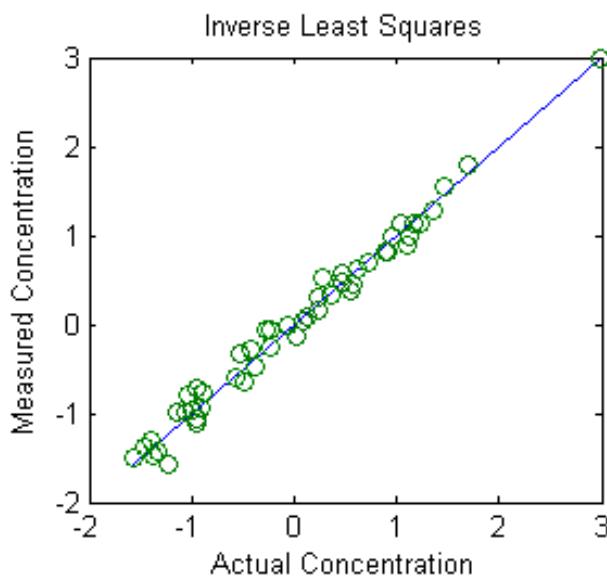
On the other hand, INLS can be better than CLS if there are *unsuspected shifts* in peak position and/or peak width between calibration and measurement (for example caused by drifting spectrometer calibration or by changing temperature, pressure, or solution variables) because INLS can track and compensate 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](#).

The [cls2.m](#) function is similar to [cls.m](#), except that it also measures the baseline (assumed to be flat), using 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...].

Weighted linear regression: The downloadable Matlab/Octave function "[tfit.m](#)" simulates the measurement of the absorption spectrum of a mixture of three components by *weighted* linear regression (on line 61), 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 237).

The **Inverse Least Squares (ILS)** technique is demonstrated in Matlab by [this script](#) and the graph above. The math, described above on page 154, is similar to the Classical Least Squares method, and can be done by any of the Matlab/Octave or spreadsheet methods described in this section. This example is a real data set derived from the [near infrared \(NIR\) reflectance spectroscopy](#) of agricultural wheat samples analyzed for protein content. In this example there are 50 calibration samples measured at 6 wavelengths. These calibration samples had already been analyzed for [protein content](#) by a reliable [reference method](#). The purpose of this calibration is to establish whether near-infrared reflectance spectroscopy, which can be measured on wheat paste preparations much more quickly than reliable, but possibly laborious and time-consuming, wet chemical methods, correlates to their protein content as determined by the reference method. These results indicate that it does, at least for this set of 50 wheat samples, and therefore is it 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 for the protein content), but this is a very common analytical situation in industrial and agricultural *quality control*, where large numbers of samples of a similar predictable type must often be tested quickly and cheaply.

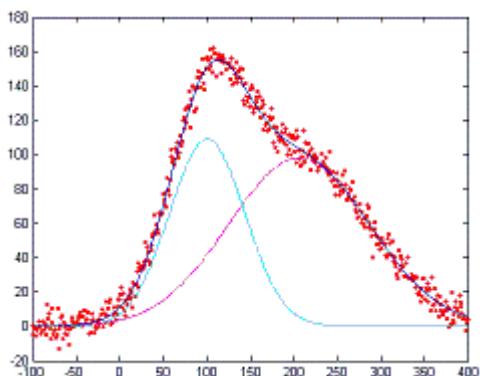
Note: you can right-click on any of the m-file links above and select **Save Link As...** to download them to your computer for use within Matlab.



Curve fitting C: Non-linear Iterative Curve Fitting

Least squares curve fitting, described in "Curve Fitting A" on page 126, is simple and fast, but it is limited to situations where the dependent variable can be modeled as a polynomial with linear coefficients. We saw that in some cases a non-linear situation can be converted into a linear one by a coordinate transformation, but this is possible only in some special cases, it may restrict the range of allowable dependent data values, and, in any case, the resulting coordinate transformation of the noise in the data can result in imprecision in the parameters measured in this way.

The most general way of fitting any model to a set of data is the [iterative method](#), (a.k.a. "spectral deconvolution" or "peak deconvolution"). It is a kind of "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 only grudgingly applied. 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.



Iterative methods proceed in the following general way:

- (1) Select a model for the data;
- (2) Make first guesses of all the non-linear parameters;
- (3) A computer program computes the model and compares it to the data set, calculating a fitting error;
- (4) If the fitting error is greater than the required fitting accuracy, the program systematically changes the parameters and loops back around to the previous step and repeats until the required fitting accuracy is achieved or the maximum number of iterations is reached. This continues until the fitting error is less than the specified error.

One popular technique for doing this, called the [Nelder-Mead Modified Simplex](#), is essentially 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. 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 at <https://terpconnect.umd.edu/~toh/spectrum/FittingAnimation.gif> 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 (page 152) for each trial iteration). In order to allow the process to be observed in action, this animation is *slowed down artificially* by plotting step-by-step, making a bad initial guess and adding a "pause ()". The entire process normally takes only about 0.05 seconds on a standard desktop PC, depending mainly on the number of nonlinear variables that must be iterated.

The main difficulty of the iterative methods is that they sometime fail to converge at an optimum solution in difficult cases. The standard approach to handle this is to restart the algorithm with another set of first guesses; repeat that several times and take the one with the lowest fitting error (that process is automated in the peak fitting functions described on page 343). Iterative curve fitting also takes longer than linear regression - with typical modern personal computers, an iterative fit might take fractions of a second where a regression would take fractions of a millisecond. Still, this is fast enough for many purposes.

The precision of the model parameters measured by iterative fitting (page 174), 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 134) are applicable.

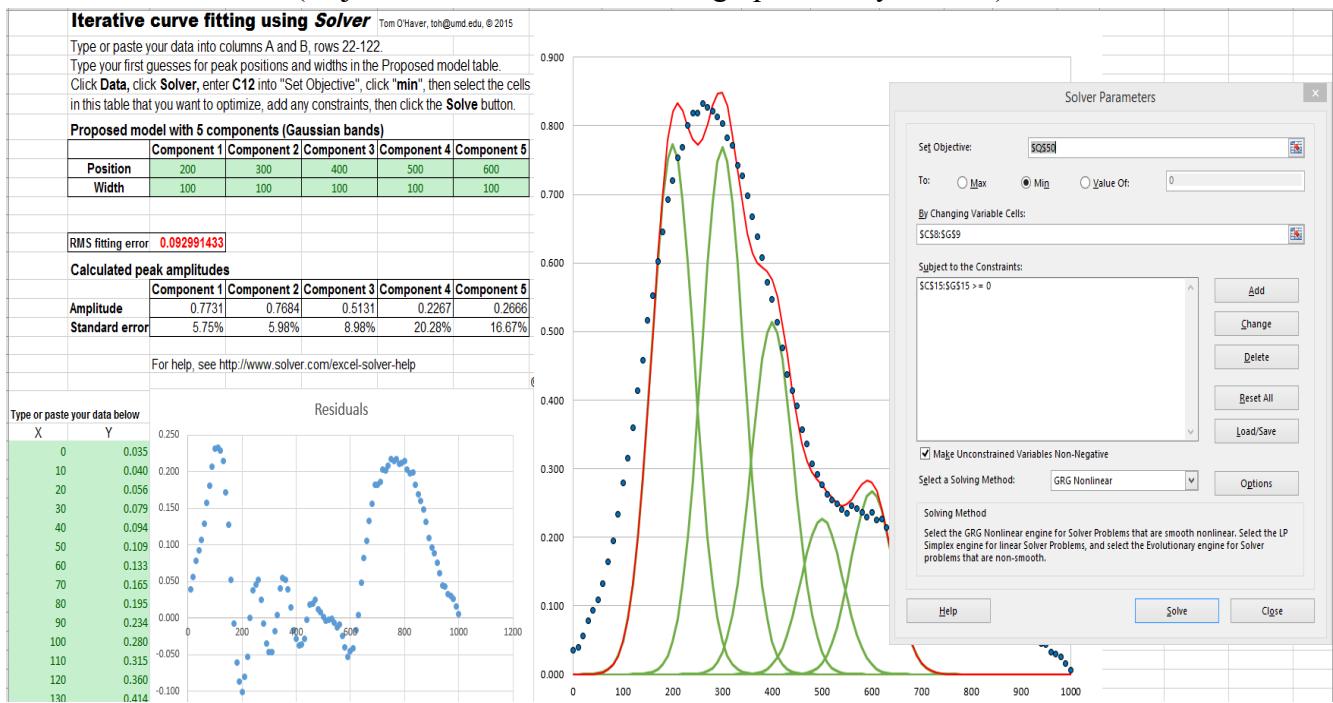
Note: the term "spectral deconvolution" or "band deconvolution" is often used to refer to this technique, but in this essay, "deconvolution" specifically means *Fourier* deconvolution, an independent concept that is treated on page 96. In Fourier deconvolution, the underlying peak shape is *unknown*, but the broadening function is assumed to be *known*; whereas in iterative least-squares curve fitting it's 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; don't get them confused. See page 268: [Fourier deconvolution vs curve fitting](#).

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 \(screen shot\)](#) 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

152) 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 278.

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 fit with an increasing sequence of [two Gaussians](#), [three Gaussians](#), [four Gaussians](#), and [five Gaussians](#). As you look at this sequence of screenshots, you'll see the percent fitting error decrease, the R^2 value become 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 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.

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 [freeware](#) and commercial programs, including [PeakFit](#), [Data Master 2003](#), [MyCurveFit](#), [Curve Expert](#), [Origin](#), [ndcurvemaster](#), and the [R language](#).

If you use a spreadsheet for this type of curve fitting, you have 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's 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](#), [5Gaussians](#), 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).

In some cases, it is useful to add *constraints* to the variables determined by iteration, for example to constrain them to be greater than zero, or constrained between two limits, or equal to each other, 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 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 343) *automatically* adapts to any number of data points and is easily set to over 40 different model peak shapes (graphic on page 369) and any number of peaks simply by changing the input arguments. Using my *Interactive Peak*

Fitter function [ipf.m](#) in Matlab (page 361), you can *press a single keystroke* to instantly change the peak shape, number of peaks, baseline mode (page 182), or to re-calculate the fit with a different start value or with a bootstrap subset of the data. That's far quicker and easier than the spreadsheet. But on the other hand, 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's 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.

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 an [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 many 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](#). (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](#)). Incandescent lightbulbs, of the type that used to be common in household lighting, are examples of black-body radiators.

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 138), 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 in in [Demofitgauss2.m](#) (shown on the left).

To accommodate the possibility that the baseline may shift, we can add a column of 1s to the A matrix, just as was done in the CLS method (page 152). This has the effect of introducing an additional "peak" into the model that has an amplitude but no position or width. The baseline amplitude is returned along with the peak heights in the global vector "c"; [Demofitgaussb.m](#) and [fitgauss2b.m](#) illustrates 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. Updated to Matlab 6, 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 $2n$, one entry for each iterated variable ([position1 width1 position2 width2....etc.]). The "for" loop (lines 5-7) constructs a $n \times \text{length}(t)$ 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 $A\backslash y'$ with $\text{abs}(A\backslash y')$ 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 $2 \times n$ 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: in order 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 path. Functions, on the other hand, 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).

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 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. It's all about the bottom line. This makes fitshape.m a good platform for experimenting with different mathematical expression 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...").

Variable shape types, such as the Voigt profile, Pearson, [Breit-Wigner-Fano](#), Gauss-Lorentz blend, and the exponentially-broadened Gaussian and Lorentzian, are defined not only by a 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 [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. (These are self-contained functions). Version 7 of the downloadable Matlab/Octave function [peakfit.m](#) includes 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).

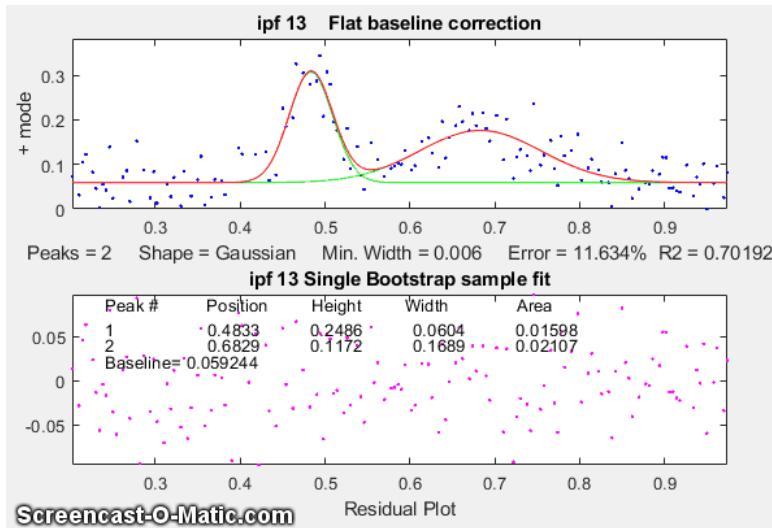
If you do *not* know the shape of your peaks, you can use peakfit.m or ipf.m (page 361) to try different shapes to see if one of the standard shapes included in those programs fits the data; try to find a peak in your data that is typical, isolated, and that has a good signal-to-noise ratio. For example, the Matlab functions [ShapeTestS.m](#) and [ShapeTestA.m](#) 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 R2 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 still it is 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 237), a model of the instrumentally-broadened transmission spectrum is fit to the observed transmission data, using a [fitting function](#) that performs Fourier convolution (page 93) 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 Fitters for Matlab and Octave



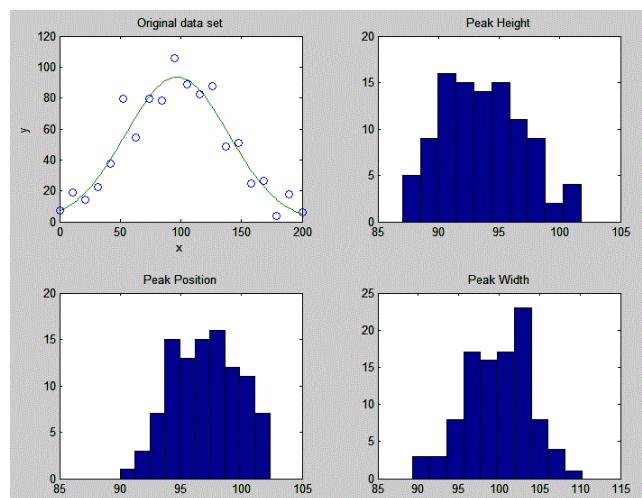
exponentially-broadened Gaussian, exponentially-broadened equal-width Gaussians, bifurcated Gaussian, Lorentzian, fixed-width Lorentzians, equal-width Lorentzians, exponentially-broadened Lorentzian, bifurcated Lorentzian, logistic distribution, logistic function, triangular, alpha function, Pearson 7, exponential pulse, up sigmoid, down sigmoid, Gaussian/ Lorentzian blend, Breit-Wigner-Fano, and Voigt profile shapes. (A graphic that illustrates the basic peak shapes available is on page 369).

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](#), for Matlab only (page 361). For adding as an element on your own programs and for automating the fitting of large numbers of signals, [peakfit.m](#) is better; [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.

These functions can optionally estimate the expected standard deviation and interquartile range of the peak parameters using the bootstrap sampling method (page 134). See [DemoPeakfitBootstrap](#) for a self-contained demonstration of this function.

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 don’t 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,

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](#) (page 134). A simple demonstration of bootstrap estimation of the variability of an iterative least-squares fit to a single noisy Gaussian peak is given by the custom downloadable 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. An typical example for `BootstrapIterativeFit(100,100,100,20,10,100)`; is displayed in the figure on the right.



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. An typical example for `BootstrapIterativeFit(100,100,100,20,10,100)`; is displayed in the figure on the right.

```
>> 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's 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 take into account 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's 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's 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 35). 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        0.11571
```

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's also instructive to compare the iterative least-squares method with *classical least-squares curve fitting*, discussed on page 152, 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* unknowns 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 are *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's 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 three 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 259).

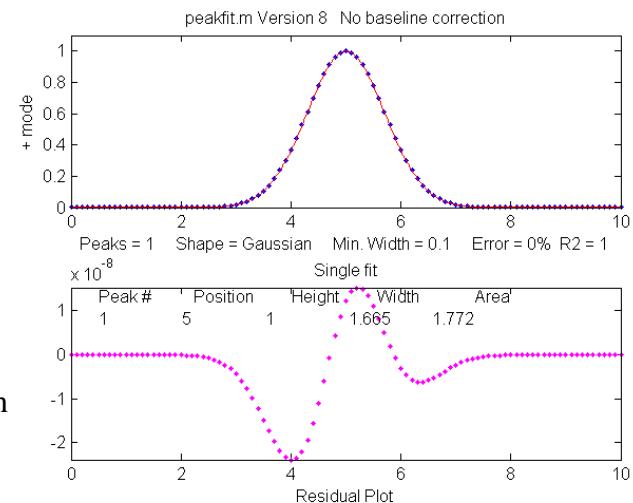
Note: If you are reading this book online, you can right-click on any of the m-file links and select **Save Link As...** to download them to your computer, then place them in the Matlab path for use within Matlab.

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. (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 343), as shown on the right. (The 5th input argument for the peakfit function specifies the shape of peaks to be used in the fit; "1" means Gaussian).



```
>> 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. R2 is the "R-squared" or coefficient of determination, which is exactly 1 for a perfect fit. Note the good agreement between the area (1.7725) with the *theoretical* area under the curve of $\exp(-x^2)$, (click for [Wolfram Alpha solution](#)), which is the [square root of pi](#).

But this same peak, when fit 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 don't 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
FitResults =
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), this peak gives a 6% fitting error and height, width and area errors of 8%, 20%, and 17%, respectively.

```

>> [FitResults,MeanFitError]=peakfit([x' y'],5,10,1,2)

```

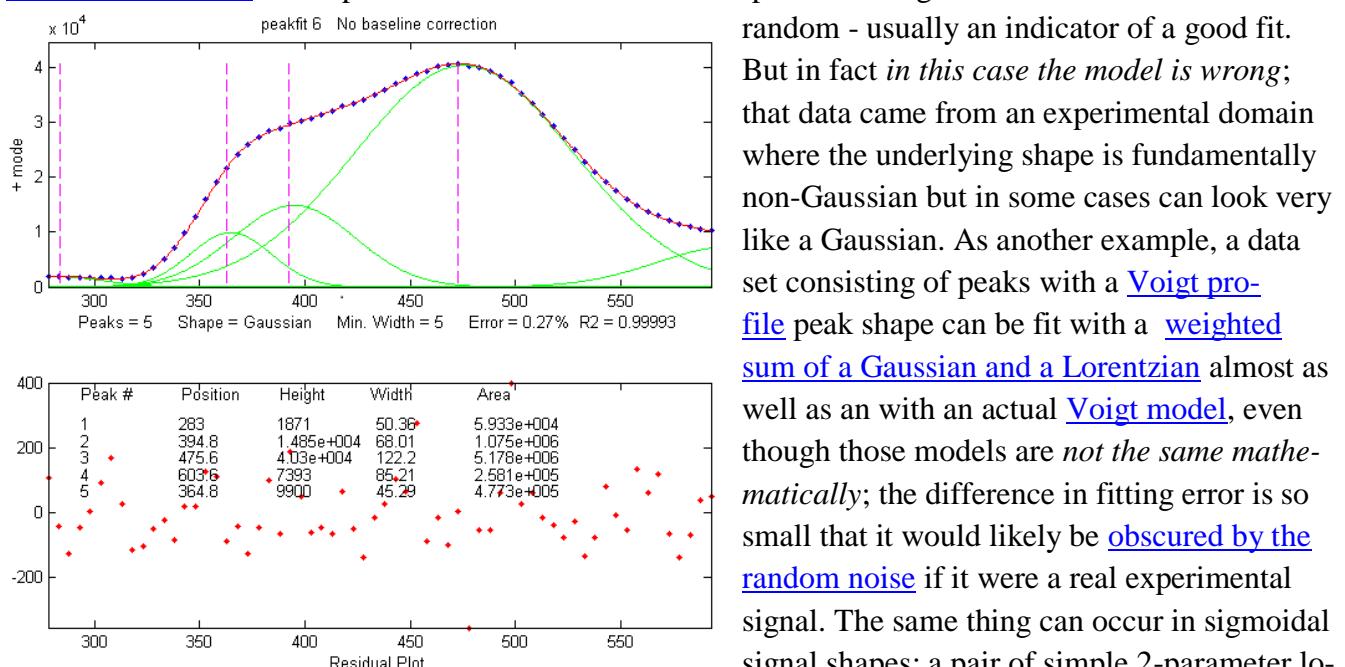
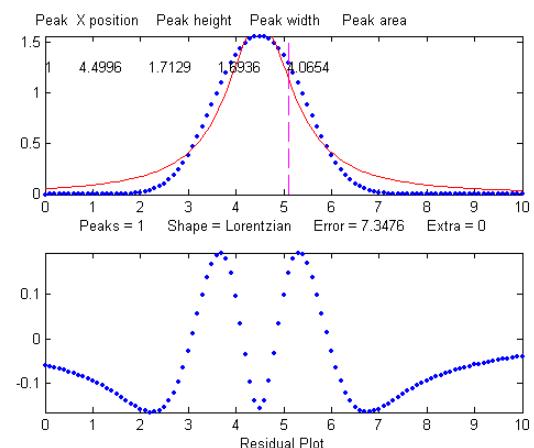
Peak#	Position	Height	Width	Area
1	5	1.0876	1.3139	2.0579

MeanFitError =5.7893

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, it's clear that the peak *height* and *width* are the parameters most susceptible to errors. The peak *positions* are measured accurately even if the model is way wrong, as long as the peak is symmetrical and 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,

[this fit of a real data set](#) to a 5-peak Gaussian model has a low percent fitting error and the residuals look



logistic functions seems to fit [this example data](#) pretty well, with a fitting error of less than 1%; you

random - usually an indicator of a good fit. But in fact *in this case the model is wrong*; that 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 fit 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 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. In such cases you cannot depend solely on what *looks* like a good fit to determine whether the fit is 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.

Sometimes the accuracy of the model is *not* so important. In *quantitative analysis applications*, where the peak height or areas measured by curve fitting is used only to determine the concentration of the substance that created the peak by constructing a *calibration curve* (page 386), using laboratory prepared standards solutions of known concentrations, the necessity of using the exact peak model is lessened, as long as 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 exactly the same* for the unknown samples and the known calibration standards, so the error will cancel out and, 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's 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.

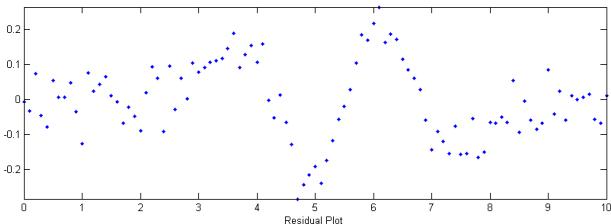
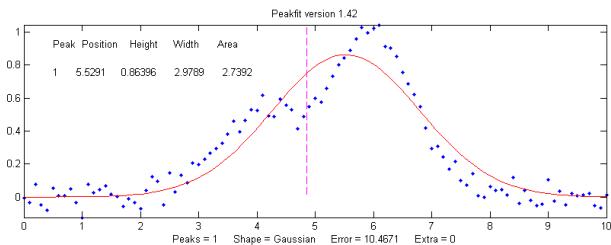
Number of peaks. Another source of model error occurs if you have the wrong *number of peaks* in your model, for example if the signal actually has two peaks but you try to fit it with only one peak. In the example below, a line of Matlab code generates a simulated signal with of two Gaussian peaks at $x=4$ 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's assume that, on the basis of previous experience or some preliminary trial fits, you have established that the optimum peak *shape* model is Gaussian, but you don't 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)
```

```
FitResults =
Peak#    Position    Height      Width      Area
 1        5.5291    0.86396    2.9789    2.7392
MeanFitError = 10.467
```



```
>> [FitResults,MeanFitError]=peakfit([x' y'],5,10,1,1)
```

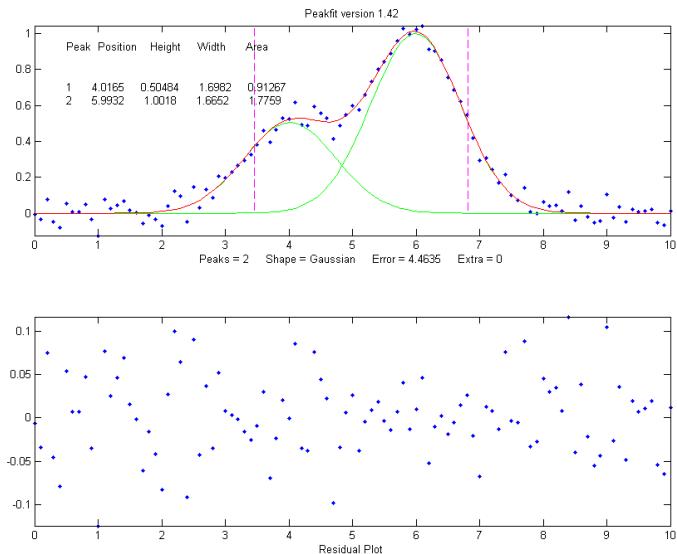
```
FitResults =
    Peak#      Position      Height      Width      Area
      1          4.0165     0.50484    1.6982    0.91267
      2          5.9932     1.00018    1.6652    1.7759
MeanFitError = 4.4635
```

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 much lower (less than half) of the error with only one peak. In fact, the fitting error is just about what we 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 we 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), we can actually 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 actually 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, and adding a third

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 right.

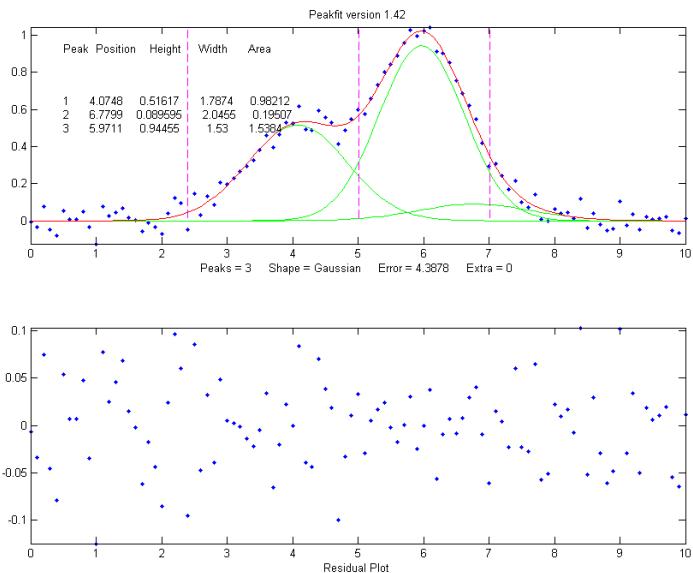
However, a fit with *two* peaks yields much better results (The 4th input argument for the peakfit function specifies the number of peaks to be used in the fit).



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)
FitResults =
    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 actually lower than for the 2-peak fit, but only slightly lower, and *the residuals are no less visually random* than 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 actually present in the signal: it increases the parameter measurement errors of the peaks that are actually present. Again, we can prove this because we know beforehand the true values of the peak parameters: clearly the peak positions, heights, and widths of the two real peaks than are actually in the signal (peaks 1 and 3) are significantly less accurate than those of the 2-peak fit.



Moreover, if we 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 actually 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)
FitResults =
    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 we 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 we know the right answers, we 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 we were to run a 2-peak fit on the same new data, we get much better measurements of the peak heights.

```
>> [FitResults,MeanFitError]=peakfit([x' y'],5,10,2,1)
FitResults =
    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 don't *know* the right answers beforehand, so that's why it's 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/unnecessary terms](#)". The Matlab/Octave function [testnumpeaks.m](#) ($R = \text{testnumpeaks}(x, y, \text{peakshape}, \text{extra}, \text{NumTrials}, \text{MaxPeaks})$) 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 we 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 to each other, on the basis of 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.

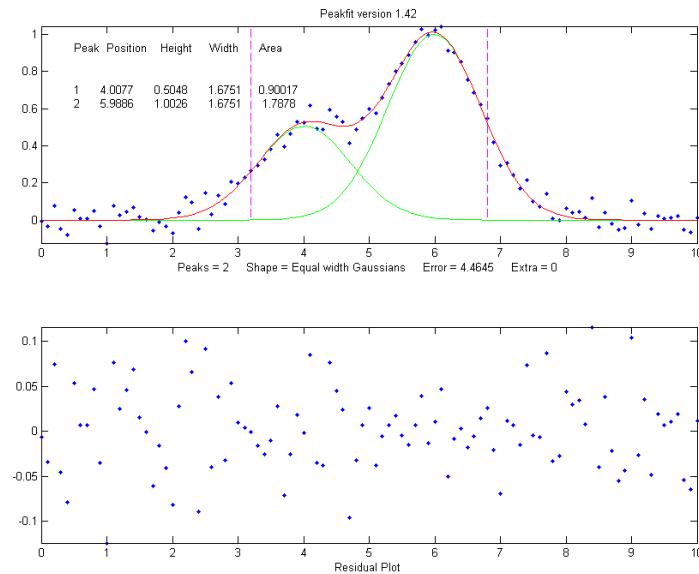
In the current simulation, the true peaks widths are in fact both equal to 1.665, but all the results above show that the *measured* peak widths are close but not quite equal, due to random noise in the signal. The unequal peak widths are a consequence of the random noise, not real differences in peak width. But we can introduce an *equal-width* constraint into the fit by using peak shape 6 (Equal-width Gaussians) or peak shape 7 (Equal-width Lorentzians). Using peak shape 6 on the same set of data as the previous example:

```
>> [FitResults,MeanFitError]=peakfit([x' y'],5,10,2,6)
FitResults =
    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 really 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 we knew that the peak shape was Gaussian (although we could have verified that choice by trying other candidate peaks shapes). We determined that the number of peaks was 2 by inspecting the residuals and fitting errors for 1, 2, and 3 peak models. And then we 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. *Not every experiment can be expected to yield peaks of equal width, but when it does, it's 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 *width 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:



```

>> [FitResults,MeanFitError]=peakfit([x' y'],0,0,2,11,0,0,0,0,[1.6661.666])
FitResults =
    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 peaks 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's better to make use of that constraint. For example, see [Example 35](#) (page 357) 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). Finally, note that if the peak *positions* are also known, and only the peak *heights* are unknown, you don't 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 152) 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 R2 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 with 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 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.

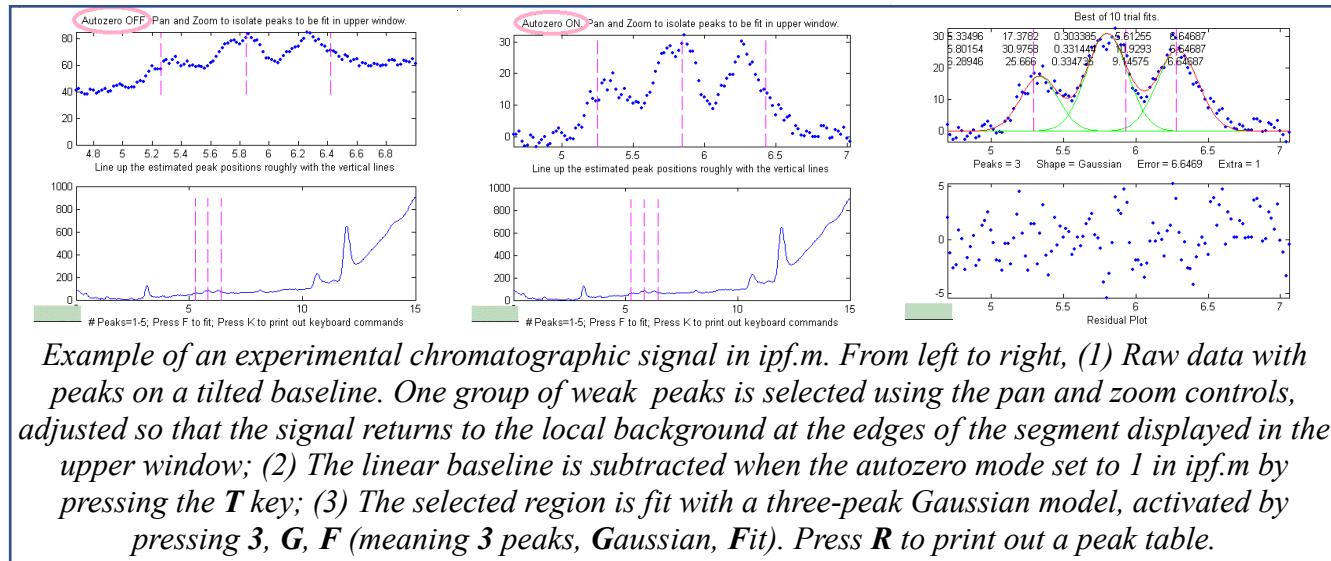
Here's a screen video ([MorePeaksLowerFittingError.mp4](#)) of a real-data experiment using the interactive peak fitter [ipf.m](#) (page 361) with a complex experimental signal in which several different fits were performed using models from 4 to 9 variable-width, equal-width, and fixed-width Gaussian 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, 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 particular experiment is needed: how many peaks are really expected; are the peak widths really expected to be constrained? Note that in this particular case the residuals (bottom panel) are *never really random* and always have a distinct "wavy" character, suggesting that the data *may have been smoothed* before curve fitting (not a good idea: see <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 252.

b. Background correction

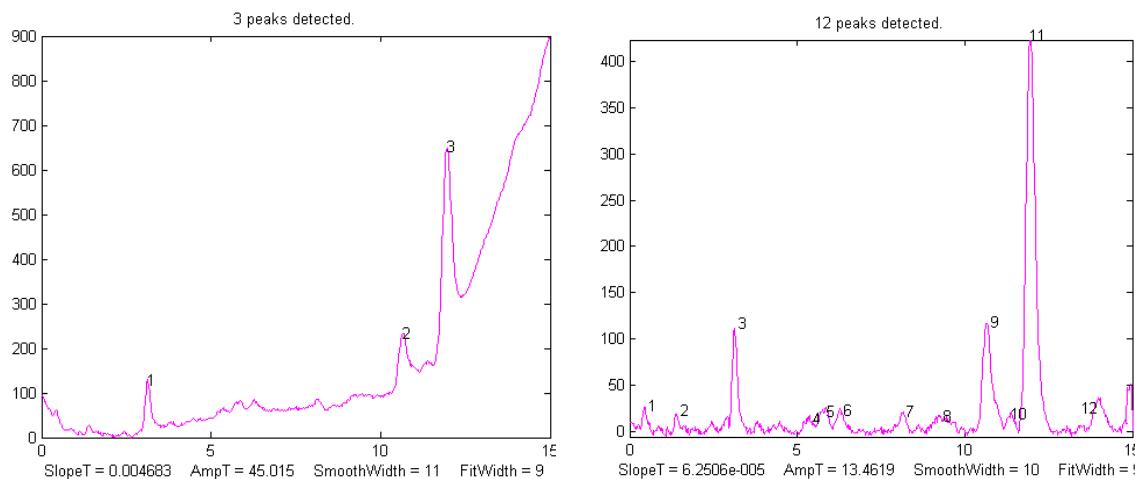
The peaks that are measured in many scientific instruments are sometimes superimposed on a non-specific background or baseline. Ordinarily the experiment protocol is designed to minimize the background or to compensate for the background, for example by subtracting a "blank" signal from the signal of 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 depends 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 peak. 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.

There are various methods 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 region of 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 "autozero" modes in the [ipf.m](#), [iSignal.m](#), and [iPeak.m](#) functions, which are selected by the T key to cycle thorough *OFF*, *linear*, *quadratic*, and *flat* 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.

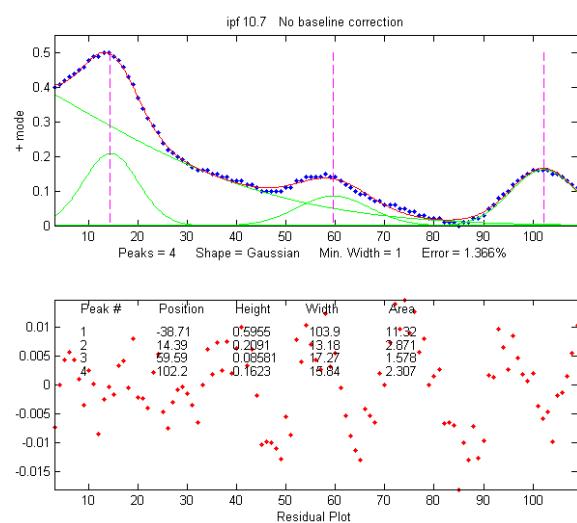


Alternatively, it may be better to subtract the background from the *entire* signal first, before further operations are performed. As before, the simplest assumption is that the background is piece-wise linear, that is, can be approximated as a series of small straight line segments. This is the basis 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 clicks 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 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 return 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 will work in that case.



In many 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, 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 shouldn't use the equal-width shapes for this, because the background peak is broader than the other peaks).

In another real-data example of an [experimental spectrum](#), the linear baseline subtraction ("autozero") 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](#) can employ a peakshape input argument that is a *vector of different shapes*, which can be useful for baseline correction. As an example, consider a weak Gaussian peak on 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)
FitResults =
    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 \(graphic\)](#).

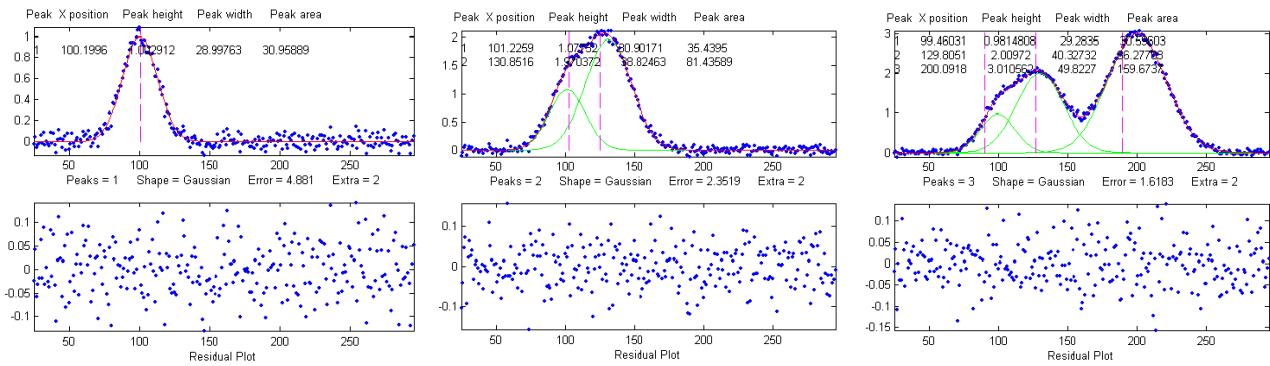
The downside to including the baseline as a variable component is that it increases the number of degrees of freedom, increases the execution time, and increases the possibility of unstable fits. Specifying start values can help.

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 mean 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).

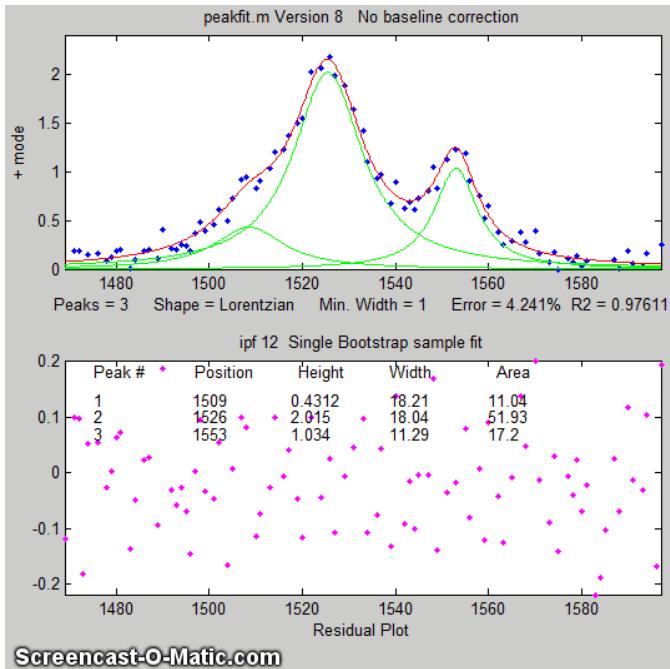
A better way to estimate the parameter errors is to record multiple samples of the signal, fit each of those separately, compute the model parameters from each fit, and calculate the standard error of each parameter. However, if that is not practical, it is possible to simulate such measurements by adding 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's 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:

- (1) a single peak at $x=100$ with a peak height of 1.0 and width of 30 is fit 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.
- (2) The same peak, with the same noise level but with another peak overlapping it, **reduces** the relative fit error to 2.4% (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 134) to estimate the uncertainty of the peak heights, positions, and widths, as illustrated on the left and as described in detail on page 134. The latest version of the [keypress operated interactive version](#) of ipf.m (page 361) has added a function (activated by the 'v' key) that estimates the expected standard deviation of the peak parameters using this method. Click on the figure to open an animation.

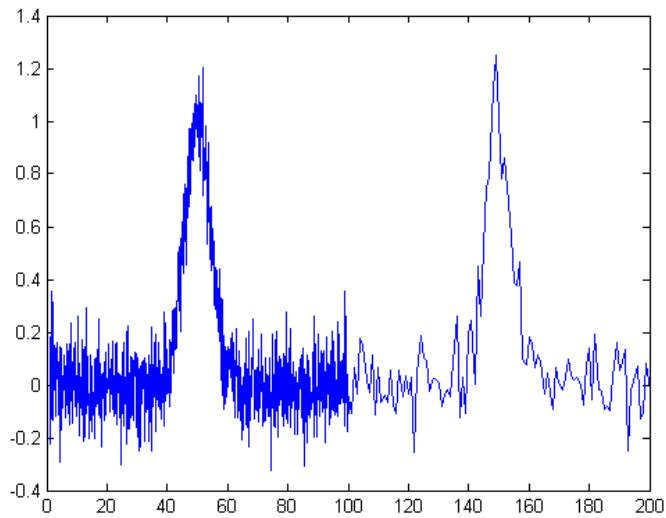
One way to reduce the effect of noise is to take more data. If the experiment makes it possible to reduce the x-axis interval between points, or to take multiple readings at each x-axis values, then

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 right, it's 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, and 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's 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

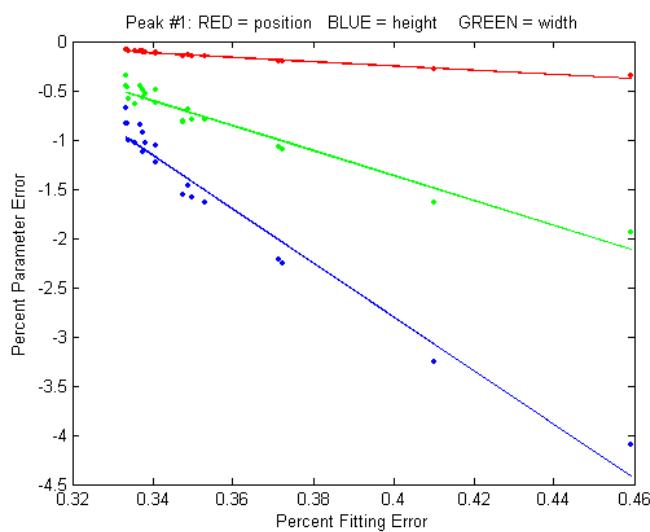
So far, this discussion has applied to white noise. But other noise colors (page 26) have different effects. Low-frequency weighted (“pink”) noise has a *greater* effect on the accuracy of peak parameters measured by curve fitting, and, in a nice symmetry, high-frequency “blue” noise has a *smaller* effect on the accuracy of peak parameters that would be expected on the basis of its standard deviation, because the information in a smooth peak signal is concentrated at low frequencies (page 87). An example of this occurs when you apply curve fitting is to a signal that has been deconvoluted (page 96) to remove a broadening effect. This is why smoothing before curve fitting does not help (page 197), because the peak signal information is concentrated in the *low* frequency range, but smoothing reduces mainly the noise in the *high* frequency range.

Sometime 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 as long as 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 361) 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 to the right, 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

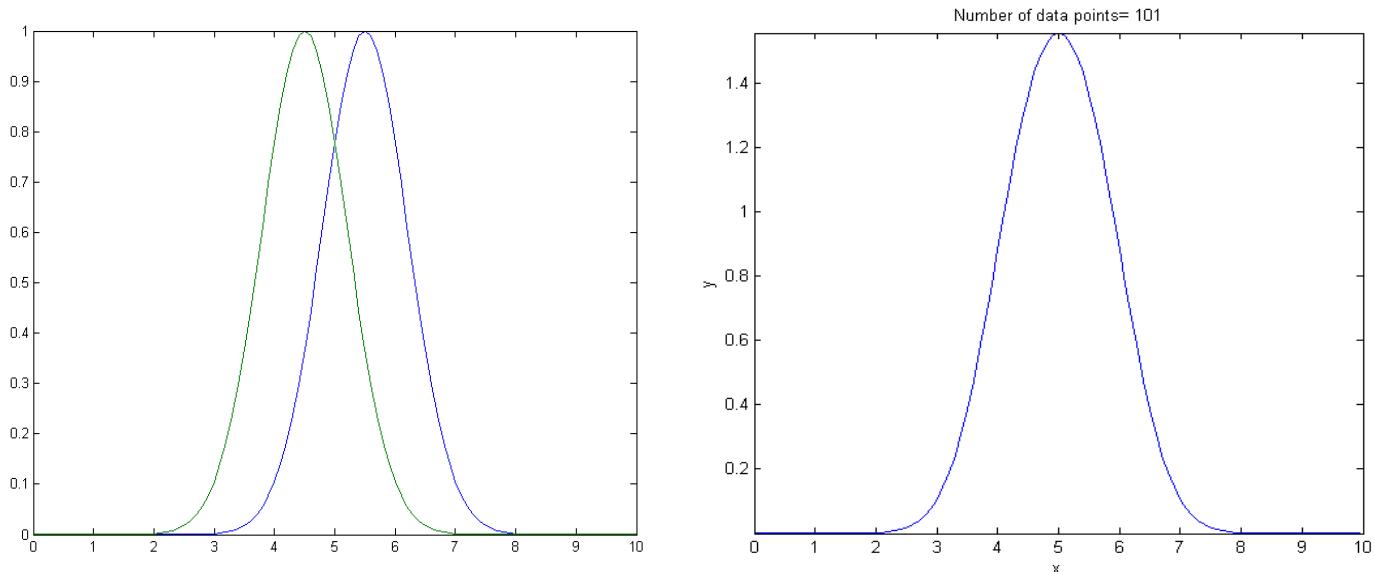


random small variations in the first guesses, rather than by random noise in the signal. In many practical cases there is enough random noise in the signals that the iterative fitting errors within one sample of the signal are small compared to the random noise errors between samples.

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 134).

Selecting the optimum data region of interest. When you perform a peak fitting using [ipf.m](#) (page 361), 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 absolutely 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 uncertainty in the measured parameters.

A difficult case. As a dramatic example of the ideas in the previous sections, consider this simulated

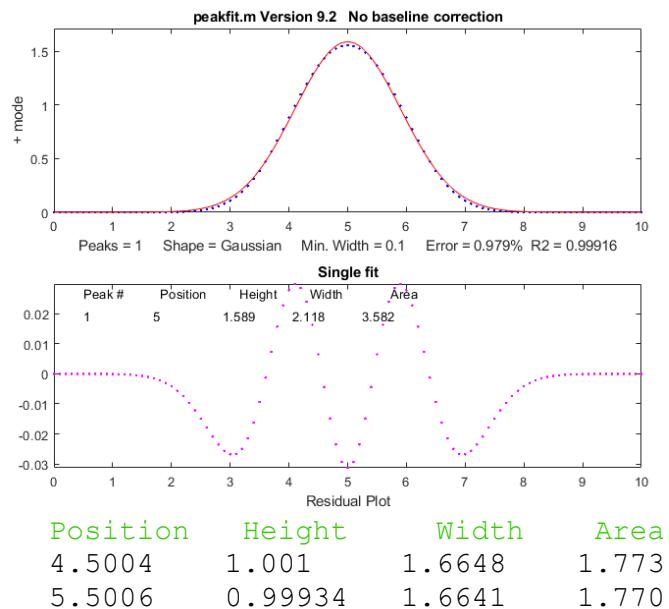


example signal, consisting of two Gaussian peaks of equal height = 1.00, shown on the left, that overlap closely enough so that their sum, shown on the right, is a single symmetrical peak that *looks very much like* a single Gaussian.

```
>> x=[0:.1:10]';
>> y=exp(-(x-5.5).^2)+exp(-(x-4.5).^2);
```

An attempt to fit this with a *single* Gaussian yields a fit with roughly a 1% fitting error and noticeably wavy but smooth residual, suggesting that there is no random noise in the data but that the model is not right.

```
>> peakfit([x y], 5, 19, 1, 1)
```



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)
```

222	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 361). 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 particular sample of the noise is not perfectly symmetrical (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 are 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 make 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 134), 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 133) 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):

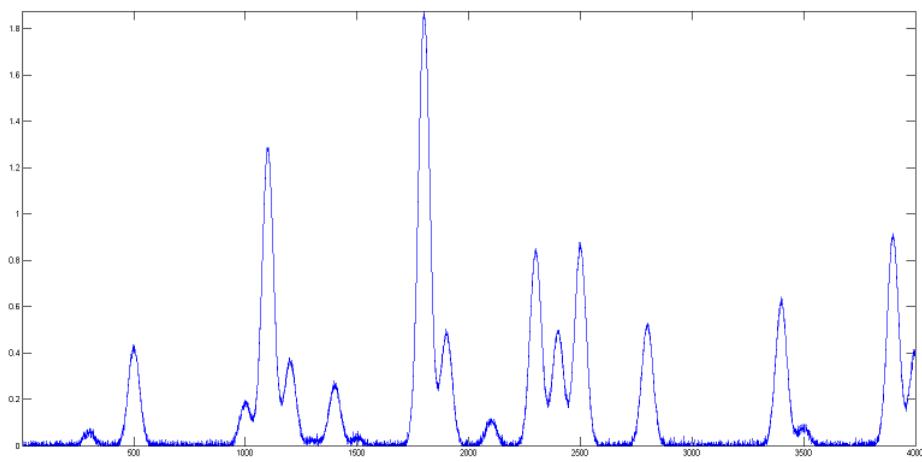
```
peakfit([x, y], 5, 10, 2, 6, 0, 10, [4 2 5 2], 0, 0).
```

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 peak are very different in height, see page 285.

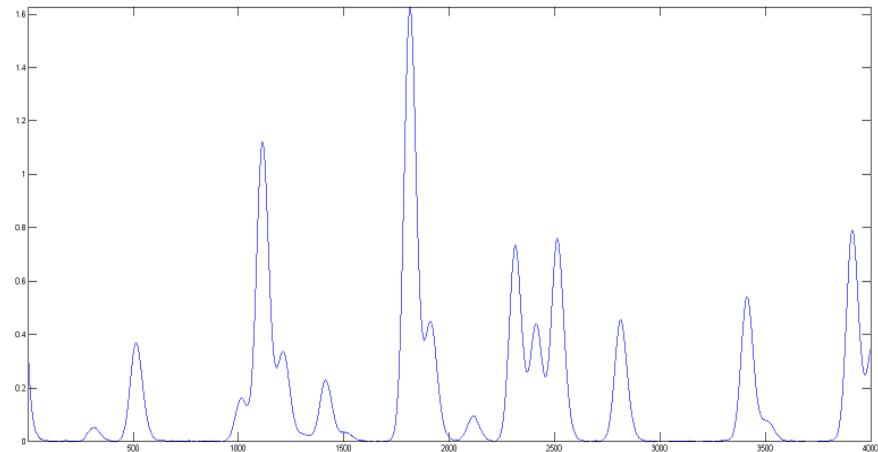
So, to sum up, we can make the following observations about the accuracy of model parameters: (1) the parameter errors depend on the accuracy of the model chosen and on number of peaks; (2) the parameter errors are directly proportional to the noise in the data (and worse for low-frequency or pink noise); (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.

Fitting signals that are subject to exponential broadening.



[DataMatrix2](#) (figure on the 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 path, then type "load [DataMatrix2](#)" at the command prompt to load it into the Matlab workspace.



[DataMatrix3](#) (figure on the left) is a [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 it is usually desirable 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 361), or the "ExpLorentzian", if the underlying peaks are Lorentzian. Right-click and select "Save" to download this signal, put it in the Matlab path, then type

"load [DataMatrix3](#)" to load it into the Matlab workspace.

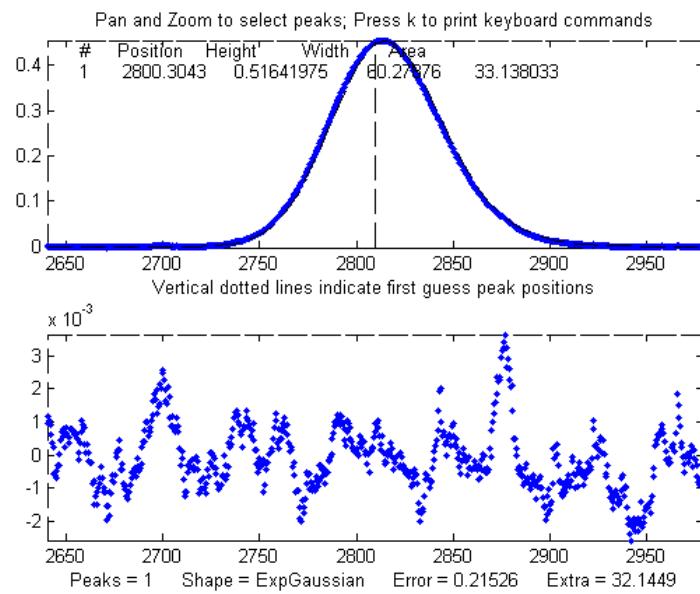
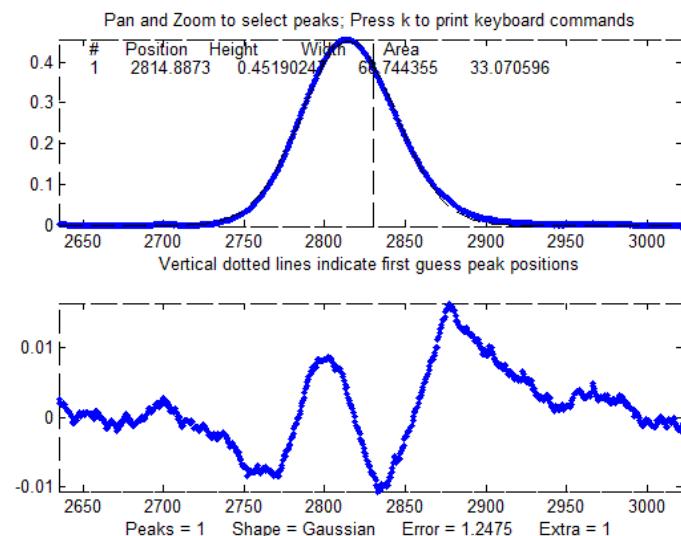
The example illustrated on the right focuses on the single isolated peak whose "true" peak position, height, width, and area in the original unbroadened signal, are 2800, 0.52, 60, and 33.2 respectively. (The relative standard deviation of the noise is $0.01/0.52=2\%$.) In the broadened signal, the peak 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 effected by the broadening).

```
>> load DataMatrix3
>> ipf(DataMatrix3);
Peak Shape = Gaussian
Autozero 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 0.45100549 68.441262 32.859436
```

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 peak position, height, and width to an accuracy of a fraction of 1%*. 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). Of course, 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 have to 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. (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 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. 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 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:

```

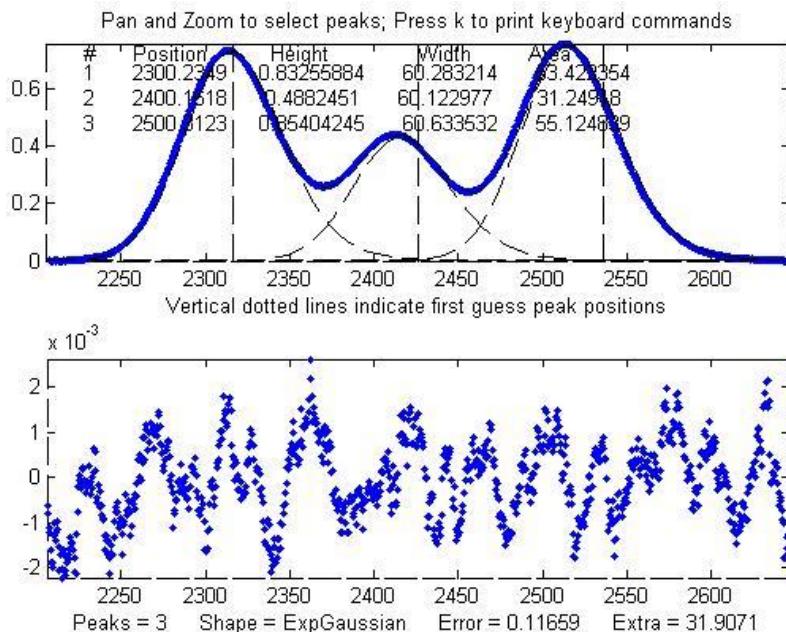
Peak Shape = Exponentially-broadened Gaussian
Autozero ON
Number of peaks = 1
Extra = 32.6327
Fitted range = 2640 - 2979.5 (339.5) (2809.75)
Percent Error = 0.21696

```

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



heights in this particular group of peaks - the noise is the same everywhere in this signal.

Peak Shape = Exponentially-broadened Gaussian

Autozero OFF

Number of peaks = 3

Extra = 31.9071

Fitted range = 2206 - 2646.5 (440.5) (2426.25)

Percent Error = 0.11659

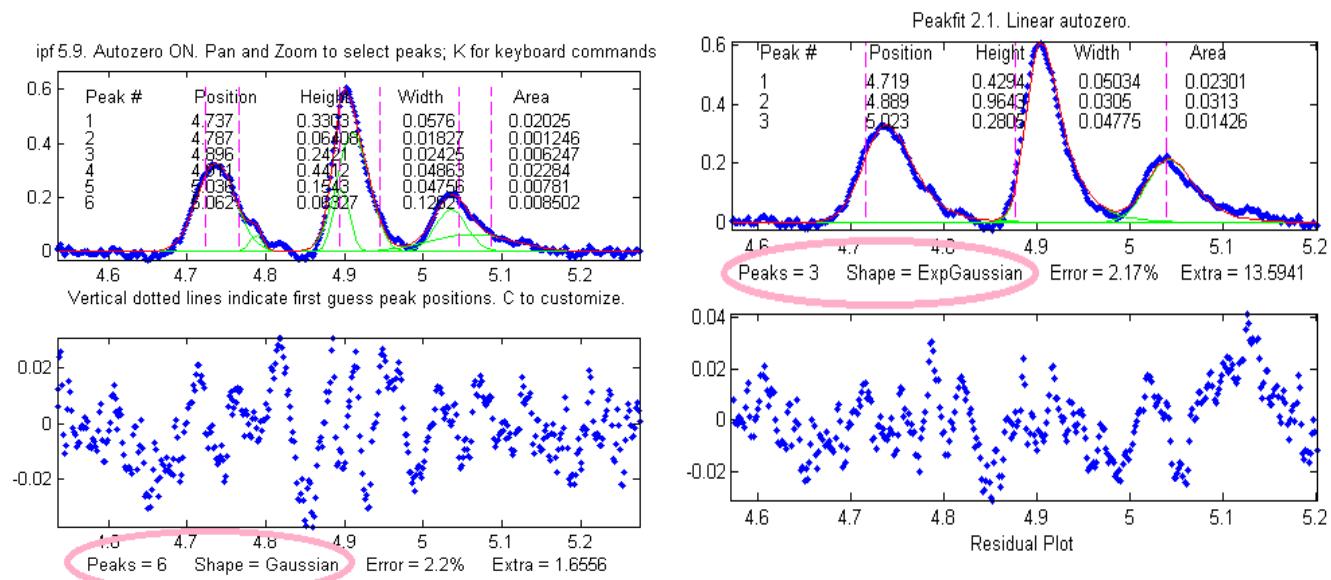
Peak#	Position	Height	Width	Area
1	2300.2349	0.83255884	60.283214	53.422354
2	2400.1618	0.4882451	60.122977	31.24918
3	2500.3123	0.85404245	60.633532	55.124839

The residual plots in both of these 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.

Other peaks in the same signal, under the broadening influence of the same decay constant, can be fit 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

One final caveat: peak asymmetry similar to 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 fit 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.

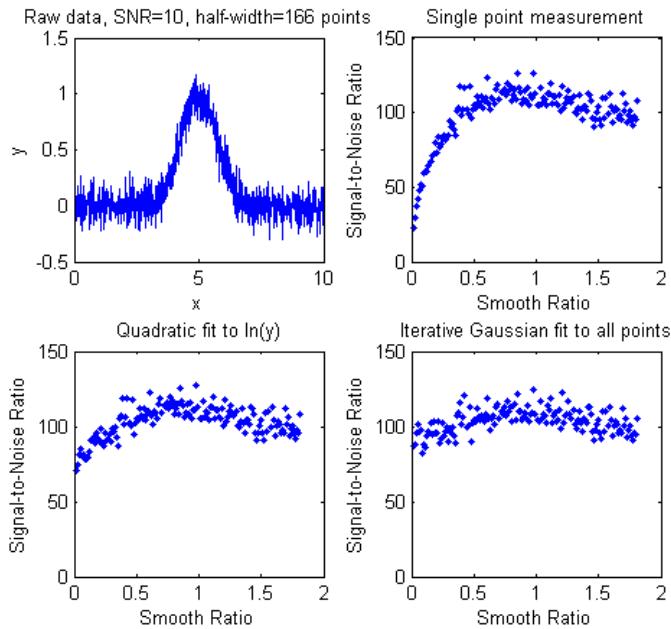


based on fitting error alone. However, you can decide that by inspecting the other peaks in the signal: in most experiments, exponential broadening applies to every peak in the signal, and the broadening is either constant or changes gradually over the length of the signal. On the other hand, it is relatively unlikely that a smaller side peak that varies in this way will accompany every peak in the signal. So, if only one or a few of the peaks exhibit asymmetry, and the others are symmetrical, it's 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's more likely to be a broadening factor that applies to the entire signal. The two figures here provide an example from real experimental data. On the left, three asymmetrical peaks are each fit with two symmetrical Gaussians (six peaks total). On the right, those three peaks are fit with one exponentially broadened Gaussian each (three peaks total). In this case, the three asymmetrical peaks all have the same asymmetry and can be fit with the same decay constant ("extra"). Moreover, the fitting error is slightly lower for the three-peak exponentially broadened fit. *Both of these 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 361), press Shift-X and type a negative values.

An alternative to this type of curve fitting for exponential broadened peaks is to use the [first-derivative addition technique](#) 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 be 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 between-signal variations in peak parameters (page 134). [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:

- simply taking the single point at the center of the peak as the peak height;
- using the [gaussfit](#) method to fit the top half of the peak (see page 137), and
- fitting the entire signal with a Gaussian using the iterative method.

The results of 150 trials with independent white noise samples are shown on the left: 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 effected by smoothing and the iterative method hardly at all. So the bottom line is that you should not smooth 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 to smooth before fitting are 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 if the signal is contaminated with high-amplitude narrow spike artifacts, in which case a [median-based pre-filter](#) can remove the spikes without much change to the rest of the signal. And, in another application 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 common 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 57).

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 first [smooths](#) the first derivative of the signal, before looking 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 carefully adjusting the smooth width, slope threshold, and amplitude threshold, it's possible to detect only the desired peaks and ignore peaks that are too small, too wide, or too narrow.

Moreover, this technique can be extended to estimate

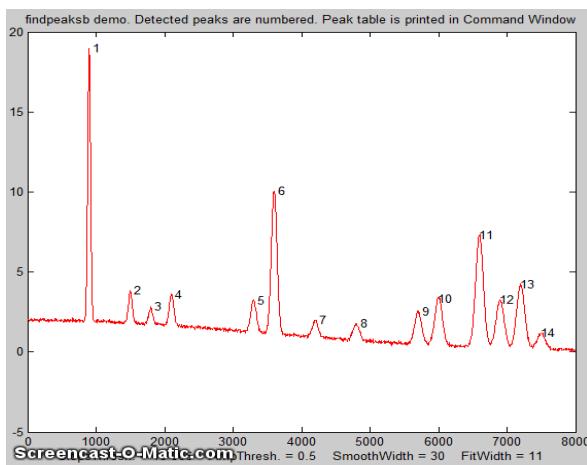
the position, height, and width of each peak by [least-squares curve-fitting](#) of a segment of the *original unsmoothed signal* in the vicinity of the zero-crossing. Thus, even if heavy smoothing of the first derivative is necessary to provide reliable discrimination against noise peaks, the peak parameters 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 is capable of measuring peak positions and heights quite accurately, but the measurements of peak widths and areas is 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:

[findpeaksx](#), [findpeaksxw](#), [findpeaksG](#), [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. Don't 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`), (page 216), for adjusting the peak detection criteria interactively to optimize for any particular peak type (Matlab only). *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 308.

[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

```
P=findpeaksx(x, y, SlopeThreshold, AmpThreshold, SmoothWidth, PeakGroup,  
smoothtype)
```

```
P=findpeaksxw(x, y, SlopeThreshold, AmpThreshold, SmoothWidth, PeakGroup,  
smoothtype)
```

These are simple and fast Matlab/Octave command-line functions to *locate and count* the positive peaks in noisy data sets. They detect 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 on the basis of peak width. Larger values of this parameter will neglect broad features of the signal. A reasonable initial value for Gaussian peaks is $0.7 * \text{WidthPoints}^{-2}$, where WidthPoints is the *number of data points* in the half-width ([FWHM](#)) of the peak.

AmpThreshold - Discriminates on the basis of 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 is 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 35)

If smoothtype=1, rectangular (sliding-average or boxcar)

If smoothtype=2, triangular (2 passes of sliding-average)

If smoothtype=3, pseudo-Gaussian (3 passes of sliding-average)

Basically, higher values yield greater reduction in high-frequency noise, at the expense of slower execution. For a comparison of these smoothing types, see page 51.

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's better to use [findpeaksG.m](#).

Speed demonstration; detecting 12,000 peaks in less than 1 second (in Matlab, on a typical desktop PC):

```
>> 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 0.339553 seconds.
NumPeaks = 12028
```

Gaussian peak measurement

```
P=findpeaksG(x, y, SlopeThreshold, AmpThreshold, SmoothWidth, FitWidth,
smoothtype)

P=findpeaksL(x, y, SlopeThreshold, AmpThreshold, SmoothWidth, FitWidth,
smoothtype)
```

These Matlab/Octave functions locate the positive peaks in a noisy data set, performs a [least-squares curve-fit](#) of a Gaussian or Lorentzian function to the top part of the peak, and computes 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 that 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.

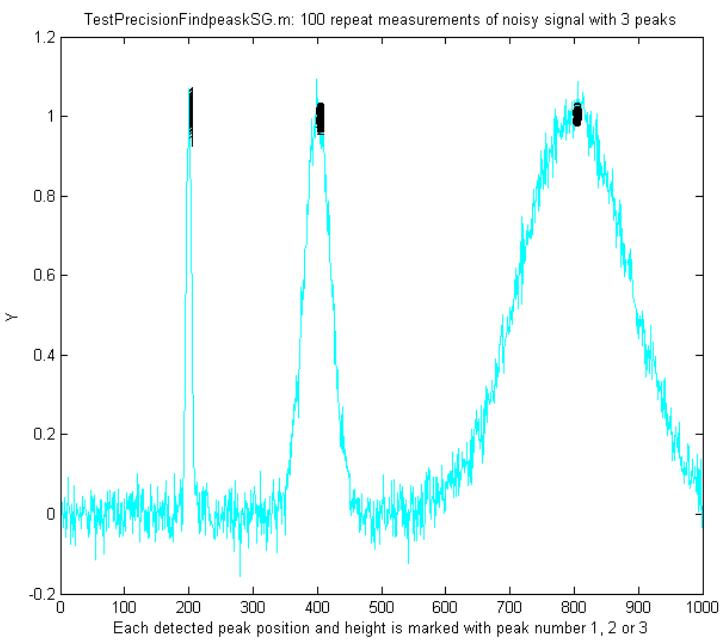
[findpeaksSG.m](#) is a *segmented* variant of the findpeaksG function, with the same syntax, except that the four peak detection parameters can be *vectors*, dividing up the signal into regions that are optimized for peaks of different widths. Any number of segments can be declared, based on the length 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). 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)

```

[FindpeaksSL.m](#) is the same thing for Lorentzian peaks. In the graphic example shown 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, findpeakssG, 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).



One difficulty 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 findpeakssG.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 of the peaks 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, a visual estimate of the following signal indicates about 20 peaks, so you use 20 as the 3rd input argument:

```

x=[0:.1:50];
y=10+10.*sin(x).^2+randn(size(x));
[P,A]=autofindpeaks(x,y,20);

```

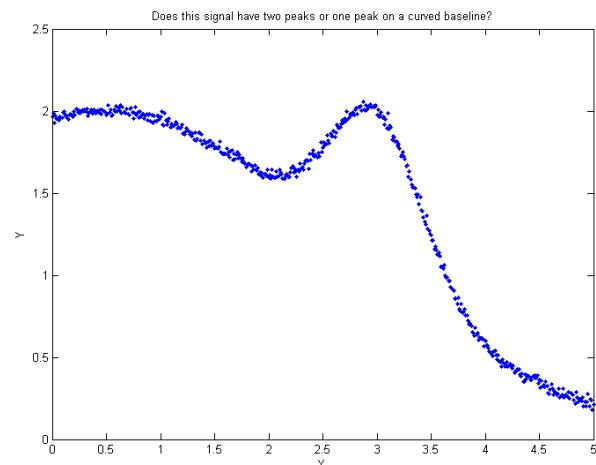
Then you can use A as the peak detection parameters for other peak detection functions, such as `P=findpeaksG(x,y,A(1),A(2),A(3),A(4),1)` or `P=measurepeaks(x,y,A(1),A(2),A(3),A(4),1)`. You will probably want to fine-tune the *amplitude* threshold A(2) manually for your own needs, but that's the one that's easiest to know.

Type "help autofindpeaks" and run the examples there. [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 (Click for [animated graphic](#)).

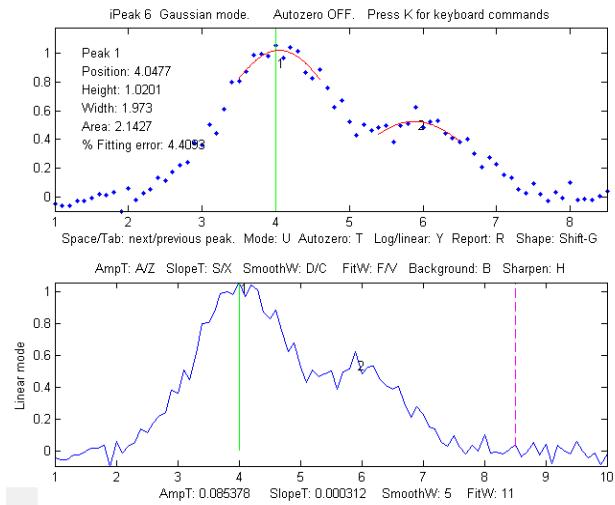
Optimization of peak finding

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 216), 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.



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

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 in a vector 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, let's create a noisy series of peaks (plotted on the right) and apply both of these `findpeaks` functions to the resulting data.

```
x=[0:.1:100];
y=5+5.*sin(x)+randn(size(x));
plot(x,y)
```

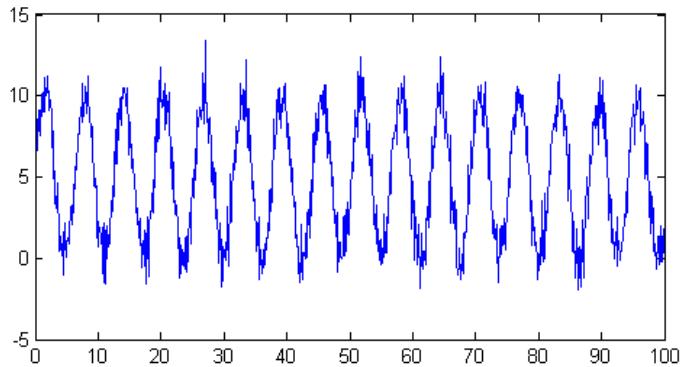
Now, most people just looking at this plot of data would count *16 peaks*, with peak heights averaging about 10 units. Every time the statements above are run, the random noise is different, but you would still count the 16 peaks, because the signal-to-noise ratio is 10, which is not that bad. But the `findpeaks` function in the *Signal Processing Toolbox*,

```
[PKS,LOCS]=findpeaks(y,'MINPEAKHEIGHT',5,'MINPEAKDISTANCE',11)
```

In the example shown on the left (using the interactive peak detector *iPeak* program described on page 216), 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. We 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



counts anywhere from 11 to 20 peaks, with an average height (PKS) of 11.5.

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; my function is better for peaks that have more data points.

The demonstration script [FindpeaksSpeedTest.m](#) compares the speed of the Signal Processing Toolkit `findpeaks`, `findpeaksx`, and `findpeaksG` on the same large test signal with many peaks:

Function	Number of peaks	Elapsed time	Peaks per second
<code>findpeaks</code> (SPT)	160	0.16248	992
<code>findpeaksx</code>	158	0.00608	25958
<code>findpeaksG</code>	157	0.091343	1719

Finding valleys

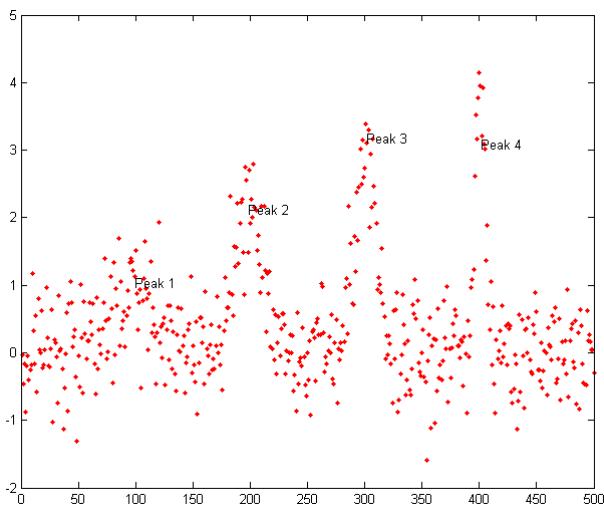
There is also a similar function for finding *valleys* (minima), called [findvalleys.m](#), which works the same way as `findpeaksG.m`, except that it locates *minima* instead of *maxima*. Only valleys above the AmpThreshold (that is, more positive or less negative) are detected; if you wish to detect 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 the measurements of peaks

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 peak of *Lorentzian* shape, use [findpeaksL.m](#) instead (the only difference is that the reported peak heights, widths, and areas will be more accurate if the peak are actually 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 361).

For a direct comparison of the accuracy of `findpeaksG` vs `peakfit`, run the demonstration script [peakfitVSSfindpeaks.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'll 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 an 5th peak that is not really there. On the other hand, `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.

[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; and
- (3) it has provision for background subtraction (when the input argument "autozero" 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, AUTOZERO)`. 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 `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 autozero modes 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;
- "AUTOZERO" 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 (a) plain *findpeaksG*, to (b) *findpeaksb* without background subtraction (AUTZERO=0), and to (c) *findpeaksb* with background subtraction (AUTZERO=1).

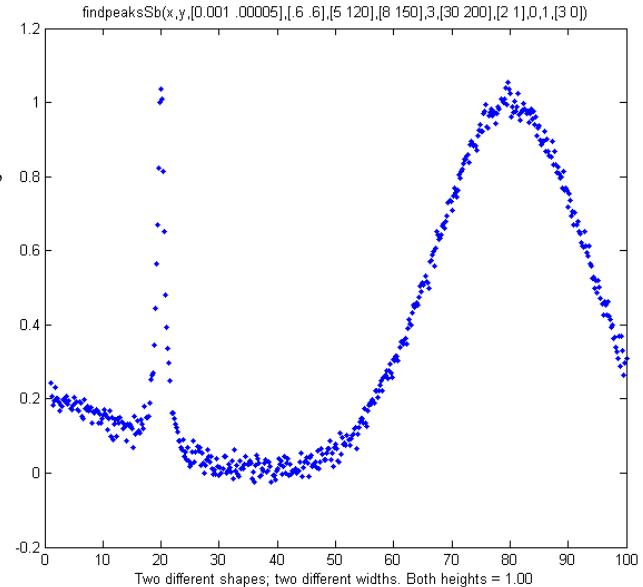
```
x=1:.2:100; Heights=[1 2 3]; Positions=[20 50 80]; Widths=[3 3 3];
y=2-(x./50)+modelpeaks(x,3,1,Heights,Positions,Widths)+.02*randn(size(x));
plot(x,y);
disp('          Peak      Position      Height      Width      Area
% error')
PlainFindpeaks=findpeaksG(x,y,.00005,.5,30,20,3)
NoBackgroundSubtraction=findpeaksb(x,y,.00005,.5,30,20,3,150,1,0,0)
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.

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, autozero, 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 the both peaks is measured accurately. First, we 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; autozero=[3 0];
```



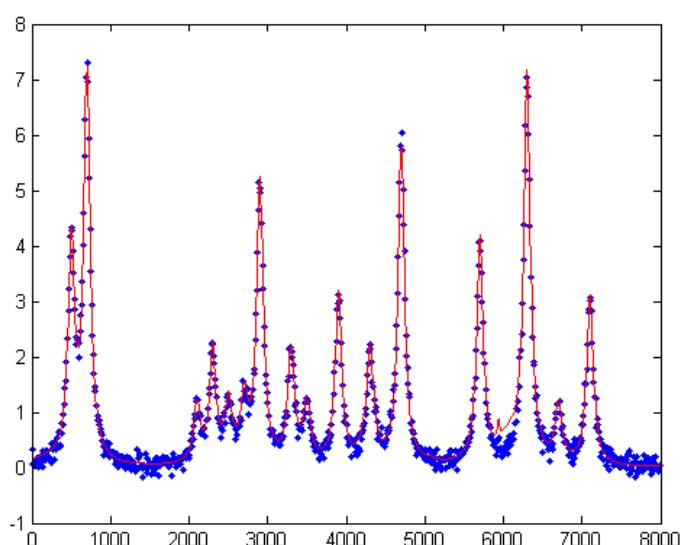
```
>> findpeaksSb(x,y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup,
smoothtype, window, PeakShape, extra, NumTrials, autozero)

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`, so as to deal better with *overlap of the adjacent overlapping peaks*. The syntax is

```
FPB=findpeaksb3 (x,y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup,
smoothtype, PeakShape, extra, NumTrials, AUTOZERO, ShowPlots).
```



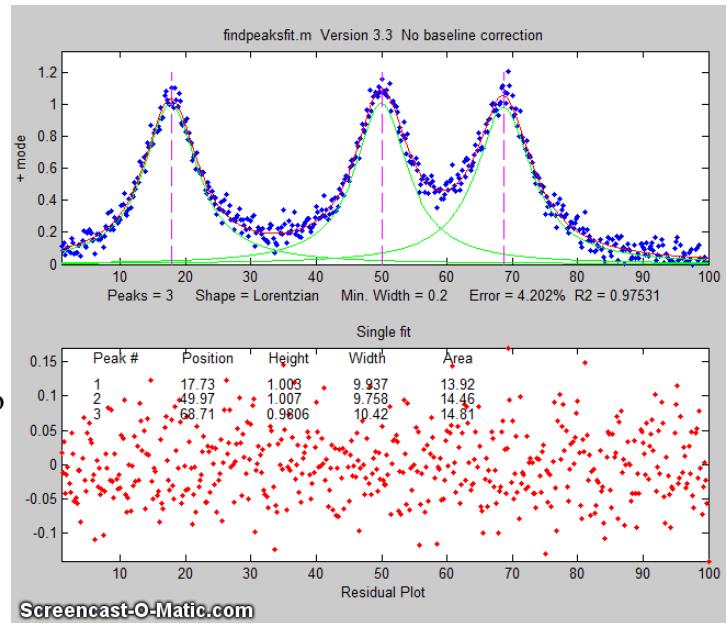
The demonstration script [DemoFindPeaksb3.m](#) shows how `findpeaksb3` works with irregular clusters of overlapping Lorentzian peaks, as in the example on the left (type "help `findpeaksb3`") for more. The demonstration script [FindpeaksComparison](#) shows how `findpeaksb3` compares to the other peak 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 determined by `findpeaksG` as 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`. The syntax is

```
function [P, FitResults, LowestError, BestStart, xi, yi] =
findpeaksfit(x, y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup,
smoothtype, peakshape, extra, NumTrials, autozero, fixedparameters, plots)
```

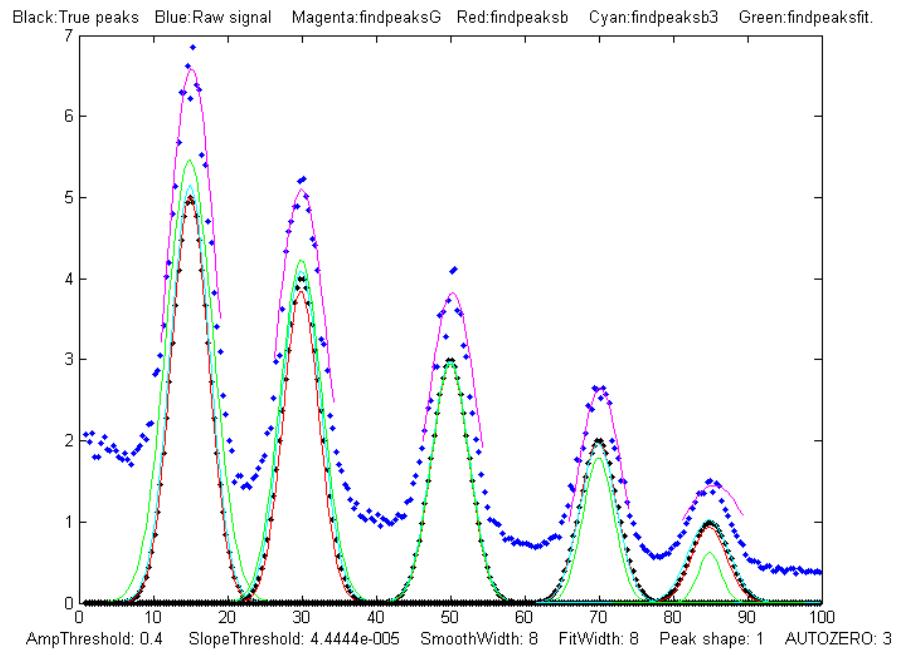
The first seven input arguments are exactly the same as for the [findpeaksG.m](#) function; if you have

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 361) to fit the peaks in your signals, then you can use those same input argument values for `findpeaksfit.m`. The demonstration script [findpeaksfitdemo.m](#), 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. Requires the [findpeaksfit.m](#) and [lorentzian.m](#) functions installed. This script was used to generate [the GIF animation](#) shown on the right. 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 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 similar to 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.

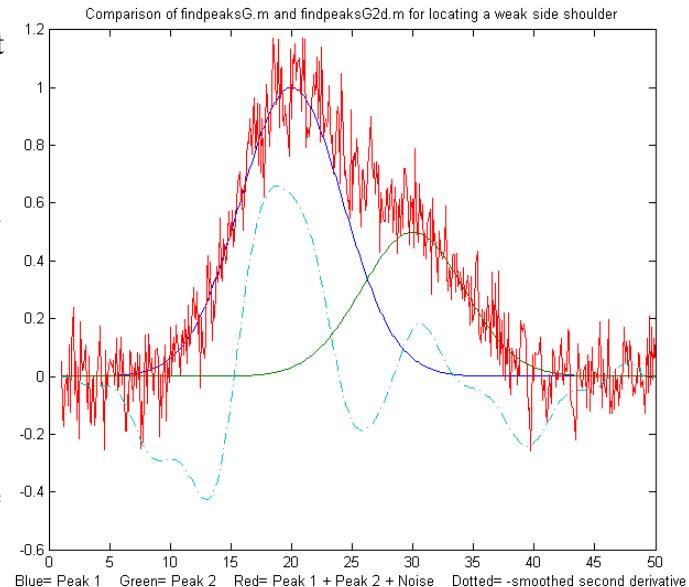
Average absolute percent errors of all peaks				
	Position error	Height error	Width error	Elapsed time, sec
findpeaksG	0.35955	38.5733	25.7977	0.17447
findpeaksb	0.388283	8.50246	14.3295	0.78245
findpeaksb3	0.27187	3.7445	3.0474	1.2146
findpeaksfit	0.519302	8.04176	24.035	1.017

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.

Other related functions

[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](#).

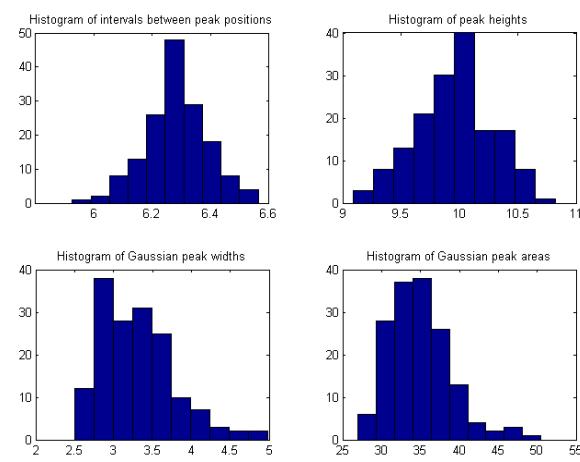
[\[M,A\]=autopeaks.m](#) is a peak detector for peaks of arbitrary shape; it's basically a combination of [autofindpeaks.m](#) and [measurepeaks.m](#). It has similar syntax to [measurepeaks.m](#), except that the peak detection parameters (SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, and smoothtype) can be omitted and the function will calculate trial values in the manner of [autofindpeaks.m](#). Using the 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. Like [measurepeaks](#), 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 (page 111), 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 path.

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 a 8th input argument to control the display and plotting. Version 2, March 2016, adds median and mode. 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:

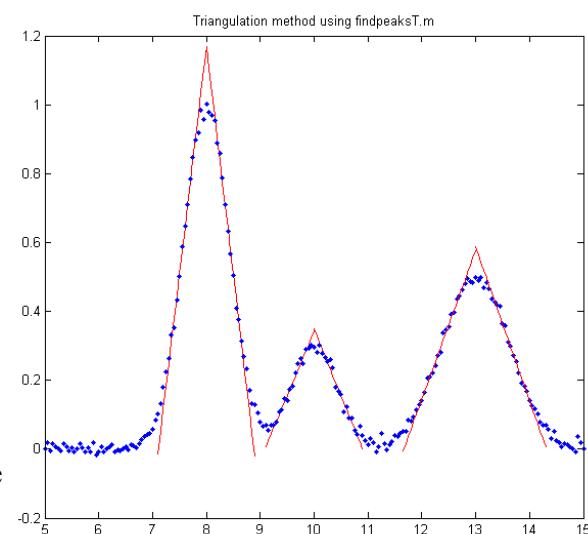
```
x=[0:.1:1000];y=5+5.*cos(x)+.5.*randn(size(x));
figure(1);P=findpeaksplot(x,y,0,8,11,19,3);tablestats(P,1);
```

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...
```

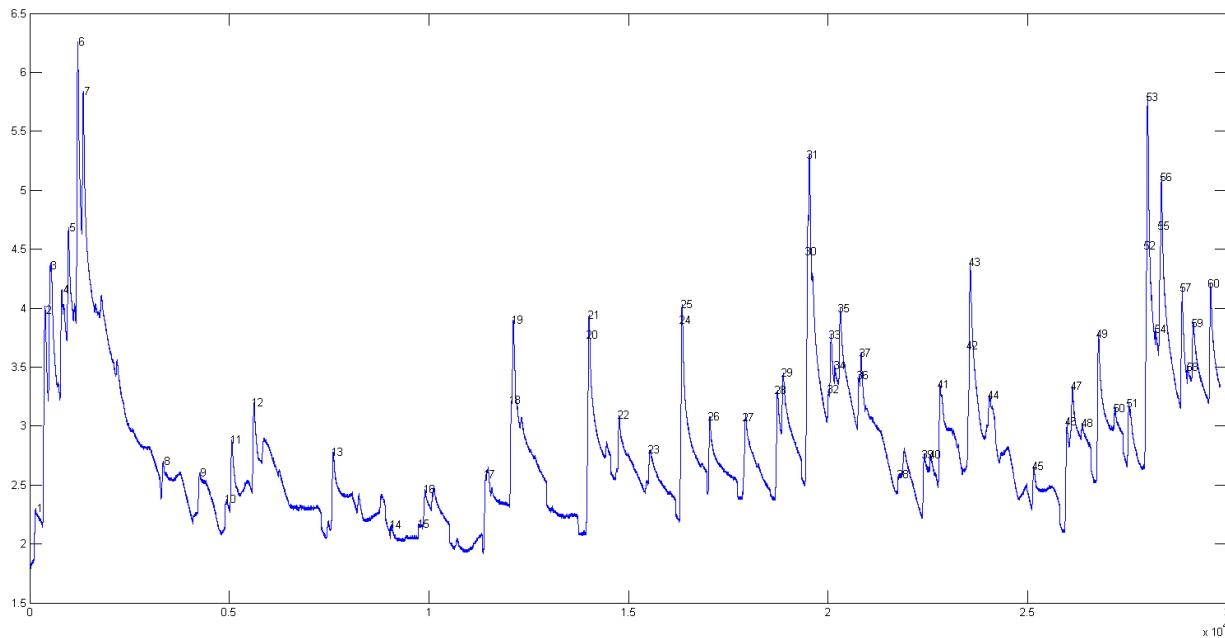
Peak start and end. Defining the "start" and "end" of the peak (the x-values where the peak begins and ends) is a bit arbitrary because typical peak shapes approach the baseline asymptotically far from the peak maximum. You might define the peak start and end points as the x values where the y value is some small fraction, say 1%, of the peak height, but then the random noise on the baseline is likely to be a large 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 138), 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 $x = p \pm \sqrt{w^2 \log(1/a)/(2 \sqrt{\log(2)})}$ where p is the peak position and w is the peak width (full width at half maximum). So, for example if $a = .01$, $x = p \pm w * \sqrt{(\log(2)+\log(5))/(2 \log(2))} = 1.288784 * w$. Lorentzian peaks [can be shown](#) to reach a fraction a of the peak height at $x = p \pm \sqrt{[(w^2 - a w^2)/a]/2}$. If $a = .01$, $x = p \pm (3/2 \sqrt{11}) * w = 4.97493 * w$. 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**. (For greater accuracy with overlapping peaks, use [**peakfit.m**](#) or the Interactive Peak Fitter ([**ipf.m**](#), page 361) and calculate the start and end from the peak positions and width using the formulas above). The advantage of curve fitting is that it can measure peak areas, even of overlapping peaks, *without defining the peak start and stop times*.

[**findpeaksT.m**](#) and [**findpeaksTplot.m**](#) are variants of findpeaksG that measure the peak parameters by *constructing a triangle around each peak* with its sides tangent to the sides of the peak, as shown on the right ([script that generated this graphic](#)). This method mimics the geometric construction method that was formerly used to measure peak parameters manually before the age of computers. Peak height is taken as the apex of the triangle, which is slightly higher than the peak of the underlying curve. The performance of this method is



poor when the signals are very noisy or if the peaks overlap, but in a few 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* (see the demo function [triangulationdemo.m](#) for some examples: [click for graphic](#)).

[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. See [findsteps.png](#) for a real example. And [findstepsplot.m](#) plots 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 height 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\(m,threshold\)](#): "Numbers Above Threshold": 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 that it detects 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 the examples above) and then use Matlab/Octave notation and built-in functions to extract specific information from that matrix. *The powerful combination of functions and matrix/vector "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). Also works with `median`.

`max(P(:,3))` returns the maximum peak height. 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; `sortrows(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. The downloadable function [val2ind.m](#) (syntax `[index,closestval] = val2ind(v,val)`) 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 path). It's 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

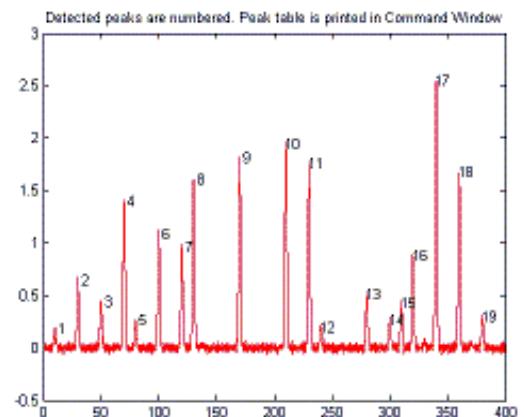
position (column 2) is closest to 7.5. `P(val2ind(P(:,3), max(P(:,3))), :)` returns the row vector of peak parameters of the highest peak in peak table P.

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 widths (column 4) are less than 5.8.

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:

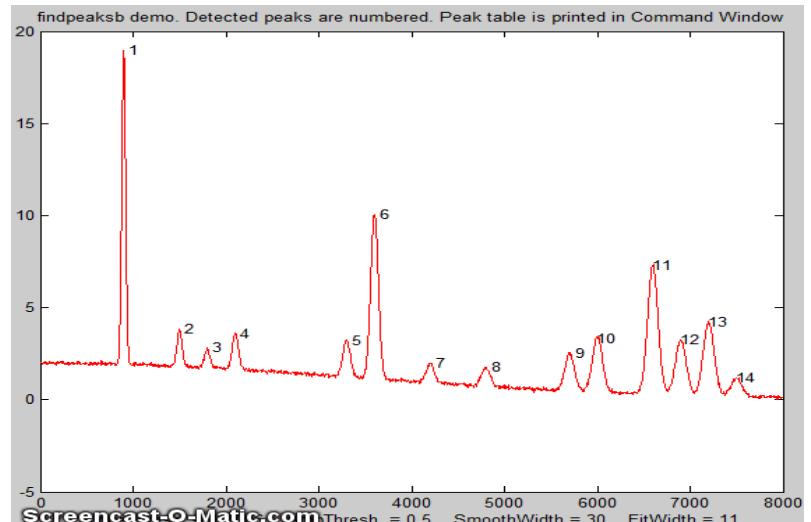
```
Peak # Position Height Width Area
Measuredpeaks =
1      799.95   6.9708  51.222  380.12
2      1199.4    3.9168  50.44   210.32
3      1600.6    2.9955  49.683  158.44
4      1800.4    2.0039  50.779  108.33
.....etc.
```



[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 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 very useful in 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's 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 152).

Example: Download [idpeaks.zip](#), extract it, and place the extracted files in the Matlab or Octave 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]  
.....etc...
```

iPeak, for Matlab

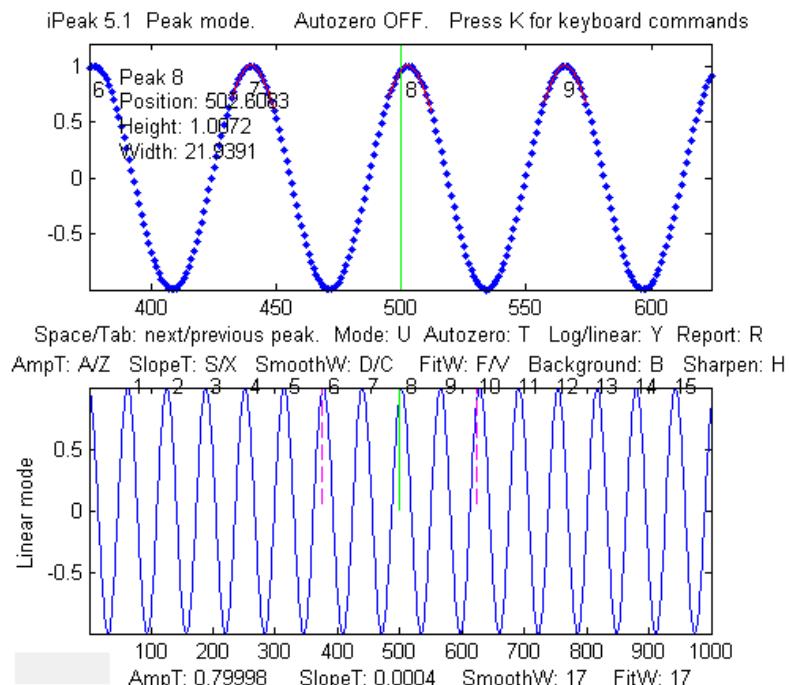
iPeak is a keyboard-operated interactive peak finder for time series data, based on the "findpeaksG.m" and "findpeaksL.m" functions, for Matlab only. The interactive keypress operation works even if you run [Matlab in a web browser](#), but not on [Matlab Mobile](#) or in Octave. Its basic operation 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 the A/Z, S/X, D/C, and F/V keys.

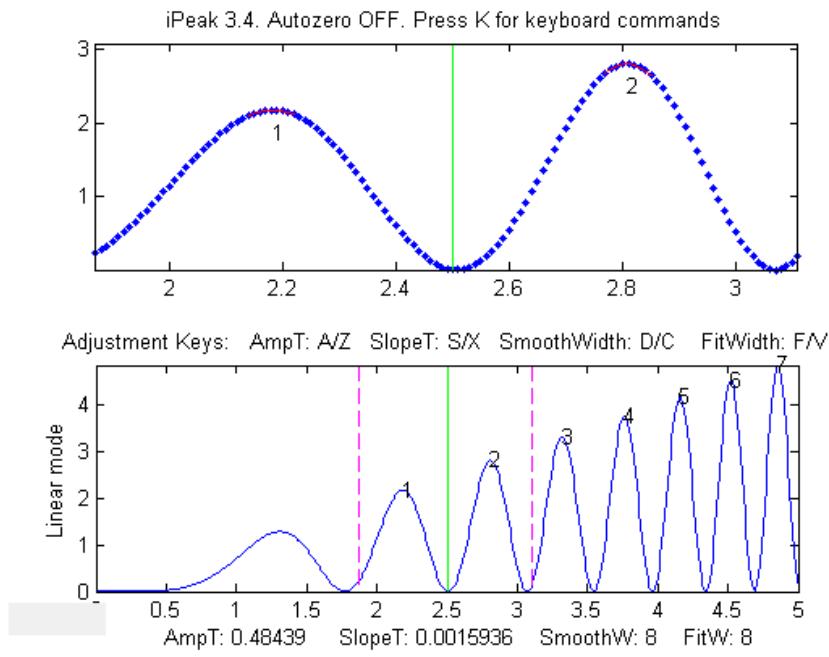
Example 1: One input argument; data in single vector:

```
>> y=cos(.1:.1:100);
>> ipeak(y)
```



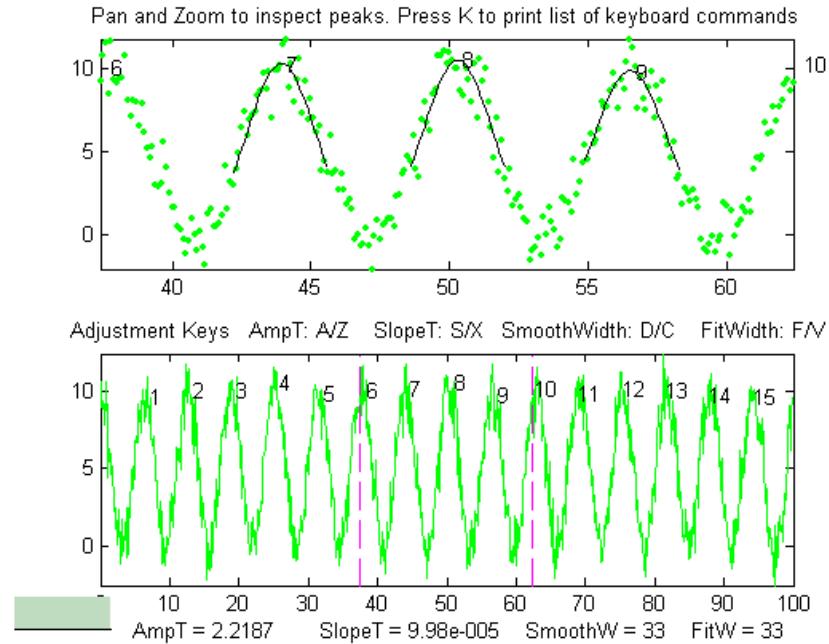
Example 2: One input argument; data in two columns of a matrix:

```
>> x=[0:.01:5]';
>> y=x.*sin(x.^2).^2;M=[x y];
>> ipeak(M)
```



Example 3: Two input arguments; data in separate x and y vectors:

```
>> x=[0:.1:100];
>> y=(x.*sin(x)).^2;
>> ipick(x,y);
```

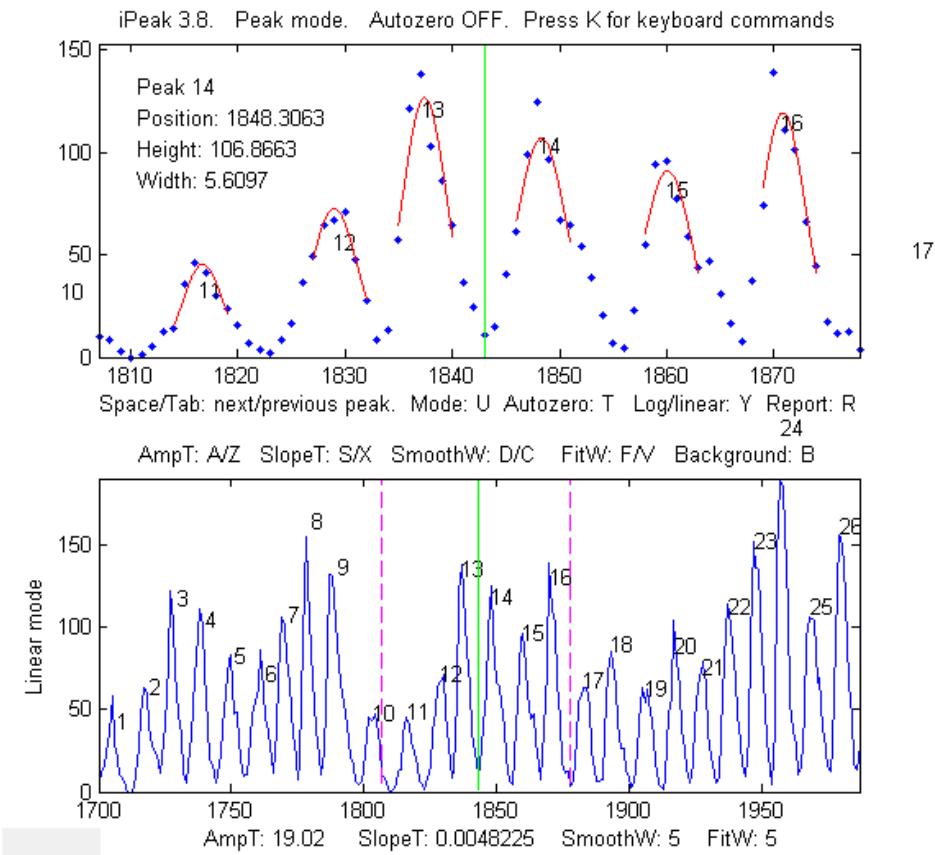


Example 4: When you start *iPeak* using the simple syntax above, the initial values of the peak detection parameters are calculated as described above, but if it starts off by picking up far *too many* or *too few* peaks, you can add an additional input argument (after the data) to control peak sensitivity.

```
>> 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)
```

The 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 effects only the *starting* values for the peak detection parameters. (It's just a quick way to set reasonable initial values of the peak detection parameters, so you won't have so much adjusting to do).

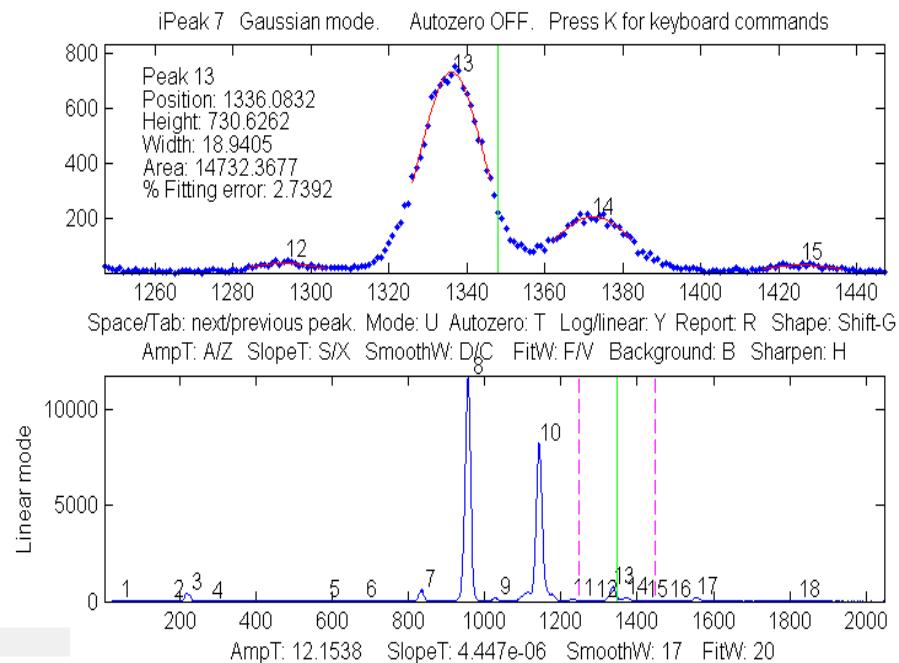
```
>> load sunspots
>> ipeak(year,number,20)
```



*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 displays 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.

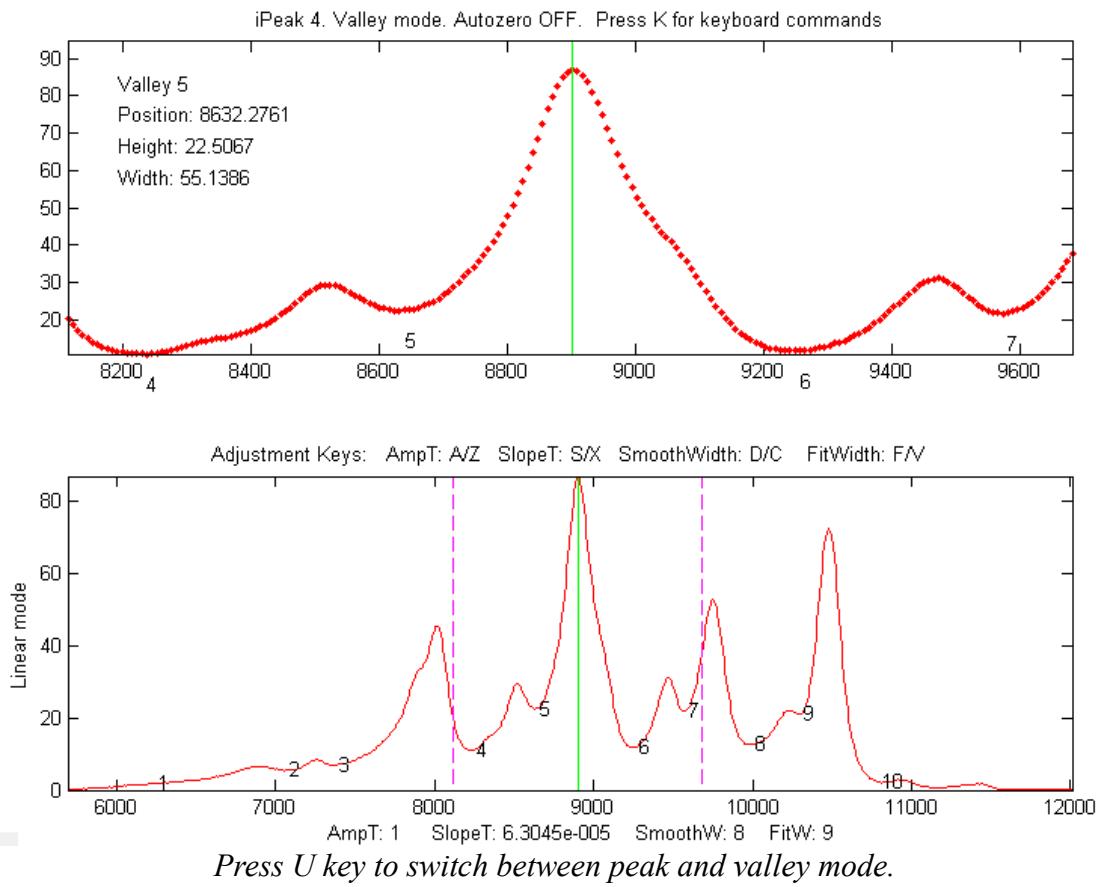


Adjust the peak detection parameters AmpThreshold (**A/Z** keys), SlopeThreshold (**S/X**), SmoothWidth (**D/C**), FitWidth (**F/V**) so that it detects the desired peaks and ignores those that are too small, too broad, or too narrow to be of interest. You can also type in a specific value of AmpThreshold 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. Don't 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.



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. 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

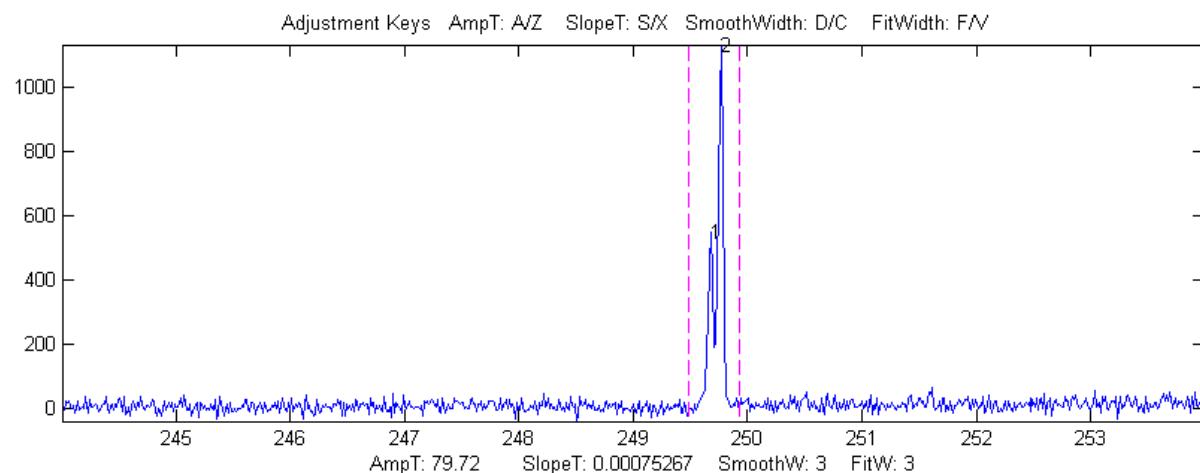
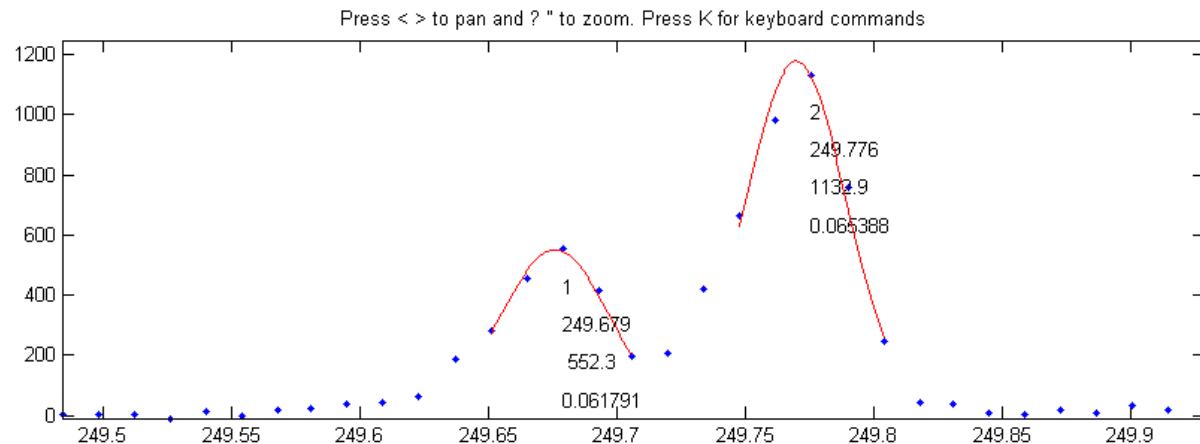
149 peaks detected

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).

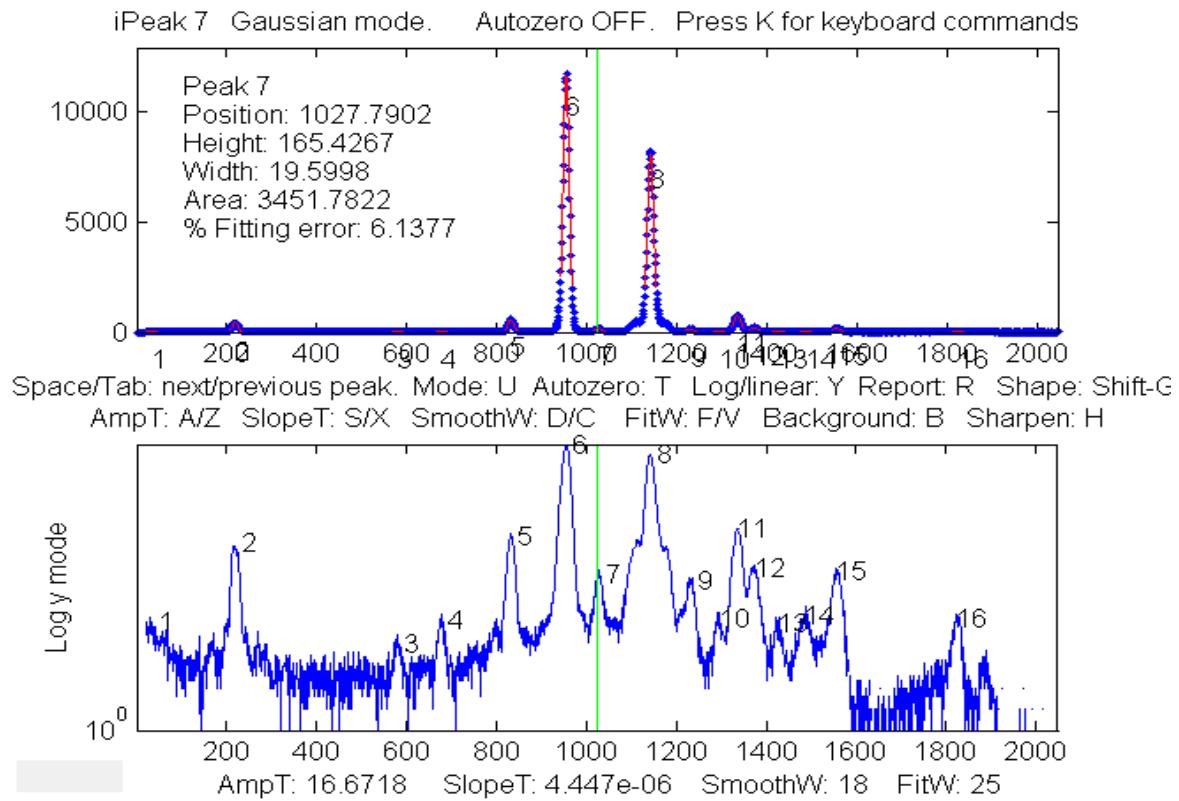
```
>> ipeak(datamatrix, 0, .5, .0001, 20, 20);
```



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*, to *linear*, to *quadratic*, to *flat mode*, 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).

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. 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 function to automate processing, 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 most or all of the input arguments in place, so you can 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 `iSignal.m` (page 323) and **Shift-Ctrl-P** transfers the current signal to Interactive Peak Detector (`iPeak.m`), if those functions are installed in your Matlab path.

Ensemble averaging. 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 in order 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 window 2 and saved as "EnsembleAverage.mat" in the current directory. See the script [iPeakEnsembleAverageDemo.m](#).

Normal and Multiple Peak fitting: The **N** key applies [iterative curve fitting](#) to the *detected peaks that are displayed in the upper window* (referred to here as "Normal" 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, 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 182). Make sure that the AmpThreshold, SlopeThreshold, SmoothWidth are adjusted so that each peak is numbered once. Only numbered peaks are fit. Then press the **N** key, which will display the following menu of peak shapes (graphic on page 369):

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: $y=\text{ones}(\text{size}(x)) / (1+((x-pos) / (0.5.*width)) .^2)$	
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: $y=\text{ones}(\text{size}(x)) / (1+((x-pos) / ((0.5.^2/m).*wid)) .^2) .^m$..	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): $y=.5+.5*\text{erf}((x-tau1) / \sqrt{2*tau2}))$.10	
Down Sigmoid $y=.5-.5*\text{erf}((x-tau1) / \sqrt{2*tau2}))$	23
Triangular.....	21

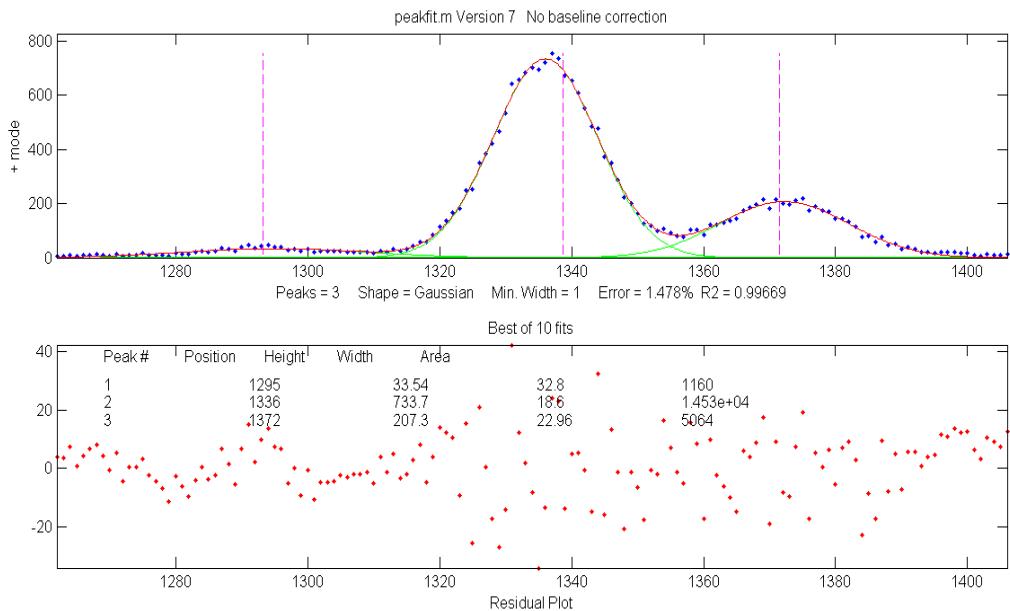
Type the number for the desired peak shape from this table and press **Enter**, then type in a number of repeat trial fits and press **Enter** (the default is 1; start with that and then increase if necessary). If you have selected a variable-shape peak (e.g. numbers 4, 5, 8 ,13, 14, 15, 18, 20, 30-33), 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

```



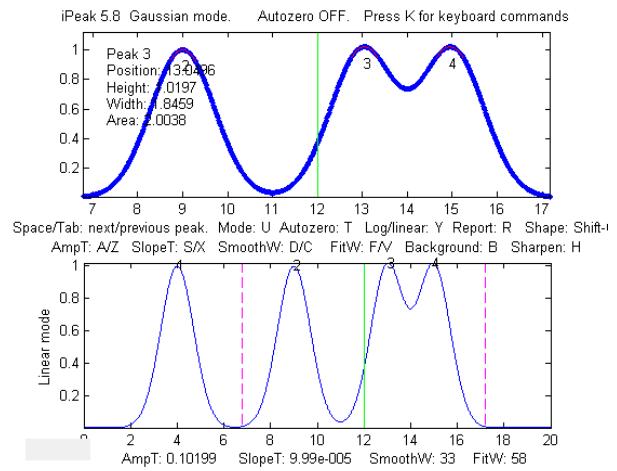
Normal Peak Fit (N key) applied to a group of three overlapping Gaussians peaks

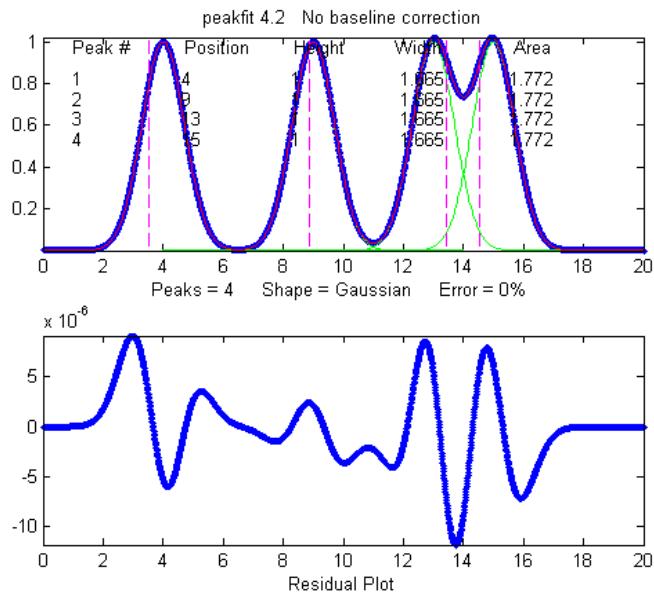
There is also a "Multiple" peak fit function (**M** key) that will attempt to apply iterative curve fitting to *all the detected peaks in the signal simultaneously*. Before using this function, it's best to turn *off* the automatic baseline correction (**T** key) and use the *multi-segment baseline correction 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 adjusted so that all the peaks in the selected region are detected and numbered once. (For more flexible curve fitting, use [ipf.m](#), page 361, 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 the peaks are Gaussian, but 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 sharpen function before pressing **M** (version 4.0 and above only). This does not effect the signal itself, only the peak detection.

Note 2: If you plan to use a variable-shape peak (numbers 4, 5, 8, 13, 14, 15, 18, or 20) for the **Multiple** peak fit, it's a good idea to obtain a reasonable value for the requested "extra" shape parameter by performing a **Normal** peak fit on an isolated single peak (or small group of partly-overlapping peaks) of the same shape, then use that value for the **Multiple** curve fit of the entire signal.

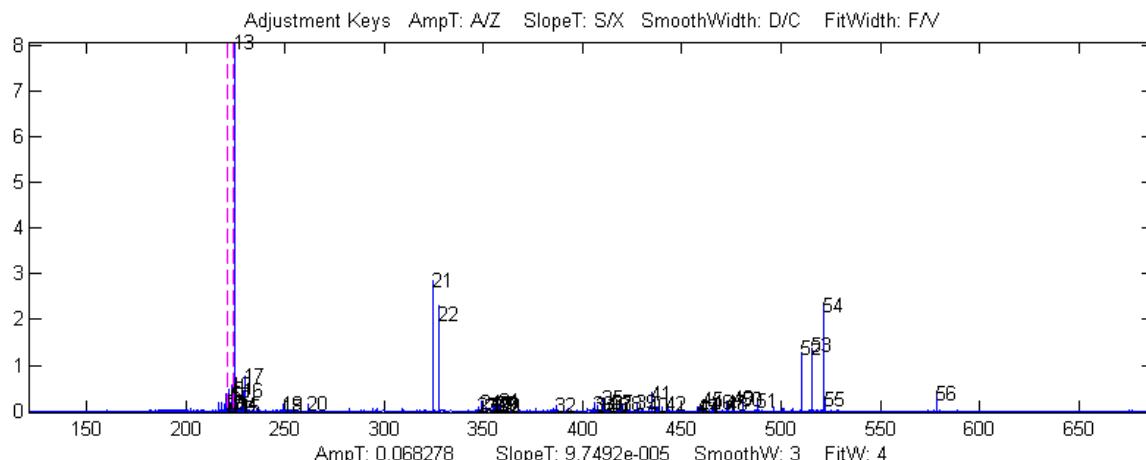
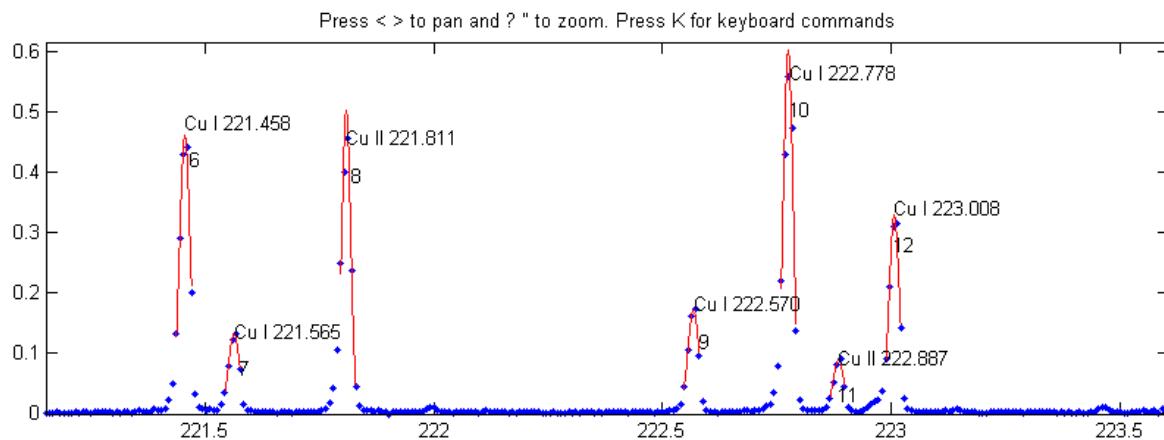
Note 3: If the peak shape varies across the signal, you can either use the **Normal** peak fit to fit each section with a different shape rather than the **Multiple** peak fit, or you can use the *unconstrained* shapes

that fit the shape individually for each peak: Voigt (30), ExpGaussian (31), Pearson (32), or Gaussian/Lorentzian blend (33).

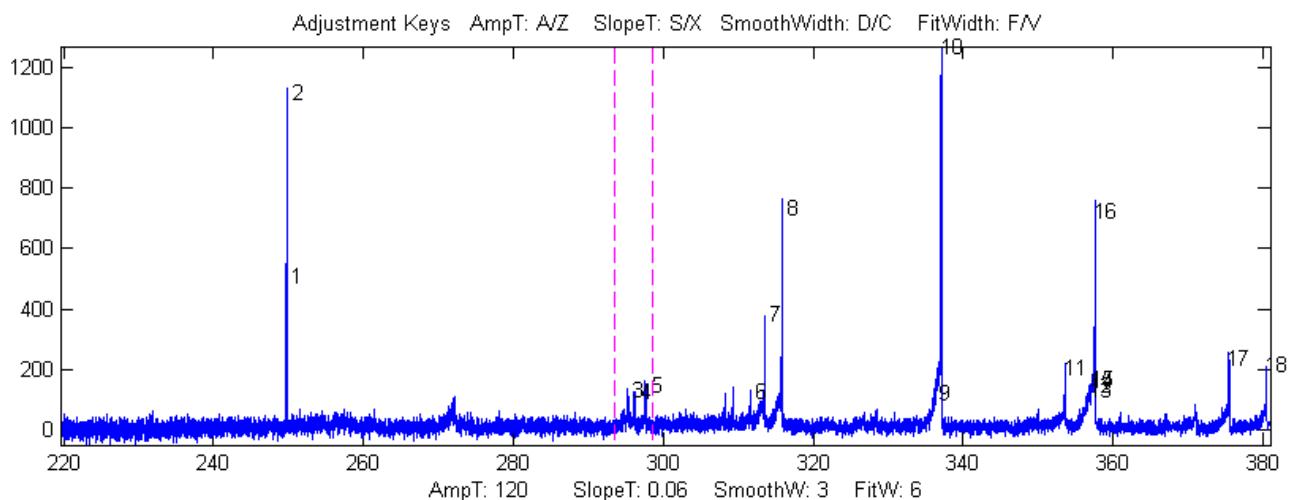
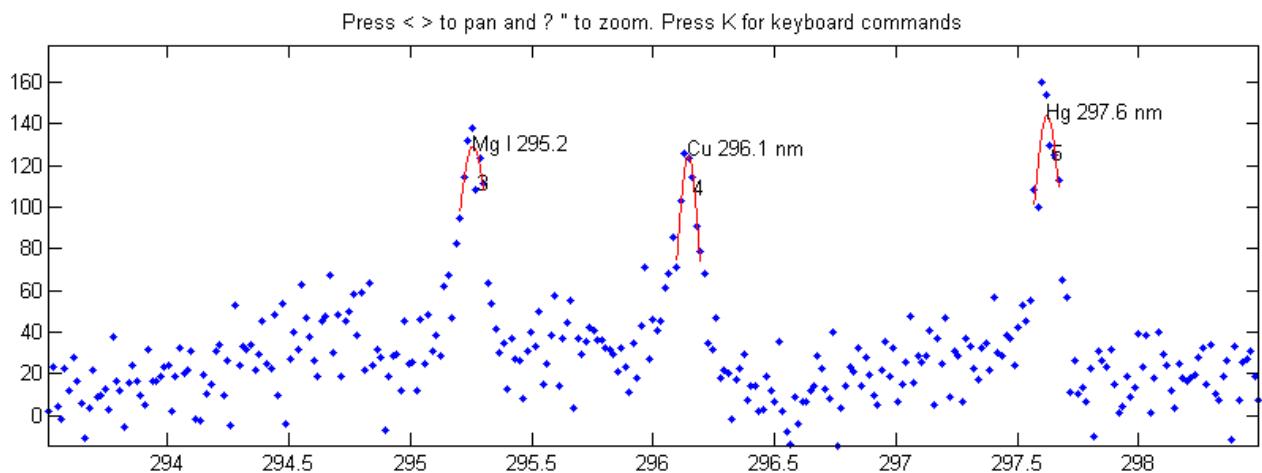
Peak identification. There is an optional "peak identification" function if optional input arguments 9 ('MaxError'), 10('Positions'), and 11 ('Names') are included. The "I" key toggles this function ON and OFF. This function compares the found peak positions (maximum x-values) to a reference database of known peaks, in the form of an 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'), 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: 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 'T' 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 spectrum (x-axis in nanometers).

```
>> 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.



An atomic emission spectrum of a sample containing trace amounts of several elements. Three small peaks near 296 nm are isolated in the upper panel and identified, based on the atomic line data in ipeakdata.mat. Press the I key to display the peak ID names.

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 nm'	[296.1]	[0.045368]	[124.6]
'Hg 297.6 nm'	[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, 10 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 have poor signal-to-noise ratios, so their identification might be in doubt (especially Iron, since its wavelength error is greater than the rest). It's up to the experimenter 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 I am really depending on the accurate wavelength calibration of the instrument. The evidence suggests that the wavelength calibration is excellent, based on the very small error for the *two well-known and relatively strong Sodium lines* at 589 and 589.59 nm. (Even so, I set MaxError to 0.2 nm in this example to loosen up the wavelength matching requirements).

'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's 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 7.9):**

```
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 just top part of
                                peaks)
Toggle sharpen mode .....H Helps detect overlapped peaks.
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=quadratic; 3=Flat.
Toggle valley mode OFF/ON...U Switch to valley mode
Gaussian/Lorentzian switch..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
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

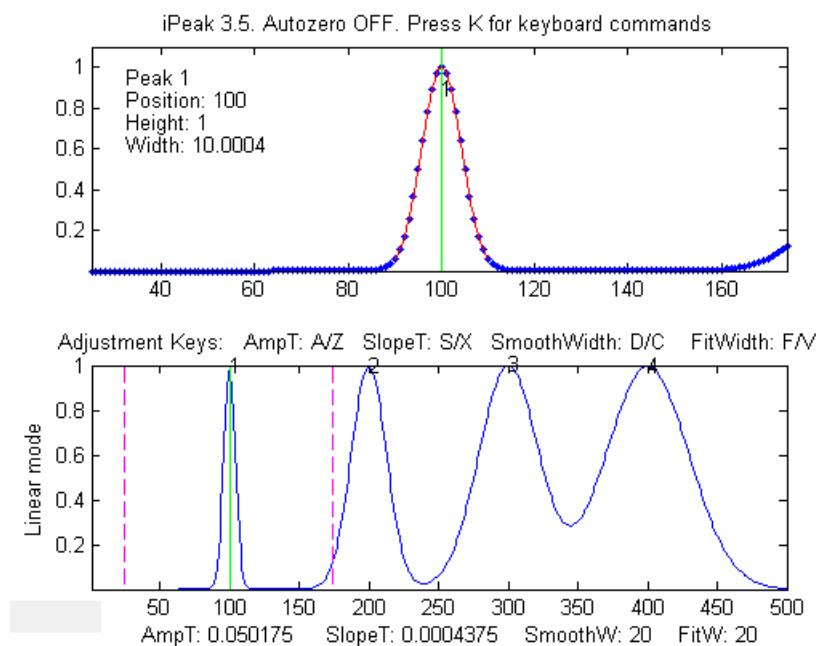
```

[Click for Animated step-by-step instructions](#)

iPeak Demo functions

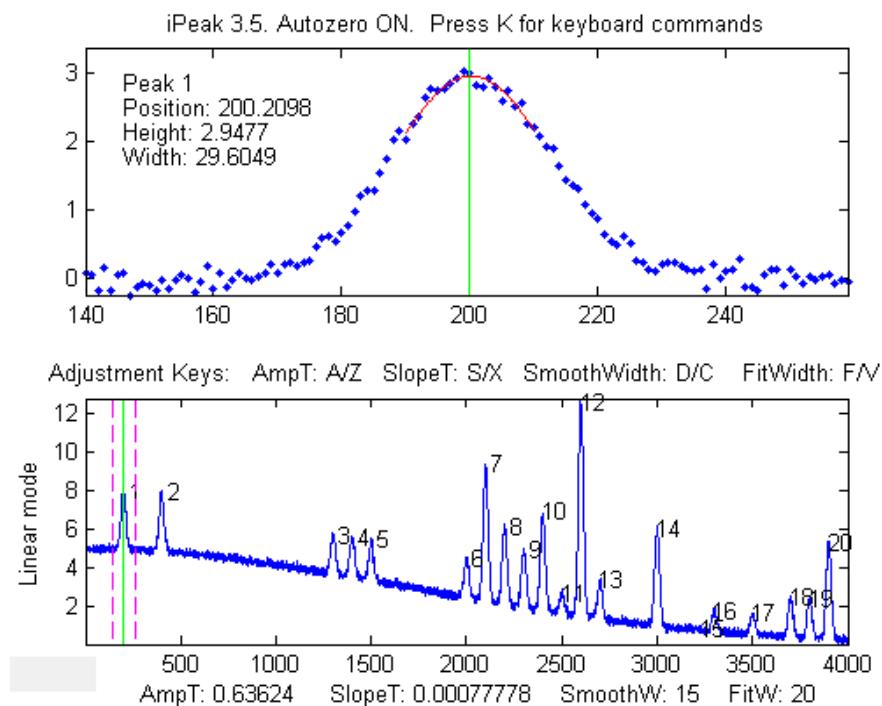
[demoipeak.m](#) is a simple demo function that generates a noisy signal with peaks, calls *iPeak*, and then prints out a table of the actual peak parameters and a list of the peaks detected and measured by *iPeak* for comparison. Before running this demo, [ipeak.m](#) must be downloaded and placed in the Matlab path. The ZIP file at <http://terpconnect.umd.edu/~toh/spectrum/ipeak7.zip> 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](#)".

ipeakdemo: effect of the peak detection parameters



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.

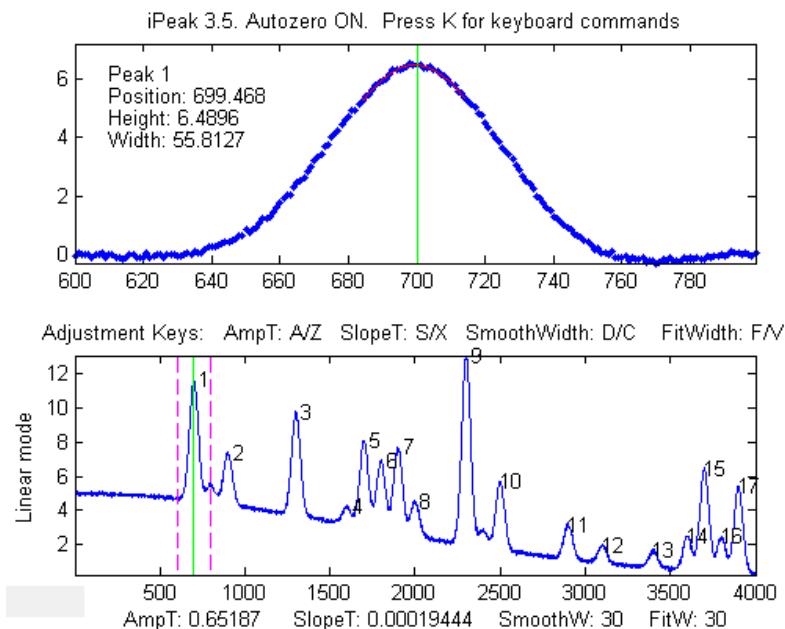
ipeakdemo1: the baseline correction mode. Demonstration of background correction, for separated, narrow peaks on a large baseline.



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 (F/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.

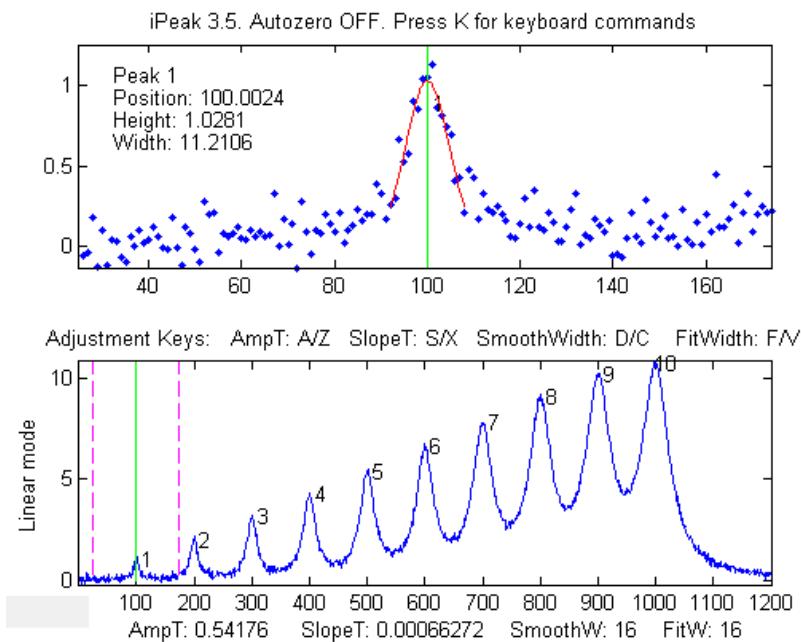
narrow peaks on a large baseline. Each time you run this demo, you will get a different set of peaks and noise. A table of the actual peak positions, heights, widths, and areas is printed out in the command window. Jump to the next/ previous peaks using the **Spacebar/Tab** keys. Hint: Set the linear 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.

[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 turn on the background correction mode (T key) and use the Normal curve fit (N key) with peak shape 1 (Gaussian).

[ipeakdemo3: Baseline shift caused by overlapping peaks](#)

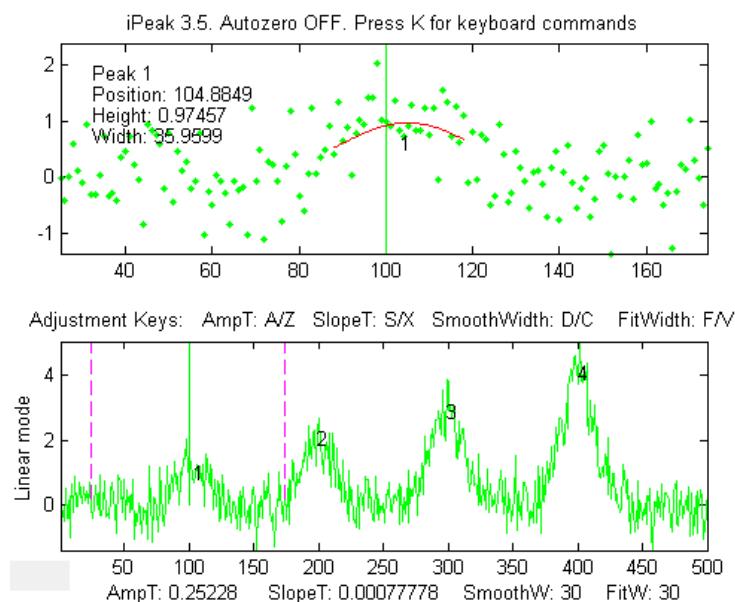


Demonstration of overlapping Lorentzian peaks, without an added background. A table of the actual peak positions, heights, widths, and areas is printed out in the command window; in this example, the true peak heights are 1,2 3,...10. Overlap of peaks can cause significant errors in measuring peak parameters, especially for Lorentzian peaks, because they have gently sloping sides that contribute to the baseline of any peaks in the region.

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

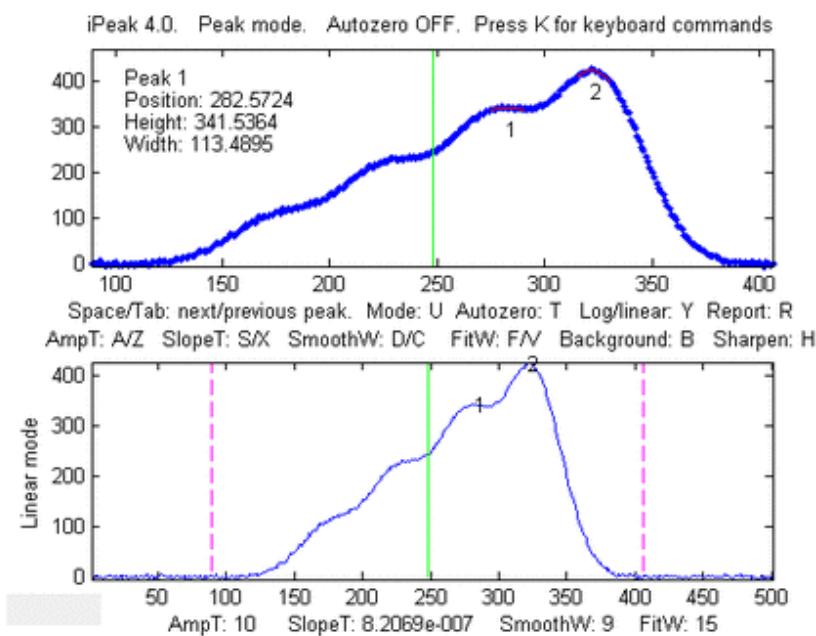


best to increase *SmoothWidth* and *FitWidth* to help reduce the effect of the noise.

Detection and measurement of four peaks in a very noisy signal. The signal-to-noise ratio of 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 on noise (white, pink, proportional, etc.).

Hint: With very noisy signals it is usually

[ipeakdemo5](#): dealing with highly overlapped peaks

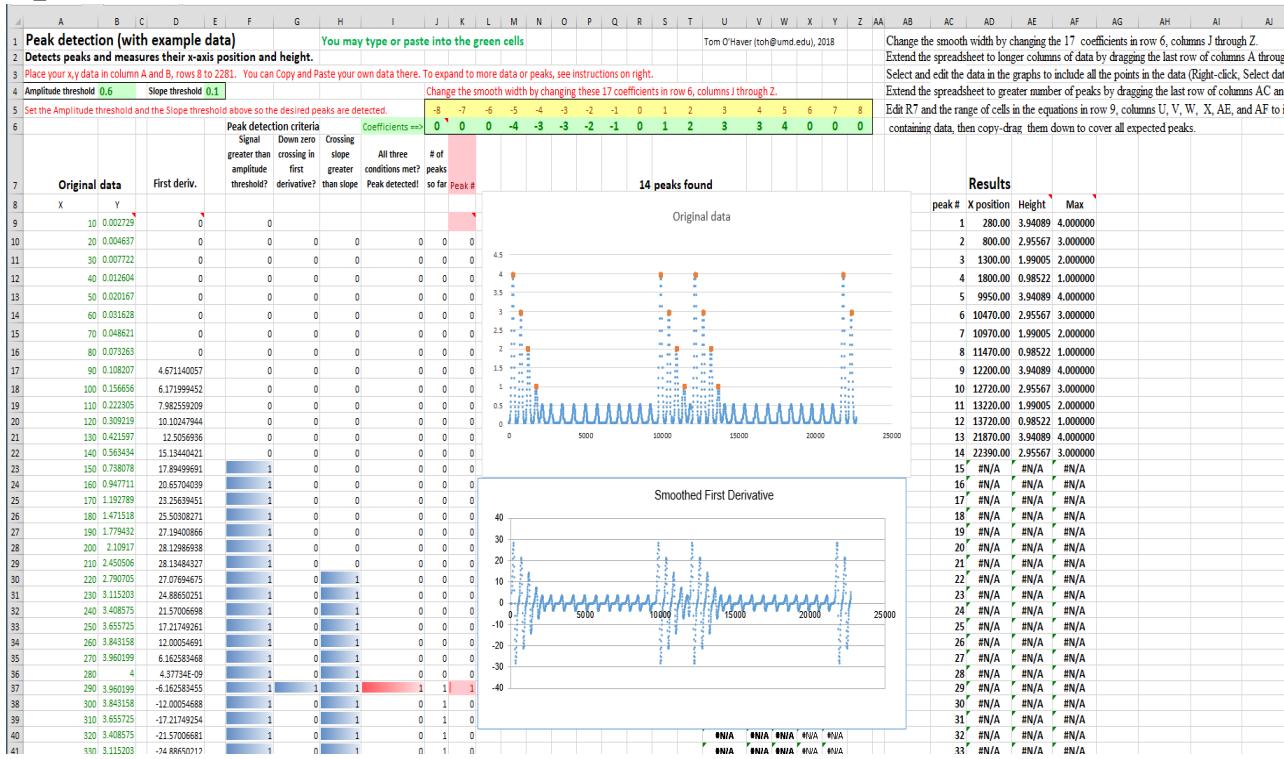


'H' key will activate the peak sharpen function that decreases peak width and increases peak height of all the peaks, making it easier to detect and number all the peaks for use by the peakfit function (N key). Note: peakfit fits the original unmodified peaks; the sharpening is used only to help locate the peaks to provide peakfit with suitable starting values.

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

Spreadsheet Peak Finders



Simple peak detection. The spreadsheet [PeakDetectionTemplate.xls](#) (or [PeakDetectionExample.xls](#) with sample data) implements the simple derivative zero-crossing peak detection method. 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 65. 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 57) 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 the spreadsheet 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 *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, 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. 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 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 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'll 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 of 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 a large number of 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 make 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. Spreadsheet programs have many flexible and easy-to-use formatting options to make displays more attractive. On the down side, a spreadsheet as complicated as PeakDetectionAndMeasurement.xlsx is far more difficult to construct than its Matlab/Octave equivalent. Much more serious, the spreadsheet is *less flexible* and harder to expand to larger signals and larger number of peaks. In contrast, the Matlab/Octave equivalent, while requiring some understanding of programming to create initially, is easy to use, faster in execution, much more flexible, and can easily handle signals and smooth/fit widths of any size. Moreover, because it is written as a Matlab/Octave *function*, it can be readily employed as an element in your own custom Matlab/Octave programs to perform even larger tasks. It's harder to do that in a spreadsheet.

To compare the computation speed of this spreadsheet peak finder to the Matlab/Octave equivalent, we 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:

```
tic
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
toc
```

The table below compares the elapsed times measured on a typical PC desktop computer. The speed advantage of Matlab is clear.

Method	Elapsed time
Excel spreadsheet	~ 1 sec
Calc spreadsheet	~ 1 sec
Matlab script	0.132 sec
Octave script	0.85 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 would be more significant. Moreover, Matlab would be the method of choice if you have a large number of separate data sets to which you need to apply a peak detection/measurement algorithm completely automatically (page 306). See also [Excel VS Matlab](#).

Hyperlinear Quantitative Absorption Spectrophotometry

Specific knowledge of the instrumentation design is often essential in designing an effective signal-processing regimen. This example shows how custom signal processing procedure for an optical measurement system is constructed by combining several of the methods and concepts introduced in this essay, which *expands the classical limits of measurement* in optical spectroscopy. This is a computational method for quantitative analysis by [multiwavelength absorption spectroscopy](#), called the transmission-fitting or "TFit" method, which is based on [fitting a model](#) of the instrumentally-broadened transmission spectrum to the observed transmission data, rather than direct "classical" calculation of absorbance as $\log_{10}(I_0/I)$. 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.* **63**, 164-169 (1991). It is included as an example of combining techniques because it uses many of the important concepts that have been covered in this essay: signal-to-noise ratio (page 21), Fourier convolution (page 93), multicomponent spectroscopy (page 151), iterative least squares fitting (page 163), and calibration (page 298).

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](#) ("hyperlinearity"), which reduces the labor and cost of preparing and running large numbers of standard solutions *and safely disposing of them afterwards*, 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, 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 [multilinear regression \(classical least squares\)](#) methods conventionally used in absorption spectroscopy, the Tfit method

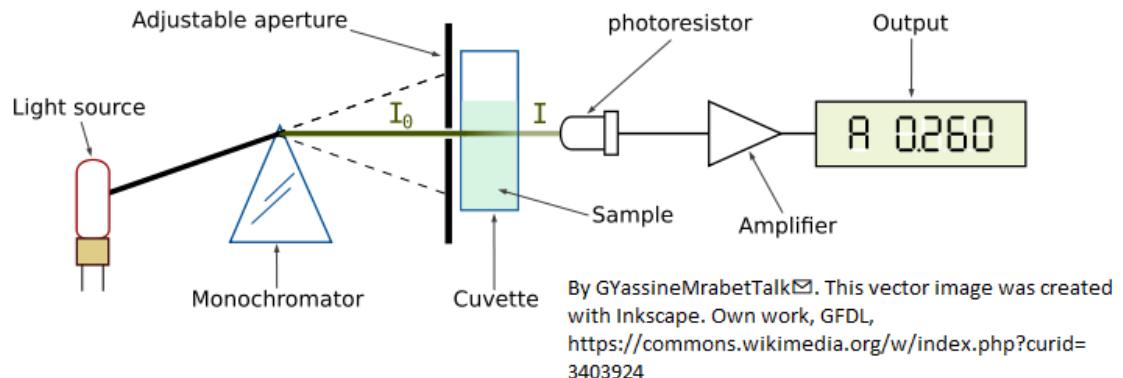
- (a) requires an accurate reference spectrum of each analyte,
- (b) utilizes multiwavelength data such as would be acquired on diode-array, Fourier transform, or automated scanning spectrometers,
- (c) applies both to single-component and [multicomponent mixture](#) analysis, and
- (d) requires that 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 values calculated by the traditional log (Io/I) method), and
- (d) It won't 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

In [optical absorption spectroscopy](#), the intensity I of light passing through an absorbing sample is given (in Matlab notation) by the [Beer-Lambert Law](#):

$$I = I_0 \cdot 10^{-(\alpha \cdot L \cdot c)}$$

where “ I_0 ” (pronounced “eye-zero”) is the intensity of the light incident on the sample, “ α ” is the absorption coefficient of the absorber, “ L ” is the distance that the light travels through the material (the

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.

In conventional applications, measured values of I and Io are used to compute the quantity called "absorbance", defined as

$$A = \log_{10}(I_0/I)$$

Absorbance is defined in this way so that, when you combine this definition with the Beer-Lambert law, you get:

$$A = \alpha \cdot L \cdot c$$

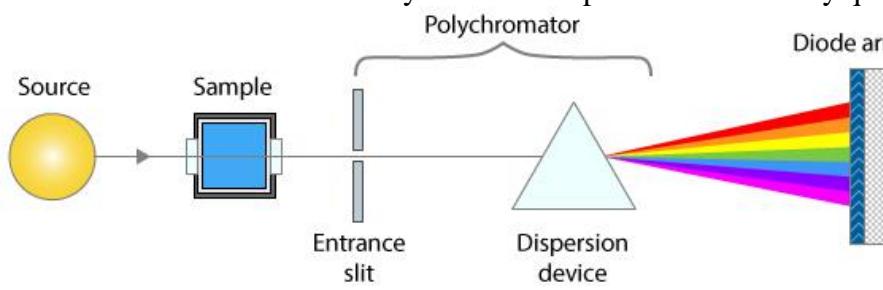
Absorbance is proportional to concentration, ideally, which simplifies analytical calibration. However, any real spectrometer has a *finite spectral resolution*, meaning that the light beam passing through the sample is not truly monochromatic, with the result that an intensity reading at one wavelength setting is actually an average over a small spectral interval. More exactly, what is actually measured is a convolution of the true spectrum of the absorber and the *instrument function*, the instrument's response as a function of 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 calculated absorbance will no longer be linearly proportional to 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 an dynamic range even as high as 5 or 6 absorbance, so that much of the theoretically optimum absorbance range is actually unusable. (c.f. <http://en.wikipedia.org/wiki/Absorbance>). So, in conventional practice, greater sample dilutions and shorter 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 actually measured. Here's 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's well known that the absorption peaks of molecules are more-or-less 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 than molecular bands, have a finite non-zero width, mainly due to their velocity (temperature or Doppler broadening) and 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 of the *same* energy, ideally corresponding to the average energy distribution of the collection of atoms or molecules being measured. But many - actually most - of the atoms or molecules in the light path would have an energy *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, but then the simple calculation of absorbance as $\log_{10}(I_0/I)$ 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 then using the conventional $\log_{10}(I_0/I)$ absorbance would result in very substantial non-linearity over most of the absorbance range 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 theory-

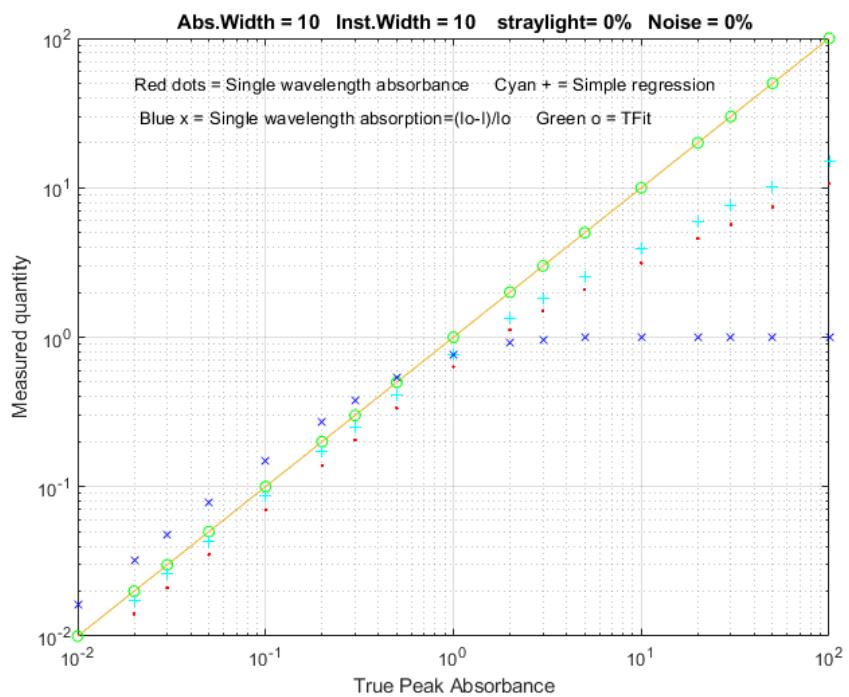


based approach would be to perform the curve fitting in the *spectral domain* where the source of the non-linearity arises. This is possible with *modern* absorption spectrophotometers that use *array detectors*.

tors, 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 spectral resolution is determined by both the entrance slit width and by the range of optical detector elements that are summed 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 - initially equal to the conventional $\log_{10}(I_0/I)$ 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 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](#). The TFit 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 important, 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](#). So the additional mathematical work needed to compute the *absorbance*, $\log(I_0/I)$, rather than the simpler relative *absorption*, $(I_0-I)/I_0$, was justified because of the better linearity of absorbance with respect to concentration and path length, and the log calculation could easily be performed simply 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 more mathematically complex than a logarithm and cannot be done without the aid of software (at least a [spreadsheet](#)) and [some sort of computational hardware](#), but 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 analytical curve linearity of simple relative absorption (blue x), logarithmic absorbance (red dots), multilinear regression or CLS method (cyan +) based on absorbance, and the TFit method (green o). This plot was created by the Matlab/Octave script [TFitCalCurveAbs.m](#).



Bottom line: The TFit method is based on the Beer-Lambert Law; it simply calculates the absorbance in a different way that does not require the assumption that stray light and polychromatic radiation effects are zero. Because it allows larger slit widths and shorter focal lengths to be used, it yields greater 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 $\log(I_0/I)$ 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 unthinkable. That was then; this is now. *Tfit is the 21st century way to do quantitative absorption spectrophotometry.*

Note: 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, in particular *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.

Spreadsheet implementation

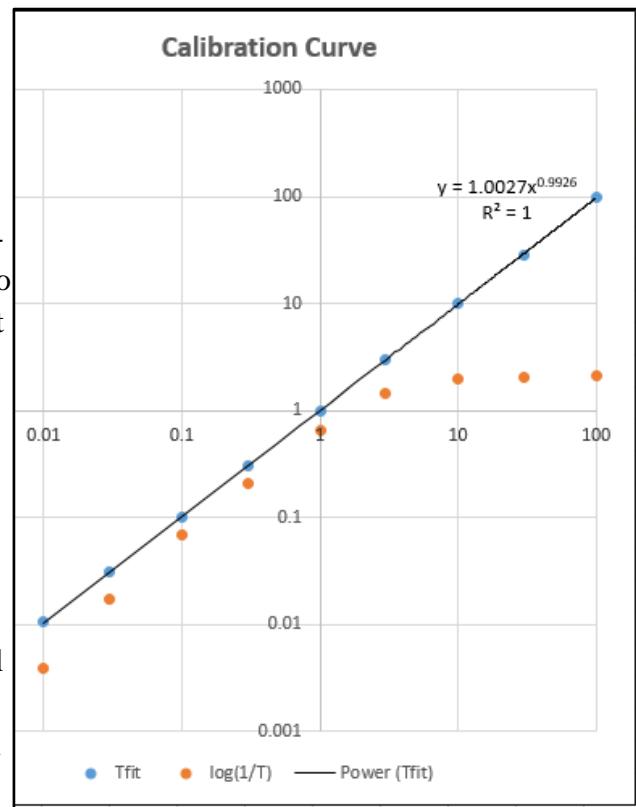
The Tfit method can also be implemented in an *Excel* or *Calc* spreadsheet; it's a bit more cumbersome than the [Matlab/Octave implementation](#), but it works. The shift-and-multiply method is used for the convolution of the reference spectrum with the slit function, and the "Solver" add-in for Excel and Calc is used for the iterative fitting of the model to the observed transmission spectrum. It's 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 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 278). 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 R² value for the TFit method, in the upper right corner of 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's convenient to use the conventional $\log_{10}(I_0/I)$ estimate of absorbance for `start`. The other arguments (described above) are passed on to FitM. In this example, fminsearch 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, for a **single-component measurement** where the true absorbance is 1.00, using only *4-point spectra* for illustrative simplicity (of course, 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 too low: $\log_{10}(1/.38696)=0.4123$. 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,In-  
stFun,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 $\log_{10}(I_0/I)$ absorbance, which is easily calculated and which is a 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 [multilinear regression methods](#). This can be true if there are 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 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 of 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, although the TFit method does make the *computer* work harder, the computation time on a typical laboratory personal computer is only a fraction of a second (roughly 25 μ sec 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 even be performed, with some loss in speed, on a \$35 single-board computer (see page 304).

Demo function for [Octave](#) or Matlab

The [**tfit.m**](#) function is a self-contained command-line 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

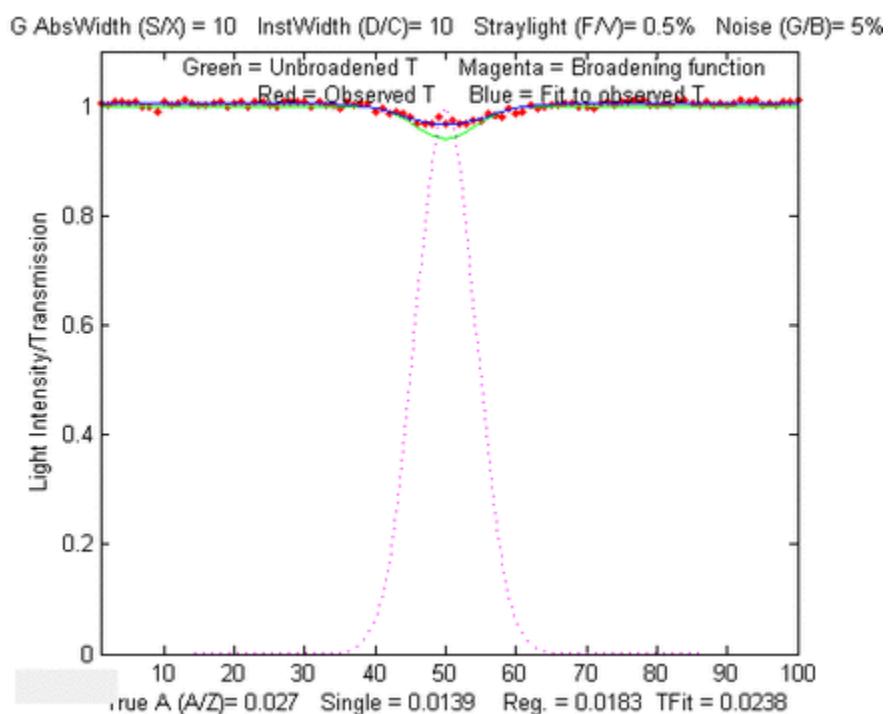
>> tfit(10)
      10      1.4858     2.2244     9.5123    9.9979

>> 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 for Matlab, version 2.1



[TFitDemo.m](#) (for animation go to <https://terpconnect.umd.edu/~toh/spectrum/TFitAnimated.gif>) is a keypress-operated interactive explorer for the Tfit method (for Matlab only), 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. 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 path, then type "TFitDemo" at the Matlab command prompt. 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 below the graphs show that, at every absorbance and for either a Gaussian or a Lorentzian peak shape, the TFit method gives a 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 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 independent random noise
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 Figure 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 the most common type of absorption spectrometer, using a continuum light source and a dispersive spectrometer, there are actually *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 effects 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 making the assumption 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 simple script that computes the statistics of the TFit method compared to single-wavelength (SingleW), simple regression (SimpleR), and weighted regression (WeightR) methods. 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. 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 is 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
```

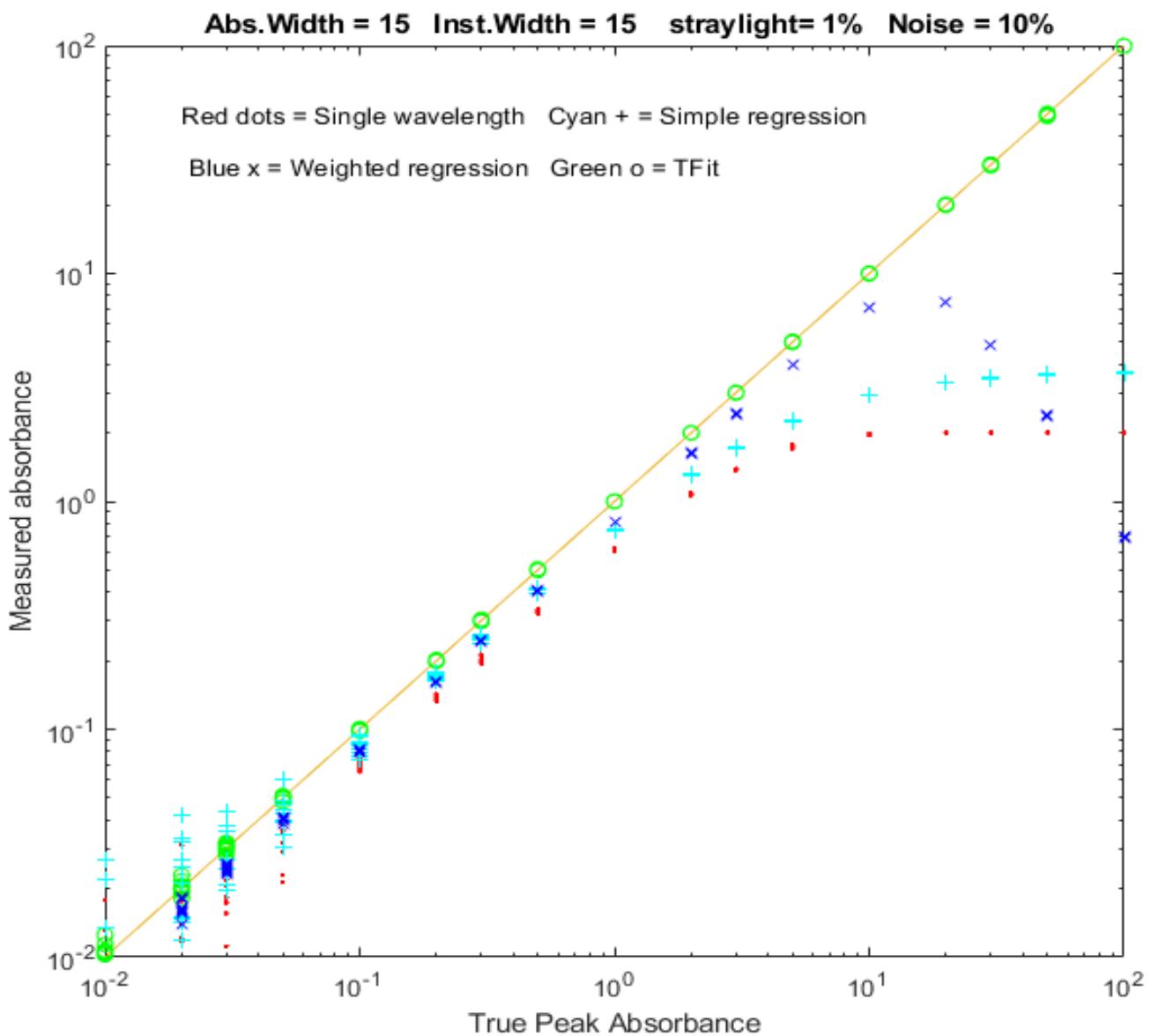
As you can see, the Tfit method offers improved accuracy *and* precision.

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 linearity of Tfit is especially important in a regulated lab where [quadratic least-squares fits are discouraged](#).

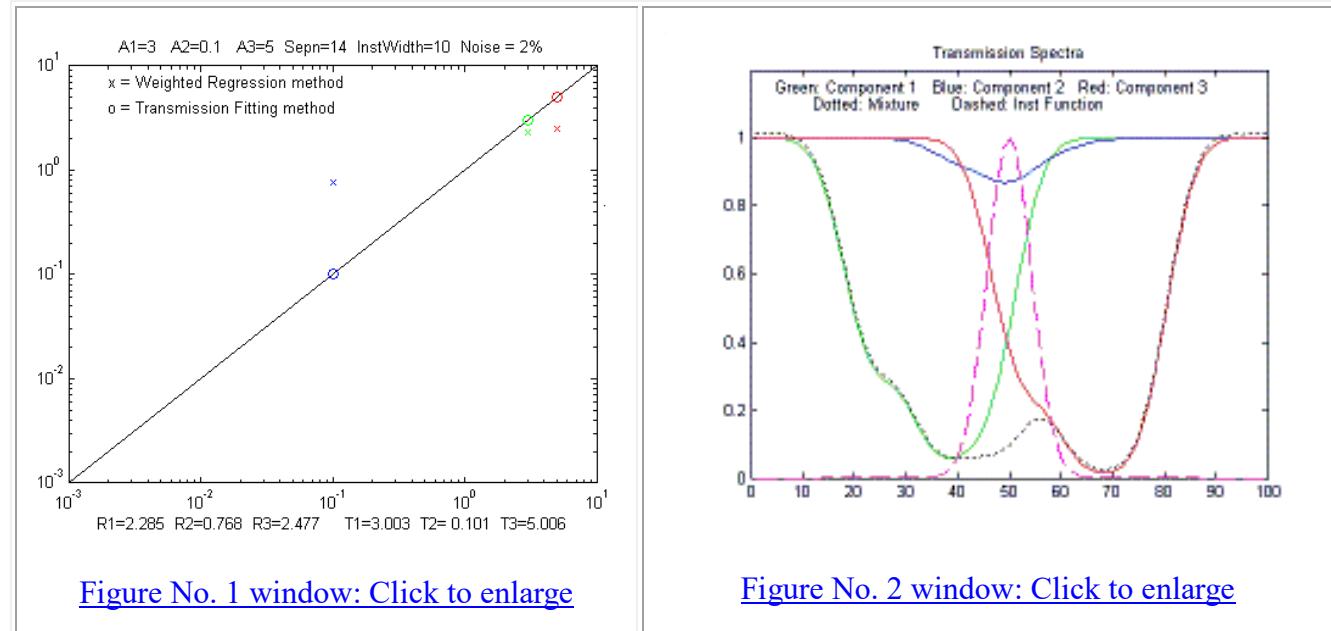
This calibration curve function is included as a keypress command (**M** key) in [TFitDemo.m](#).



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

for Matlab/[Octave](#) (and [TFit3Demo.m](#), for Matlab only)



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 be equal to the sum of the absorbances of the separate components. In practice, this requires that the absorbers do not interact chemically; that is, 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 multicomponent methods as well as to the T-Fit method.

[TFit3.m](#) is a 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 (A_1 , A_2 , and A_3), spectral overlap between the component spectra ("Sepn"), width of the instrument function ("InstWidth"), and the noise level ("Noise"). Compares quantitative measurement by weighted regression and TFit methods. 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 don't 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 (shown in blue) is almost completely buried by the stronger absorption bands of components 1 and 3 on either side, giving a much weaker absorbance (0.1) than the other two components (3 and 5, respectively). Even in this 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 methods works better (although of course *nothing* works if the spectral differences between the components is too small). Note: in this program, as in all of the above, when you change InstWidth, the photon noise automatically changes accordingly just as it would in a real variable-slit dispersive spectrophotometer.

You can also download the [newer self-contained keyboard-operated version](#) that works in recent versions of Matlab:

KEYBOARD COMMANDS

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

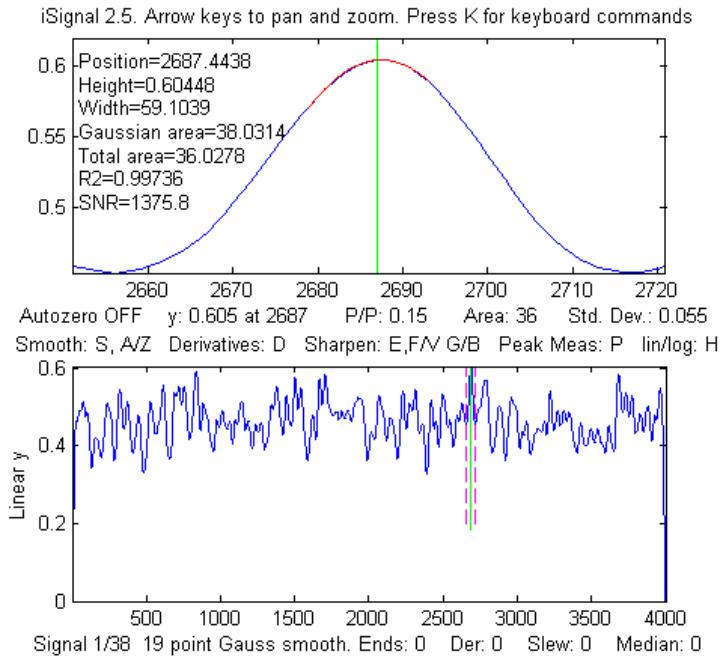
Sample table of results (by pressing the Tab key):

	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

Note for [Octave](#) users: the current versions (October, 2012 or later) of [fitM.m](#), [TFit3.m](#), [TFitStats.m](#) and [TFitCalDemo.m](#) work in [Octave](#) as well as in Matlab. However, the interactive features of TfitDemo.m and Tfit3Demo.m work only in Matlab; Octave users can use the command-line function [tfit.m](#) (single absorbing component) or [TFit3.m](#) (mixture of 3 absorbing components) instead. See page 399 or <http://tinyurl.com/cey8rwh> for a list of and links to these and other Matlab and Octave functions.

Tutorials, Case Studies and Simulations.

A. Can smoothed noise may 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 323) displaying a computer-generated 4000-point signal consisting of pure random noise that has been smoothed with a 19-point Gaussian 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 to look nice. Don’t fool yourself.

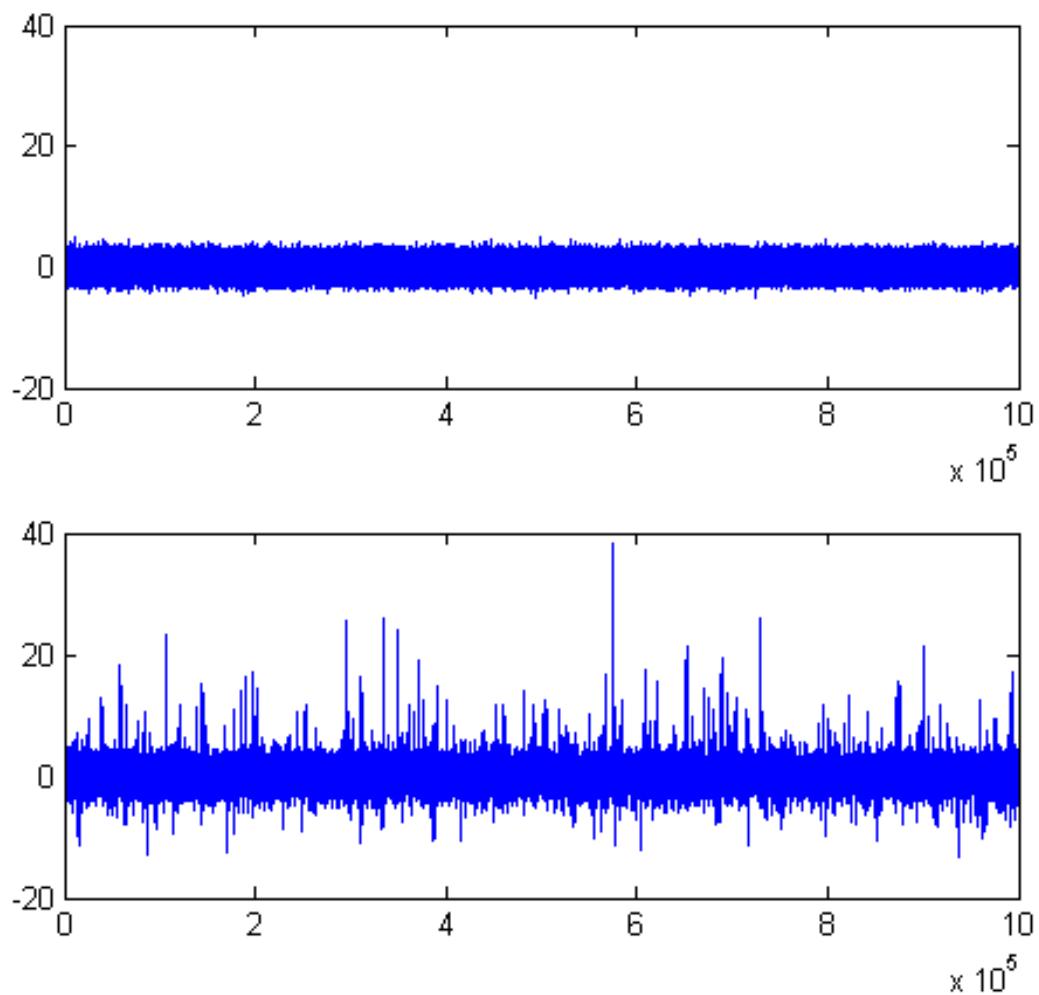
containing only normally-distributed white noise. Then it uses 'fastsmooth.m' to smooth that noise, resulting in a 'signal' with a standard deviation of about 0.3 and a maximum value around 1.0. That signal is then submitted to [iPeak](#) (page 216). If the peak detection criteria (e.g. AmpThreshold and SmoothWidth) are set too low, many peaks will be found. But setting the AmpThreshold to 3 times the standard deviation ($3 \times 0.3 = 0.9$) 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 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.

B. Signal or Noise?

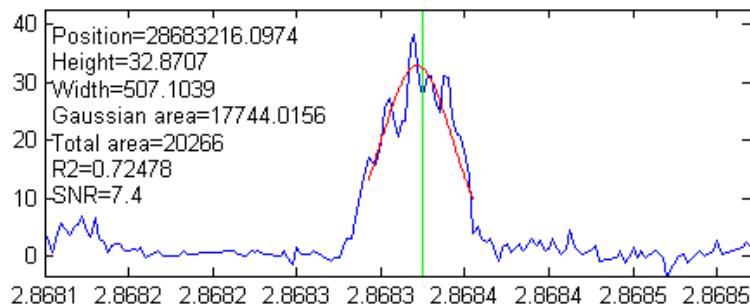
The client’s experimental signal in this case study was unusual because it did not look like a typical signal when plotted; in fact, it looked a lot like noise at first glance. The figure below compares the raw signal (bottom) with the same number of points of normally-distributed white noise (top) with a mean of zero and a standard deviation of 1.0 (obtained from the Matlab/Octave 'randn' function).



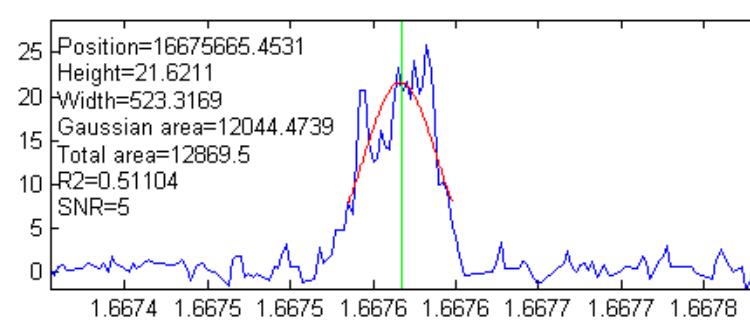
As you can see, the main difference is that the signal has more large 'spikes', especially in the positive direction. This difference is evident when you look at the [descriptive statistics](#) of the signal and the randn function:

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. Most of these turned out to be the peaks of interest for this signal; they look like spikes only because the length of the signal

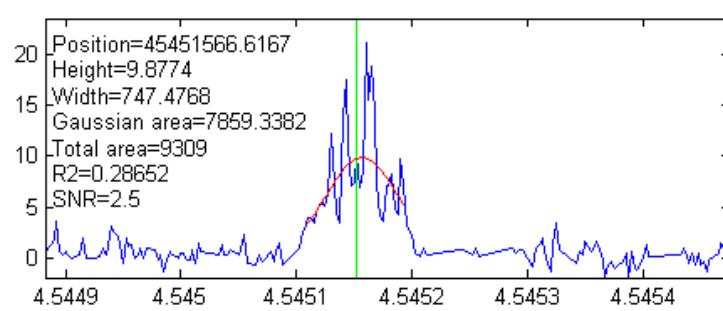


(over 1,000,000 points) causes the peaks to be compressed into one screen pixel or less when the entire signal is plotted on the screen. In the figures on the left, iSignal (page 323) is used to "zoom in" on some of the larger of these peaks (using the cursor arrow keys). The peaks are very sparsely 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 said that a 'good' peak was 'bell shaped', with an amplitude above 5 and a width of 500-1000 x-axis units. So that means that we 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 on the left that the peak widths do indeed meet those expectations.

The interval between adjacent x-axis points is 25, so that means we can expect the peaks to have about 20 to 40 points in their widths. Based on that, we 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, we can use the `findpeaksG` or `iPeak` (page 361) 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"

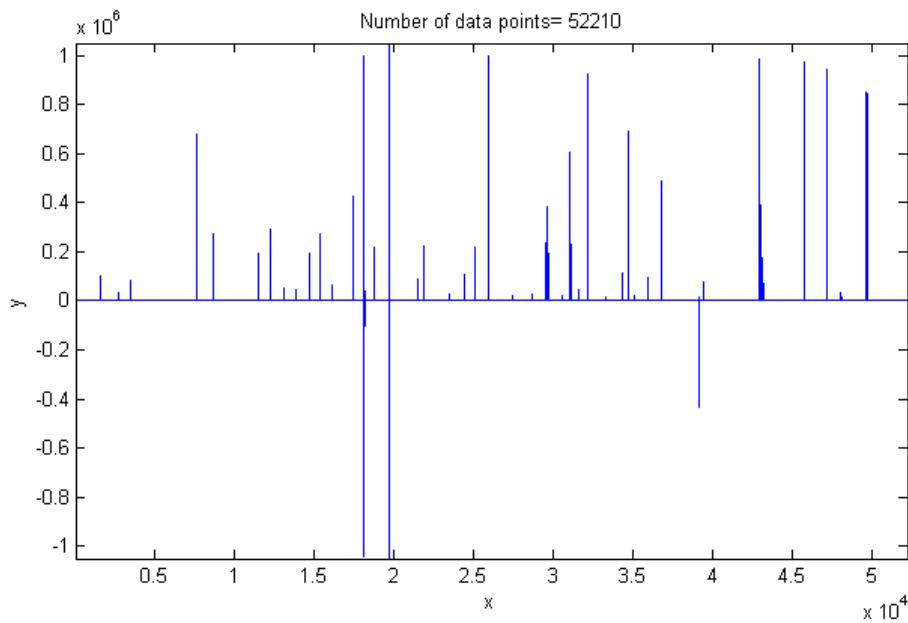
peaks. For example, using *AmplitudeThreshold*=5, *SlopeThreshold*=.001, *SmoothWidth*=25, and *FitWidth*=25, these function detect and measure 76 peaks above an amplitude of 5 and with an average peak width of 523. The interactive [iPeak](#) function (page 361) is especially convenient for exploring 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 in reality the criteria for good and bad peaks is at least partly subjective, so it's 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, height of 32, and width of 500; the bootstrap method determines that the standard deviations are 4, 0.92, and 9.3, respectively.

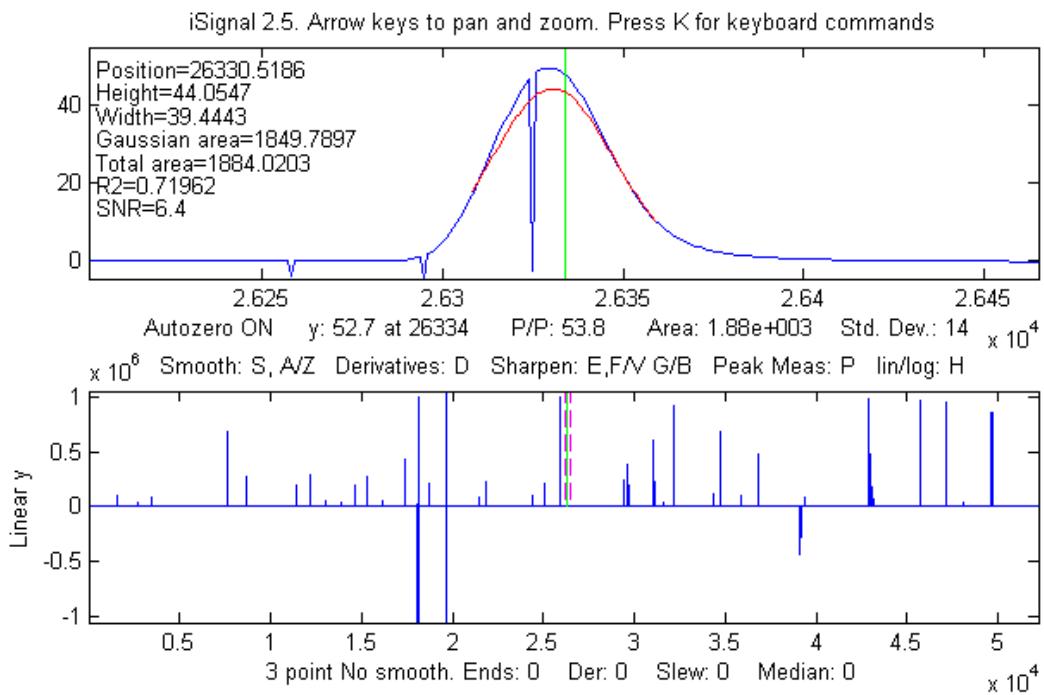
Because the signal in the case study was so large (over 1,000,000 points), the interactive programs such as [iPeak](#), [iSignal](#), and [ipf](#) may be sluggish in operation, especially if your computer is not fast computationally or graphically. If this is a serious problem, it may be best to break 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.

C. Buried treasure

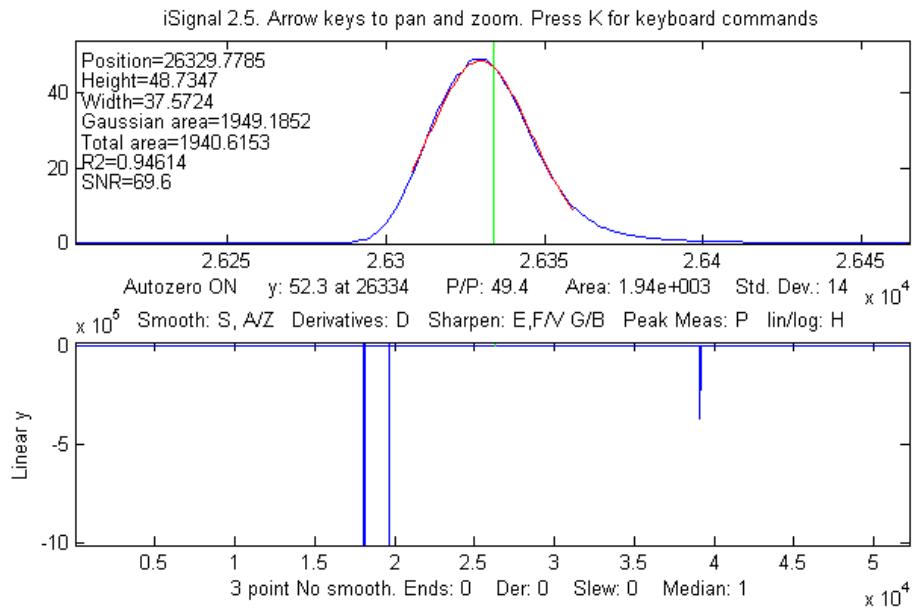
The experimental signal in this case study had a number of narrow spikes rising above a *seemingly flat baseline*.



Using [iSignal](#) (page 323) to investigate the signal, I found that the visible positive spikes were *single points* of very large amplitude (up to 10^6), whereas the regions *between* the spikes were not really flat but contained bell-shaped peaks that were so much smaller (below 10^3) that they were not even visible on this scale. For example, using [iSignal](#) to zoom in to the region around $x=26300$, you can see one of those bell-shapes peaks with a small single-point negative-going spike artifact near its peak.

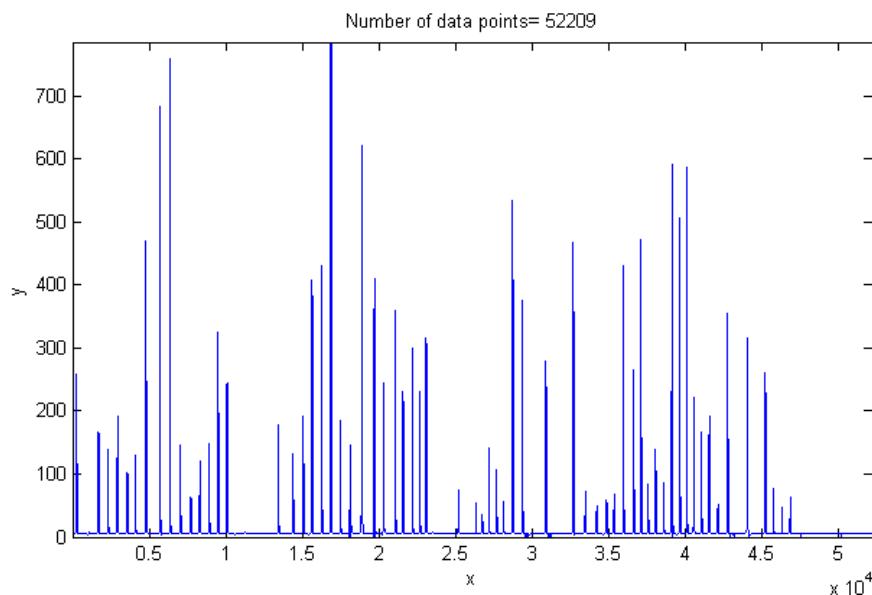


Very narrow spikes like this are common artifacts in some experimental signals; they are easy to eliminate by using a [median filter](#). The [iSignal](#) function (page 323) has such a filter, activated by the “M” key. The result (on the next page) shows that the single-point spike artifacts have been eliminated, with little effect on the character of the bell-shaped peak.

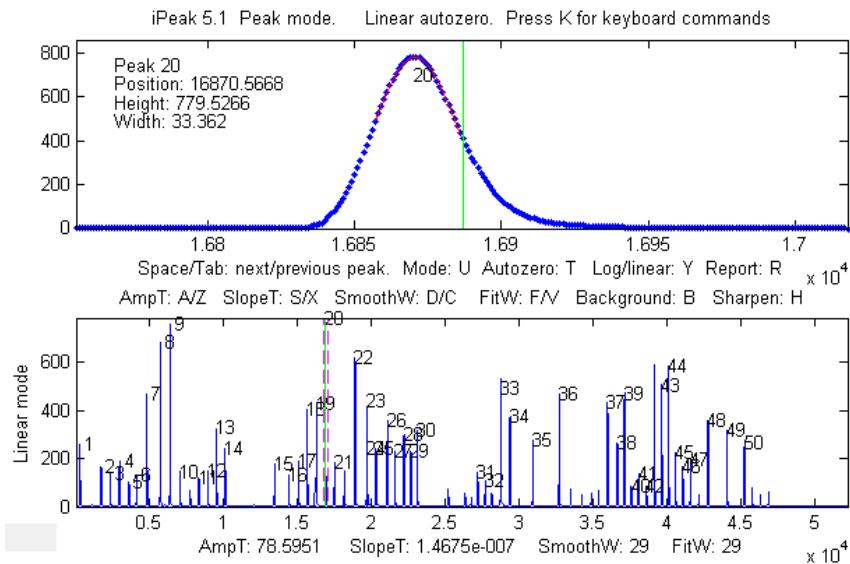


Other filter types, like most forms of smoothing (page 35), would be far less effective than a median filter for this type of artifact and would distort the peaks.

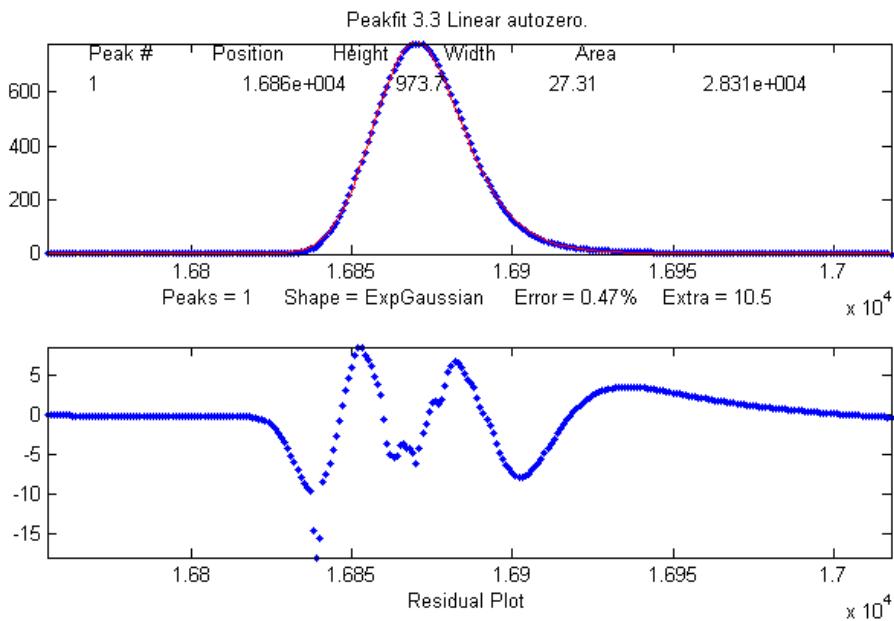
The negative spikes in this signal turned out to be steep *steps*, which can either be reduced by using iSignal's [slew-rate limit](#) function (the ` key) or manually eliminated by using the semicolon key (;) to set the selected region between the dotted red cursor lines to zero. Using the latter approach, 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 361) can automate measurement of peak positions and heights for the entire signal.



If required, individual peaks can be measured more accurately by fitting the whole peak with *iPeak*'s "N" key (page 361) or with *peakfit.m* or *ipf.m* (page 361). The peaks are all slightly asymmetrical; they fit an exponentially-broadened Gaussian model (page 192) to a fitting error less than about 0.5%, as shown on the left. The smooth residual plots suggests that the signal was smoothed *before* the spikes were introduced.



Note that fitting with an exponentially-broadened Gaussian model gives the peak parameters of the Gaussian *before* broadening. *iSignal* (page 323) and *iPeak* (page 361) 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

D. The Battle Rounds: a comparison of methods

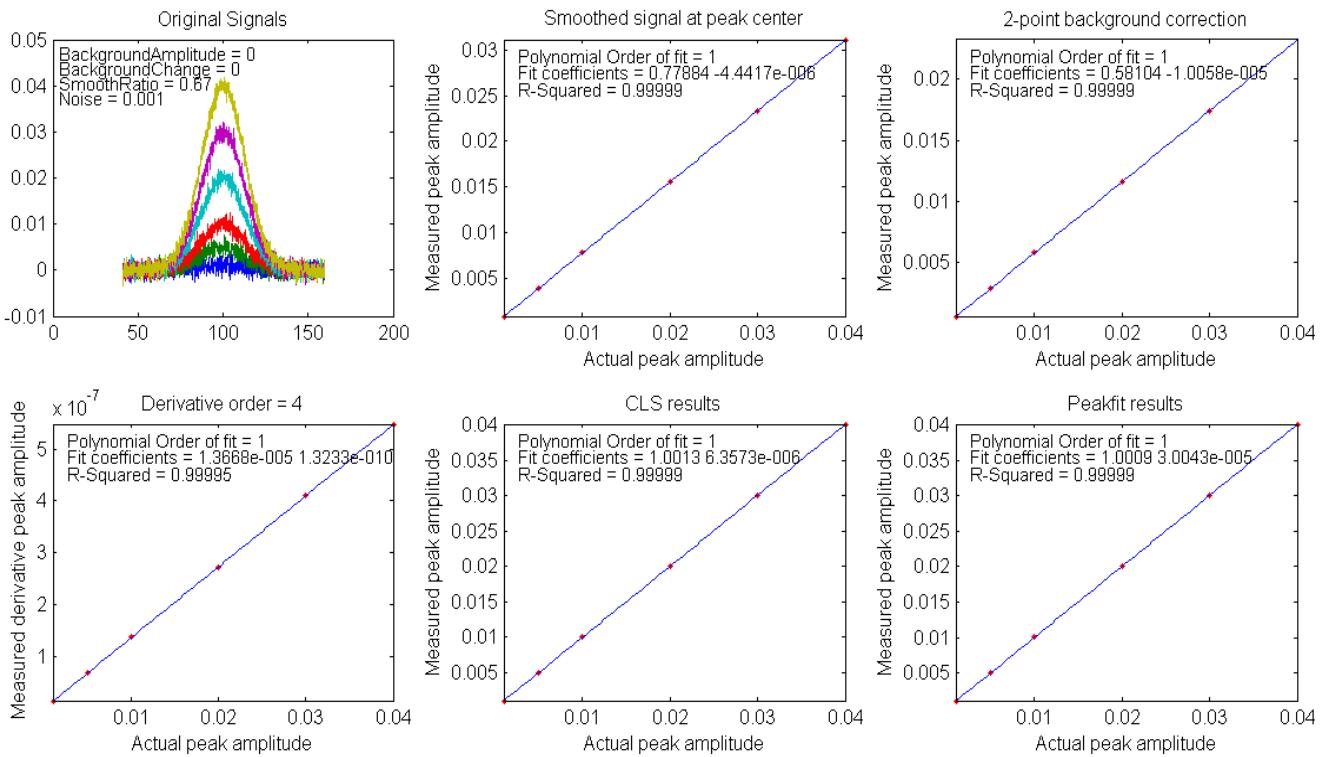
This case study demonstrates the application of several techniques described in this paper to the quantitative measurement of a peak that is buried in an unstable background, a situation that can occur in the quantitative analysis applications of various forms of spectroscopy, process monitoring, and remote sensing. The objective is to derive a measure of peak amplitude that varies linearly with the actual peak amplitude but that is not effected by the changes in the background and 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 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 35), differentiation (page 53), classical least squares multicomponent method (page 152), and iterative non-linear curve fitting (page 163).

[CaseStudyC.m](#) is a self-contained Matlab/Octave demo function that demonstrates this case. To run it, download it, place it in the path, and type “CaseStudyC” at the command prompt. Each time you run it, you’ll 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”.

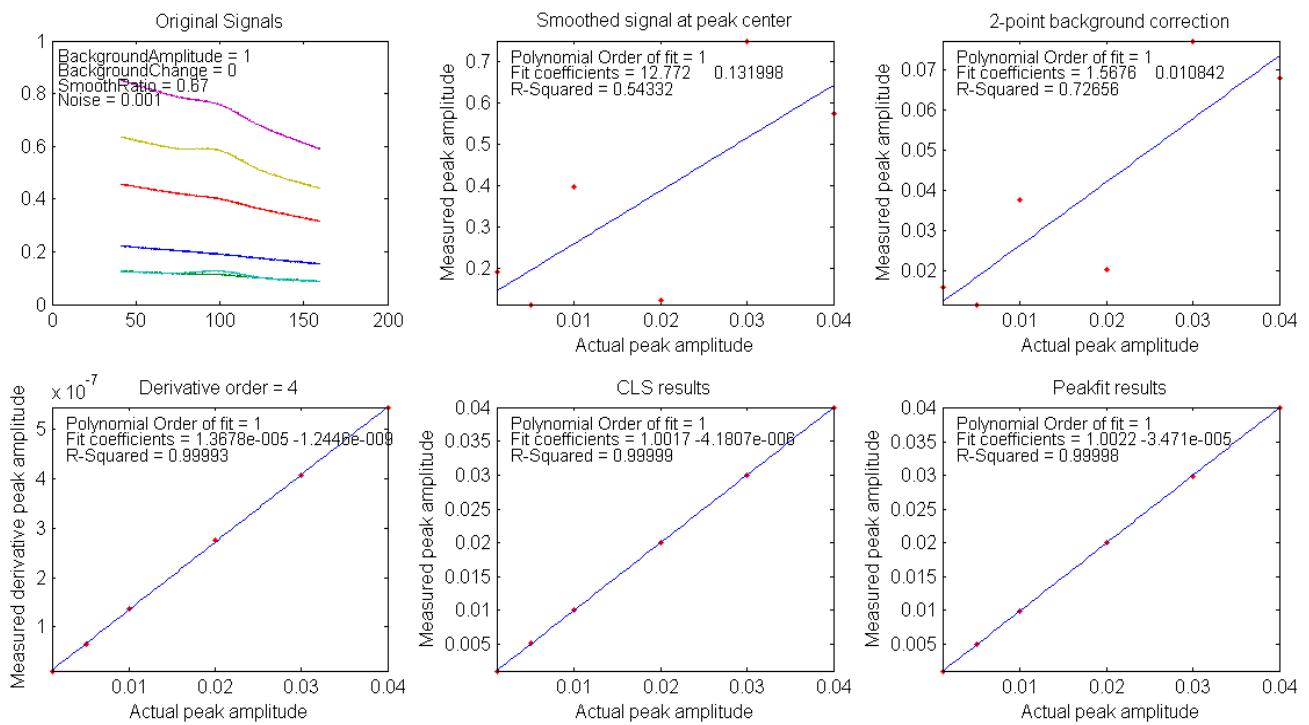
The five methods compared in the figures below are:

- 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), as long as the measured peak and the background *shapes* are independent of the concentration of the unknown.

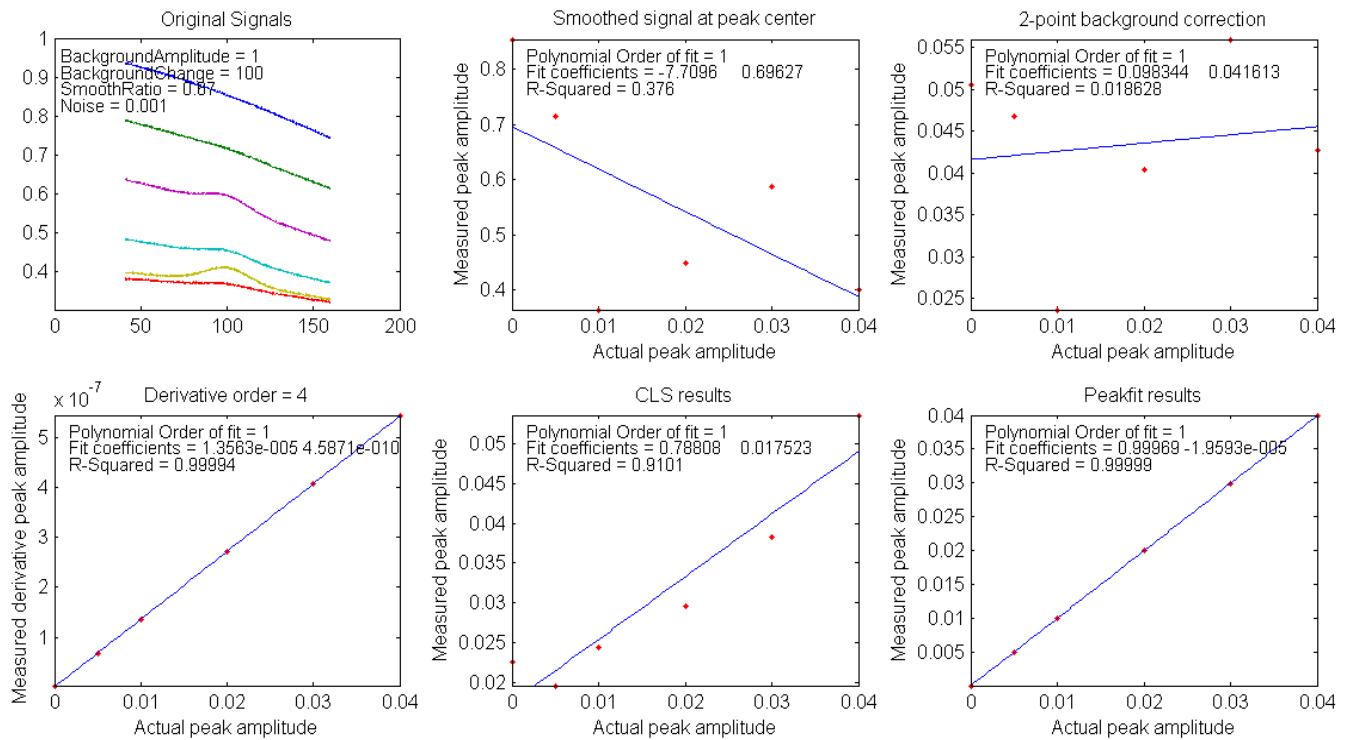
These five methods are listed roughly in the order of increasing mathematical and geometrical complexity. They are compared below by plotting the actual peak heights (set by the vector “SignalAmplitudes”) against the measure derived from that method, fitting those data to a straight line, and computing the *coefficient of determination*, R^2 , which is 1.0000 for a perfectly linear plot.



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 “Background-Change”=100, so the background varies in position, width, and amplitude (but remains broad compared to the signal). In that case, the CLS methods fails as well, because it assumes that the background varies only in amplitude. However, if we go one step further ([click to view](#)) and set “Background-Change”=1000, the background shape is now so unstable that even the INLS method fails, but still the derivative method 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, in the same way, for example, that a hand-held calculator hides the complexity of long division.

E: 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 a large number of times, and all the repeats are averaged. The signal builds up, and the noise gradually averages towards zero, as the number of repeats increases.

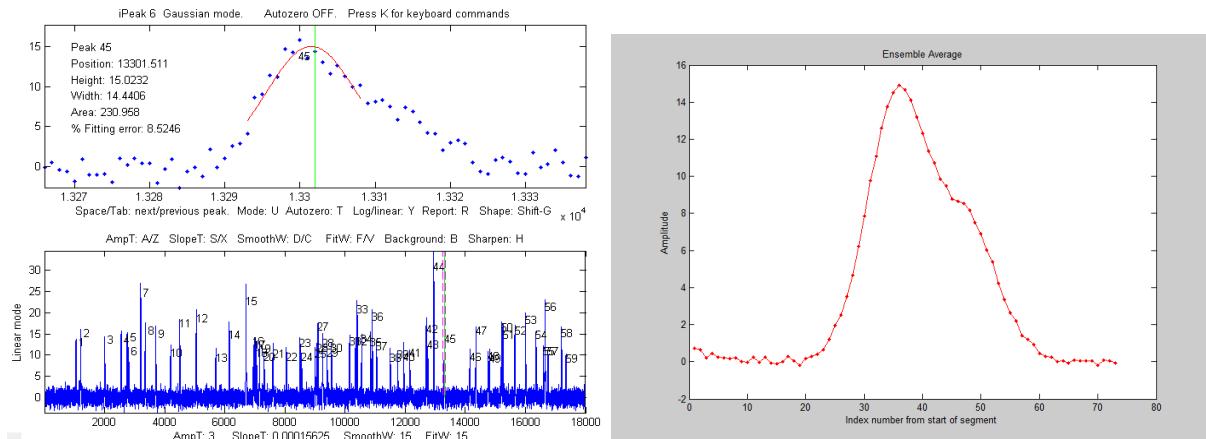
An important requirement is that the repeats be aligned or synchronized, so that in the absence of random noise, the repeated signals would line up exactly. There are two ways of managing this:

- (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 361) has a built-in ensemble averaging

function (Shift-E) can compute the average of all the repeating waveforms. It works by detecting a single peak in each repeat in order to synchronize the repeats.



The Matlab script [iPeakEnsembleAverageDemo.m](#) (on <http://tinyurl.com/cey8rwh>) demonstrates this idea, with a 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 361) 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;
```

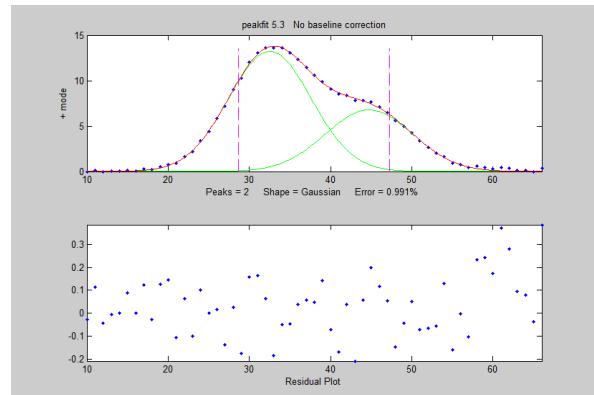
then curve-fit this averaged pattern to a 2-Gaussian model using the [peakfit.m function](#) (figure on the right):

```
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'll 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)
```

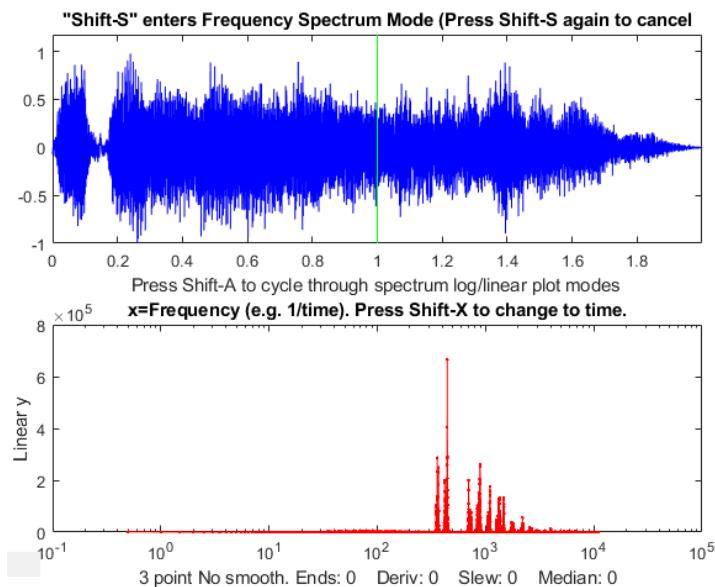


F: Harmonic Analysis of the Doppler Effect

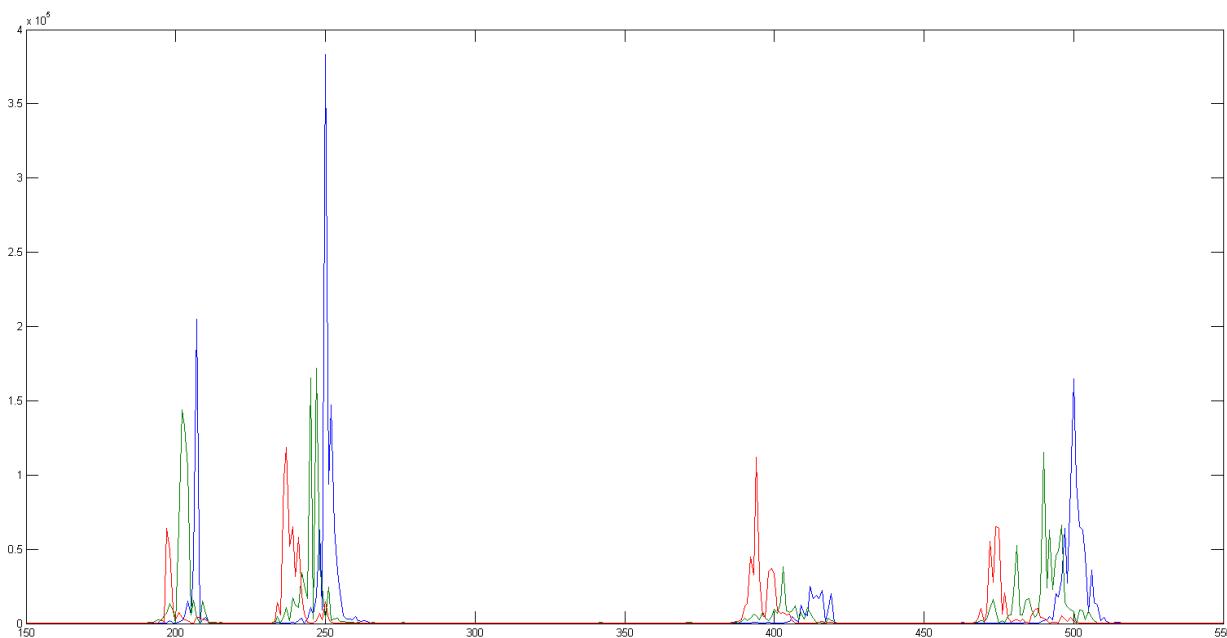
The wav file “[horngoby.wav](#)” (Ctrl-click to open) is a 2-second recording of the sound of a passing automobile horn, exhibiting the familiar [Doppler effect](#). The sampling rate is 22000 Hz. Download this file and place it in your Matlab path. You can then load this into the Matlab workspace as the variable “doppler” and display it using [iSignal](#) (page 323):

```
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's much easier to *hear* the frequency shift - press Shift-P to play the sound - than to *see* it graphically; the shift is rather small on a percentage basis, but [human hearing is very sensitive to small pitch \(frequency\) changes](#)). It helps to 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* (plotted 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 are the [fundamental frequency](#) of the lowest note of the horn and the

group of peaks near 400 are its [second harmonic](#). (Pitched sounds have a harmonic structure of 1, 2, 3... times a fundamental frequency). The group of peaks near 250 are the fundamental frequency of the next higher note of the horn and the group of peaks near 500 are 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 varies 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 361):

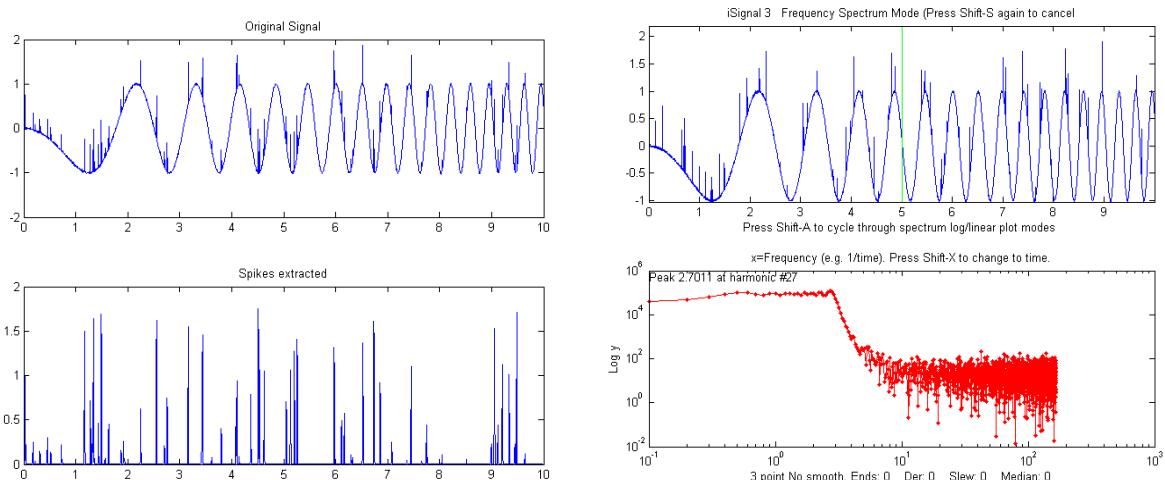
Peak	Position	Height	Width	Area
Beginning	206.69	3.0191e+005	0.81866	2.4636e+005
Middle	202.65	1.5481e+005	2.911	4.797e+005
End	197.42	81906	1.3785	1.1994e+005

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](#) and to demonstrate that the ratio of the second harmonic to the fundamental for these data is 2.0023, which is very close to the theoretical value of 2.

G: 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 recently encountered in a research application, and it opens up some interesting twists on the usual procedures.

As a demonstration, the Matlab/Octave script [SpikeDemo1.m](#) creates a waveform (top panel of 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 361) to the raw signal does not work well.



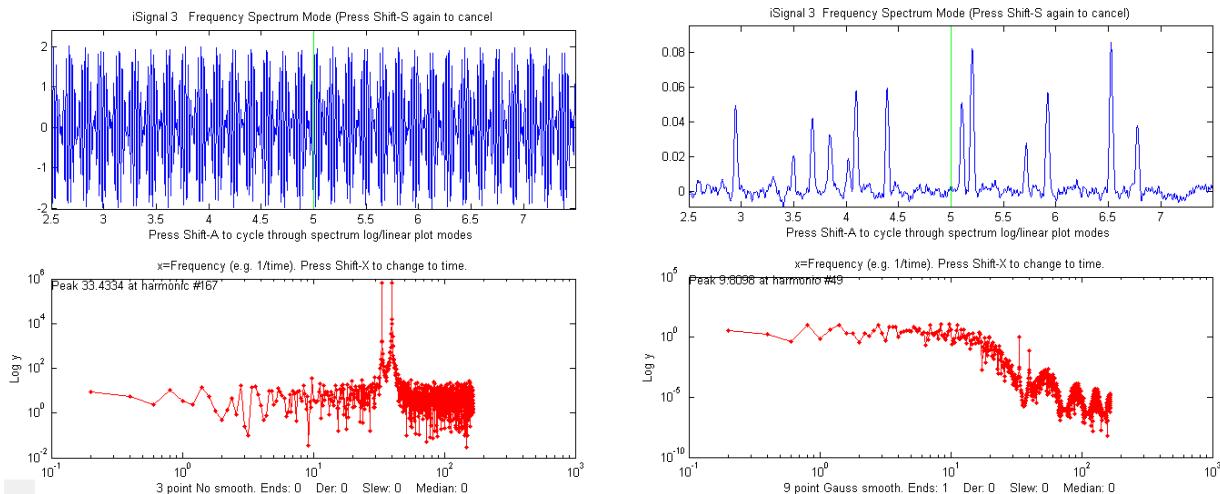
A single-point spike, called a *delta function* in mathematics, has a power spectrum that is *flat*; that is, it has *equal power at all frequencies*, just like white noise. But the oscillatory interference in this case is located in a *specific range of frequencies*, which opens some interesting possibilities. 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 361):

```
P=ipeak([x;y1],0,0.1,2e-005,1,3,3,0.2,0);
```

The second example, [SpikeDemo2.m](#), is similar except that in this case the oscillatory interference is caused by *two* fixed-frequency sine waves at a higher frequency, which completely obscure 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 up 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 cases are 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 pseudo-Gaussian 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 is *reduced by about a factor of about 1,000,000!* 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 substantially in frequency across the signal, you could use a [segmented smooth](#) rather than the standard fastsmooth). The extracted peaks can then be counted with any of the [peak finding functions](#), such as:

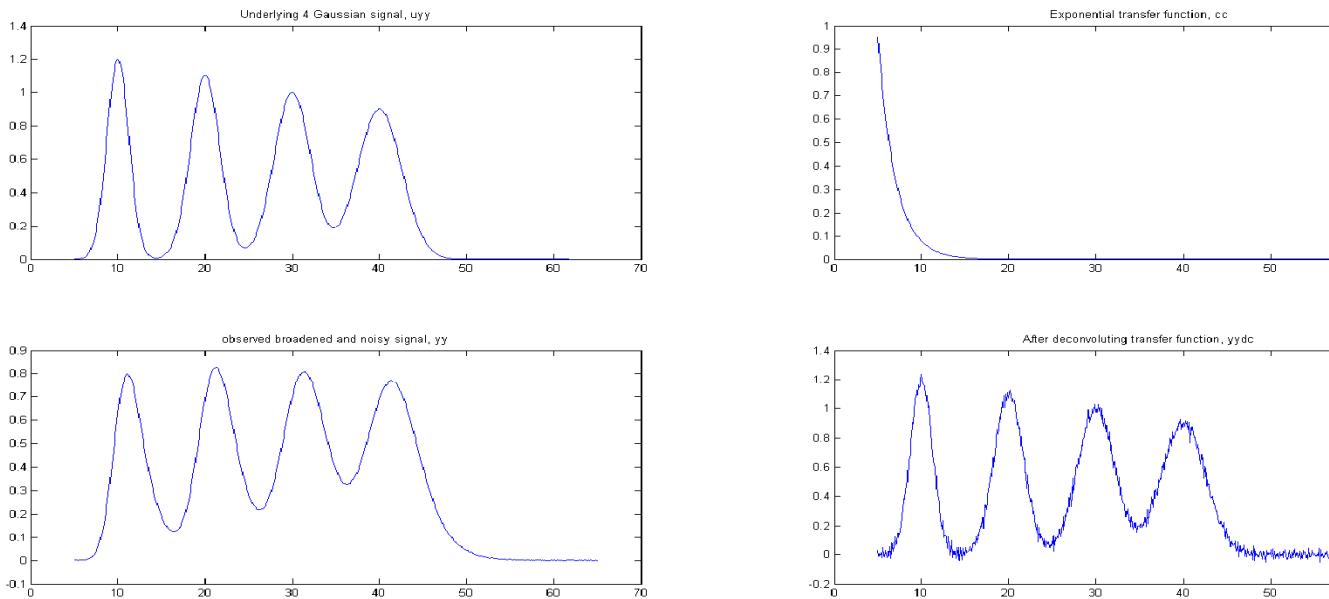
```
P=findpeaksG(x,y1,2e-005,0.01,2,5,3);
or
P=findpeaksplot(x,y1,2e-005,0.01,2,5,3);
or
```

```
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 323) by applying them to a [delta function](#). Change the smooth type, smooth width, and derivative order and other functions in order to see how the power spectrum changes.

H: 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; whereas in iterative least-squares curve fitting it's 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 on the right ([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 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=.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 actually works pretty 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 time constant here

% Attempt to recover original signal uyy by deconvoluting cc from yy
% It's 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')

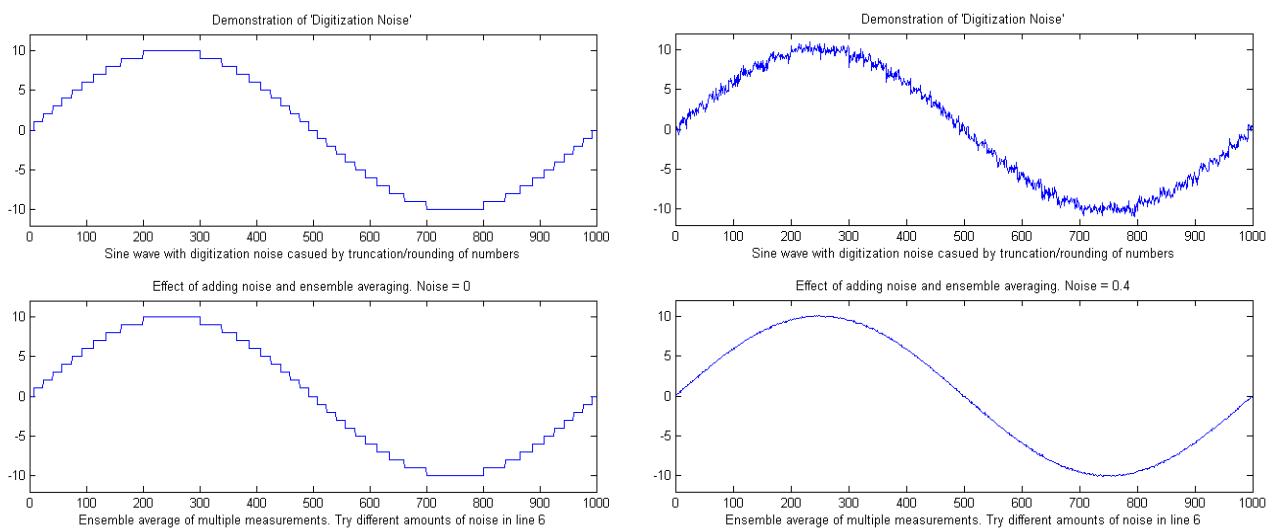
```

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 a 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 361), adjusting the exponential factor interactively to get the best fit. Note that *you have to 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 *independently variable time constant* exponentially-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.

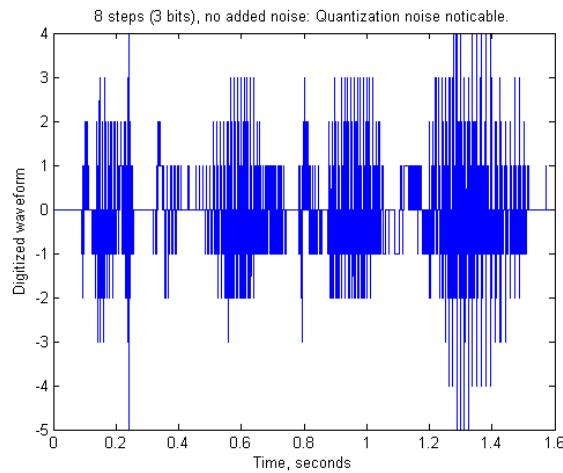
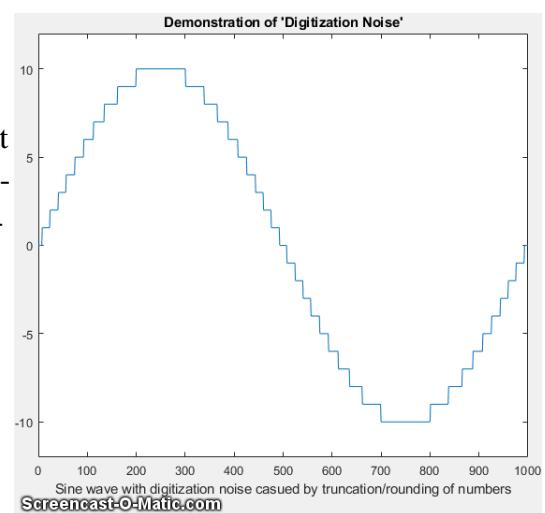
I: Digitization noise - can adding noise really help?

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 there is insufficient random noise already in the signal, it can be beneficial to add additional noise artificially! The script [Rounding-Error.m](#) illustrates this effect, as shown the [animation](#) on the right, 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. This technique is called "[dithering](#)" and it is also used in audio and in image processing.



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 left, 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's likely that the neurons of the hearing center of your brain provide that function by virtue of their response time and memory effect.

J: How Low can you Go? Performance with very low signal-to-noise ratios.

This is a simulation of several techniques described in this paper applied to the quantitative measurement of a peak that is *buried in excess of random noise*, where the signal-to-noise (S/N) ratio is *below* 2. (Ordinarily, a S/N ratio of 3 is desired for reliable detection).

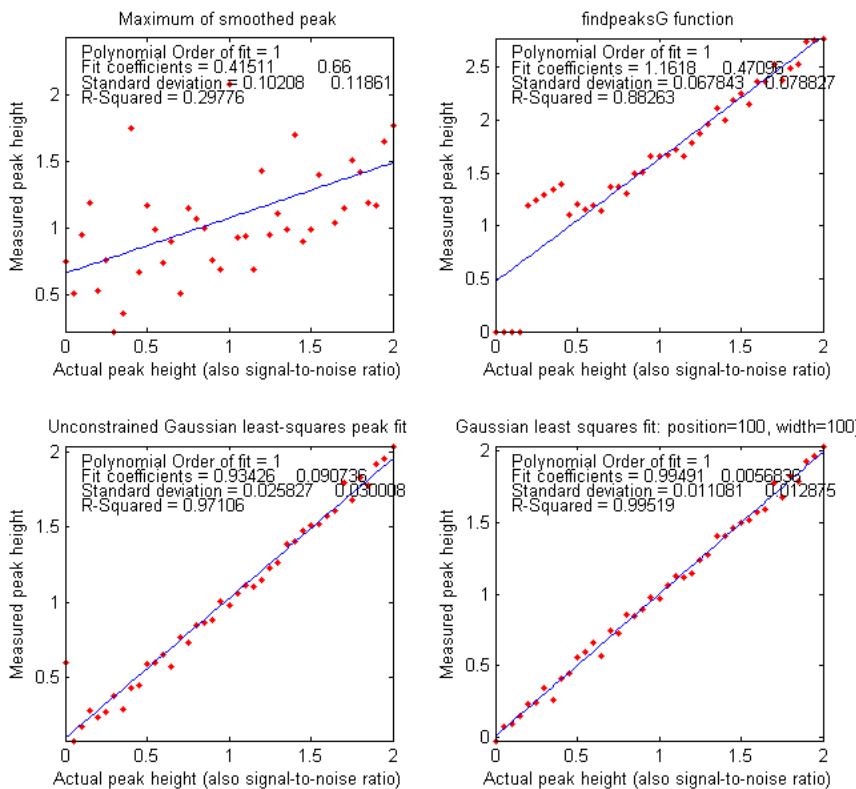
The Matlab/Octave script [LowSNRdemo.m](#) performs the simulations and calculations and compares the results graphically, focusing on the behavior of each method as the S/N ratio approaches zero. Four methods are compared:

- (1) smoothing, followed by the peak-to-peak measure of the smoothed signal (page 35);
- (2) a peak finding method based on [findpeakG](#) (page 200);
- (3) unconstrained iterative least-squares fitting (INLS) based on [peakfit.m](#) (page 163);
- (4) constrained classical least squares fitting (CLS) based on the [cls2.m](#) function (page 152);

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 printed below, 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 [findpeaksG](#) function (upper right) works OK for height for higher peak heights but fails completely below a 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 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's 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 edge in accuracy, but you have to 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.

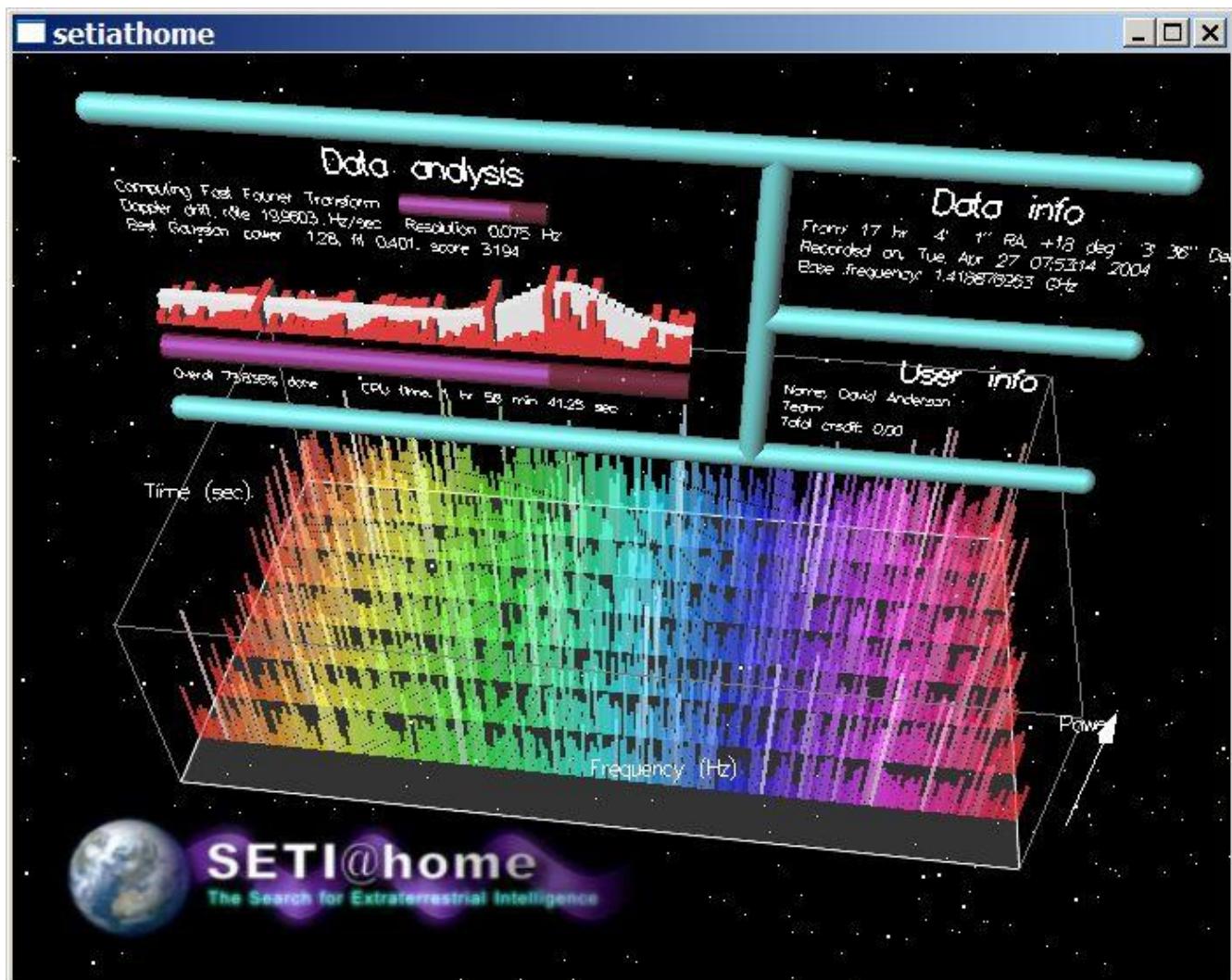
(Surprisingly, as we will see on page 298, [Measurement Calibration](#), *it is not even necessary to have an accurate peak shape model* in order to get a good correlation between measured and actual height).

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 similar to 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 that previously.

K: 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 don't know much about what to expect. In particular, we don't know exactly where to look in the sky, or what frequencies might be used, or the possible forms of the transmissions. Moreover, astronomers don't 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 has driven the development of specialized hardware and software as well as distributed computation over thousands of Internet-connected personal computers across the world using the [SETI@home](#) computational screen-saver. Although some of the [computational techniques](#) used in this search are far 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 don't 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 radiations from vast distances. In order to look at different frequencies at once, [*Fourier transforms*](#) of the raw telescope signals can be computed over multiple time segments. We previously saw a [*simulation*](#) that showed 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 equally-spaced pulses, so their detection and verification is 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 is fixed in position and is 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*](#):

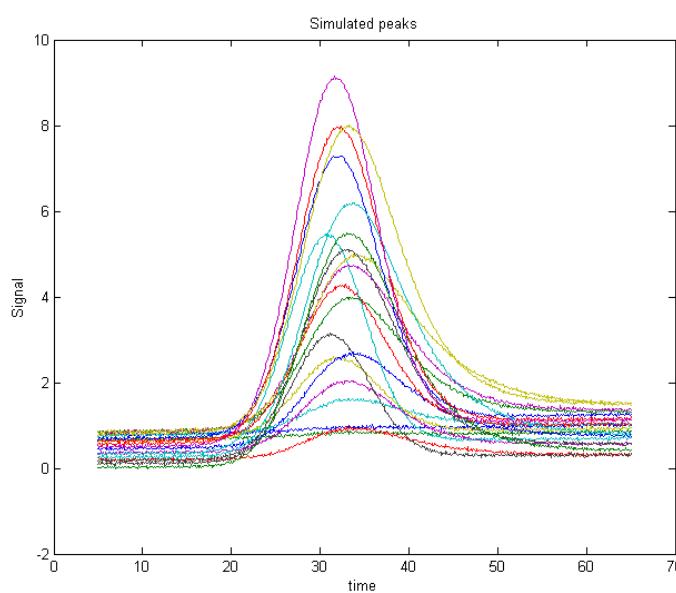
“Radio signals from a distant transmitter should get stronger and then weaker as the telescope's focal point moves across that area of the sky. Specifically, the power should increase and then decrease with a bell shaped curve ([*a Gaussian curve*](#)). [*Gaussian curve-fitting*](#) is an excellent test 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.

We also know 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).

For more on the details of SETI signal processing, see [*SETI@home*](#).

L: 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 the measurement of peak area might 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 area has different units and is numerically greater than peak height. All the measurement methods are baseline corrected; that is, 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 due to the effect of the variability of peak shape broadening ('vbr') and to the averaging out of random noise in the computation of area.

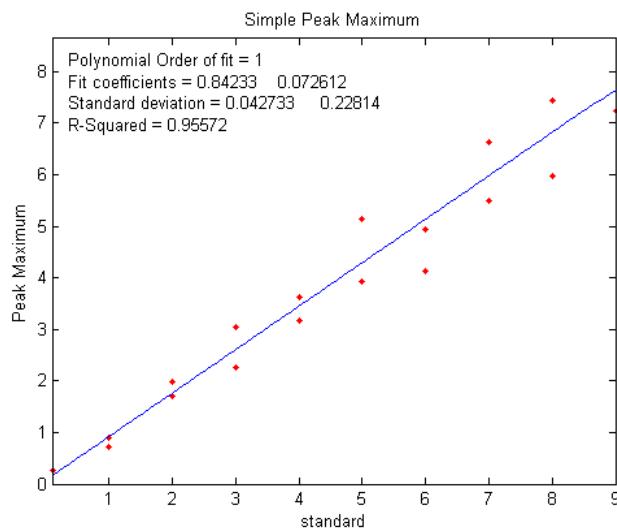


Figure window 2

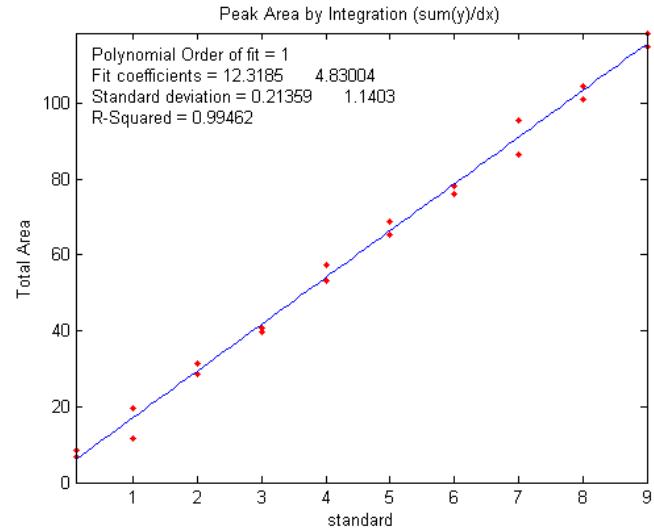


Figure window 3

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](#) than other peak area measurement methods.

N: Using macros to extend the capability of spreadsheets

Both Microsoft *Excel* and OpenOffice *Calc* have the ability to automate repetitive tasks using “macros”, 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.

To enable macros in Excel, click on File, Options, click Customize Ribbon Tab and check 'Developer' and click 'OK'. To access the macro recorder, click Developer, Record Macro, give the macro a name, click Options, assign a Ctrl-key shortcut, 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 164) 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.xls](#) 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 'GRG Nonlinear', the best one for iterative peak fitting. Finally, "SolverSolve"

starts the Solver engine. 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, 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 [TransmissionFittingCalibration-Curve.xls](#) ([screen image](#)) that creates a calibration curve for a series of standard concentrations in the TFit method, which was previously described on [page 242](#). 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
```

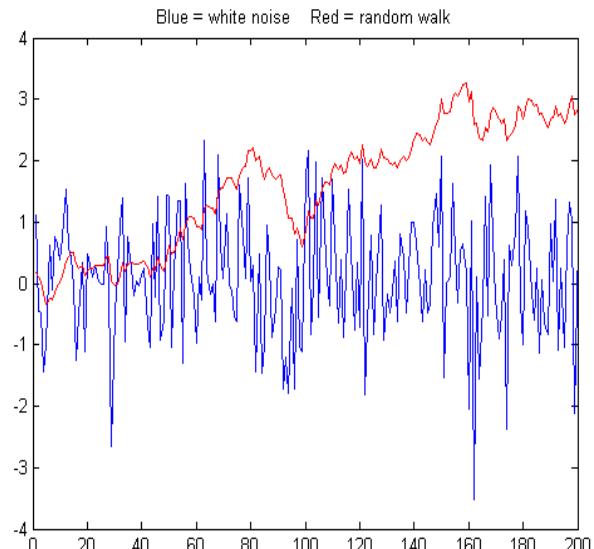
The macro in this spreadsheet repeats this chunk of code several times, once for each concentration in the calibration curve (changing only the "AF10" in the first line to pick up a different concentration from the "Results table" in column AF). This macro uses several additional instructions, to select 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 separate click, menu selection, or key press 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.

O: Random walks and baseline correction

The *random walk* was mentioned in the section on [signals and noise](#) (page 22) 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 search path 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 anchored to a mean value to which they tend to return, random walks tend to be more aimless and often drift off on one or another direction, possibly never to return. Mathematically, a random walk can be modeled as the cumulative sum of some random process, for example the 'randn' function. The graph on the right compares a 200-point sample of white noise (computed as 'randn' and shown in blue) to a random walk (computed as a cumulative sum, 'cumsum', and shown in red). *Both samples are scaled to have exactly the same standard deviation*, but their behavior is vastly different. The random walk has much more low frequency behavior, in this case wandering 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, and it cannot be reduced significantly by smoothing (See [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's not; it's just noise). But another sample might have very *different* behavior. Unfortunately, it is common to observe this behavior in experimental signals.

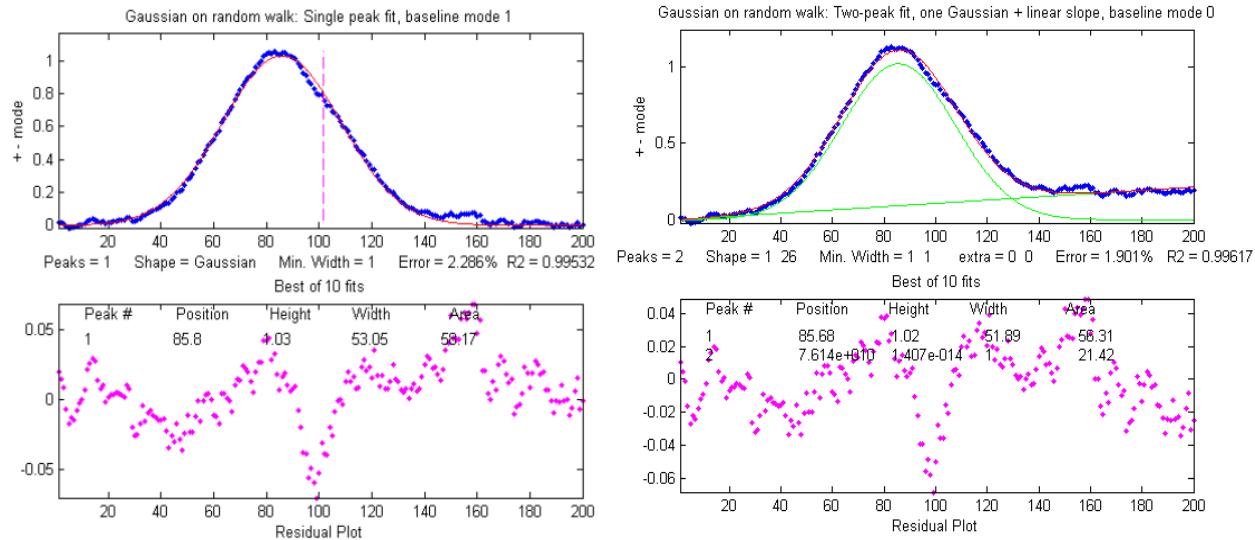


To demonstrate the measurement difficulties, the script [RandomWalkBaseline.m](#) simulates a Gaussian peak with randomly variable position and width, on a random walk baseline, with a S/N ratio is 15. The peak is measured by least-squares curve fitting methods using [peakfit.m](#) with two different methods of baseline correction in an attempt to handle the random walk:

- (a) a single-component Gaussian model (shape 1) with autozero 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);`

(b) a 2-component model, the first being a Gaussian (shape 1) and the second a linear slope (shape 26), with autozero set to 1: `peakfit ([x;y], 0, 0, 2, [1 26], [0 0], 10, 0)`.

In this particular 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.

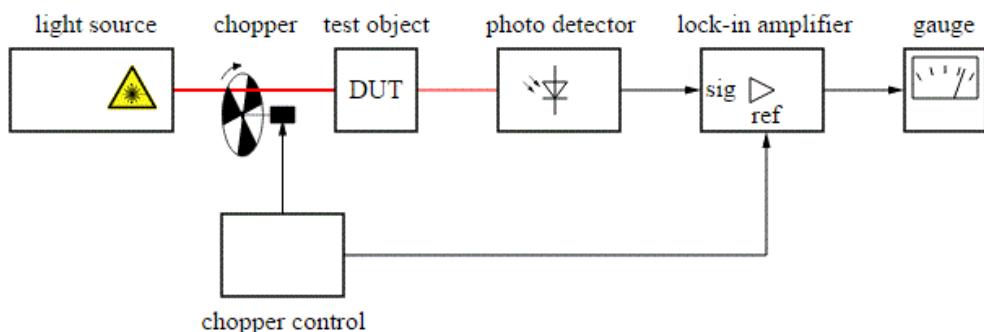
	Position Error	Height Error	Width Error
Method a:	0.2772	3.0306	0.0125
Method b:	0.4938	2.3085	1.5418

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's important to realize that fitting error does not *always* correlate with peak parameter error. Bottom line: 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.

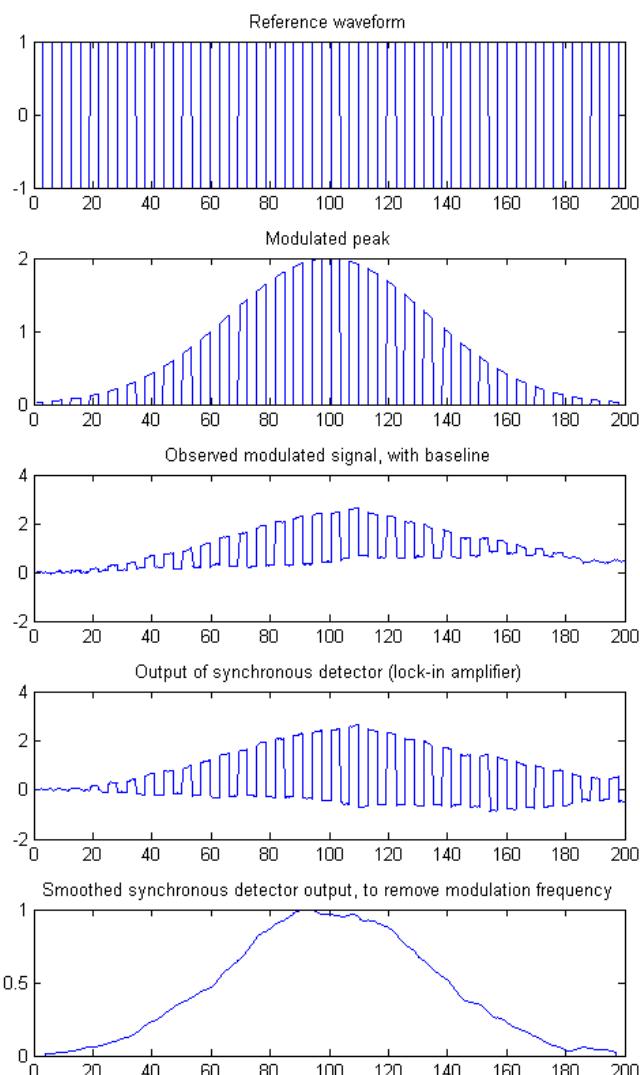
P. Modulation and synchronous detection.

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 to detect the resulting oscillation in the measured signal. The right instrumental design can reduce or eliminate some types of noise and drift.

A simple example applies to optical measurement systems like the one pictured on the right above. A



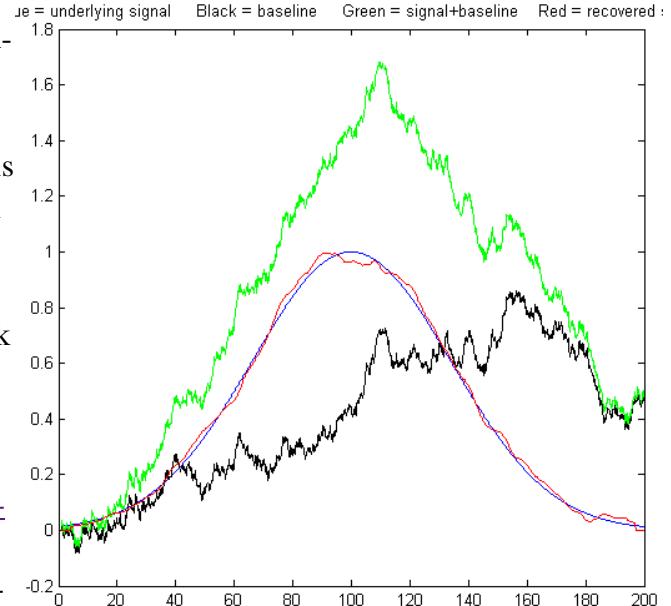
light source illuminates a test object (DUT="Device Under Test") and the resulting light from the test object is measured by photo detector. Depending on the objective of the experiment and the arrangement of the parts, the detector might measure the light *transmitted by, reflected by, scattered by, or excited by* the light beam. An optical chopper rapidly and repeatedly interrupts the light beam falling on the test object, so that the photo detector sees an oscillating signal, and the following electronic system is designed to measure *only* the oscillating component and to ignore the constant (direct current) component. The advantage of this arrangement is that any interfering signals introduced *after* the chopper - such as constant light that comes from the test object itself, or ambient light that leaks in from the outside, or any constant background signal generated by the photo detector itself - *are not oscillating* and are thus rejected. This works best if the electronics is *synchronized* to the chopper frequency. That's actually the function of the lock-in amplifier, which receives a synchronizing reference signal directly from the chopper to guarantee synchronization even if the chopper frequency were to vary (c.f. the interactive simulation on <https://terpconnect.umd.edu/~toh/models/lockin.html> and T. C.

O'Haver, "Lock-in Amplifiers," *J. Chem. Ed.* 49, March and April (1972). The lock-in amplifier is

sometimes viewed as a "black box" with almost magical abilities, but in fact it is actually performing a rather simple (but very useful) operation, as shown in this simulation.

[AmplitudeModulation.m](#) is a Matlab/Octave script simulation of 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 (updated for recent version of Matlab). As the spectrum of the sample is scanned, the light beam is amplitude modulated by the 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. The modulation frequency is many times faster than the rate at which the sample 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 280), 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) are less intense.

These various signals are compared in the figure on the right. The original 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 was 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 line 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 [peak-fit.m](#) function, which employs a least-squares method to measure the peak parameters in the original unmod-



ulated total signal (green line) and in the modulated recovered signal (green) 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's 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 of the fact that 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 the a negative-goring peak rather than a positive peak (see [AmplitudeModulationAbsorption.m](#)).

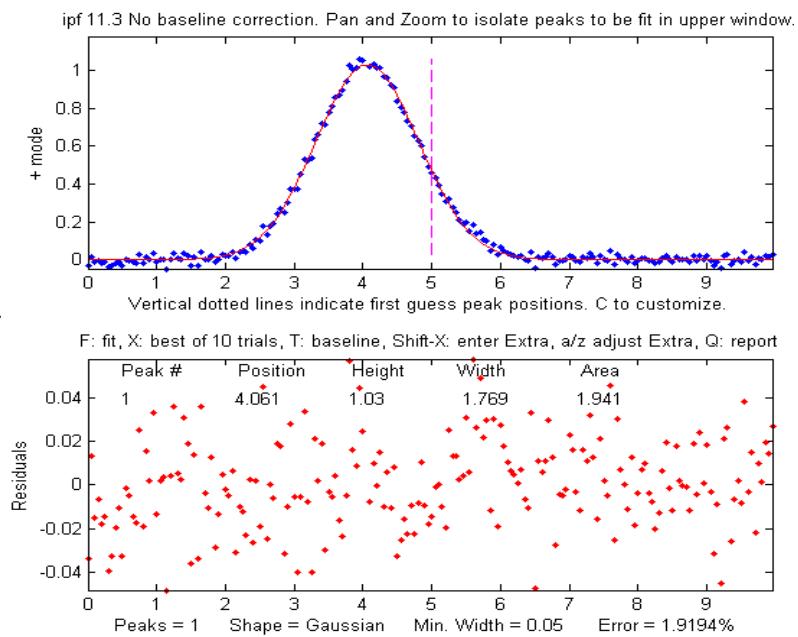
In a computer-interfaced experimental system, you may not actually need a physical lock-in amplifier. It's possible to simulate the effect in software, as is done in this simulation. You need only digitize both the modulated sample signal and modulation reference signal, and then invert the total signal 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 is oscillated over the wavelength region of an emission or absorption peak in the spectrum (reference 32); this is often used in tunable diode laser spectroscopy and 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](#).

Q: Measuring a buried peak

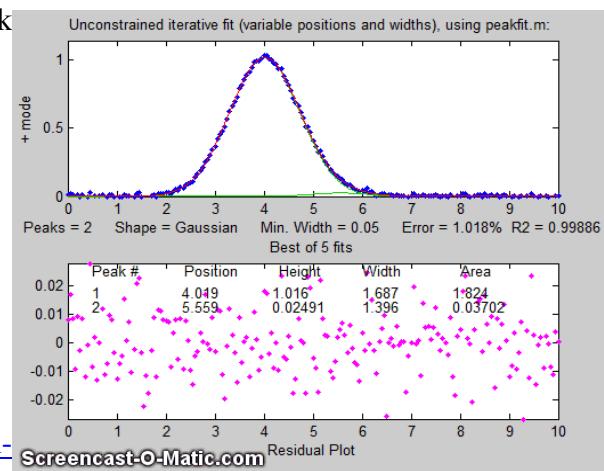
This simulation explores the problem of measuring the height of a small peak (a "child peak") that is buried in the tail of a much stronger overlapping peak (a "parent peak"), in the especially challenging case that the smaller peak *is not even visible* to the unaided eye. Three different measurement 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, we pretend that we don't necessarily know all of these facts and we will try to find methods that will extract such 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.

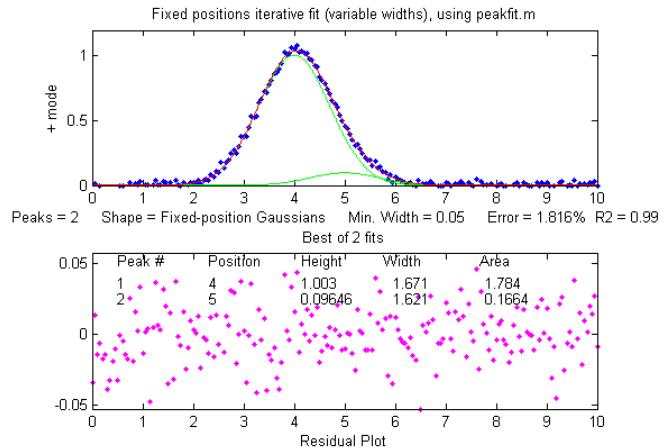
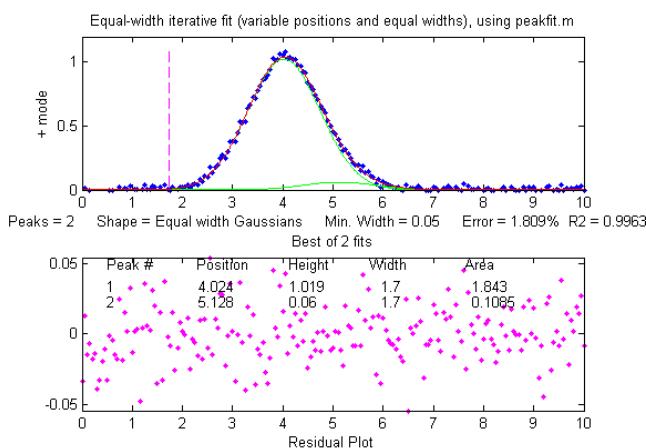


The selection of the best method will depend on what is known about the signal and the constraints that can be imposed; this will depend in your knowledge of your experimental signal. In this simulation (performed by the Matlab/Octave script [SmallPeak.m](#)), the signal is composed of two Gaussian peaks (although that can be changed if desired in line 26). The first question is: are there more than one peak there? If we 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 pretty random. (If you could reduce the noise, or [ensemble-average](#) 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 we suspect that there *should* be another peak of the same Gaussian shape just on the right side of the larger peak. We can try fitting a *pair* of Gaussians to the data (figure on the right), but in this case *the random noise is enough that the fit is not stable*. When you run [SmallPeak.m](#), the script performs 20 repeat fits (“Num-Signals” 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 ([click here to see GIF animation](#)). 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 a [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 Small-Peak.m the first time, spread out all the figure windows so they can all be seen separately and don’t overlap. That way you can compare the stability of the different methods more easily.)

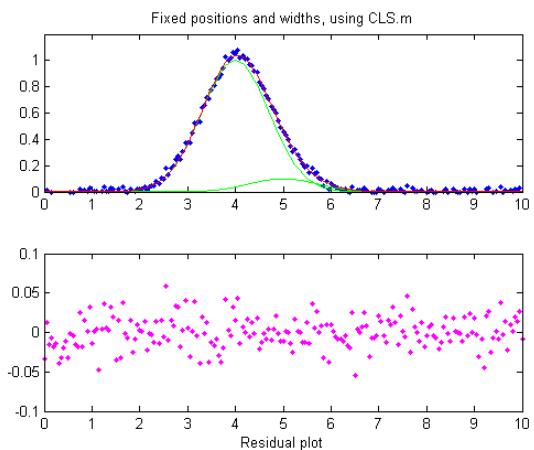


But suppose that we have reason to expect that the *two peaks will have the same width*, but we don't know what that width might be. We could try an *equal width* Gaussian fit (peak shape #6, shown in Matlab/Octave Figure window 2); the resulting fit is much more stable and shows that a small peak is located at about x=5 on the right of the bigger peak, shown below on the left. On the other hand, if we know the peak *positions* beforehand, but not the widths, we 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's 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 of 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's 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's 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's initially zero, but if you set it to something greater (e.g. 0.2) you'll 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 we know the *separation* between the two peaks, it's possible to use the findpeaksG function to search for and locate the larger peak and to calculate the position of the smaller one. 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 of these methods are computed* "NumSignals" times (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
```

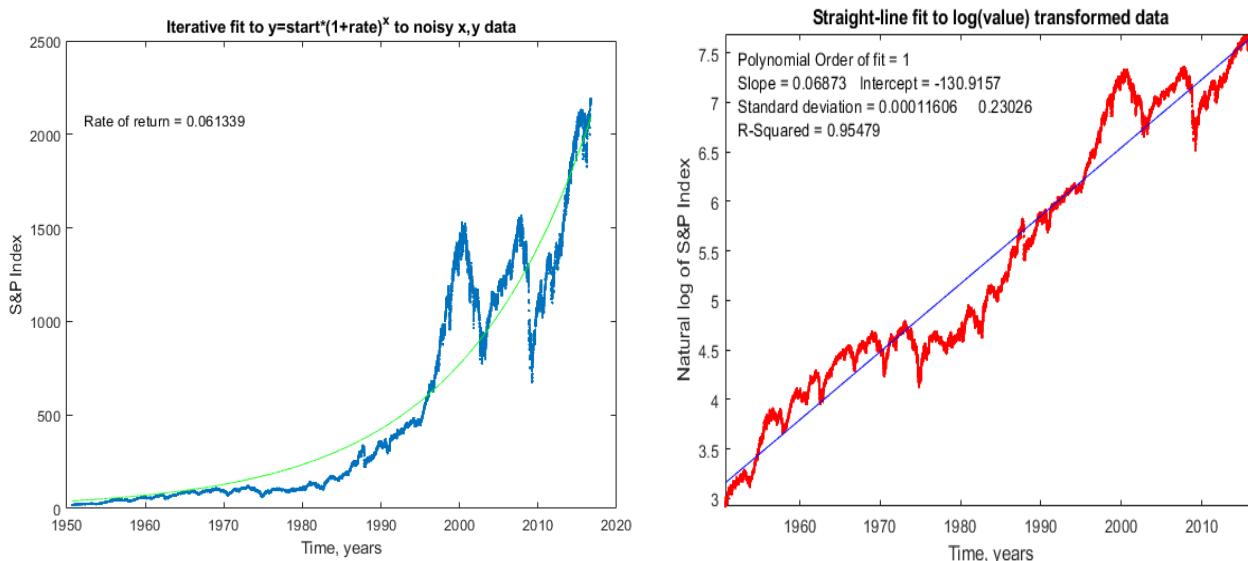
The bottom line is this: the more you know about your signals, the better you can measure them. A stable signal with *known* peak positions and widths is the most precisely measurable in the presence of random noise ("FixedP&W"), but if the positions or widths vary from measurement to measurement,

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.

R. Signal and Noise in the Stock Market

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 a large number of 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 on a daily basis, 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. Other stock indices, such as the Russel 2000, include an even larger number of smaller companies. 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 daily value, V, of the S&P 500 index vs time, T, from 1950 through September of 2016 is shown in the following graphs.



Each plot contains 16608 data points, one for each business day, shown in red. The graph on the left plots V and the graph on the right plots the *natural logarithm* of V, $\ln(V)$. There are considerable up-and-down fluctuations over time that can be related to historical events: the "stagflation" of the 1970s, the tech boom and bust of 2000, the subprime mortgage crisis of 2008. Still, the *long-term* trend of the value is upwards - the current value is over 100 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 usually go up. The most common way to model this overall long-term increase over time is based on the [equation](#)

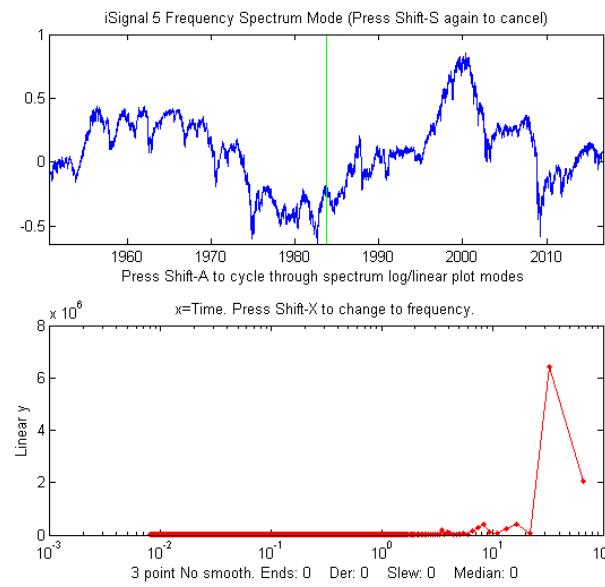
[for compound interest](#) that predicts the growth of investments that have a constant rate of return, such as savings accounts or certificates of deposit:

$$V = S^*(1 + R)^T$$

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 curve, without all the peaks and dips. The values of S and R that result in the *best fit* to the stock market data (shown by the blue lines in the graphs) can be determined 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 those to a straight line, shown on the right above.

[FitSandP.m](#) is a Matlab/Octave script that performs both of these calculations using the data in [SandPfrom1950.mat](#). When applied to the S&P 500 index data, the rate of return R is about 0.07 (or 7%), but interestingly these two methods give slightly *different results*, even though the *exact same data* are used for both, and even though *both* methods yield the *same* 7% rate if applied to noiseless *synthetic data* calculated from this expression. This difference between methods 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 the value data V over time and by the fact that the average return from 1950 to 1983 is slightly lower than that from 1983 to 2016.

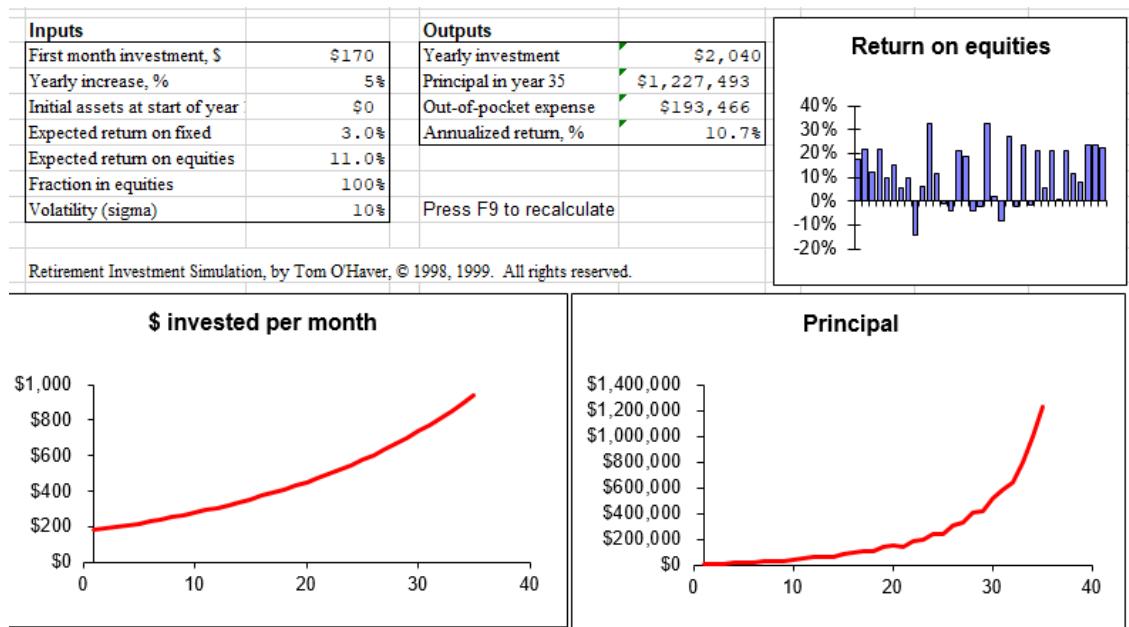


From the point of view of curve fitting, the deviations from a smooth curve described by the compound interest expression is just *noise*. But from the point of view of the stock market investor, those deviations can be an opportunity and a warning. Naturally, most investors would like to know how the 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's *most likely* (but not certain) that the *long term* behavior of the market (say, over a period of 10 years or more) will be similar to 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.

We can take a closer look at those fluctuations by inspecting the *residuals* – that is, subtracting the fitted curve from the raw data, as shown in [iSignal](#) (page 323) on the left. 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, the noise has a distinctly *low-frequency* character (page 26); the periodogram (lower panel, in red) shows peaks at 33, 16, 8, and 4 years. There are also, notably, 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 be sure of is that, in the long run, the market will rise. This is why saving for retirement by investing in the stock market, and *starting as soon as possible*, is so important: over a 30-year working life, the market is almost 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 actually 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 in an attempt 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.

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's 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; the *total returns* with dividends reinvested (https://en.wikipedia.org/wiki/S%26P_500_Index#Versions) would have been *substantially higher*, closer to 11%. (With an average total annual return of 11%, and starting with an investment of \$170 the first month - that's less than \$6 a day - and increasing it 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, as shown by the [spreadsheet](#) graphic above).

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 26) 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 = 0.07
                    Measured Rate   RSD
Coordinate transformation: 0.07112      8.9%
Iterative curve fitting:    0.07972     19.9%
```

As you can see, the two methods don't agree. In this example, the return calculated by the iterative method is higher, but it *could just have easily been the other way*. The fact is that the standard deviations are fairly large, and the iterative method always has a higher *standard deviation*, because it weights 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 saving accounts or CDs, which have much lower rates of return.

In investing in the stock market, it's 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. It's similar to the difference between [weather](#) and [climate](#); the large and dramatic short-term *weather* variations 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 the weather, but the climate changes so slowly that it's easy to conclude that it's fixed.

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>.

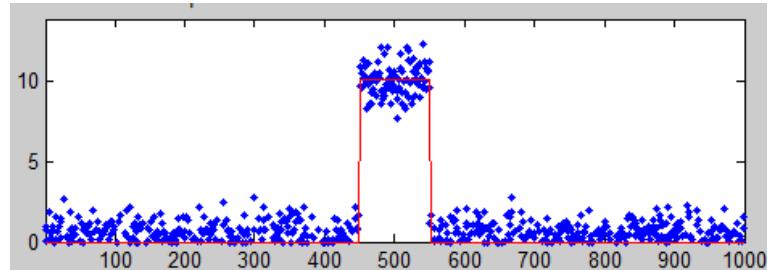
S. Measuring signal-to-noise ratio in complex signals

In the section Signals and Noise on page 22, 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’s a simple enough statement, but automating the measurement of signal and the noise in real signals is not always straightforward. Sometimes it’s 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.

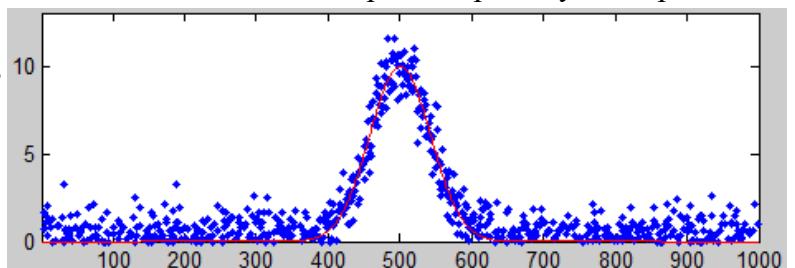
For a simple DC (direct current) signal, for example measuring a fluctuating voltage, the signal is just the average voltage value and the noise is its standard deviation. This is easily calculated in a spreadsheet or in Matlab/Octave:

```
>> 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 m_1 and m_2 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 $\text{sqrt}((\text{std}(m_1-m_2)^2)/2)$, 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’s not so hard to do by hand, but it’s harder to automate with a computer, especially if the position or width of the pulse changes. It’s 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 choose too high or two low a smooth width. And clearly all this depends on having a well-defined baseline in the data where there is only noise. It doesn’t 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 difference between the raw data and the best-fit line

over the top part of the peak. That's how [iSignal](#) (page 323) measures S/N ratios of peaks. Also, iSignal has baseline correction capabilities that allow the peak to be measured relative to the nearby 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 example shown on the right. In this example, five Gaussians are used to fit the data to the point where the residuals are random and unstructured. The residuals (shown in blue 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 fit with a polynomial or a collection of basic peak shapes and that the noise is random and mostly high-frequency compared the signal. But don't use too high a polynomial order' otherwise you are just "fitting the noise".

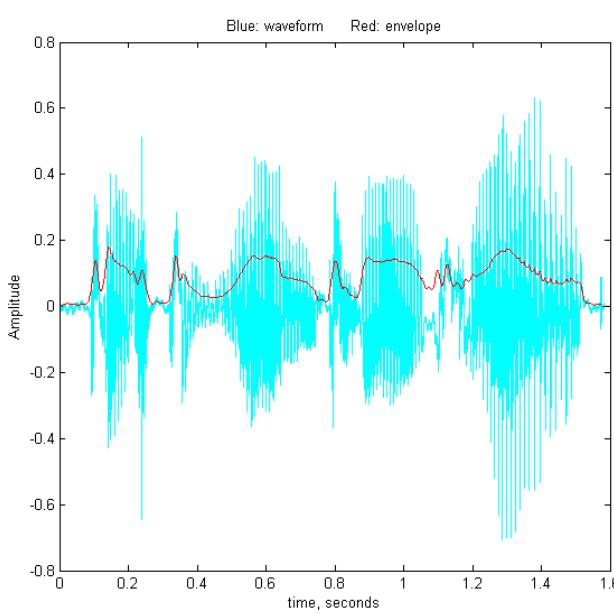
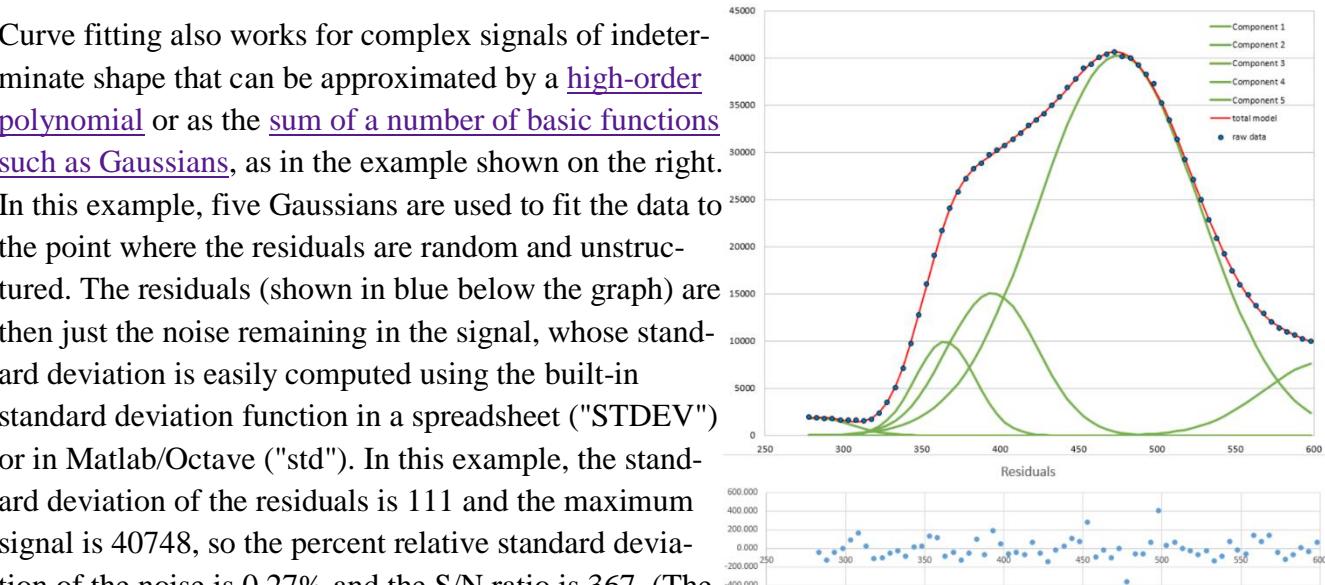
With periodic signal waveforms the situation is a bit more complicated. As an example, consider the

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 a variable "waveform", then computes the average amplitude of the waveform (the "envelope") by smoothing (page 35) the absolute value of the waveform:

```
envelope = fastsmooth(abs(waveform),  
SmoothWidth, SmoothType);
```

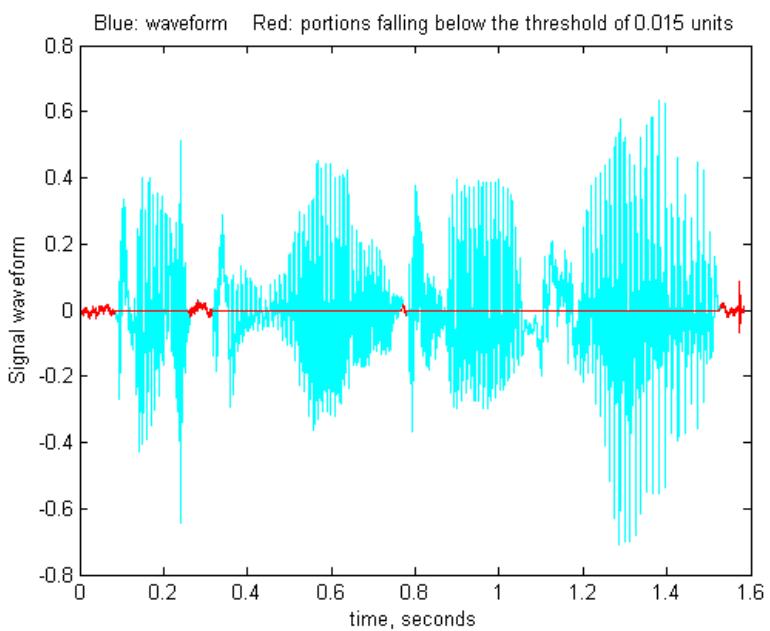
The result is plotted on the left, where the waveform is in blue and the envelope is in red. The signal is easy to measure as the maximum or perhaps the average of the waveform, but the noise is not so

evident. The human voice is not reproducible enough to get a second identical recording to subtract out the signal as above. Still, there will be often be gaps in the sound, during which the background noise



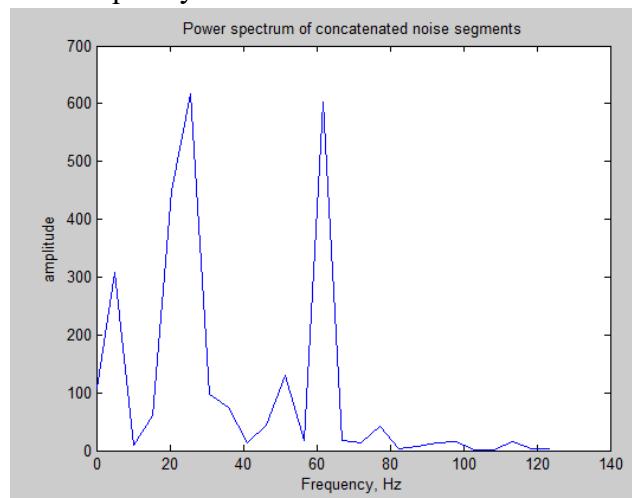
will be dominant. In an audio (voice or music) recording, there will typically be such gaps at the beginning, then the recording process has already started but the sound has not yet begun, and possibly at other short periods when there are pauses in the sound. The idea is that, by monitoring the envelope of the sound and noting when it falls below some adjustable threshold value, we 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 has to 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 are 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 323) 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 60Hz); this suggests that better shielding and grounding of the electronics might help to clean up future recordings. The lack of strong components at 100 Hz and above suggests that the vocal sounds have been effectively suppressed at this 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.

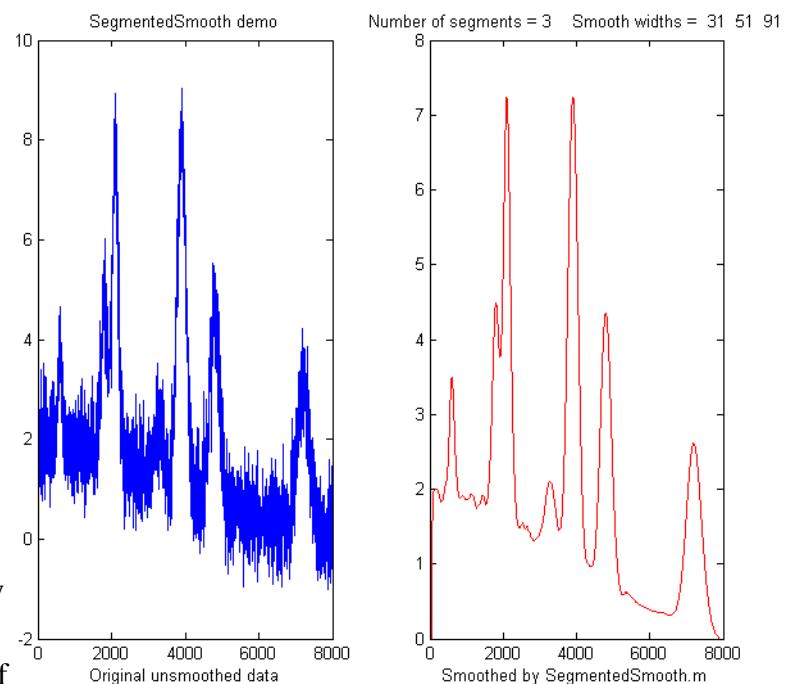
T. Dealing with wide ranging signals: segmented processing

Sometimes an experimental signal will vary so much across its x-axis range that it's impossible to find a single setting for operations like smoothing or peak detection that is optimized for all regions of the signal. You could break up the signal into pieces and treat each separately, but that could get messy. It would be easier to handle that problem with a single 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, 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*:

```
SmoothY = SegmentedSmooth(Y, smoothwidths, type, ends)
```

The function divides Y into a number of 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*. Type "help SegmentedSmooth" for other examples 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.

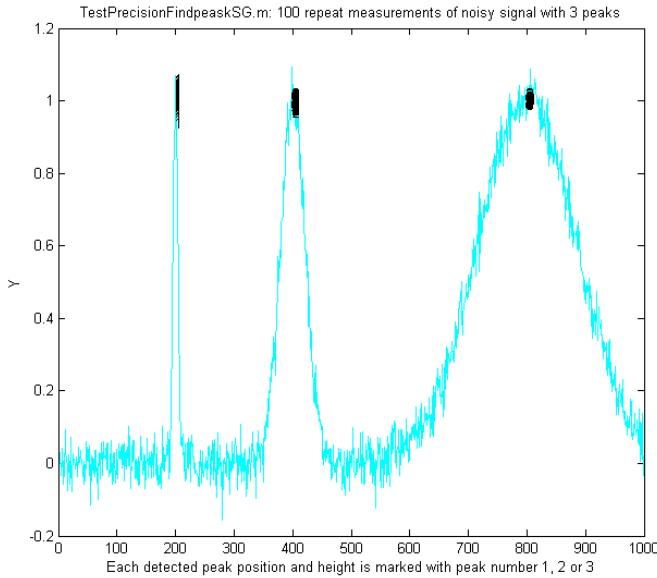


[SegmentedSmoothTemplate.xlsx](#) is a segmented multiple-width data smoothing *spreadsheet* template, functionally similar to SegmentedSmooth.m. In this version there are 20 segments.

[SegmentedSmoothExample.xlsx](#) is an example 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'll have to modify these templates for your particular 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).

[findpeaksSG.m](#) is a variant of the [findpeaksG function](#), with the same syntax, except that the four peak

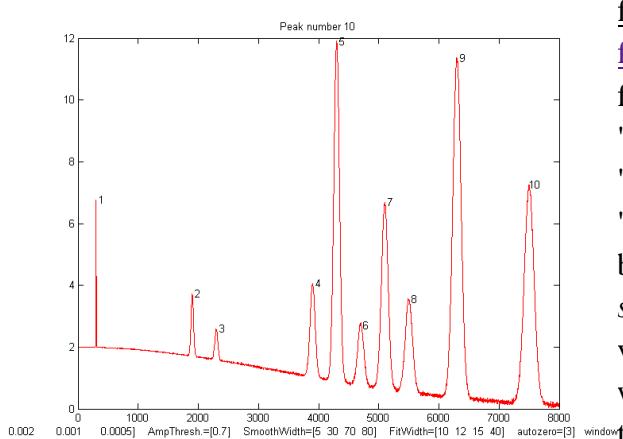
detection parameters can be *vectors*, dividing up the signal into regions that are optimized for peaks of different widths. Any number of segments can be declared, based on the length of the



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).



[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", "autozero", 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

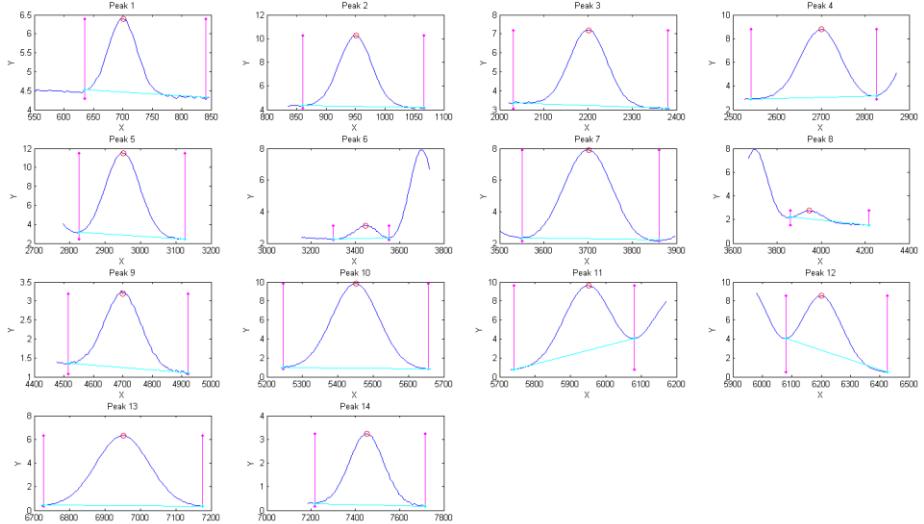
factor of 25 and that are superimposed in a curved baseline with random white noise that increases gradually across the signal. In this example, *four segments* are used, changing the peak detection and curve fitting values so that all the peaks are measured accurately.

```
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;
autozero = 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 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\(x,y,tc\)](#) deconvolves 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.

U. Measurement Calibration

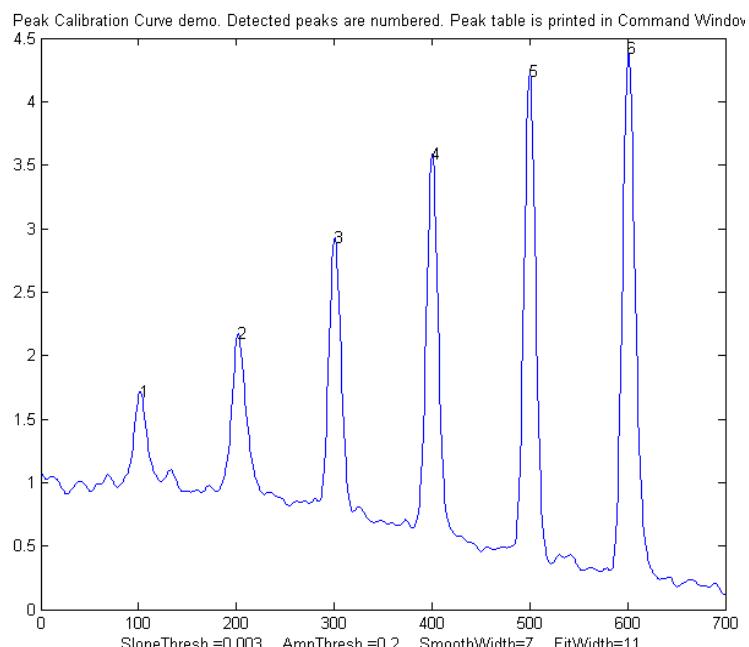
Most scientific measurements involve the use of an instrument that actually measures something else and converts it to the desired measure. Examples are simple weight scales (which actually measure the compression of a spring), thermometers (which actually measure thermal expansion), pH meters (which actually measure a voltage), and devices for measuring hemoglobin in blood or CO₂ in air (which actually measure the intensity of a light beam). These instruments are single-purpose, designed to measure one quantity, and automatically convert what they actually measure into the desired quantity and display it directly. But to insure accuracy, such instruments must 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. General-purpose instrumental techniques that are used to measure the quantity of many different chemical components in unknown samples, such as the various kinds of spectroscopy, chromatography, and electrochemistry, or combination techniques like “[GC-mass spec](#)”, 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 $(A - \text{intercept}) / \text{slope}$, where A is the measured signal (height or area) of that unknown. The predicted standard deviation in the sample concentration is $C * \text{SQRT}((\text{sdi}/(A-\text{intercept}))^2 + (\text{sds}/\text{slope})^2)$ by the [rules for propagation of error](#). All these calculations are done in [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, as [demonstrated in this animation](#) (thanks to Cecilia Yu of Wellesley College).

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 to reduce noise, or differentiation to reduce the effect of background, or measure peak area to reduce the effect of peak broadening, or use modulation to reduce the effect of low-frequency drift, then you *must* use the exact same signal processing for both the standard samples and the unknowns, because the choice of signal processing technique can have a big impact on the magnitude and even on the *units* of the resulting processed signal (as for example in the derivative technique and in choosing between peak height and peak area).

PeakCalibrationCurve.m is an 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. (We 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:



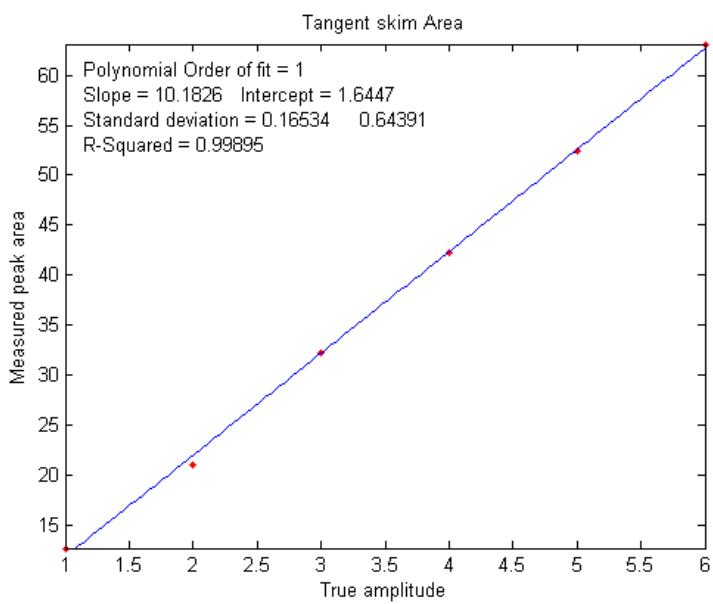
- a. *noise* - random white noise added to all the signal data points, controlled by the variable "Noise";
- b. *background* - broad curved background of *random amplitude*, tilt, and curvature, controlled by "background";
- c. *broadening* - exponential peak broadening that *varies randomly* from peak 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 111). It plots separate calibration curve 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-square would work better). The slope and intercept of the best-fit line is different for the different methods, but if the R2 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 R2 values will all be 1.000. Otherwise the measurements will not be perfect and some methods will result in better measurements - R2 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.

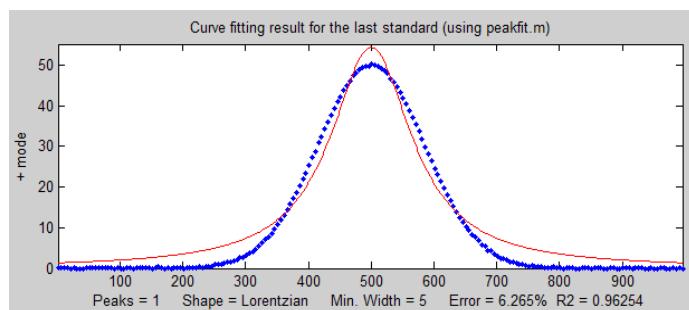
In this type of application, the peak heights and/or area measurements do not actually have to be *accurate*, but they must be *precise*. That's because the objective of an analytical method such as flow injection or chromatography is *not* to measure the peak *heights* and *areas*, but rather to measure *concentrations*, which is why calibration curves are used. [Figure window 6](#) shows the correlation between the measured tangent skim areas and the actual true areas under the peaks in the signal shown above, right; the slope of this plot shows that the tangent skim areas are actually 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* application, you may actually need to measure the peak heights and/or areas accurately, in which case curve fitting is generally the best way to go.



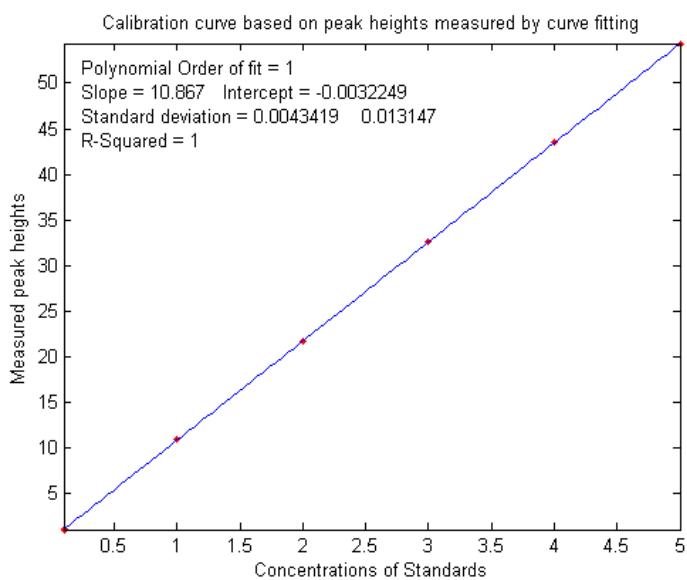
If the peaks partly overlap, the measured peak heights and areas may be effected. To reduce the problem, it may be possible to reduce the overlap by using [peak sharpening methods](#), for example the [derivative method](#), [deconvolution](#) or the [power transform method](#), as demonstrated by the self-contained Matlab/Octave function [PowerTransformCalibrationCurve.m](#).

Curve fitting the signal data. Ordinary in curve fitting, 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 *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, the peak shape of the actual peak is Gaussian (blue dots) but the model used to fit the data is Lorentzian (red line). That's an intentionally bad fit to the signal data; the R₂ 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 below 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's actually 10.867 in the figure on the left, but nevertheless the *calibration curve is still linear* and its R₂ value is 1.000, meaning that the analysis should be accurate. (Note that 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's still better to use as accurate a model peak shape as possible for the signal data, because the percent fitting error or the R₂ 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.

V. Numerical precision of computer software

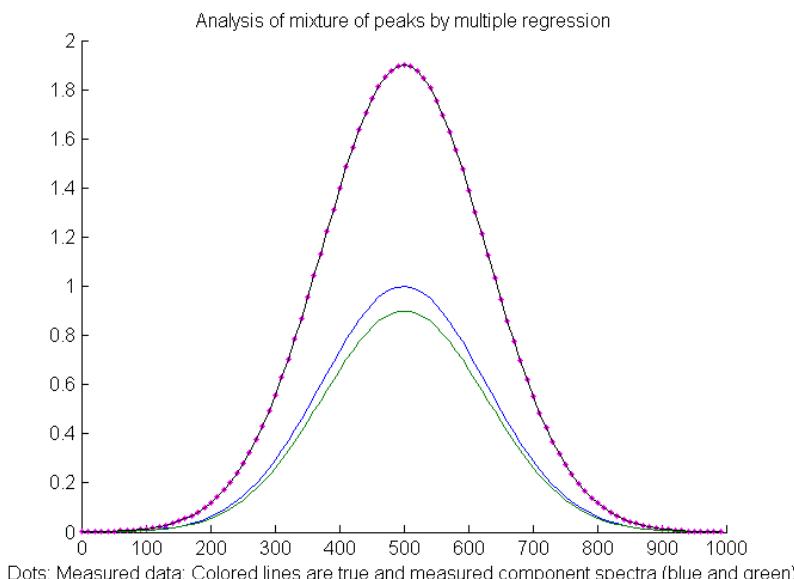
Computations carried out by computer software with non-integer numbers have a natural limit to the precision with which they can be represented; for example, the number $1/3$ is represented as $0.3333333\dots$ with a large but finite number of "3"s, whereas theoretically there are an *infinite* string of "3"s in the decimal representation of $1/3$. It's 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 significant source of error. In the vast majority of applications to scientific computation, however, 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*, Wolfram's *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 $A\b\dots$ [and] ... replace $b*inv(A)$ with b/A "

"Slow and inaccurate"? OK, now I'm scared. But how serious a problem is this really in actual applications?

To answer that question, the Matlab/Octave script [RegressionNumericalPrecisionTest.m](#) applies the CLS method to a mixture of two *very closely-spaced noiseless* overlapping Gaussian peaks (blue and green lines in the figure on the left) 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 (i.e. difficult); this is equivalent to trying to measure a mixture of two



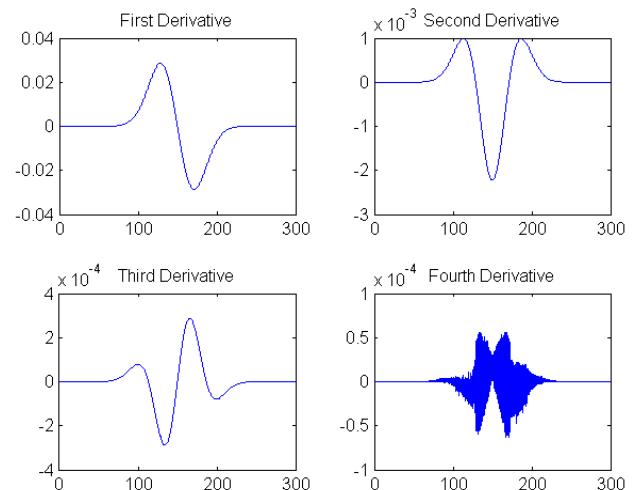
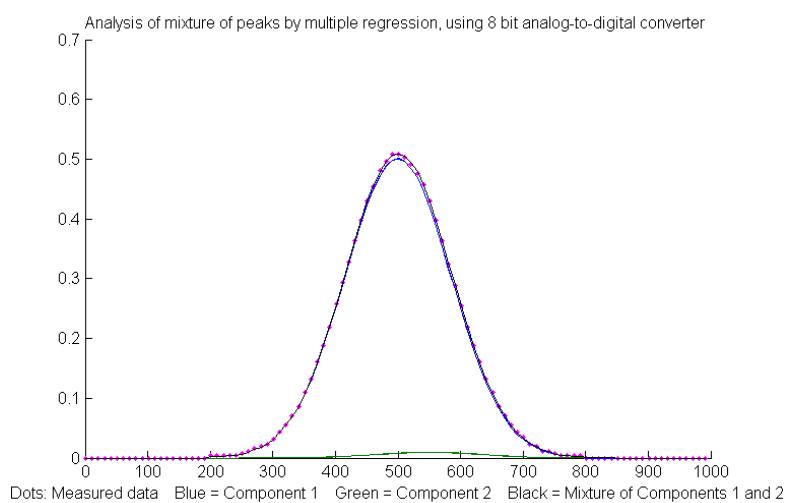
ratio is 0.0033, which is very small (i.e. difficult); this is equivalent to trying to measure a mixture of two

absorption spectroscopy peaks that are 300 nm wide and *separated by only 1 nm*, a tiny difference that you wouldn't even notice with the naked eye. The results of this script shows 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 (like that caused by small changes in the temperature of the sample or random noise in the signal, which you can simulate in line 15) produces a far greater error. So basically that warning message is the voice of a mathematician or computer programmer, not an experimental scientist.

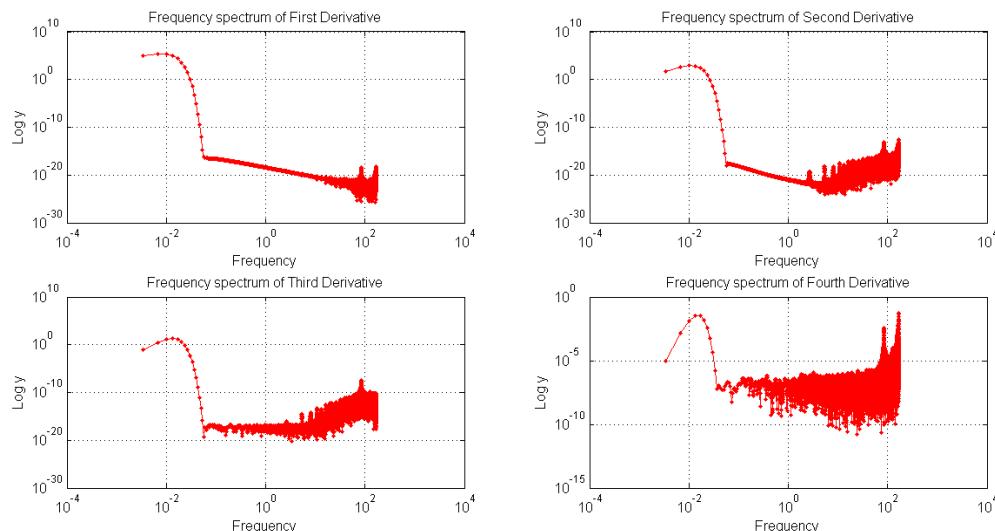
Analog-to-digital resolution. Potentially more significant than the computer's numerical resolution is the resolution of the *analog-to-digital converter* (ADC) that is used to convert analog signals (e.g. voltage) to a number. The Matlab/Octave script [RegressionADCbitsTest.m](#) demonstrates this, 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); 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 errors 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 *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 can't simply turn up the amplification 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 actually help to *add some additional random noise* (specified in line 10 in this script), as was seen on page 270.

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 self-contained Matlab/Octave



function [DerivativeNumericalPrecisionDemo.m](#) shows how the numerical precision limits of the computer effects 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 shown in the figure just below. The numerical precision limits 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 removes most of it ([click to view](#)).



W: Miniaturized signal processing: The Raspberry Pi



Signal processing does not necessarily require expensive computer systems. The Raspberry Pi is a remarkably tiny and inexpensive computer board that is about the *size of a deck of cards* and [costs \\$38!](#) Version 3 B+ in early 2018 has a 1.4GHz 64-bit quad-core ARMv8 CPU with 1GB RAM, 4 USB ports, 40 general-purpose input-output pins, HDMI port, 300 mbps Ethernet port, audio jack and composite video, video camera and display interfaces, micro SD card

slot for mass storage, VideoCore IV 3D graphics core, 802.11ac Wireless LAN, and Bluetooth 4.2. You can get it with a bunch of installed software, including a version of the Linux operating system, a simple but effective graphical desktop modeled on Windows, a Web browser, the complete [LibreOffice suite](#), Wolfram's [Mathematica](#) ([screen shot](#)), several programming languages, a bunch of games (including [Minecraft](#)), and various utilities. [All of these are installed by default](#) on the Raspberry Pi's [operating system installer](#). (There is even a smaller and cheaper model called the [Zero](#) that costs

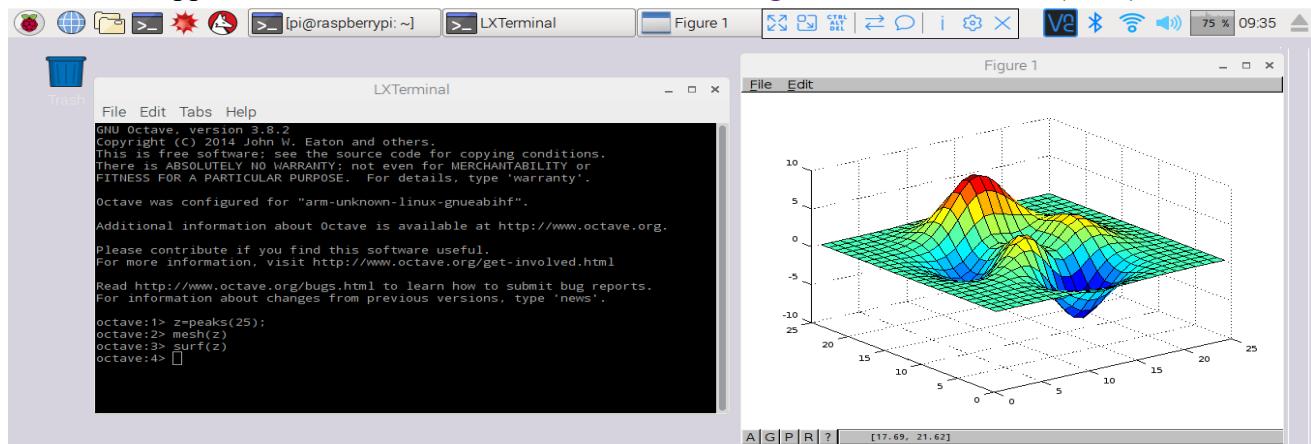
just \$5 for the card itself or \$10 with memory card and power supply; it has less memory and smaller connectors than the other models, but because of its low cost and small size, this model is ideal in situations where it might be damaged or lost, as in rocket or balloon borne experiments).

All you need for a complete computer is a 5 volt, 2 amp power supply, a USB keyboard and mouse (which you can probably pick up at a junk shop), a TV/monitor with an HDMI input, and a mini SD card (8 to 16 Gbytes) for mass storage (you can buy this card with all the software already installed or a blank one to which you can download the software yourself). In fact, if you already have a Wi-Fi network and an Internet-connected computer, tablet, or smartphone, you don't need a separate monitor, keyboard and mouse: you can log onto the Raspberry Pi via your Wi-Fi network or over the Internet, using [Putty](#) (for command-line UNIX-style access) or 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. It can also [share files with Windows](#). 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 (Python, C, C++, Java, Scratch, and Ruby). It's also ideal for "[headless](#)" applications where it is only accessed remotely via WiFi or Bluetooth, for example as a [network file server](#), [weather station](#), [media center](#) or as a networked [security camera](#).

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. Allen B. Downey's 164-page PDF book "[Think DSP](#)" has many examples of Python code in traditional engineering applications. Add-on hardware devices available at low cost, include [video cameras](#) and a piggyback [sensor board](#) that [reads and displays sensor data](#) from several built-in sensors: gyroscope, accelerometer, magnetometer, barometer, temperature, relative humidity. (It's based on the [same hardware that is currently in orbit on the International Space Station](#)).

My signal processing spreadsheets (page 428) run just fine on the version of *Calc* that comes with the Pi, as do the Calibration worksheets (page 386) and my [analytical instrument models](#) (page 316).

For school applications, Element14 markets a [Learn to Program Pack Starter Kit](#) (\$177) that includes a



license for [student version of Matlab](#) for Windows or Macintosh and a Raspberry Pi 3 with MicroSD card, power supply, and enclosure (Matlab does not currently run directly on the Pi but can

communicate with it). Even cheaper, [Octave 3.6 can run directly on a Raspberry Pi](#); the screen above shows Octave 3.6 running within the Pi's built-in graphical user interface (showing off the 3D graphic functions "mesh" and "surf").

There are many [laboratory and field applications](#), especially in combination with an [Arduino micro-controller](#). However, the [slowness of Octave \(compared to Matlab\)](#), combined with the modest speed of the Raspberry Pi 3 (altogether about 50 - 1000 times slower than Matlab on a contemporary desktop computer), may be limiting in some applications. But it's also possible to communicate with Raspberry Pi hardware remotely from a faster computer running MATLAB using the [MATLAB Support Package for Raspberry Pi Hardware](#) for Matlab R2016b, using one or more remotely accessed Raspberry Pi's for experiment control and data acquisition and local storage and doing the heavy-duty number crunching on the main computer. Or you could simply have the Pi save data or results in a [shared folder](#) that is accessed via WiFi from another computer.

[Python](#) is the primary programming language that comes with the Raspberry Pi. This language is quite different than the older languages traditionally used by scientists, such as Fortran or Pascal, and it can be confusing for people without a computer-science background. Here is a simple [real-time example of data acquisition and plotting on a Raspberry Pi](#), using the commercially available add-on [Sense Hat](#) board with a [program written in Python](#), measuring temperature as a function of time ([real-time animation](#)). If you don't have a Sense Hat, here's a [modification of the same Python program](#) that plots the running average of a random number, using the same autoscaling graphic technique, graphing a result that gradually settles down closer and closer to the average the longer you let it run. This shorter [Matlab/Octave script](#) does the same thing at the same speed. These two programs constitute a nice comparison of Matlab vs Python, illustrating the advantages and disadvantages of each.

Other competing small-scale systems include the [LattePanda](#), a tiny \$130 Windows-10 computer board with 2 Gbytes RAM and 32 Gbytes flash storage. [Many other similar products are available](#).

X: Batch processing

In situations where you have a large volume of similar types of data to process, it's useful to automate the process. Let's assume that you have already acquired data in the form of a large number of text or numerical data files of some standardized format that are stored in a known directory (folder) somewhere on your computer. For example they might be ASCII .txt or .csv 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 [unknown](#) and variable, but the data format is consistent from file to file. You could write a Matlab script or function that will process those files *one-by-one*, but you want the computer to 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, whatever), collect all the resulting terminal window output, each labeled with the file name, add the 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 with one of the data files; rather, it should just *skip that one and go on to the next*.

[BatchProcess.m](#) is a Matlab/Octave example of just such an automated process that you can use as a framework for your applications. The main things you need to change here are:

- (a) the directory name where the data are stored on your computer - (DataDirectory) in line 11;
- (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 opens a “diary” file in line 21, located in the FunctionsDirectory, with the file name “BatchProcess<date>.txt” (where <date> is the current date, e.g. 12-Jun-2017). 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 the percent fitting error and R2 value, for each data file in that directory. (Subsequent runs of the program on the same date are appended to this diary 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 to 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 a couple of coding techniques that are especially useful in automated file processing. It uses the “function forms” of the 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; rather, it reports the error for that file and skips to the next one.

After running this script, the “BatchProcess...” diary file will contain all the terminal output. Here's an excerpt from a typical diary file, in which *the first two data files in the directory yielded errors*, but the third one ("2016-08-05-RSCT-2144.txt") and all the following ones 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....
```

You could also optionally import the diary 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).

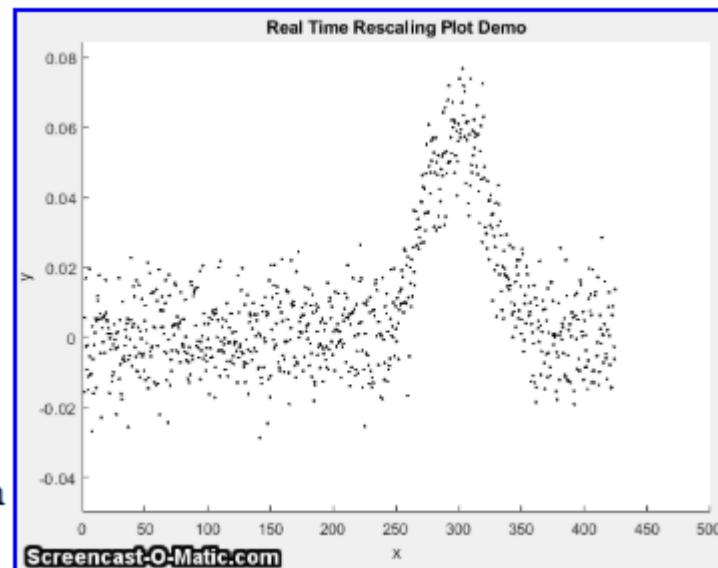
Y: Real-time signal processing

All of the signal processing techniques covered so far make the assumption that you have acquired and 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 we will look at ways to perform real-time data plotting, smoothing, differentiation, peak detection, harmonic analysis (frequency spectra), and Fourier filtering. Because the details of the data acquisition itself varies 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 without additional hardware.

I'll do this in either of two ways:

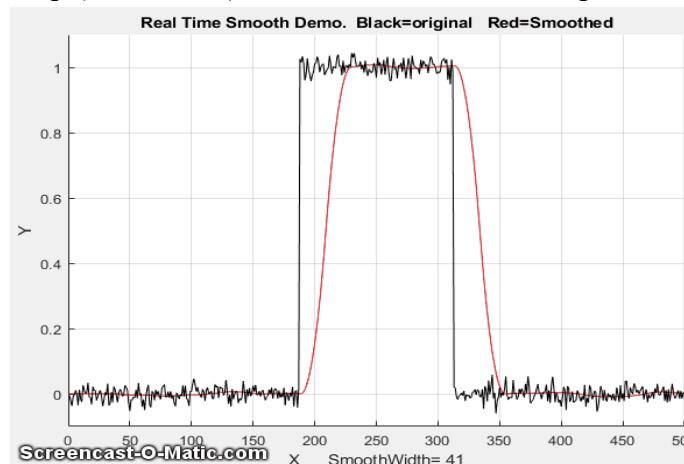
- (a) by using mouse-clicks to generate each data point, using Matlab's "ginput" function, or
- (b) by pre-calculating simulated data and then accessing it point-by-point in a loop.

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 script [realtimeplotautoscale2.m](#) uses the second method to simulate real-time data, using [pre-calculated data](#) (loaded from disk in line 13) that is accessed point-by-point in lines 25 and 26 (animation shown on the right). The script [realtimeplotdatedtime.m](#) demonstrates one way to use Matlab's 'clock' function to record the data 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 not 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's adequate for low-speed applications. For higher speeds, [specialized hardware](#) and [software](#) is available.

Smoothing. The script [RealTimeSmoothTest.m](#) demonstrates real-time smoothing (page 35), 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 '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).

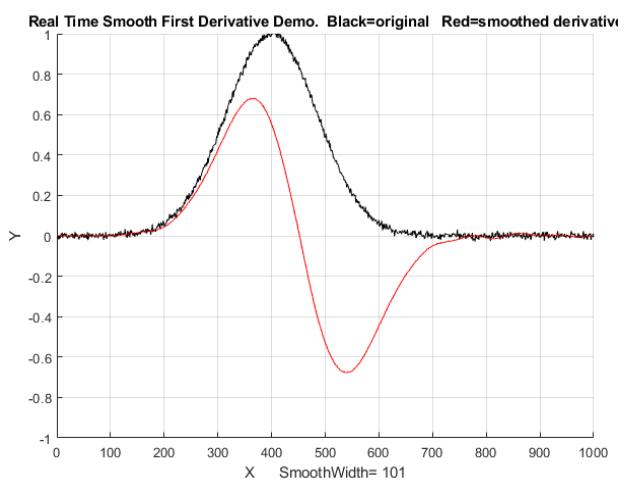
As you can see in the screen image on the left ([link to animation](#)), the smoothed data (in red) is delayed compared to the raw data, because a smoothed data point cannot be computed until a number of data points equal to the smooth width

has been acquired - 41 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 triangular 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 path.

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) and 1.4 milliseconds per point with point-by-point plotting (lines 40-50). Plotting the data points point-by-point slows it down. The script [RealTimeSmoothPostPlot.m](#) 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 (about 500,000 Hz) than plotting in real time.

Differentiation. The script [RealTimeSmoothFirstDerivative.m](#) demonstrates real-time smoothed differentiation (page 53), 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 of these scripts pre-calculate the simulated data in line 28 and then accesses 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 65](#).

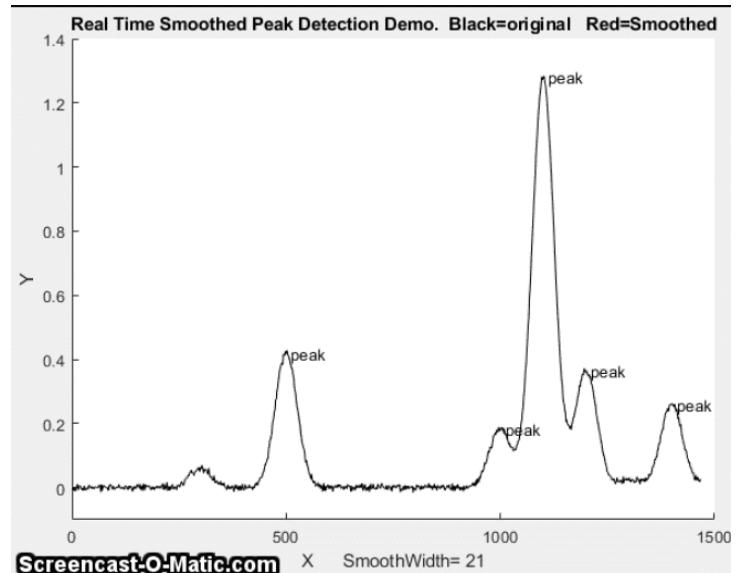


Peak detection. The little script [realtimepeak.m](#) demonstrates simple real-time peak detection based on derivative zero-crossing (page 198), using mouse clicks to simulate data. Each time your mouse clicks form a peak (that is, go up and then down again), the program will register and label the peak 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's 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 200; 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. (In the screen image above, the first visible peak, at *x*=300, is not detected because it falls below the amplitude threshold). Link to [animation](#).



Screencast-O-Matic.com

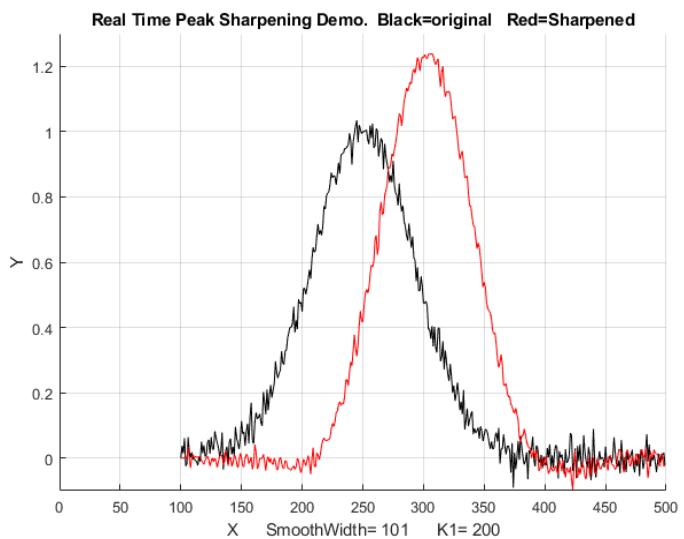
```

Peak detected at x=500.1705, y=0.42004, width= 61.7559
Peak detected at x=1000.0749, y=0.18477, width= 61.8195
Peak detected at x=1100.033, y=1.2817, width= 60.1692
Peak detected at x=1199.8493, y=0.36407, width= 63.8316
Peak detected at x=1400.1473, y=0.26134, width= 58.9345

```

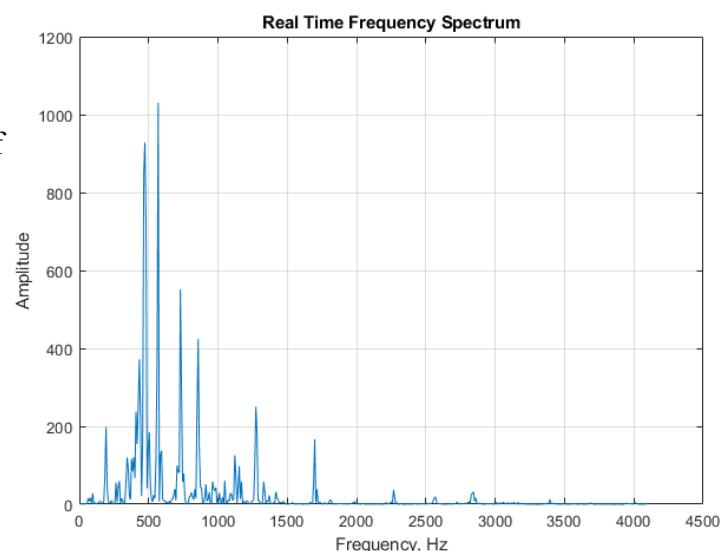
The latest version additionally [numbers the peaks](#) and saves the peak parameters of all the peaks in a matrix called PeakTable, which you can interrogate as each peak is encountered if you are looking for particular peak patterns. See page 213 for some ideas on using Matlab/Octave notation and functions to do this.

Peak sharpening. The Matlab/Octave script [RealTimePeakSharpening.m](#) demonstrates real-time peak sharpening (page 69) 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 this example on the left, the smooth width is 101 points, which accounts for the delay in the sharpened peak compared to the original.



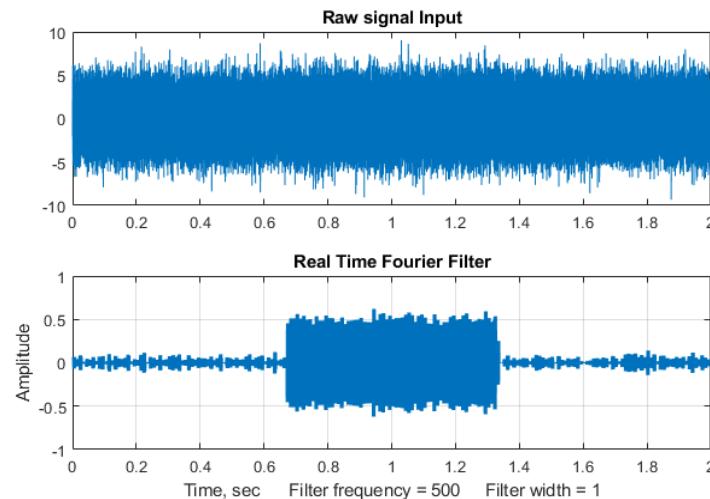
Real-Time Frequency Spectrum. The script [RealTimeFrequencySpectrumWindow.m](#) computes and plots the Fourier frequency spectrum of a signal (page 80). Like the scripts above, it loads the simulated real-time data from a “.mat file” and then accesses that data point-by-point in the processing loop. 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 a standard 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 seconds) 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's 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 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 in fact an audio recording of an 8-second excerpt of the 'Hallelujah Chorus' from Handel's Messiah with a sampling rate of 8192 Hz; 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 at the correct pitch.



Real-Time Fourier Filter. The script [RealTimeFourierFilter.m](#) is a demonstration of a real-time Fourier filter. It pre-computes a simulated signal starting in line 38, then access the data point-by-point

(line 56, 57), and divides up the data stream into segments to compute each filtered section. "WindowWidth" (line 55) is the number of data points in each segment. The larger this number, the fewer the number of segments that will be generated, but the higher will be the frequency resolution within each segment. On a standard 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, shown the figure on the left.

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 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 is quite obvious in the filtered output (lower panel). The filter center frequency (`CenterFrequency`) and width (`FilterWidth`) are set in lines 46 and 47.

To apply any of these examples to real-time data from your sensor or instrument, you need only 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 don't 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.).

Z. Dealing with variable data arrays in spreadsheets

When applying spreadsheet templates of the type described in this book to your own data, it's often necessary to modify the templates to accommodate different numbers of data points or of 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.

The MATCH function. In signal processing using spreadsheets, it's common to have x-y arrays of

	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

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. Spreadsheet formulas normally refer to cells by their row and column address, but for an x-y data set like this, it's more natural to refer to a data point by its independent variable x , rather than its row and column address. For example, suppose you want to select the data point where $x=2$, irrespective of what cells they inhabit. You can do that with the MATCH function. For example, if you set cell B2 to the desired x value (e.g. 2), then $\text{MATCH}(B2,A5:A11)+\text{ROW}(A5)$ 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.

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 " $=\text{INDIRECT}("B"&A1")$ ", 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 particular data set.

These examples were done in what is called the “A1” reference style, where the columns are referred to by letters; it’s also possible to use the “R1C1” reference style, where both the and the rows 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’s longer, but the advantage over direct addressing is that you can adjust the range by changing just two cells rather than retying the formula. It’s 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 46.

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 except 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’s 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 exactly the same for the two related functions that calculate the INTERCEPT and RSQ (the R2 value) of the regression line. I agree that it’s 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

	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		

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 R2 values of the y vs x linear regression line (page 126) 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 recalculate, without your having to edit any of the cell formulas*. Try it yourself. (You can float the mouse pointer over any cell with 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 copy data from columns A and B into columns J and K only those data points that fall between the two specific x limits.

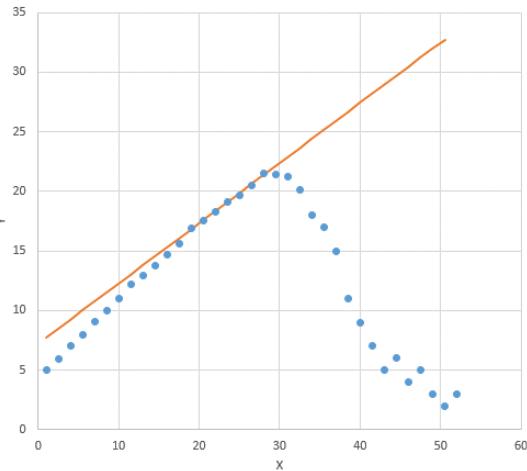
If desired, you can add more data to the end of columns A and B, limited only by the range of the match

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 234 uses this technique.

The LINEST function. Indirect addressing is particularly useful when using *array functions* such as LINEST (page 143) or the matrix algebra functions (page 156). 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 152). 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 peaks heights calculated by LINEST for the three peaks are given in row 17, columns C, D, can E, and the predicted standard deviations are in the row below. In this spreadsheet the data are actually *simulated* (over 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 157.

Blue: Plot of x and y data in Columns A and B.
Red: regression line for *selected points only*.
Select the low and high x limits in cells B2 and B3

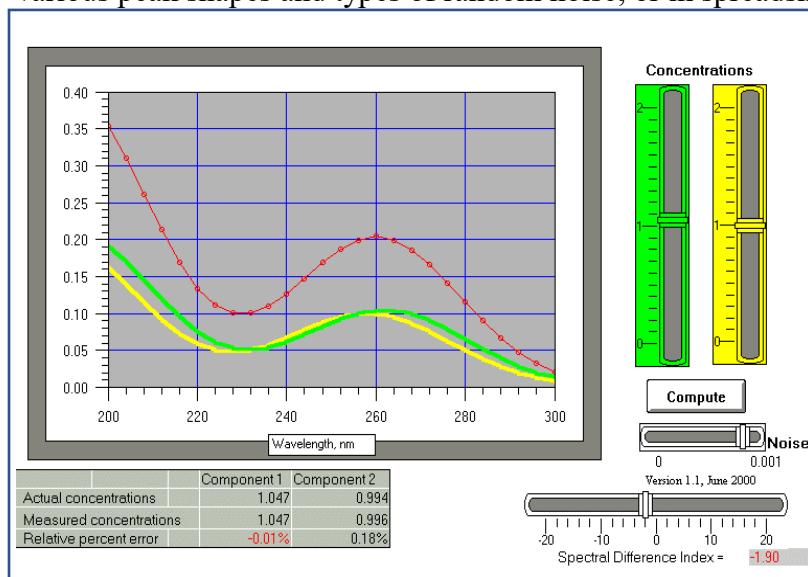


AA. Computer simulation of signals and instruments.

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 signal 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 22), which may be various colors (page 26) and amplitude dependences (page 27).

This can be done either in Matlab/Octave, using the built-in and downloadable functions (page 400) for various peak shapes and types of random noise, or in spreadsheets, which can also be used to create

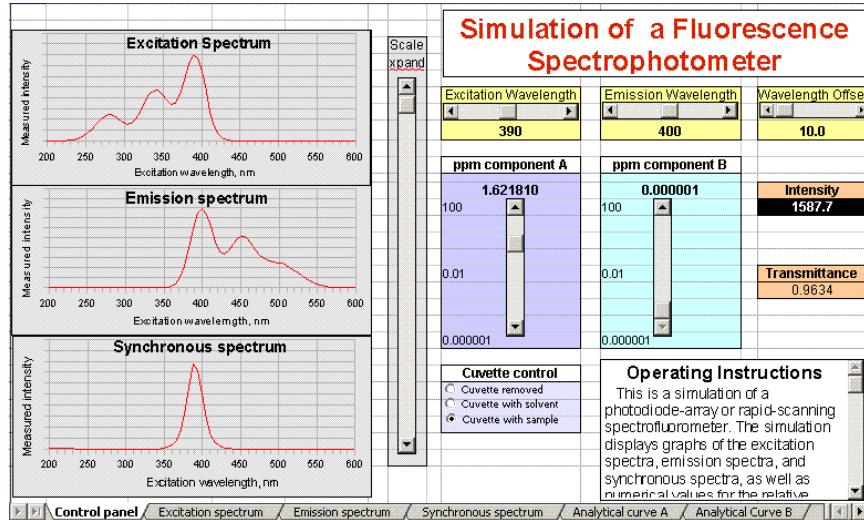


attractive and intuitive user interfaces; some spreadsheet examples include [SimulatedSignal6Gaussian.xlsx](#), [PeakSharpeningDemo.xlsx](#), [PeakDetectionDemo2.xls](#), [TransmissionFittingDemoGaussian.xls](#), [BeersLawCurveFit2.xls](#), and [RegressionDemo.xls](#) (on the left).

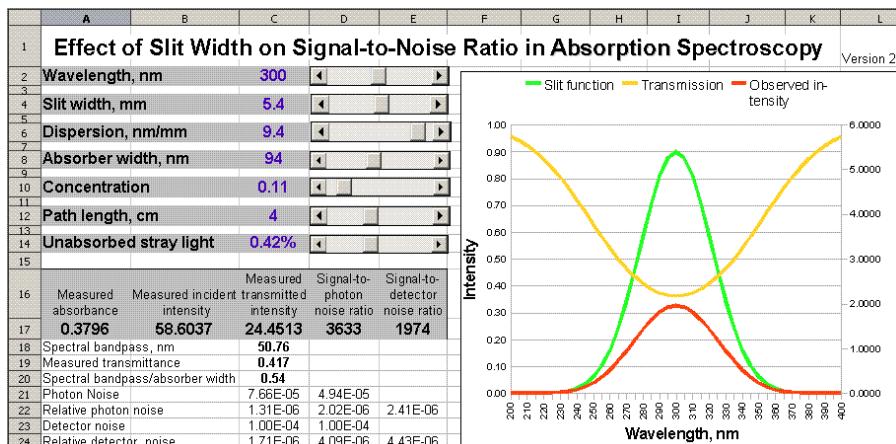
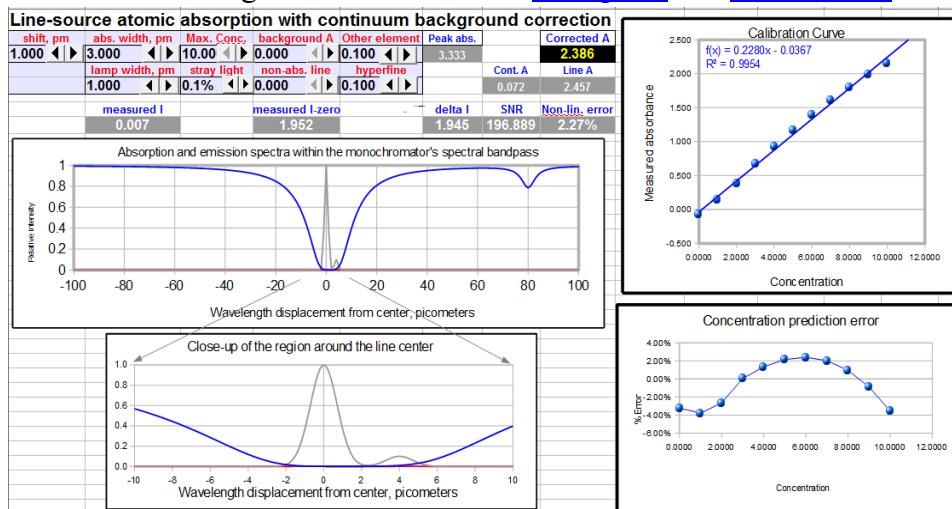
It's possible to make any aspect of a computer generated signal randomly variable from measurement to measurement, with the aim of making the simulation as close as possible to

the real signal behavior that you may have to measure. For example, in the section “The Battle Rounds: a comparison of methods” on page 259, 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 276, 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 285, 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 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's possible to simulate the fundamental aspects of such instruments as a [scanning fluorescence spectrometer](#) (above) 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 281), a [wavelength modulation](#) spectroscopy system, and even [basic analog electronic and operational amplifier circuits](#). Note that *these are not simulations of particular commercial instruments*. 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 the instrument's internal operation.

AB. Who uses this book and its associated web site, documents and software?

In the last few years, my signal processing web site (<http://terpconnect.umd.edu/~toh/spectrum/>) has been accessed from Internet Service Providers in over **162 countries** and 6 non-region-specific categories (e.g. satellite providers), including 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). Breath of Internet access is often an issue. For example, I've got fewer views from Cuba than from 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.

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've 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 recently have been [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 download. (Interestingly, that is not the case with [Interactive Computer Models for Analytical Chemistry Instruction](#), which has only 75% encryption).

There have been over 100,000 downloads of my software and documentation files, currently averaging about 500 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.zip](#), [CurveFitter....xlsx](#), [iSignal6.zip](#), [ipeak7.zip](#), [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 via 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 log(page views) and log(population) is very poor*, with a coefficient of determination (log-log correlation coefficient or R² value) of only 0.36 (n=163 countries; over 160,000 total page loads over the period from 2008 to 2017; [graphic link](#)). Note that because of the very large range of population sizes, I did a *log-log* correlation (page 389) in order 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

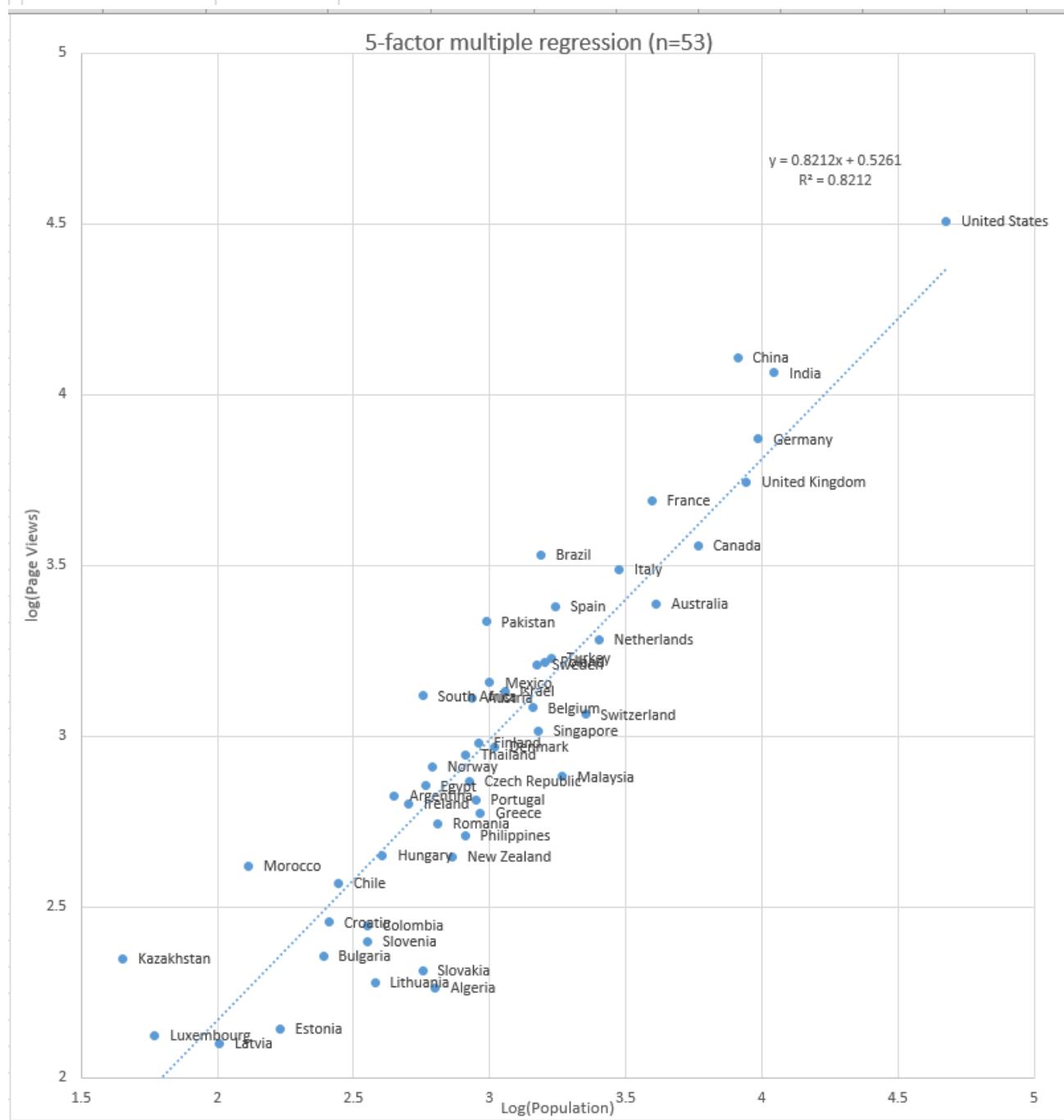
- the number of *English speakers* in each country,
- the number of *Internet users* in each country,
- the number of *universities* in each country, and
- the total *research and development budget* of each country.

All of 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 R² 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 a R² 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 (Graphic below).

Satellite Provider	6	Log-log correlations to hits	n	R2
Asia/pacific Region	105	population	163	0.4
Anonymous Proxy	64	English speakers	92	0.579
Europe	955	Number of universities	146	0.6814
Non-country hits	1130	Internet users	145	0.6812
Country hits	159297	R&D spending	72	0.75
		Patent applications, 2008-2012	133	0.591

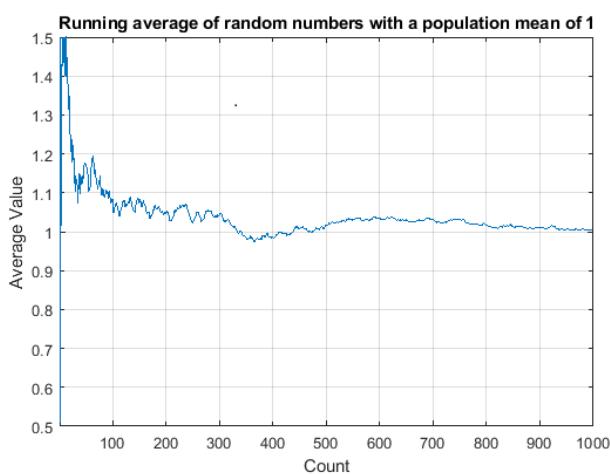


For an Excel spreadsheet with all these data and calculations (between 2008 and 2015), see [FinalCountriesSummary.xlsx](#)

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 don't write 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, I have only incomplete information about the application areas where my programs are being applied. A list of specific application areas on which readers are working is given in [applications.pdf](#). As of December, 2018, my site and its programs had been cited in over [360 published papers and patents](#), whose titles also indicate areas of application (page 440).

AC. The Law of Large Numbers

[The Law of Large Numbers](#) is a theorem that describes large collections of numbers subject to independent and identically distributed random variation, such as the result of performing the same experiment a large number of times. The average of the results obtained from a large number of trials should be close to the expected value, 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 why gambling casinos are able to make money; their games are designed to give the casino a small advantage in the long run but highly variable results in the short term, guaranteeing plenty of winners, to encourage the gamblers. 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. And that's why it's 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.



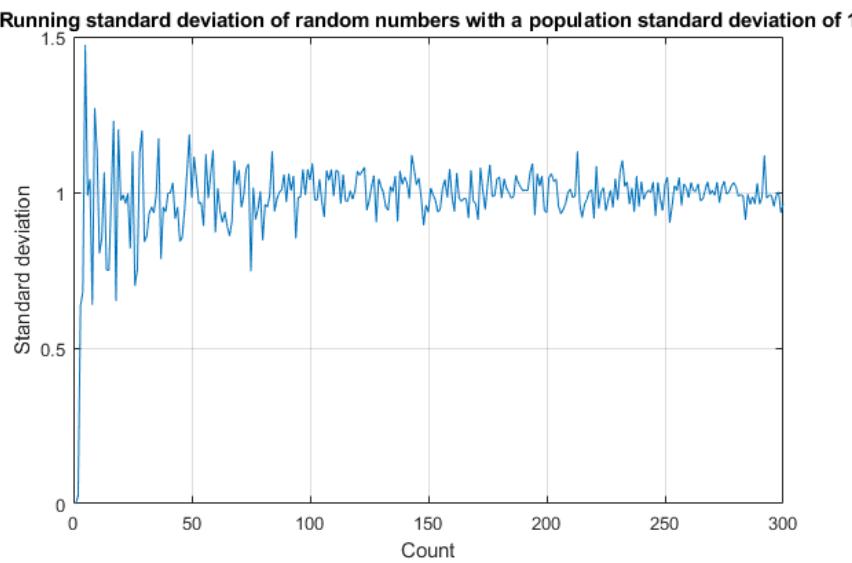
But “The average … will tend to become closer as more trials are performed” does *not* mean that the average becomes *steadily and irreversibly* closer. In fact, the average can wander around quite a bit. Take the example on the left, which shows the running average of a set of normally distributed independent random numbers with a population mean of 1.000 and a standard deviation of 1.000, as more and more numbers from that population are averaged, up to 1000. (This is generated by the Matlab script [RunningAverage.m](#)). Note that the average wanders around, reaching and crossing over the true

population average twice in this case before ending up near 1.0 after 1000 points are accumulated. But if you ran this script again, the final average may *not* be so close to 1.0. In fact, the predicted standard deviation of the average of 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 fall [within 6% of the true standard deviation](#).

The uncertainty of uncertainty. The situation is even worse if you wish to estimate the *standard deviation* of a population from small samples. The Matlab script

[RunningStandardDeviation.m](#) simulates this for the same population in the previous example. As shown in the graph on the left, the sample standard deviation wanders around alarmingly for small sample and only settles down slowly. Even worse, the standard deviation for very small samples is [biased down](#), often returning values far lower than usual.



There is a well-documented tendency for people to *overestimate* the quality of small number 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 people tend to believe in a false "[Law of Small Numbers](#)", the name they coined for the mistaken belief that a small sample drawn from a large population is representative of that large population. We'd 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's tempting to stop there, "while you're ahead". This is called "[confirmation bias](#)".

Of course in many practical experimental measurements, you may really be constrained to a rather small number of repeated measurements. There may be a fixed number of data points and no possibility of gathering more. Or the cost, in money or in time, of gathering more data may be excessive. For example, the process of calibrating an analytical instrument for quantitative measurement (pages 298, 386) may involve the preparation and measurement of several standard samples or solution 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. 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. The bottom line is, if you are limited to a small number of data points, do not over-represent the precision of your results. To use the [3-sigma rule](#) (page 29) to determine uncertainty ranges for a set of data, the distribution must be normal (Gaussian) and you need to know the standard deviation. For small set of data, both are uncertain.

Signal processing software details

Interactive smoothing, differentiation, and signal analysis (iSignal)

iSignal is a Matlab-only function, written as a single self-contained m-file, for performing smoothing, differentiation, peak sharpening, interpolation, baseline subtraction, Fourier frequency spectrum, least-squares peak fitting, and other useful functions on time-series data.

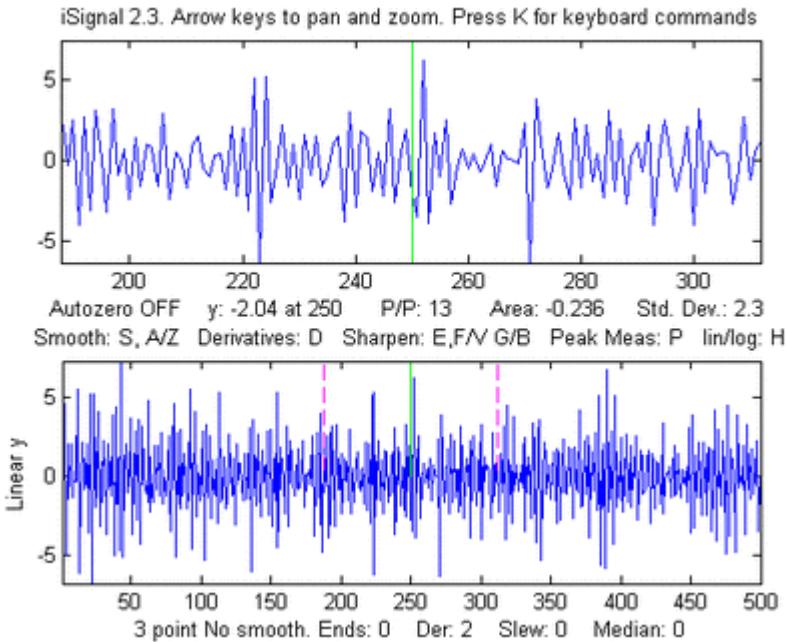


figure window), but not on [Matlab Mobile](#) or in Octave. The demo script "[demoisignal.m](#)" is a self-running demonstration of several features of the program and will test for proper installation; the title of each figure describes what is happening. Its basic operation of iSignal is similar to [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,  
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), with the initial pan and zoom settings optionally controlled by input arguments 'xcenter' and 'xrange', respectively. 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 don't 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, pseudo-Gaussian (3 passes of sliding-average)

If **SmoothMode**=4, [Savitzky-Golay](#) smooth (thanks to [Diederick](#)).

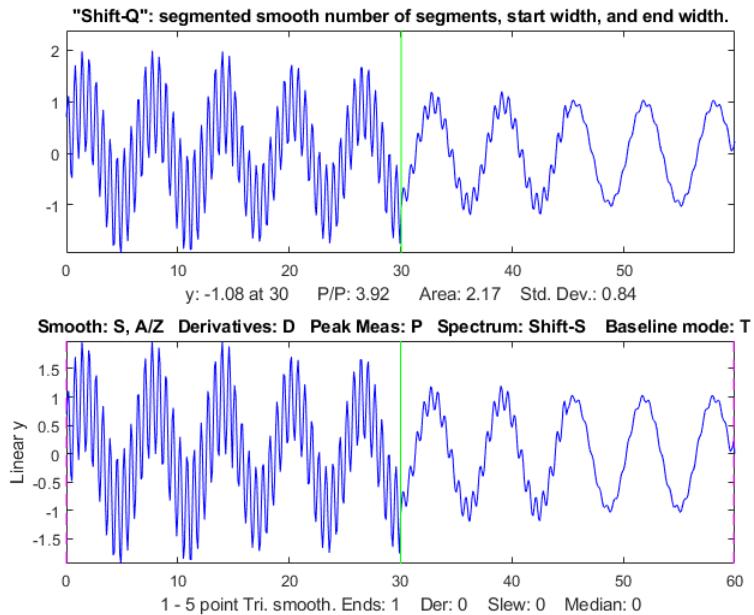
The **A** and **Z** keys (or optional input argument **SmoothWidth**) control the **SmoothWidth**, w .

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 (page 295) 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 percent-age factor. (To return to an ordinary single segment smooth, enter 1 as the number of segments). See the picture of a 4-segment smooth on the right, 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.

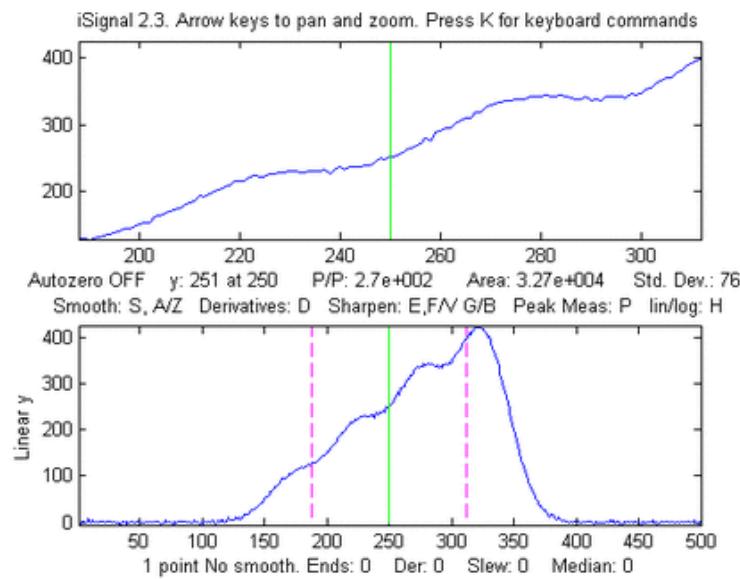
Differentiation

The **D** / **Shift-D** keys (or optional input argument “**DerivativeMode**”) increase/ decrease the derivative order. The default is 0. Careful optimization of smoothing of derivatives is critical for 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 $\text{PeakWidth}^2/25$ and for **Sharp2** is $\text{PeakWidth}^4/800$ (or $\text{PeakWidth}^2/6$ and $\text{PeakWidth}^4/700$ for Lorentzian peaks), where **PeakWidth** is the full-width at half maximum of the peaks *expressed in number of data points*. However, you don't need to do the math yourself; *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.

(The optimum settings depends 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). You can fine-tune the

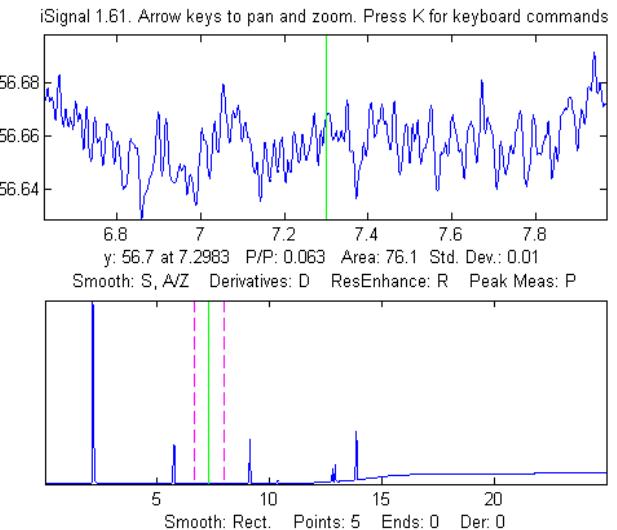
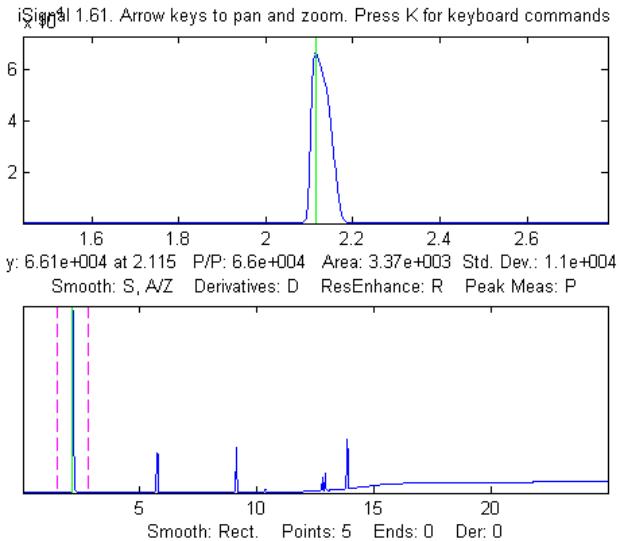


with the **F/V** and **G/B** keys and the smoothing with the **A/Z** keys. [Click for GIF animation of this figure](#).

You can expect a decrease in peak width (and 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 press **E** to toggle the sharpen mode off and on. Note: only the Savitzky-Golay smooth mode is used for peak sharpening.

Signal measurement

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.



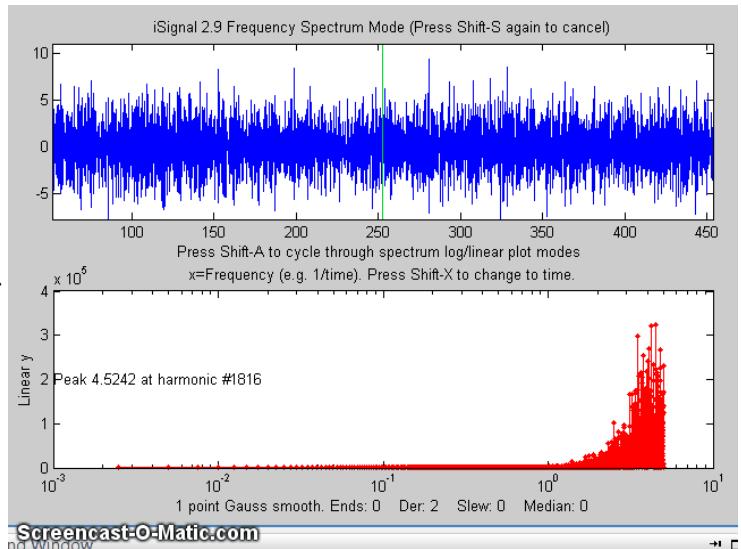
Signal-to-noise ratio (SNR) measurement of a signal with very high SNR. Left: The peak height of the largest signal peak is measured by placing the green center cursor on the largest peak; peak-to-peak signal=66,000. Right: The noise is measured on a flat portion of the baseline: standard deviation of noise=0.01, therefore the SNR=66,000/.01 = 6,600,000

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. The demo script demoisignal42.m demonstrates this version.

Frequency Spectrum mode

The **Frequency Spectrum mode**, toggled on and off by the **Shift-S** key, computes the Fourier frequency spectrum (page 80) 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. 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; click the figure to see an animation of this. 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. [Click for GIF animation.](#)



To save the spectrum as a new variable, call iSignal with the output arguments `[pY, Spectrum]`:

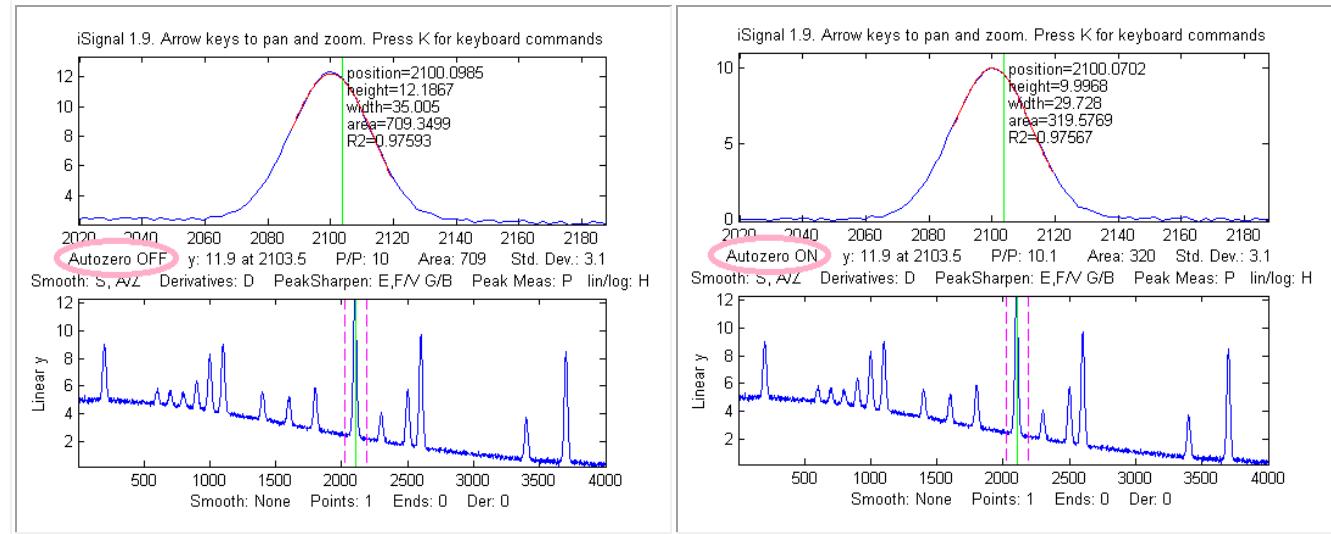
```
>> x=0:.1:60; y=sin(x)+sin(10.*x);  
>> [pY,Spectrum]=isignal([x;y],30,30,4,3,1,0,0,1,0,0,0,1);  
>> plot(Spectrum(:,1),Spectrum(:,2)) or plotit(Spectrum)  
or isignal(Spectrum); or ipf(Spectrum); or ipeak(Spectrum)
```

Shift-Z toggles on and off peak detection and labeling on the frequency/time spectrum; 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 thorough four modes (page 182): *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 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 signal in the first 1/10th of the points and the last 1/10th of the points. Try to adjust the pan and zoom to include some of the baseline at the beginning and end of the segment in the upper window, allowing 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 autozero mode will have a big effect on the measured peak height, width, and area, but very little effect on the peak x-axis position, as demonstrated by the two figures on the next page.



In addition to the four autozero 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; click along the presumed 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 138) 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 to the central peak over the segment that is colored red in the upper panel. (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 most accurate if the peaks are Gaussian. Other peak shapes, or very noisy peaks of any

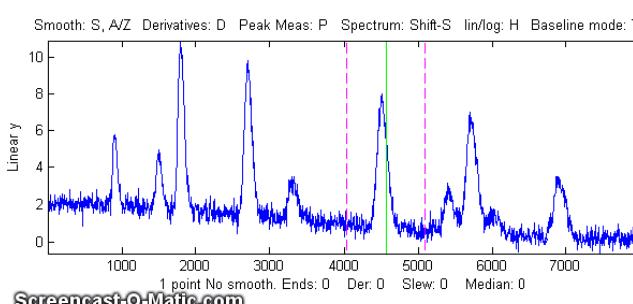
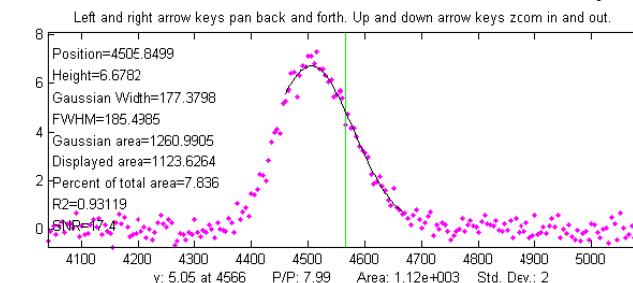
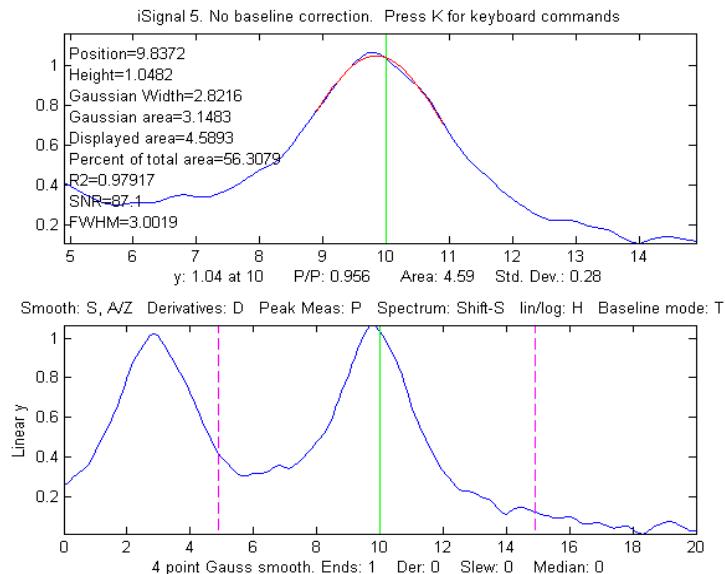
shape, will give only approximate results. However, the position and height, and area values are pretty good for any peak shape as long as the "RSquared" value is at least 0.99. The "SNR" is the signal-to-noise-ratio of the peak under the green cursor; it's the ratio of the peak height to the standard 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 autozero mode (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 actually measured *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.

Version 5.8 (shown below on the left)

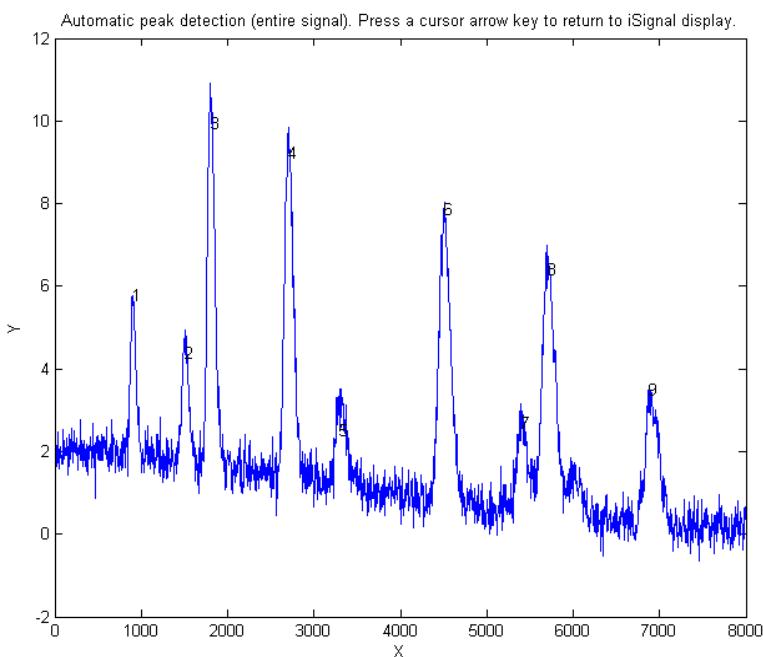
adds [direct measurement](#) of the *full width at half maximum* ('FWHM') of the *central peak* in the upper panel (the peak marked by the green vertical line); this works for peaks of *any shape*, but it is computed only for the central peak and only if the half-maximum points fall within the zoom region 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 width 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 *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 percent 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, as long as 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 198). 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 163) 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 182). 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.

Version 5 of 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

Fourier convolution and deconvolution (Version 5.7 and later)

Shift-V 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 asks you for the width or the time constant (in the units of the independent variable x.):

Convolution/deconvolution menu

1. Convolution
2. Deconvolution

Select mode 1 or 2: 2

Shape of convolution/deconvolution function:

1. Gaussian
2. Exponential

Select shape 1 or 2: 1

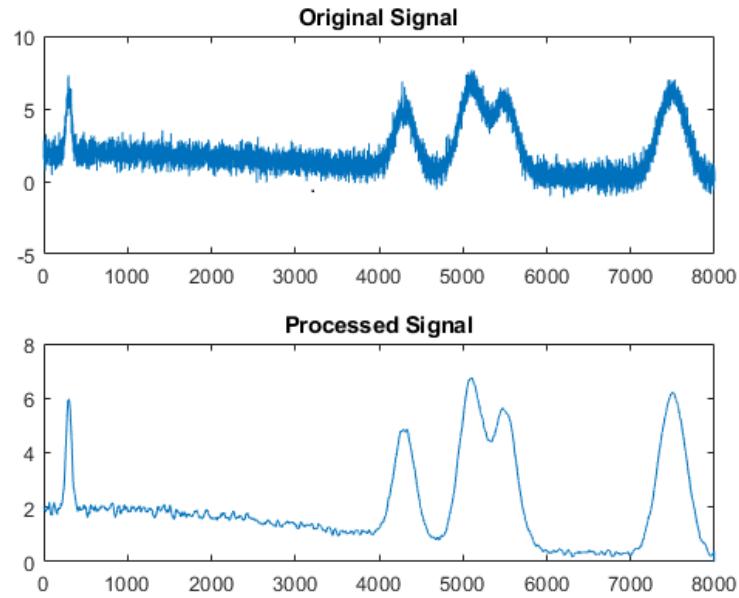
Enter the Gaussian convolution width: .05

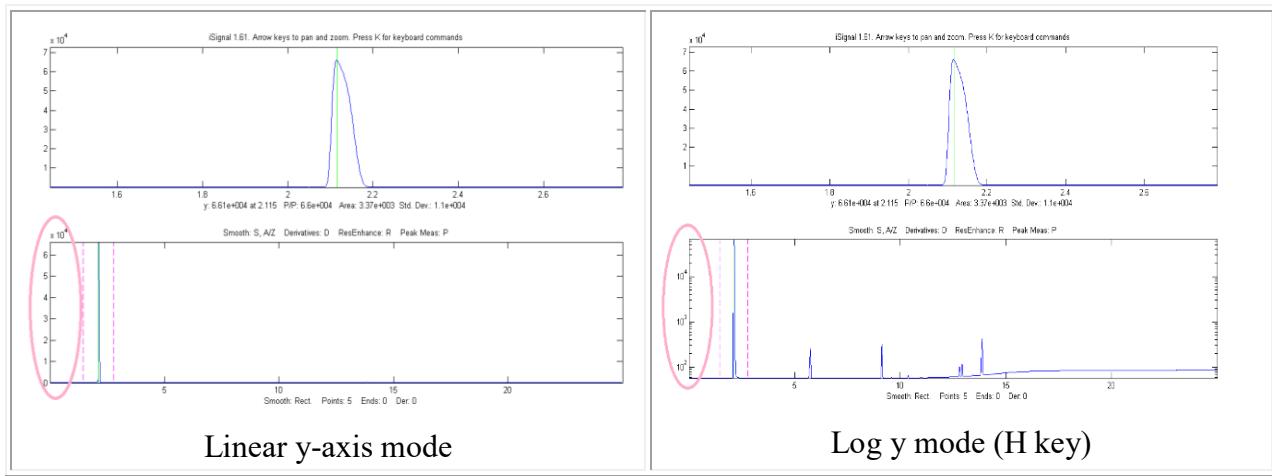
Saving the results

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 (right). The **Tab** key restores the original signal and cursor settings. The ";" key sets the selected region to zero (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). Press '+' key to take the absolute value of the entire signal (and follow this by a smooth to create an amplitude modulation "detector"). In version 5.7, **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. In version 5.95, 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 (see page 73).





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.

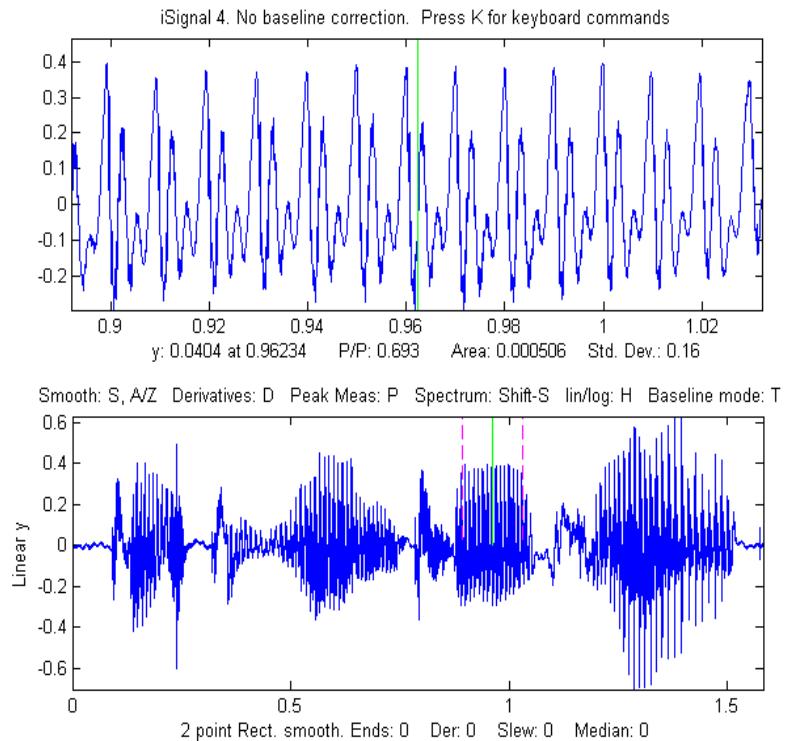
Playing data as audio.

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 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 80) 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 *little effect on the intelligibility*. This is because the sequence of frequency components in the signal 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 Fitter (ipf.m, page 361) and **Shift-Ctrl-P** transfers the current signal to Interactive Peak Detector (iPeak.m, page 361), if those functions are installed in your Matlab path.

Press **K** to see *all* the keyboard commands.

EXAMPLE 1: Single input argument; data in a 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 (xrangle) in the last two input arguments. Using data in the ZIP file:

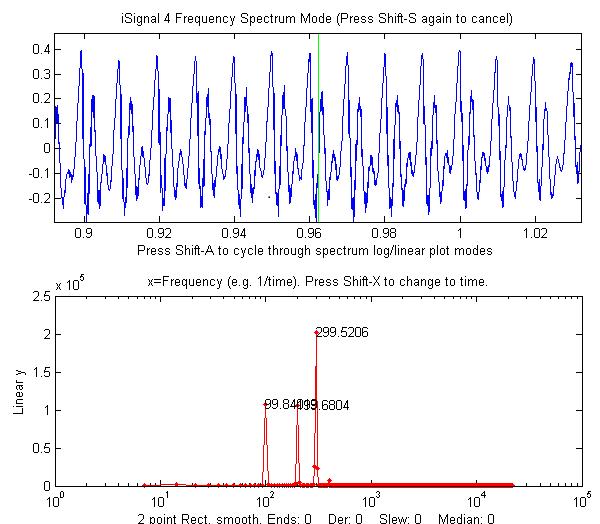
```
>> load data.mat
>> isignal(DataMatrix,180,40); or
>> isignal(x,y,180,40);
```

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:

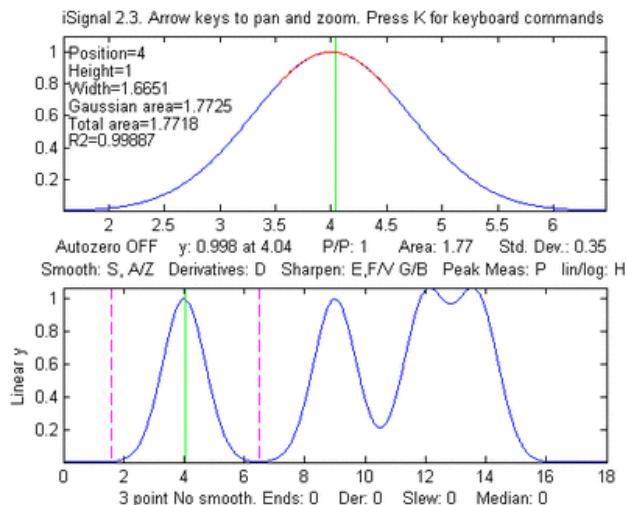
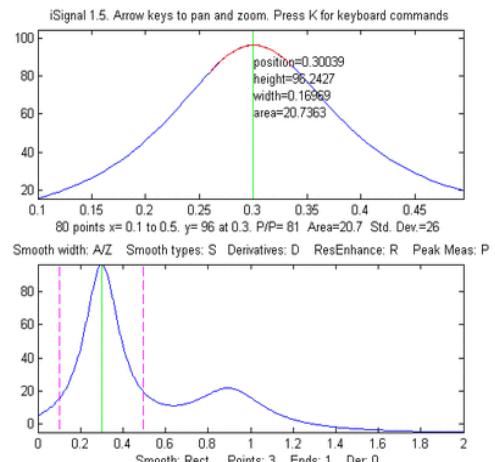
```
>> isignal(Data,0.9,0.5,1,3,1,4);
```

(Click figure on right to see GIF animation.)

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 to compare)



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 111), 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.

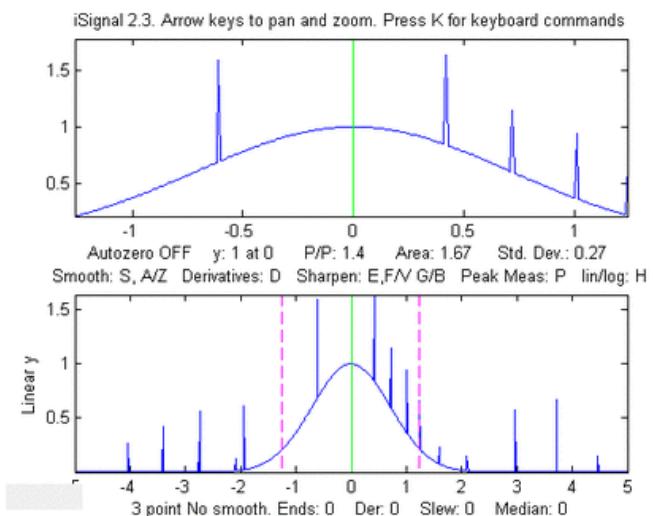
```
x=-5:.01:5;
y=exp(-(x).^2);
for n=1:1000,
    if randn()>2, y(n)=rand() +y(n),
    end,
end;
isignal(x,y);
```

EXAMPLE 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.

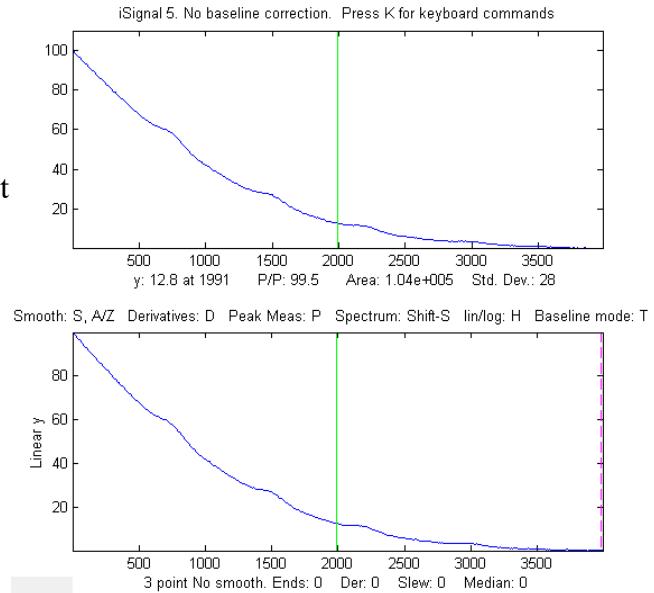
```
>> x=[0:.01:20];
>> y=exp(-(x-4).^2)+exp(-(x-9).^2)+exp(-(x-13).^2)+exp(-(x-15).^2);
>> isignal(x,y);
```

The first peak (at x=4) is isolated, the second peak (x=9) is slightly overlapped with the third



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 work "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 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 Gaussian 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 6.1):

Pan 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 (on/off).....**Shift-G** Temporarily displays grid on
 the plots
Adjust smooth width.....**A,Z** (A=>more, Z=>less)
Set smooth width vector.....**Shift-Q** for segmented smooth
Adjust smooth type.....**S** (cycles through None, Rectangular,
 Triangle, Gaussian, and Savitzky-Golay)
Toggle smooth ends.....**X** (0=ends zeroed 1=ends smoothed (slower)
Adjust derivative order.....**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 (0=OFF)...**M** median filter eliminates sharp spikes
Toggle peak parabola.....**P** fits parabola to center, labels vertex
Fits peak in upper window...**Shift-F** (Asks for shape, number of peaks,
 Number of trials, etc.)
Fit polynomial.....**Shift-o** Fits polynomial to data in
 upper panel
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
Display Waterfall spectrum..**Shift-W** Allows choice of mesh, surf,
 contour, or pcolor
Transfer power spectrum.....**Shift-T** Replaces signal with power spectrum
Click graph to print x,y....**Shift-C** Click graph to print coordinates
Lock in current processing..**Shift-L** Replace with processed version
Convolution/Deconvolution...**Shift-V** Convolution/Deconvolution menu
Power transform method.....^ (**Shift-6**) Raises the signal to a power
Print peak report.....**R** prints position, height, width, area
Toggle overlay mode.....**L** Overlays original signal as dotted line
Display current signals.....**Shift-B** Original (top) vs Processed
 (bottom)
Toggle log y mode.....**H** semilog plot in lower window

Cycle baseline mode.....**T** none, linear, quadratic, or flat
 baseline mode
 Restores original signal....**Tab** key resets to original signal
 and modes
 Baseline subtraction.....**Backspace**, then click baseline at
 multiple points
 Restore background.....**** to cancel previous background
 subtraction
 Invert signal.....**-** Invert (negate) the signal (flip + / -)
 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 N
 Interpolate signal.....**i** Interpolate (re-sample) to N points
 Print keyboard commands.....**K** prints this list
 Print signal report.....**Q** prints signal info and current settings
 Print isignal arguments.....**W** prints isignal (current arguments)
 Save output to disk.....**O** as .mat file with processed signal
 matrix and (in spectrum mode) the
 frequency spectrum.
 Play signal as sound.....**Spacebar** or **Shift-P** Play upper panel
 segment through computer sound system
 Set sound sample rate.....**Shift-R** for the Shift-P command.
 Switch to ipf.m.....**Shift-Ctrl-F** Transfer current signal to
 Interactive Peak **Fitter**
 Switch to iPeak.....**Shift-Ctrl-P** Transfer current signal to
 Interactive Peak **Detector**

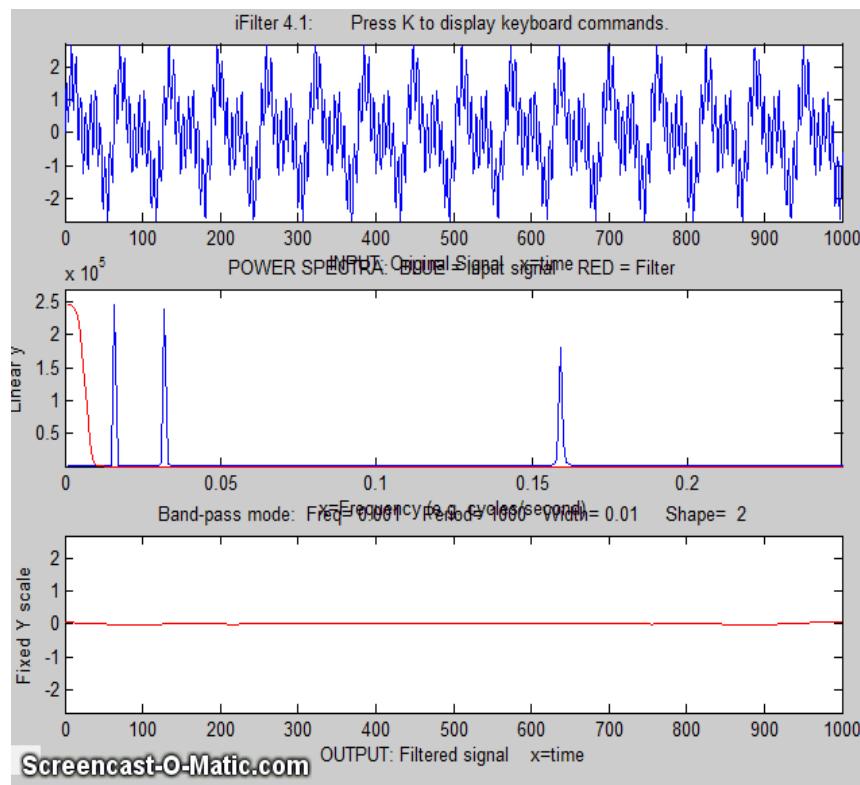
[ProcessSignal](#), a Matlab/Octave command-line function that performs smoothing and differentiation on the time-series data set x,y (column or row vectors). Type "help ProcessSignal". It returns the processed signal as a vector that has the same shape as x, regardless of the shape of y. The syntax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

iFilter, keyboard-operated interactive Fourier filter

iFilter is a keyboard-operated interactive Fourier filter function (page 107) 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](#) or in 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 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)
```



Example 1: Click figure above to see animation

Periodic waveform with 2 frequency components at 60 and 440 Hz.

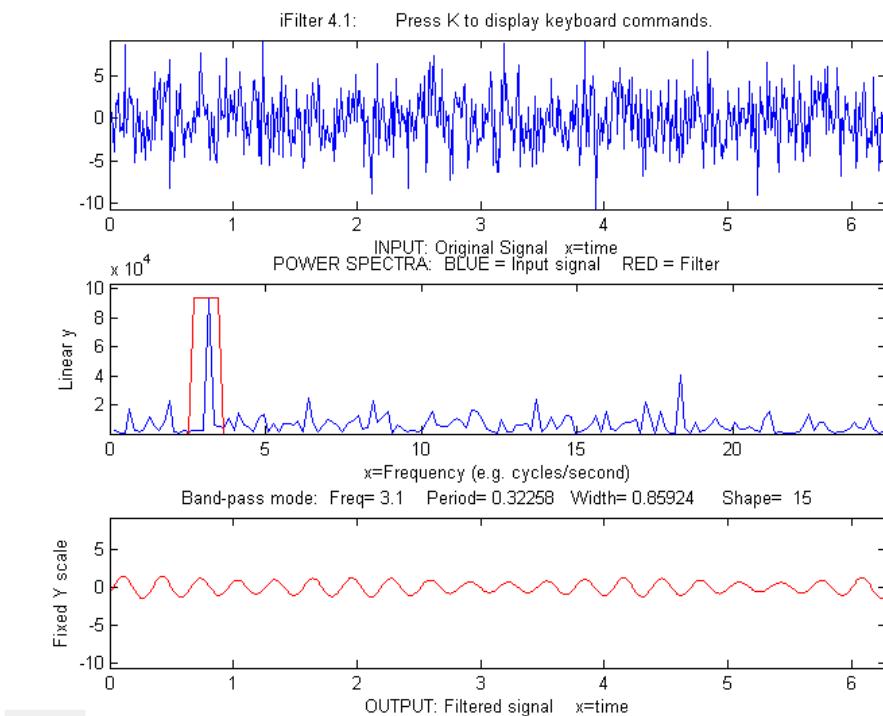
```
x=[0:.001:2*pi];  
y=sin(60.*x.*2.*pi)+2.*sin(440.*x.*2.*pi);  
ifilter(x,y);
```

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.

```
x=[0:.01:2*pi]';
y=sin(20*x)+3.*randn(size(x));
ifilter(x,y,3.1,0.85924,15,1,'Band-pass');
```

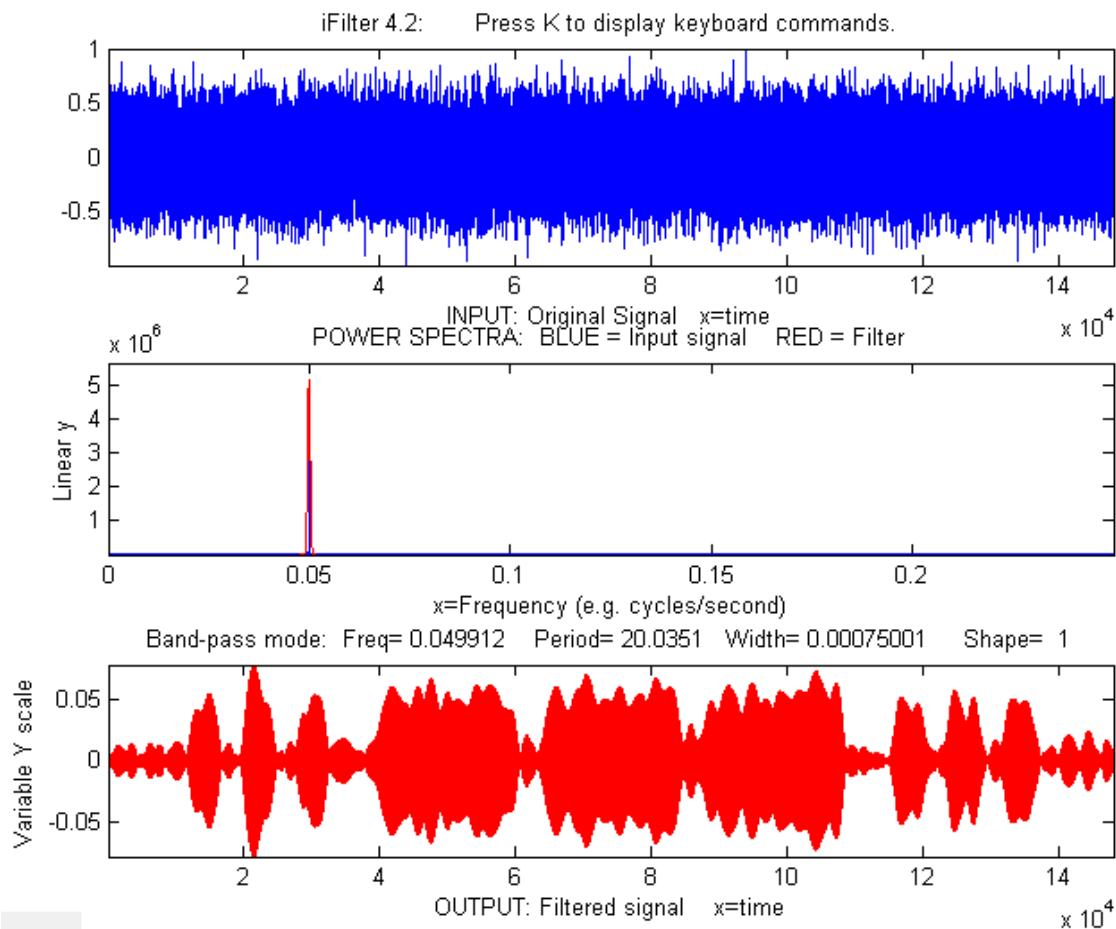


Example 4: Square wave with band-pass vs Comb pass filter

```
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');
```

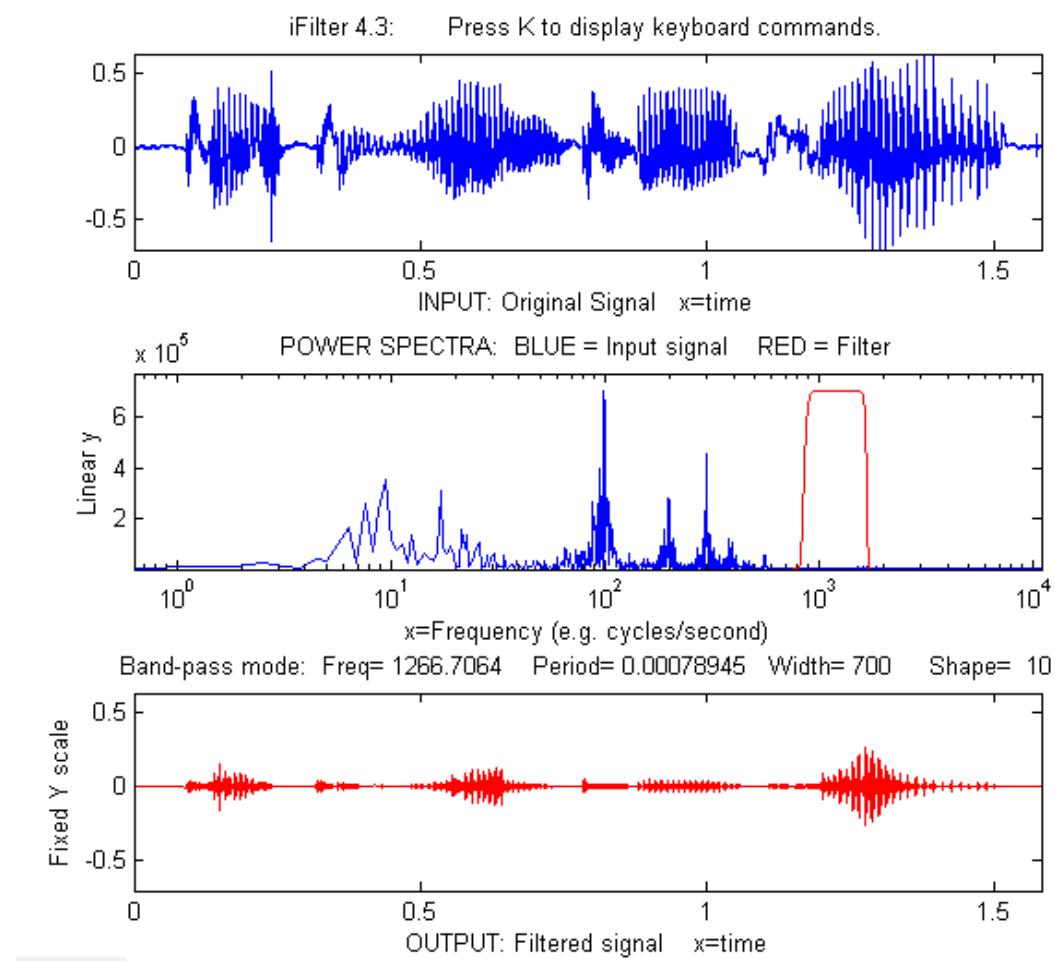
Example 5: [MorseCode.m](#) uses iFilter to demonstrate the abilities and limitations of Fourier filtering. It creates a pulsed [fixed frequency sine wave](#) that [spells out “SOS” in Morse code](#) (dit-dit-dit/dah-dah-dah/dit-dit-dit), adds random white noise so that the [SNR is very poor](#) (about 0.1 in this example). The white noise has a frequency spectrum that is [spread out over the entire range of frequencies](#); 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 1.5825 sec duration audio recording of the spoken phrase "Testing, one, two, three", previously recorded at 44001 Hz and saved in both WAV format ([download link](#)) and in ".mat" format ([download link](#)). This data file is loaded into *iFilter*, which is initially 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. That passband misses most of the 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. 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.



iFilter 4.3 keyboard controls (if the figure window is not topmost, click on it first):

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**

Matlab/Octave Command-line Peak Fitters

A Matlab peak fitting program for time-series signals, which uses an unconstrained non-linear optimization algorithm (page 163) 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 **9.2**.

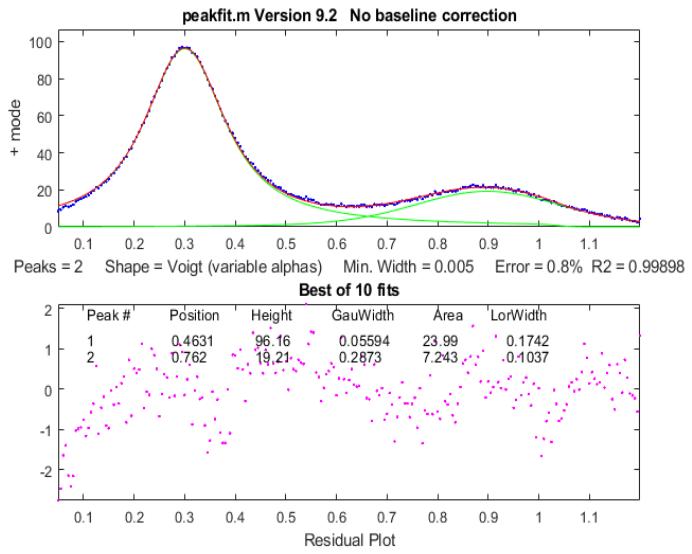
(2) a keypress operated interactive version (**ipf.m**, page 361) for Matlab only. The current version number is 13.

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 306); 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 369 ; custom peak shapes can be added (see page 380). See pages 376 and 381 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 or download, click [peakfit.m](#)). It takes data in the form of a 2 x 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, AUTOZERO, 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, and a test script ([testpeakfit.m](#) or [autotestpeakfit.m](#)) that runs all the examples sequentially to test for proper operation. For a discussion of the accuracy and precision of peak parameter measurement using peakfit.m, click [here](#).

The peakfit.m functionality can also be accessed by the keypress-operated interactive functions [ipf](#) (page 361), [iPeak](#) (page 361), and [iSignal](#) (page 323).

Version 9.2: June, 2018. Modified Voigt peak shapes to report separate Gaussian and Lorentzian widths as the 4th and 6th columns of FitResults. See **Example 21** below. Previous version m 9.1 (January, 2018) added peak shape **50** which implements the multilinear regression ("classical least squares") method for cases in which the peak shapes, position, 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 $y = maxy * (1 + (miny - 1) / (1 + (x / ip)^{\text{slope}}))$; **46**=quadratic baseline (see Example 38); **47**=blackbody emission; **48**=equal-width exponential pulse; **50**=multilinear regression (known peak positions and widths). The function [ShapeDemo](#) demonstrates most of the basic peak shapes (graphic on page 369) 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 values can usually be approximate average values based on your experience. If you leave this off, or set start=0, the program will generate its own start values (which is often good enough). See examples 22, 28, 40, and 42 for situation where specifying the start values is useful.

```
peakfit(signal, center, window, NumPeaks, peakshape, extra, NumTrials, start, autozero)
```

As above, but "autozero" sets the baseline correction mode in the last argument: **autozero**=0 (default) does *not* subtract baseline from data segment; **autozero**=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); **autozero**=2, like mode 1 except that it computes a *quadratic* curved baseline; **autozero**=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, 2)
```

Use zeros as placeholders to use the default values of input arguments. In this case, **autozero** is set to 2, but all others are the default values.

```
peakfit(signal, center, window, NumPeaks, peakshape, extra, NumTrials, start, autozero, 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 (multilinear 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, autozero, 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, autozero, 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, autozero, 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, autozero, 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 that 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 134.

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:** a matrix containing bootstrap standard deviations and interquartile ranges for each peak parameter of each peak in the fit (page 134).

Examples

Note: test script [testpeakfit.m](#) runs all the following examples automatically; just press Enter to continue to the next one. (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  
FitResults =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 auto-zero mode (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 peaks 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 174 - 182.

Example 12. Baseline correction. Demonstrations of the four autozero modes, for a single Gaussian on large baseline, with position=10, height=1, and width=1.66. The autozero mode is specified by the 9th input argument (0,1,2, or 3).

Autozero=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)
FitResults =
    1          10          1.8561          3.612          5.7641
FitError = 5.387
```

Autozero=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)
FitResults =
    1          9.9984          0.96161          1.5586          1.5914
FitError = 1.9801
baseline = 0.0012608          1.0376
```

Autozero=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)
FitResults =
    1          9.9996          0.81762          1.4379          1.2501
FitError = 1.8205
baseline = -0.046619          0.9327          -3.469
```

Autozero=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)
FitResults =
    1          10          1.0001          1.6653          1.7645
FitError = 0.0037056
baseline = 0.99985
```

In the following example, the baseline is strongly sloped, but straight. In that case the most accurate result is obtained by using a 2-peak fit, fitting the baseline with a variable-slope straight line (**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)
FitResults =
```

	1	10	1	1.6651	1.7642
--	---	----	---	--------	--------

```

2           4.485       0.22297       0.05       40.045
FitError = 0.093

```

In this example, the baseline is *curved*, so you may be able to get good results with **autozero=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)
FitResults =
    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 111):

```

>> 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 effected by exponential broadening, which does not by itself change the true peak areas, but does shift peak positions and increases 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 may improve 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)
ans=      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
```

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)
FitResults =
          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)
FitResults =
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 9.2)

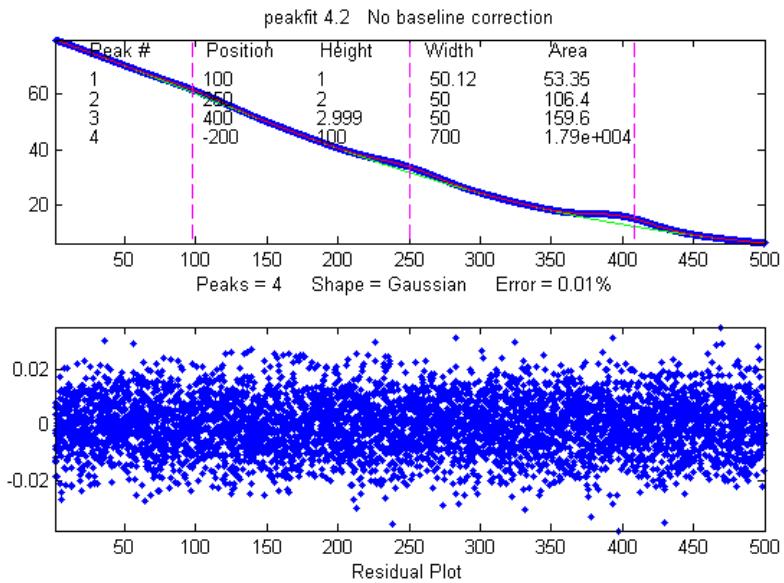
```

>> disp('Peak #          Position          Height          Gau-
Width          Area          LorWidth')
>> [FitResults,FitError]=peakfit(humps(0:.01:2),60,120,2,30,1.7,5,0)

FitResults =
%   1     48.555      95.09      7.1121      2413.8      16.485
%   2     78.173      19.602     20.377      769.04      19.775
GOF = 0.72829      0.99915

```

Example 22. [peakfitdemob.m](#). Illustrated on the right. Measurement of three weak Gaussian peaks at $x=100, 250, 400$, superimposed in a very strong curved baseline plus noise. 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)
FitResults =
    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])
FitResults =
    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 = RMS noise = 1.

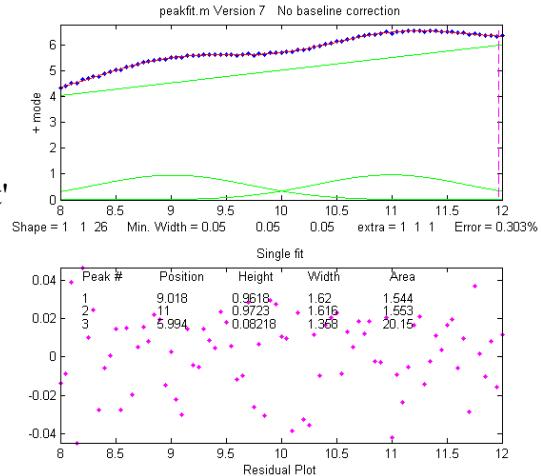
```
>> x=[-10:.05:10];y=exp(-(x).^2)+randn(size(x));
>> P=peakfit([x;y],0,0,1,1,0,10);
```

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)
FitResults =
    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 we 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],start)
```



See example 38 (page 357) 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). In version 9.2, this returns separate Gaussian and Lorentzian widths:

```
x=1:.1:30; y=modelpeaks2(x,[13 13],[1 1],[10 20],[3 3],[20 80]);
disp('Peak#      Position      Height      GauWidth      Area      LorWidth');
[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;
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 time constant exponentially-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)
```

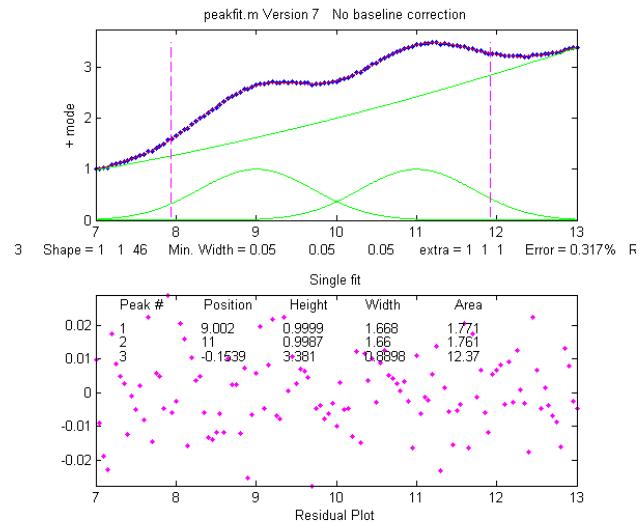
Example 37: 3-parameter logistic (Gompertz), shape 43. (Version 7.9 and above only). Parameters labeled Bo, Kh, and L. FitResults extended to 6 columns.

```
>> 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)
```

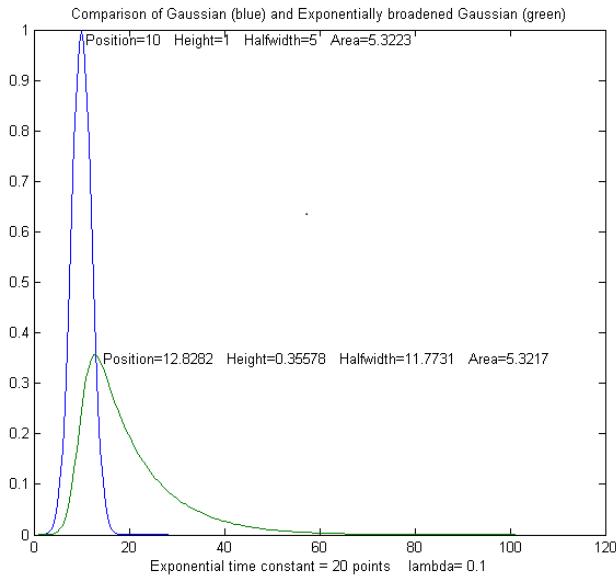
Example 38: Shape 46, 'quadslope'. Two overlapping Gaussians (position=9,11; heights=1; widths=1.666) on a curved baseline, using a 3-peak fit with peakshape=[1 1 46], default NumTrials and start.

```
>> 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:

```
[FitResults,GOF]=peakfit([x;y],0,0,4,1,0,10)
```

Second attempt specifying approximate start values in the 8th input argument gives almost perfect 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 R2 remains close to 1.00.

How do you find the right 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's 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 actually 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 the downloadable 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 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) ;
```

This can be *nested* into a single line of code (which is shorter but harder to read):

```
HeightOfClosestPeak=FitResults (val2ind(FitResults (:,2),TargetPosition),3) ;
```

For an example of this use in a practical application, see [RandomWalkBaseline.m](#).

Demonstration script for peakfit.m

[DemoPeakFit.m](#) is a demonstration script for peakfit.m. It generates an overlapping Gaussian peak signal, adds normally-distributed 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:

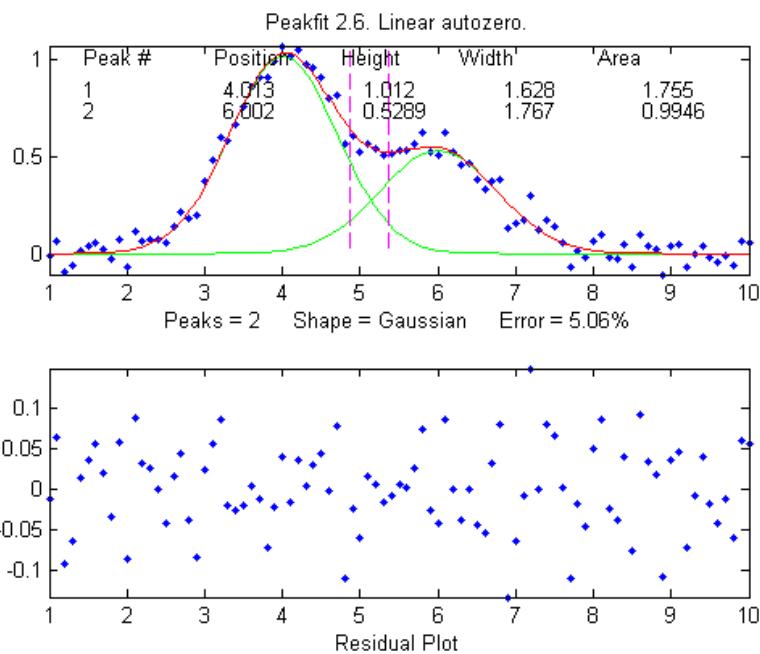
Position	Height	Width	Area
0.048404	0.07906	0.12684	0.19799
0.023986	0.38235	-0.36158	-0.067655

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 simple script that demonstrates how to apply multiple curve fits to a signal that is changing with time. The signal contains two noisy Gaussian peaks (similar to the illustration at 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 $100 \times 2 \times 5$ 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's 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 306.



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 alone, 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

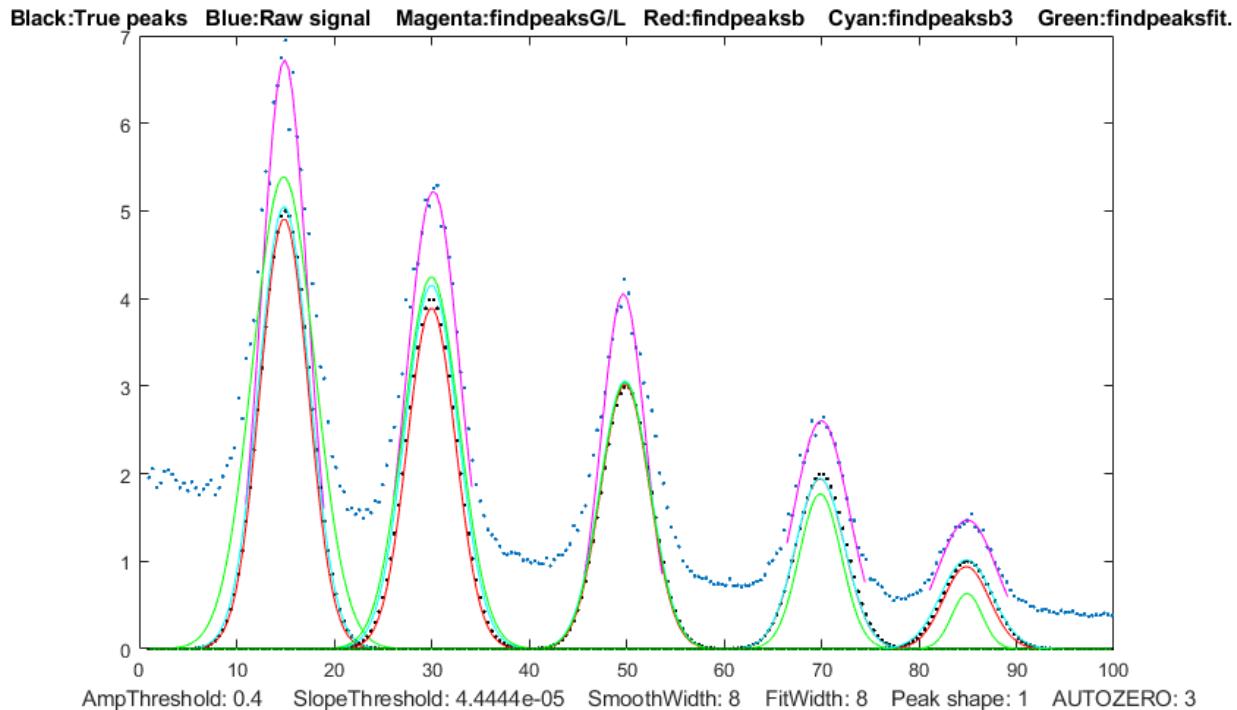
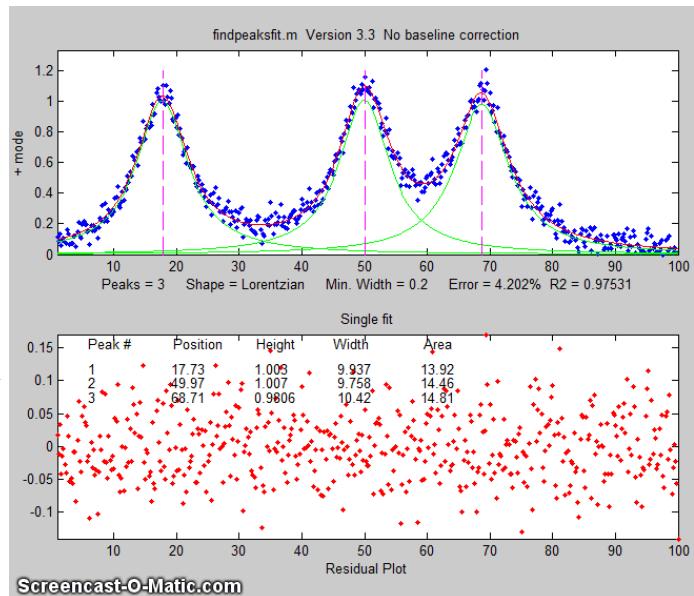
```
function [P,FitResults,LowestError,residuals,xi,yi] = findpeaksfit(x, y,
SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, smoothtype,
peakshape, extra, NumTrials, autozero, fixedparameters, plots)
```

The first seven input arguments are exactly the same as for the [findpeaks....](#) functions (page 200); 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 343); if you have been using peakfit.m or [ipf.m](#) to fit peaks in your signals, you can use those same input argument values for findpeaksfit.m. Type "help findpeaksfit" for

more information. This function is included in the [ipf13.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	0.13528
findpeaksb	0.28246	2.7755	3.4747	1.739
findpeaksb3	0.28693	2.2531	2.9951	1.7763
findpeaksfit	0.341892	12.7095	18.3436	0.98308

5 peaks found.

Interactive Peak Fitter (ipf.m)

[ipf.m](#) (Version 13.2, December 2018) is a Matlab-only version of the peak fitter for x,y data that uses keyboard commands and the mouse cursor. Animated instructions on its use are available online at <https://terpconnect.umd.edu/~toh/spectrum/ifpinstructions.html>. It is a self-contained Matlab function, in a single m-file. The interactive keypress operation will also work if you run [Matlab Online in a web browser](#), but it does not work on [Matlab Mobile](#) or in Octave. The flexible input syntax allows you to specify the data as separate x,y vectors or as a 2xn 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. 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);
```

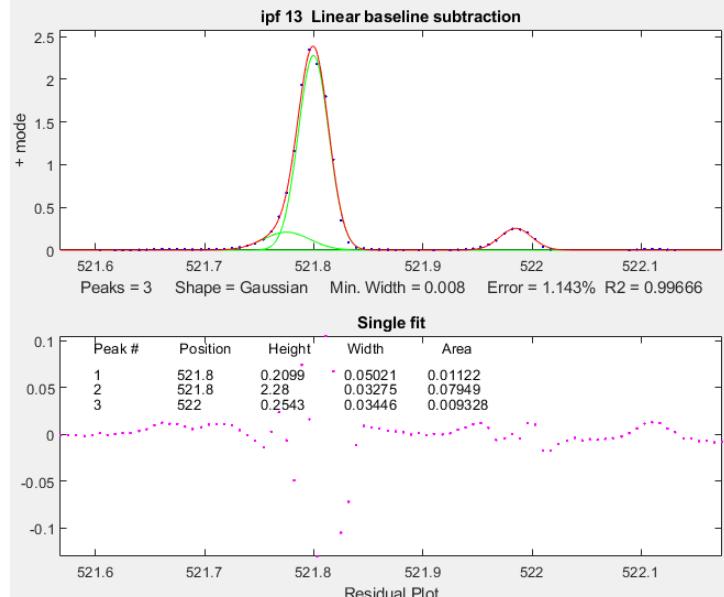
3 input arguments:

```
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 (figure on the right), 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 don't 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.



ping 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 don't 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.

Recent version history. Version 13.2: December, 2018. Modified "d" key to save model data to disc as SavedModel.mat. Version 13, November 2017: added new peak shapes, 24 total 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 flat-top peaks). 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, August 2014 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 path; Version 10.4, June, 2014. 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 autozero mode 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 (last fit result reported) includes only the area within the upper window, so it does vary. (But 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];
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 13.2):

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
 \therefore 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)
 u exponential pUlse
 $y=\exp(-\tau_1.*x).*(1-\exp(-\tau_2.*x))$
 Shift-U Alpha: $y=(x-\tau_2)./$
 $\tau_1.*\exp(1-(x-\tau_2)./\tau_1)$
 s Up Sigmoid (logistic function):
 $y=.5+.5*\text{erf}((x-\tau_1)/\sqrt{2*\tau_2})$
 Shift-D Down Sigmoid
 $y=.5-.5*\text{erf}((x-\tau_1)/\sqrt{2*\tau_2})$
 ~` 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 autozero mode.....**t** selects none, linear,
 quadratic, or flat baseline mode

+ or +/- peak mode.....**+=** Flips between + peaks only and
 +/- peak mode

Invert (negate) signal.....**Shift-N**

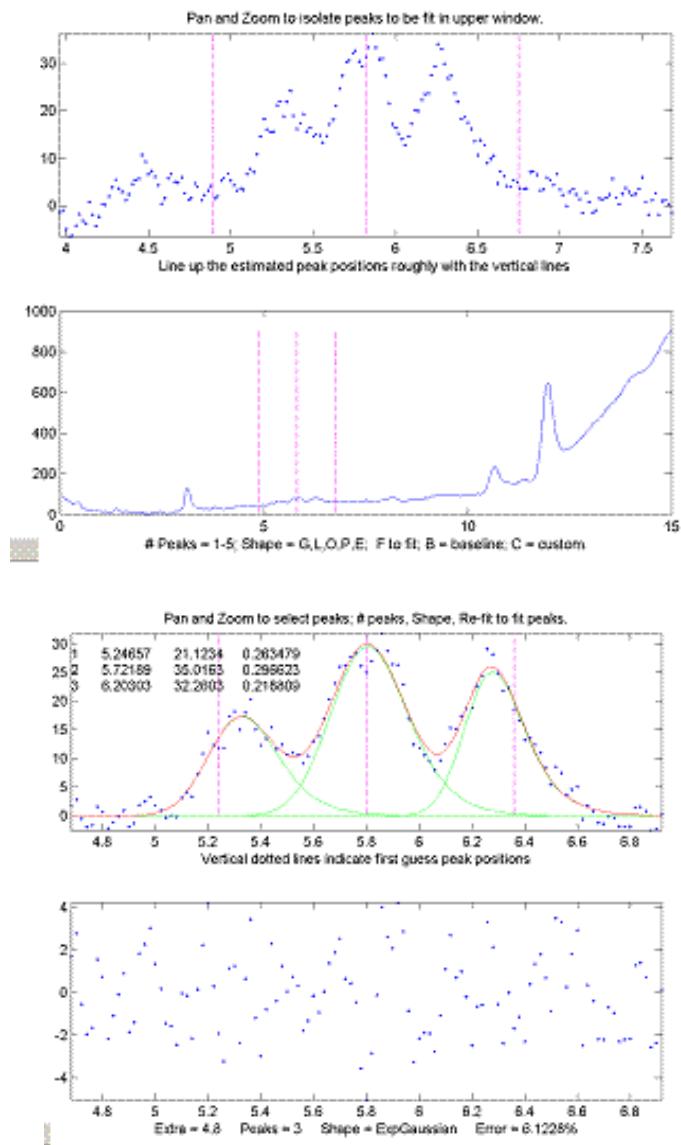
Toggle log y mode OFF/ON.....**m** Log mode plots and fits
 log(model) to log(y).

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 369)

Practical examples with experimental data:

1. Fitting weak and noisy chromatographic peaks with exponentially modified Gaussians.



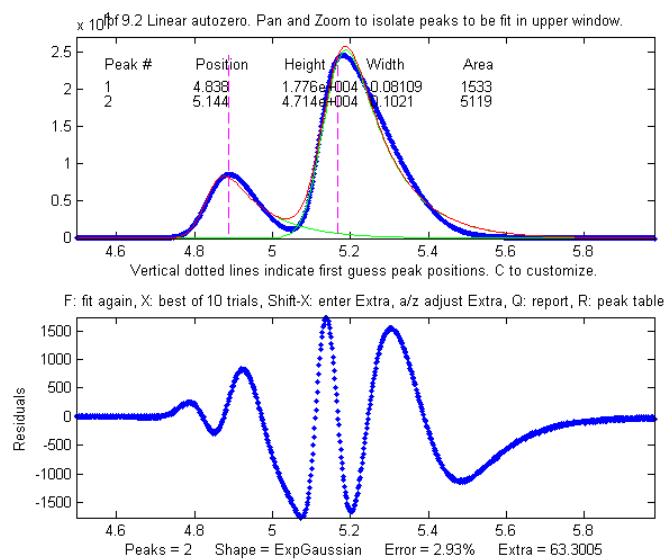
a. In this example, pan and zoom controls are used to isolate a segment of a chromatogram that contains three very weak peaks near 5.8 minutes. The lower plot shows the whole chromatogram and the upper plot shows the segment. Only the peaks in that segment are subjected to the fit. Pan and zoom are adjusted so that the signal returns to the local baseline at the ends of the segment

b. Pressing **T** selects autozero mode 1, causing the program to subtract a linear baseline 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 134) indicates that the relative standard deviation of the measured peak heights is predicted to be less than 3%.

2. Measuring the areas of overlapping peaks. In this example, a sample of room air is analyzed by gas chromatography ([data source](#)). The resulting chromatogram (next page, right) shows two slightly overlapping 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 111) 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 (which a commonly-encountered peak shape in chromatography)

gives a fairly 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.

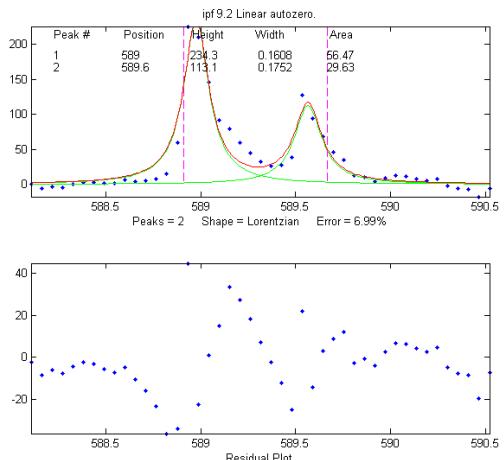
```
Percent Fitting Error =
2.9318% Elapsed time = 11.5251 sec.
Peak# Position Height Width
1      4.8385    17762   0.081094
2      5.1439    47142   0.10205
```



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	56.473
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 *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 (perhaps by [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 134) 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).

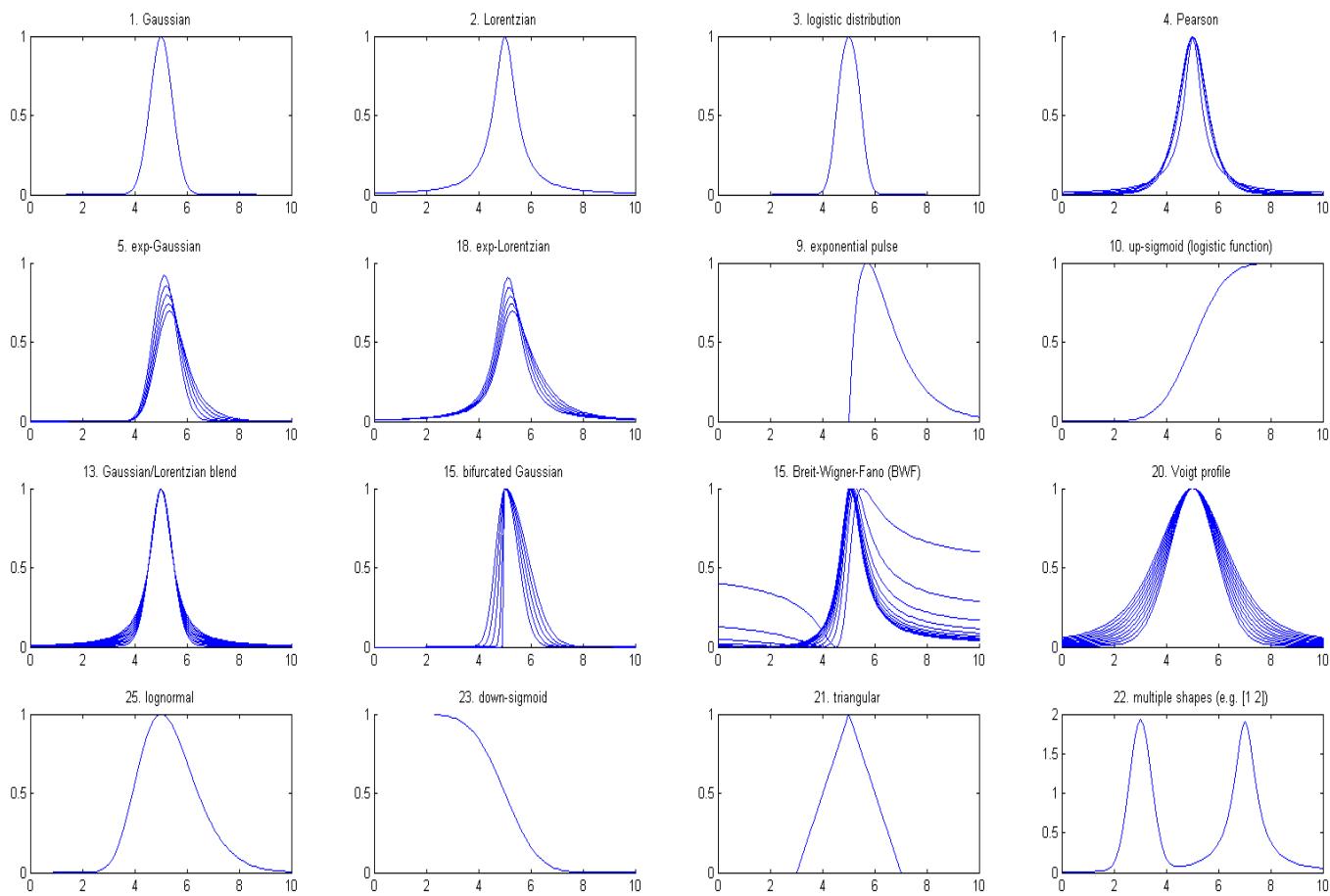
4. How many peaks to model? In the second and third examples above, the number of peaks in the model was suggested by the data and by the expectations of each of those experiments (two major gasses in air; sodium has a well-known doublet at that wavelength). In the first example, no *a priori* expectation of number of peaks was available, but the data suggested three obvious peaks, and the residuals were random and unstructured with a 3-peak model, suggesting that no additional models peaks were needed. In many cases, however, the number of model peaks is not so clearly indicated. I a previously described example on page 182, 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's impossible to know which are the "real peaks" and what is just "fitting the noise".

Operating instructions for ipf.m (version 13.2).

[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 peaks shapes are



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 a 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**.

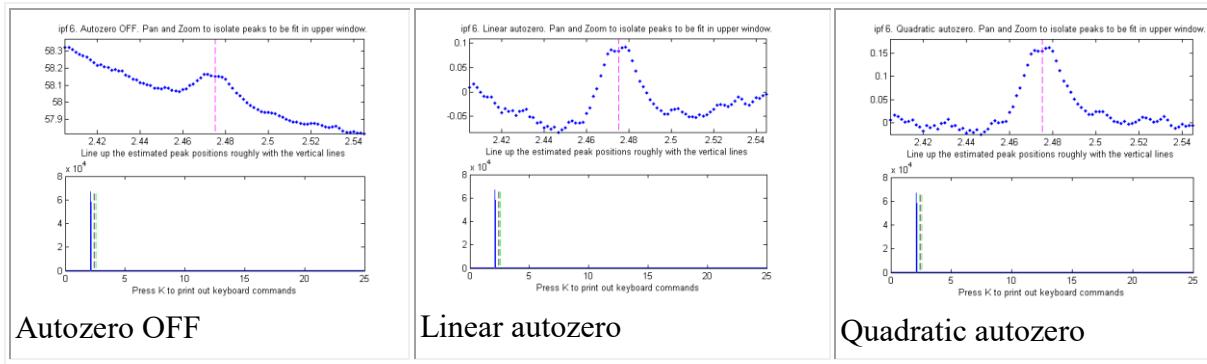
7. Press **F** to initiate the curve-fitting calculation. Each time you press **F**, another fit of the selected model to the data is performed with slightly different starting positions, so that you can judge the stability of the fit with respect to changes in starting first guesses. Keep your eye on 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 or 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 #13, 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 = 0.19741 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 *four modes*: *none*, *linear*, *quadratic*, and *flat* baseline correction. 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. Use the quadratic baseline correction if the baseline is curved, as in these examples:



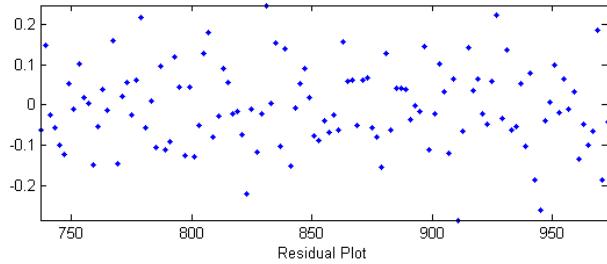
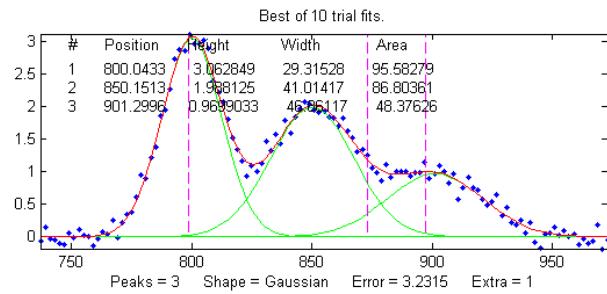
10. If you prefer to set the baseline manually, press the **B** key, then click on the baseline to the LEFT the peak(s), then click on the baseline to the RIGHT the peak(s). The new baseline will be subtracted and the fit re-calculated. (The new baseline remains in effect until you use the pan or zoom controls). Alternatively, you may use the 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 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's 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 is off-scale ([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 a 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's 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. 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 if you press **X** again and again, 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 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 you 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 134). 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 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...`). The program displays the results a table in the command window. For example, for a three-peak fit (to the same three peaks used by the `Demoipf` demonstration script described in the next section) and using 10 as the number of trials:

```
Number of fit trials per bootstrap sample (0 to cancel): 10
Computing.... May take several minutes.
```

Peak #1	Position	Height	Width	Area
Mean:	800.5387	2.969539	31.0374	98.10405
STD:	0.20336	0.02848	0.5061	1.2732
STD(IQR) :	0.21933	0.027387	0.5218	1.1555
% RSD:	0.025402	0.95908	1.6309	1.2978
% RSD (IQR) :	0.027398	0.94226	1.6812	1.1778
Peak #2	Position	Height	Width	Area
Mean:	850.0491	1.961368	36.4809	75.9868
STD:	6.4458	0.14622	3.0657	4.7596
STD(IQR) :	0.52358	0.058845	1.4303	3.9161
% RSD:	0.73828	7.2549	8.2035	6.2637
% RSD (IQR) :	0.71594	7.08002	7.9205	6.1537
etc				

20. If the RSD and the RSD (IQR) 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" (obviously 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.

Notice that the RSD of the peak *position is best* (lowest), followed by height and width and area. This is a typical pattern, seen before. 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.

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 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 sample 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's best to try to increase the fit stability by choosing a better model (e.g. using an [equal-width or fixed-width model](#), or a fixed-position shape, if appropriate), adjusting the fitted range (pan and zoom keys), 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. Note: it's 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 means 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 way 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 much higher standard deviation of its peak height when the **V** key is used.

19. **Shift-o** fits a simple polynomial (linear, quadratic, cubic, etc.) to the segment of the signal displayed 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 a maximum height), you can make the program ignore the saturated points by pressing **Shift-M** and entering the maximum Y values to keep.

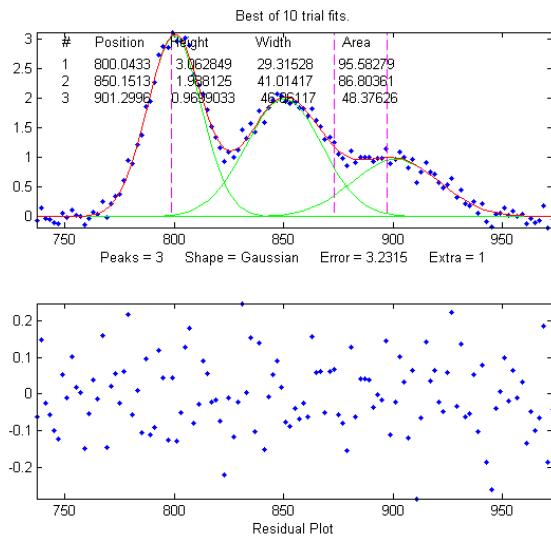
Y values above this limit will simply be ignored; peaks below this limit will be fit as usual.

21. To constrain the model to peaks above a certain width, 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 this m-file, right-click on the links, select **Save Link As...**, and click **Save**. To run it, place both [ipf.m](#) and [Demoipf](#) in the Matlab 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 in 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 fit 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 are excellent for peak position, good for peak height, and least good for peak width and area. It's 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. We 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 strong 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, (e.g. a small laptop with 1.6 GHz, dual core Athlon CPU with 4 Gbytes RAM, vs a desktop 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](#). The 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 Windows 10, 64 bit, 3 GHz core i5, with 12 GBytes RAM computer, using Matlab 7.8 R2009a, Matlab 2017b Home, Matlab Online R2018b in Chrome, Matlab Mobile on an iPad), and Octave 3.6.4. 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.

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 2xn matrix, with x in the first row or column. Or it may be that first guesses ("start") need to be provided for that particular fit.
2. It's 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 Guardians, Lorentzian, etc. (see page 174). It's worth spending some time finding and verifying a suitable function for your data. In particular, if the peak widths of each group of peaks is expected to be the same or nearly so, select the "equal-width" shapes; equal-width fits (available for the Gaussian and Lorentzian shapes) are faster, easier, and much more stable than regular variable-width fits. But it's 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 graphic](#)). But in fact in this case the model is *wrong*; that 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's 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 fits your data. (page 174). Using too many peaks will 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* don't 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 on the basis of 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 particular 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 has to 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, in order 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 bounces 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 also uses those best-fit values as the *starting first guesses* 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; for 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 134). [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 don't 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](mailto: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 in the **F**, **X**, **N**, and **V** keys:
 - **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 refine the fit. Takes about 10 times longer than the F key.
 - **N key:** Slightly varies the starting values and performs a 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 a number of 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:	$y = (x-sp0int) / pos * \exp(1 - (x-sp0int) / pos)$
Up Sigmoid:	$y = .5 + .5 * \text{erf}((x-tau1) / \sqrt{2*tau2}))$
Down Sigmoid	$y = .5 - .5 * \text{erf}((x-tau1) / \sqrt{2*tau2}))$
Gompertz:	$y = Bo * \exp(-\exp(Kh * \exp(1) / Bo) * (L-t) + 1))$
FourPL:	$y = 1 + (\miny - 1) / (1 + (x / ip)^{\text{slope}})$
OneMinusExp:	$y = 1 - \exp(-wid * (x-pos))$
EMG (shape 39)	$y = s * \lambda * \sqrt{\pi/2} * \exp(0.5 * (s * \lambda)^2 - \lambda * (t - mu)) * \text{erfc}((1/\sqrt{2}) * (s * \lambda - ((t - mu) / s)))$

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:

$$y = 87.67 \cdot \exp\left(-\left(\frac{(x-0.30284)}{(0.60056120439323 \cdot 0.23732)}\right)^2\right)$$

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

The equation would be:

$$y = 96.9405 \cdot \left(\frac{1}{1 + ((x - 0.3012) / (0.5 \cdot 0.1843))^2} \right) + 21.1237 \cdot \left(\frac{1}{1 + ((x - 0.8931) / (0.5 \cdot 0.2488))^2} \right)$$

It's 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 $y = \text{peak 1 (Lorentzian)} + \text{peak 2 (Gaussian)}$:

$$y = 97.5771 \cdot \left(\frac{1}{1 + ((x - 0.3018) / (0.5 \cdot 0.1876))^2} \right) + 18.8877 \cdot \exp\left(-\left(\frac{(x - 0.8953)}{(0.60056120439323 \cdot 0.3341)}\right)^2\right)$$

How to add a new peak shape to peakfit.m or ipf.m

It's easier than you think to add your own custom peak shape to peakfit.m or ipf.m, assuming that 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. Pick a shape to sacrifice that has the *same number of variables and constraints* as 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 (Shift-R), 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's 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 *name* of the function and the *mathematics* of the assignment statement (e.g. $g = 1 - (1./wid) .* \text{abs}(x-pos);$). 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" un-checked* 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).

That's it! Your new shape will now be shape 21 (or whatever was the shape number of the old shape you sacrificed) and, in ipf.m, will be activated by the same keystroke used by the old shape (e.g. Shift-T for the triangular, key number 84).

If you wish, you can change the keystroke assignment in ipf.m; first, find a key or Shift-Key that is not yet assigned (and which gives an "UnassignedKey" error statement when you press it 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. But it's simpler just to use the old key.

Which to use? *peakfit*, *ipf*, *findpeaks...*, *iPeak*, or *iSignal*?

I designed *iPeak* (page 361), *iSignal* (page 323), *peakfit* (page 343), and *ipf* (page 361) each with a different emphasis, although there is some overlap in their functions. Briefly, iSignal combines a number of basic functions, including smoothing, differentiation, spectrum analysis, etc.; findpeaks... and iPeak focus on finding multiple peaks in large signals; and 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 only in Matlab, whereas findpeaks..., peakfit, and are *command-line* functions and they work in Matlab and Octave. The interactive functions are better for exploration and trying out different setting directly, 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 have several features in common.

- (a) The **K** key displays the keyboard controls for each program.
- (b) The pan and zoom keys are the four cursor keys – left and right for pan, up and down for zoom.
- (c) **Ctrl-Shift-A** selects the entire signal (that is, zooms out all the way).
- (d) **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.
- (f) All these programs use the **Shift-Ctrl-S**, **Shift-Ctrl-F**, and **Shift-Ctrl-P** keys to transfer the current signal, as a [global variable](#), to iSignal.m, ipf.m, and iPeak.m, respectively.

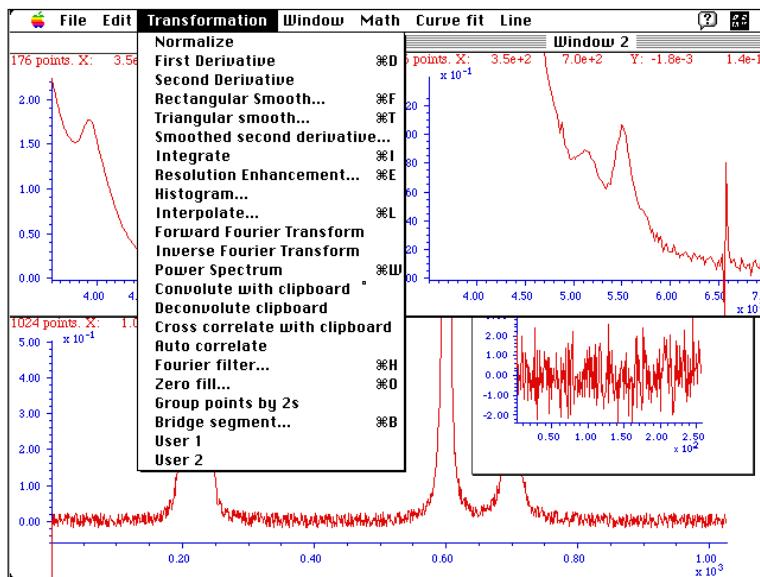
First time here? Check out these *animated Web demos* of [ipeak.m](#) and [ipf.m](#). Or download these Matlab demo functions that compare ipeak.m with peakfit.m for signals with a [few peaks](#) and signals with [many peaks](#) and that shows how to adjust ipeak to detect [broad or narrow peaks](#). 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](#). [peakfitVSSfindpeaks.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 don't 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 don't work on [Matlab Mobile](#) or in Octave.

S.P.E.C.T.R.U.M.: Simple freeware signal processing program for Macintosh OS 8.1

Some of the figures in this book are screen images from S.P.E.C.T.R.U.M. (Signal Processing for Experimental Chemistry Teaching and Research/ University of Maryland), a Macintosh program that I developed in 1989 for teaching signal processing to chemistry students. It runs only in Macintosh OS 8.1 and earlier and on Windows 7 PCs and various specific [Linux](#) distributions using the [Executor emulator](#).



SPECTRUM is designed for post-run (rather than real-time) processing of "spectral" or time-series data (y values at equally-spaced x intervals), such as spectra, chromatograms, electrochemical signals, etc. The program enhances the information content of instrument signals, for example by reducing noise, improving resolution, compensating for instrumental artifacts, testing hypotheses, and decomposing a complex signal into its component parts.

SPECTRUM was the winner of two [EDUCOM/NCRIPAL](#) national software awards in 1990, for Best Chemistry software and for Best Design.

Features

- Reads one- or two- column (y-only or x-y) text data tables with either tab or space separators.
- Displays fast, labeled plots in standard resizable windows with full x- and y-axis scale expansion and a mouse-controlled measurement cursor.
- Addition, subtraction, multiplication, and division of two signals.
- Two kinds of smoothing.
- Three kinds of differentiation.
- Integration.
- Resolution enhancement (peak sharpening).

- Interpolation
- Fused peak area measurement by perpendicular drop or tangent skim methods, with mouse-controlled setting of start and stop points (page 111).
- Fourier transformation
- Power spectra
- Fourier filtering
- Convolution and deconvolution
- Cross- and auto-correlation
- Built-in signal simulator with Gaussian and Lorentzian bands, sine wave and normally-distributed random noise.
- A host of other useful functions, including: inspect and edit points, normalize, histogram, interpolate, zero fill, group points by 2s, bridge segment, superimpose, extract subset of points, concatenate, reverse X-axis, rotate, set X axis values, reciprocal, log, ln, antilog, antiln, standard deviation, absolute value, square root.

SPECTRUM can be used both as a research tool and as an instructional aid in teaching signal processing techniques. The program and its associated tutorial was originally developed for students of analytical chemistry, but the program could be used in any field in which instrumental measurements are used: e.g. chemistry, biochemistry, physics, engineering, medical research, clinical psychology, biology, environmental and earth sciences, agricultural sciences, or materials testing.

Machine Requirements

SPECTRUM runs only on older Macintosh models running OS 7 or 8, minimum 1 MByte RAM, any standard printer. Color screen desirable. SPECTRUM has been tested on most Macintosh models and on all versions of the operating system through OS 8.1. No PC version or more recent Mac version is available or planned, but if you have some older model Macs laying around, you might find this program useful. SPECTRUM was written in Borland's Turbo Pascal in 1989. That firm has long been out of business, neither the Turbo Pascal compiler nor the executable code generated by that compiler runs on current Macs, and therefore there is no way for me to update SPECTRUM without completely re-writing it in another language.

SPECTRUM also runs on Windows 7 PCs using the [Executor emulator](#), which since 2008 has been made available as [open source](#) software.

Download links

The full version of SPECTRUM 1.1 is available as freeware, and can be downloaded from <http://terpconnect.umd.edu/~toh/spectrum/>. There are two versions:

SPECTRUM 1.1e: Signals are stored internally as *extended-precision* real variables and there is a limit of 1024 points per signal. This version performs all its calculations in extended preci-

sion and thus has the best dynamic range and the smallest numeric round-off errors. The download address of this version in HQX format is <http://terpconnect.umd.edu/~toh/spectrum/SPEC-TRUM11e.hqx>.

SPECTRUM 1.1b: Signals are stored internally as *single-precision* real variables and there is a limit of 4000 points per signal. This version is less precise in its calculations (has more numerical round-off error) than the other version, but allows signals with data more points. The download address of this version in HQX format is <http://terpconnect.umd.edu/~toh/spectrum/SPEC-TRUM11b.hqx>.

The two versions are otherwise identical.

There is also a documentation package ([located at http://terpconnect.umd.edu/~toh/spectrum/SPEC-TRUMdemo.hqx](http://terpconnect.umd.edu/~toh/spectrum/SPEC-TRUMdemo.hqx)) consisting of:

1. Reference manual. MacWrite format (Can be opened from within MacWrite, Microsoft Word, ClarisWorks, WriteNow, and most other full-featured Macintosh word processors). Explains each menu selection and describes the algorithms and mathematical formulae for each operation. The SPECTRUM Reference Manual is also available separately in in [HTML](#) and [PDF](#) format .

2. Signal processing tutorial. MacWrite format (Can be opened from within MacWrite, Microsoft Word, ClarisWorks, WriteNow, and most other full-featured Macintosh word processors). Self-guided tutorial on the applications of signal processing in analytical chemistry. This tutorial is also available on the Web at (<http://terpconnect.umd.edu/~toh/Chem498C/SignalProcessing.html>)

3. Tutorial signals: A library of prerecorded data files for use with the signal processing tutorial. These are plain decimal ASCII (tab-delimited) data files.

These files are binhex encoded: use Stuffit Expander to decode and decompress as usual. If you are downloading on a Macintosh, all this should happen completely automatically. If you are downloading on a Windows PC, shift-click on the download links above to begin the download. If you are using the ARDI Executor Mac simulator, download the "HQX" files to your C drive, launch Executor, then open the downloaded HQX files with Stuffit Expander, which is pre-loaded into the Executor Macintosh environment. Stuffit Expander will automatically decode and decompress the downloaded files. Note: Because it was developed for academic teaching application where the most modern and powerful models of computers may not be available, SPECTRUM was designed to be "lean and mean" - that is, it has a *simple* Macintosh-type user interface and very small memory and disk space requirements. It will work quite well on Macintosh models as old as the Macintosh II, and will even run on older monochrome models (with some cramping of screen space). It does not even require a math co-processor.

What SPECTRUM does not do: this program does *not* have a [peak detector](#), [multiple linear regression](#), or an iterative [non-linear curve fitter](#).

Worksheets for Analytical Calibration Curves

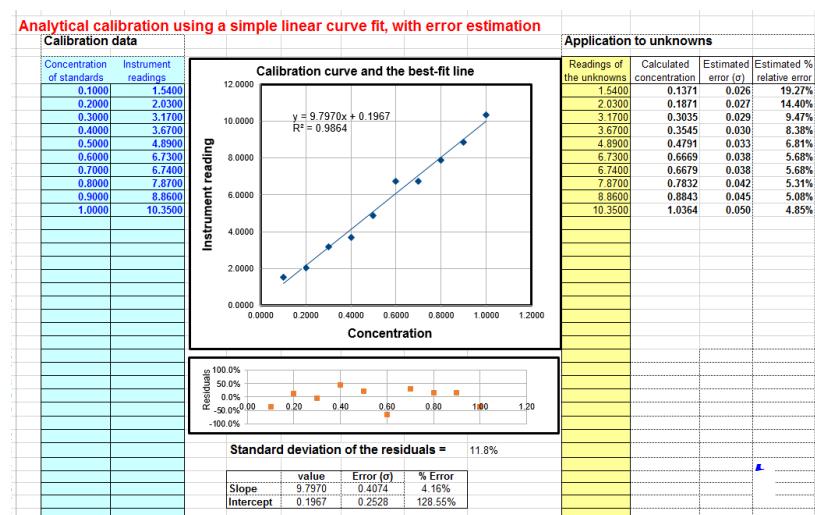
These are fill-in-the-blanks spreadsheet templates for performing the calibration curve fitting and concentration calculations for analytical methods using the calibration curve method (page 298). All you have to do is to type in (or paste in) the concentrations of the standard solutions and their instrument readings (e.g. absorbances, or whatever method you are using) and the instrument readings of the unknowns. The spreadsheet automatically plots and fits the data to a straight line, quadratic or cubic curve (page 126), 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.

Background

In analytical chemistry, the accurate quantitative measurement of the composition of samples, for example by various types of spectroscopy, 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, a series of external standard solutions is prepared and measured. A line or curve is fit to the data and the resulting equation is used to convert readings of the unknown samples into concentration. An advantage of this method is that the random errors in preparing and reading the standard solutions are averaged over several standards. Moreover, 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).

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 $A = \text{slope} * C + \text{intercept}$. 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 de-scribed and listed on page 142. You need a minimum of *two* points on the calibration curve. The concentration of unknown samples is given by $(A - \text{intercept}) / \text{slope}$ 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.

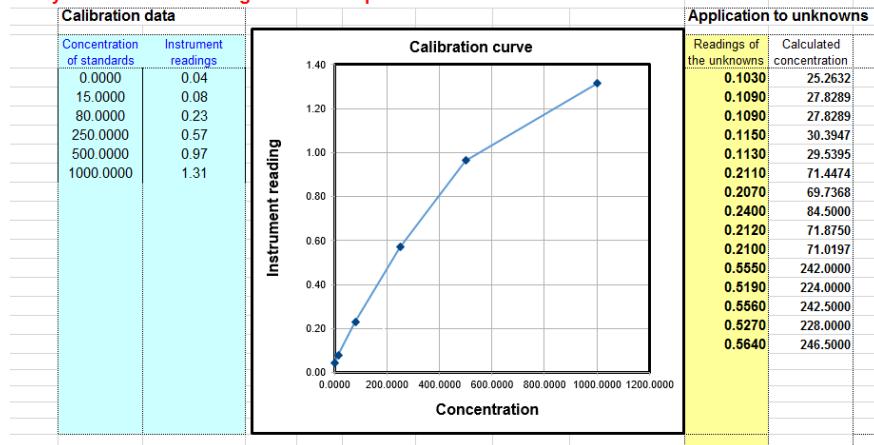


Linear interpolation calibration. In the linear interpolation method, sometimes called the bracket

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 the entire calibration set. The concentration of the sample C_x is calculated by $C_{1s} + (C_{2s} - C_{1s}) * (S_x - S_{1s}) / (S_{2s} - S_{1s})$, where S_{1x} and S_{2s} are the signal readings given by the two

standards that are just above and just below the unknown sample, C_{1s} and C_{2s} are the concentrations of those two standard solutions, and S_x is the signal given by the sample solution. This method may be useful if none of the least-squares methods are capable of fitting the entire calibration range adequately (for instance, if it contains two linear segments with different slopes). It works well enough as long as 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.

Analytical calibration using a linear interpolation bracket method



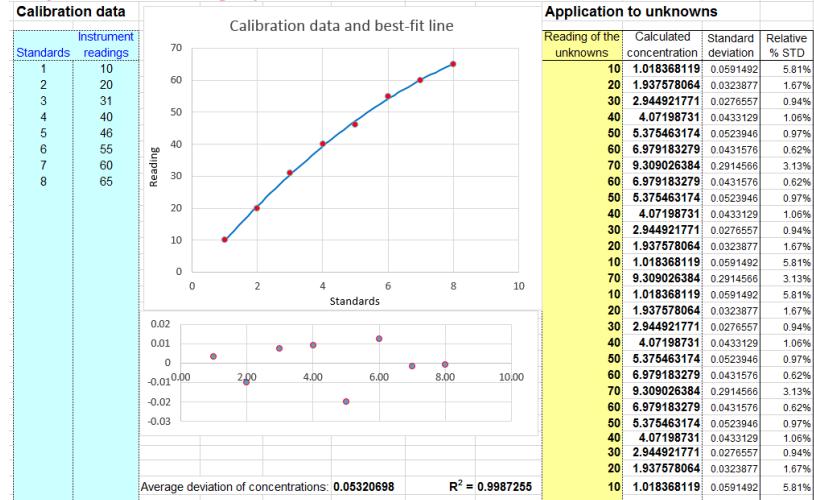
A quadratic fit of measured sig-

nal A (y-axis) vs concentra-
tion C (x-axis). The model equation
is $A = aC^2 + bC + c$. This method
can compensate for non-linearity in
the instrument response to concen-
tration. This fit uses the equations
described and listed on page 142.

You need a minimum of *three*

points on the calibration curve. The
concentration of unknown samples
is calculated by solving this equa-
tion for C using the classical "quad-

Analytical calibration using a quadratic curve fit with bootstrap error estimation



quadratic formula", namely $C = (-b + \sqrt{b^2 - 4ac}) / (2a)$, 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 the 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).

Weighted fits. A weighted curve fit applies more weight (emphasis) to some points than others, which is especially useful when the calibration curve spans a very large range of concentrations.

There are weighted versions of the linear ([CalibrationLinearWeighted.xls](#)) and quadratic ([CalibrationQuadraticWeighted.xls](#)) templates. There is also a weighted version of the drift-corrected calibration template ([CalibrationDriftingQuadraticWeighted.xls](#));

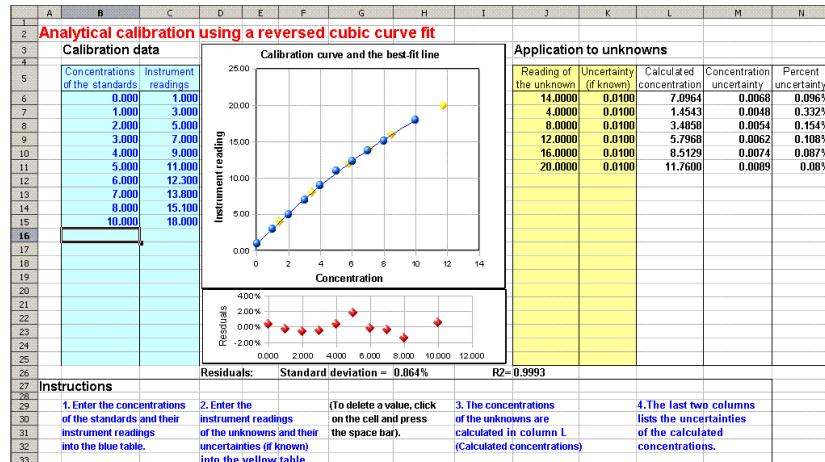
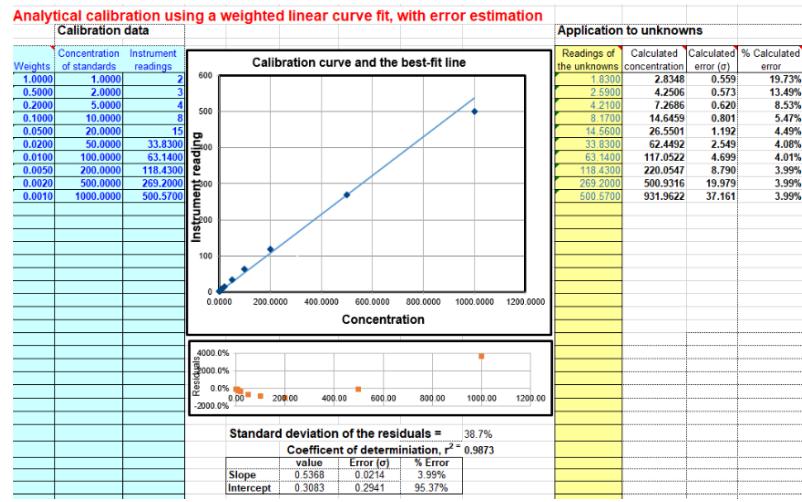
see #7 below. A weight (usually between zero and 1) for each point must be entered in Column A.

There are pre-calculated weights for $1/X$, $1/X^2$, $1/Y$, and $1/Y^2$ weighting in columns Z to AC (in the linear template) or AK to AN (in the quadratic template); you can either Copy and Paste (numbers only) these into column A, or you can enter =Z6 or =AK6 into cell A6, then "drag copy down" that cell to the last data points in column A. (Alternatively, you can enter equations into column A that calculate weights in any way you wish). If you want to disregard (ignore) one or more data points, make their weights zero. To make the calibration *unweighted*, simply make all the weights 1.0.

A reversed cubic fit of concentration C (y-axis) vs measured signal A (x-axis). The model equation

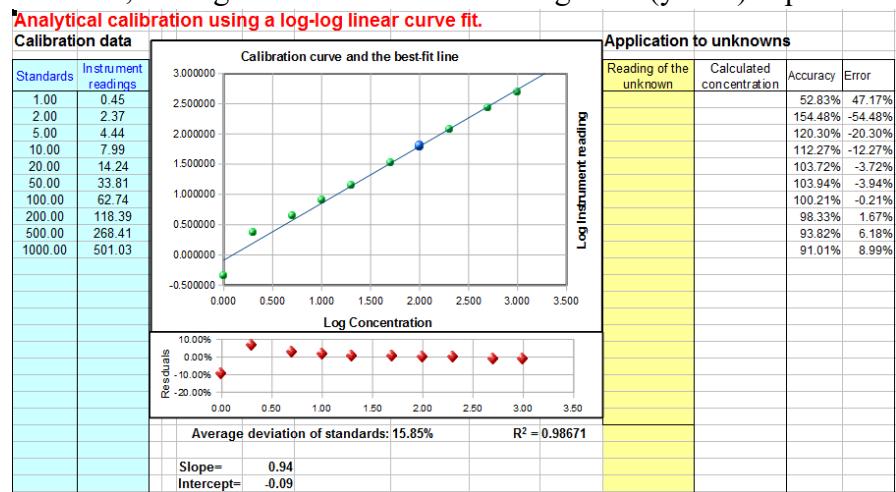
is $C = aA^3 + bA^2 + cA + d$. This method can compensate for more complex non-linearity than the quadratic fit. A "reversed fit" flips the usual order of axes, by fitting concentration as a function of measured signal. The aim is to avoid the need to [solve a cubic equation](#) when the calibration 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 [NormalIVsReversedQuadFit2.ods](#) ([Screen shot](#)), 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



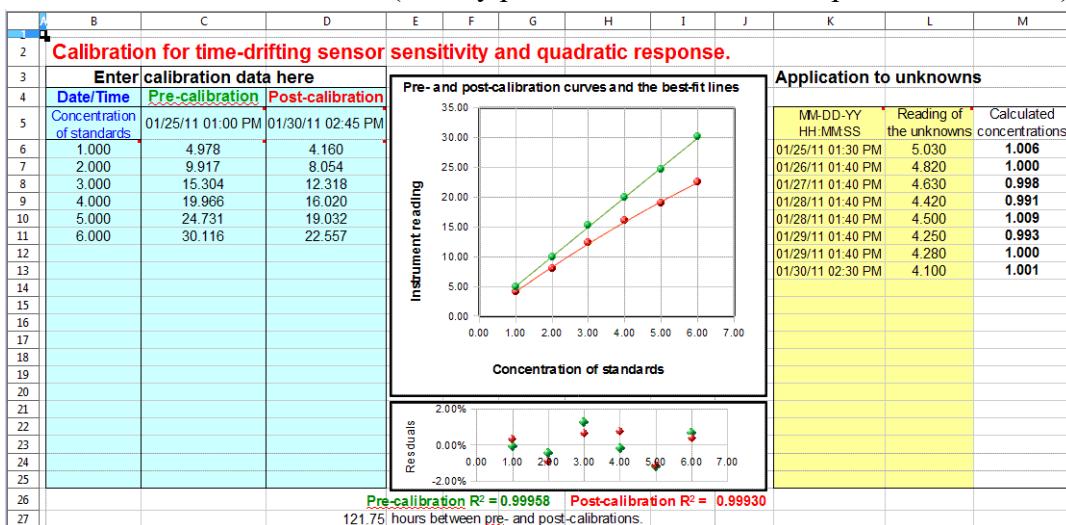
points on the calibration curve. The concentration of unknown samples is calculated directly by $aA^3+bA^2+c*A+d$, where A is the measured signal, and a , b , c , and d are the four coefficients from the cubic fit. The math is shown and explained better in the template [CalibrationCubic5Points.xls \(screen image\)](#), 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; 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 concentra-



tion. (These additional steps do not introduce any additional error, because the log and anti-log conversions can be made quickly and without significant error by the computer). Log-log calibration is well suited for data with very large range of values because it distributes the relative fitting error more evenly among the calibration points, preventing the larger calibration points to dominate and cause excessive errors in the low points. (In that sense, it is similar to a weighted fit). 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 laboratories operate under rules that [do not allow the use of nonlinear least-squares fits to calibration data](#), so the use of log-log transformation might help in such cases). However, because of the use of logarithms, the data set can 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 of 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's 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's 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 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 392.)

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 131, "[Reliability of curve fitting results](#)". The linear calibration spreadsheet (download in [Excel](#) or OpenOffice [Calc](#) format) performs a classical algebraic error-propagation calculation (page 132) 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 134). You must have a 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.

Comparison of calibration methods

In order to compare these various different methods of calibration, I will take one set of real data and

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

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 five different models, using the spreadsheet templates described above, and then the equation of the fit, solved for concentration, is 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 concentrations 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 particular data set*.

The five methods used are (1) [linear unweighted](#); (2) [linear weighted](#); (1/x weighting); (3) [quadratic unweighted](#); (4) [quadratic weighted](#) (1/x weighting); and (5) [log-log linear](#).

Comparison of concentration errors	
Unweighted linear fit	196%
Weighted linear fit	48%
Unweighted Quadratic fit	50%
Weighted Quadratic fit	6%
Log-log linear fit	20%

In the PDF version of this book, each of these is hot-linked to the corresponding spreadsheet. [Comparison-OfCalibrations.xlsx](#) summarizes the results. For this particular data set, the best method is the 1/x weighted quadratic, but that does not mean that this method will be the best in every situation. These particular 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 386).
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's 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's 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 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).

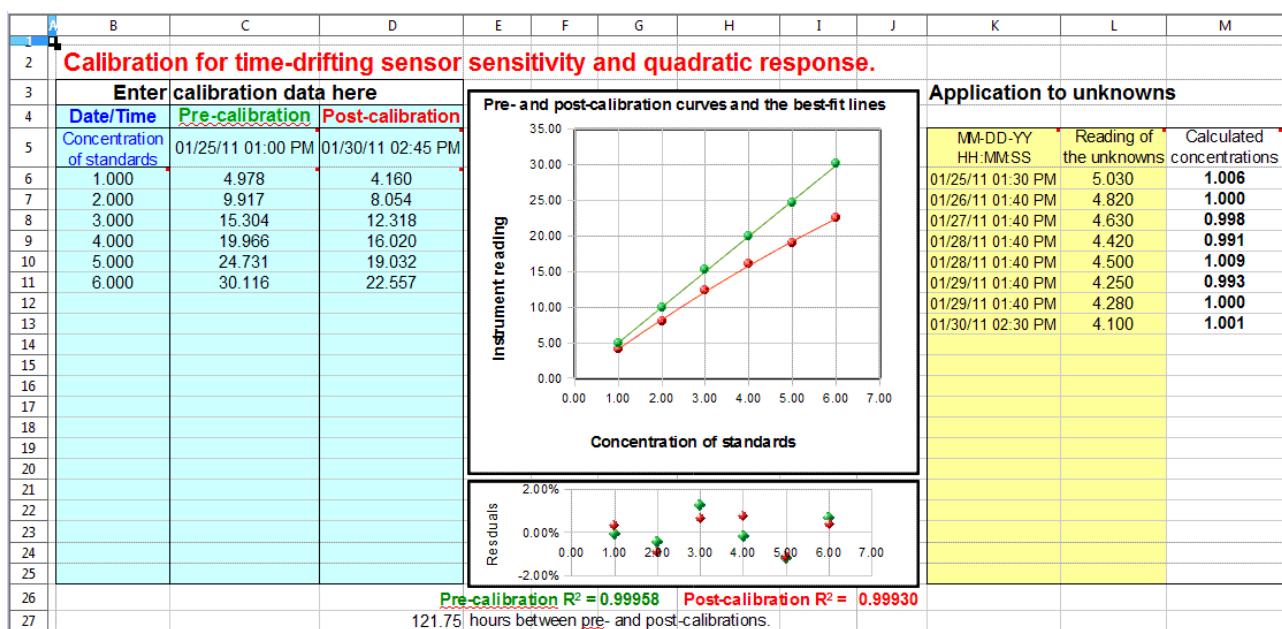
6. The linear calibration spreadsheet also calculates the coefficient of determination, R^2 , which is an indicator of the "goodness of fit", in cell C37. R^2 is 1.0000 when the fit is perfect but less than that when the fit is imperfect. The closer to 1.0000 the better.

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 the another curve fitting techniques 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 instruments readings for the first (pre-) calibration into column **C** and the date/time of that calibration into cell **C5**; enter the instruments 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 don't 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 the 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 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 actually 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 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 won't 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 don't 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's what we 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 does the slope line 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's 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.

Actually, 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 go right 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's not really recommended, because if one of your calibration points is really off by a huge error, the curve fit *will still look perfect*, and you'll have *no clue* that something's wrong. *You really must use more standards than that so that you'll 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're using a curve-fit equation, you'll still get a value of concentration calculated for

any signal reading you put in, even above the highest standard. However, it's risky to do that, because you really don't 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's best to add another standard at the high end of the calibration curve.

9. Question: *What's the difference between using a single standard vs multiple standards and a graph?*

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 as fluorescence or emission spectroscopy?*

Answer: The only difference is the units of the signal. In absorption spectroscopy you use *absorbance* (because it's 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's 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 is off a little bit, it will have no significant effect as long as 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 effected 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's 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's very likely that you'll 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's 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 stock 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 solution 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 of the accuracy of the volumetric glassware (volumetric flasks, pipettes, solution delivery devices) and on the precision of their use by the persons 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's 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 slit width or the path length, or using another instrumental method (such as a graphite furnace atomizer rather than flame atomic absorption).

19. How can I find the error in a specific quantity using least square 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's important to realize that these error computations are only estimates*, because they are based on the assumption 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'll 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'll 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 a large number of 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 $(\text{Signal} - \text{intercept}) / \text{slope}$, 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 $(\text{Signal} - \text{intercept})$ is computed by the rule for addition and subtraction; second, the error in $(\text{Signal} - \text{intercept}) / \text{slope}$ 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's important to realize that these error computations are only estimates, for the reason given in #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)?

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 pretty darned close to 1.0 for quantitative results in analytical chemistry.

Catalog of signal processing functions, scripts and spreadsheet templates

This is a complete list of all the functions, scripts, data files, and spreadsheets used in this essay, 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 the names of downloadable to download that item, 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).

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. 200 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 files 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 that 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 here, download it into your path, and try again. The script [testallfunctions.m](#) is intended to test for the existence of all/most of the functions in this collection. If it comes to a function that is not installed on your system, or if one of them does not run, it will stop with an error, alerting you of the problem. It takes about 5 minutes to run in Matlab on a contemporary PC (slower in Octave).

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](#) $y = \text{gaussian}(x, pos, wid)$
[exponentially-broadened Gaussian](#) (variable shape)
[bifurcated Gaussian](#) (variable shape)
[Flattened Gaussian](#) (variable shape)
[Clipped Gaussian](#) (variable shape)
[Lorentzian](#) (aka 'Cauchy') $y = \text{lorentzian}(x, pos, wid)$

[exponentially-broadened Lorentzian](#) (variable shape)
[Clipped Lorentzian](#) (variable shape)
[Gaussian/Lorentzian blend](#) (variable shape)
[Voigt profile](#) (variable shape)
[lognormal](#)
[logistic distribution](#) (for 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](#)
[rectanglepeak](#)
[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:
 $y=Bo \cdot \exp(-\exp((Kh \cdot \exp(1) / Bo) * (L-t)) + 1)$
[FourPL](#), 4-parameter logistic, $y = \text{maxy} + (\text{miny}-\text{maxy}) ./ (1+(x./ip).^{\text{slope}})$
[OneMinusExp](#), Asymptotic rise to flat plateau: $g = 1 - \exp(-\text{wid}.*(x-\text{pos}))$
[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 Bi-Lorentzian 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 369)

[SignalGenerator.m](#) is a script 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 here 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).

[val2ind\(x,val\)](#) Returns the index and the value of the element of vector x that is closest to "val". If more than one element is equally close, returns vectors of indices and values, Example: If $x=[1\ 2\ 4\ 3\ 5\ 9\ 6\ 4\ 5\ 3\ 1]$, then $\text{val2ind}(x,6)=7$ and $\text{val2ind}(x,5.1)=[5\ 9]$.

[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 $x=x_0$, 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 x_0 is omitted, it determines the halfwidth of the largest peak. Example: $x_0=500$; $\text{width}=100$; $x=1:1000$; $y=\exp(-1.*((x-x_0)/(0.60056120439323.*\text{width})).^2)$; $\text{halfwidth}(x,y,x_0)$. 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 has the ability to correct for some types of baseline 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's the standard deviation with outliers removed. Syntax is $b = \text{IQrange}(a)$.

[rmnan\(a\)](#), which stands for "**R**e**M**ove **N**ot **A** Number", removes NaNs ("Not a Number") 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\(a\)](#) Re**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.

[makeodd\(a\)](#): Makes the elements of vector "a" the next higher odd integers. This can be useful in computing smooth widths to insure that the smooth will not shit the maximum of peaks. For example, $\text{makeodd}([1.1\ 2\ 3\ 4.8\ 5\ 6\ 7.7\ 8\ 9]) = [1\ 3\ 3\ 5\ 5\ 7\ 9\ 9\ 9]$

[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 over-sampled signal. Mentioned on Smoothing (page 34) and iSignal (page 337)

[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 $\text{val2ind}(x,6)=7$ and $\text{val2ind}(x,5.1)=[5\ 9]$. For some examples of how this can be used, see [page 213](#).

[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\(m,threshold\)](#): "Numbers Above Threshold": 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 [try-poly\(x,y\)](#), [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 real time (Click for [animated graphic](#)).

[plotit](#), version 2, (previously named 'plotfit'), 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,data-style,fitstyle). It can also fit polynomials to the data and compute errors. [Click here](#) or type "help plotit" at the Matlab/Octave prompt for examples.

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, 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 22. 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 $\text{sqrt}((\text{std}(m1-m2).}^2)/2)$.

[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 $1.73*(\text{RAND}() - \text{RAND}()) + \text{RAND}() - \text{RAND}()$ approximates normally-distributed random numbers with zero mean and a standard deviation of 1. See page 22.

[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. A rare example where adding noise is actually beneficial. See page 270.

[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 270.

[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 22](#)
[EnsembleAverageDemo.m](#) is a Matlab/Octave script that demonstrates ensemble averaging to improved 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. [EnsembleAverageFTFGaussian.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 25](#).

[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 22](#).

[iPeakEnsembleAverageDemo.m](#) is a demonstration of 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 sig-

nal, 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 significantly improved.

[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 [Appendix J: 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 280.](#)

[AmplitudeModulation.m](#) is a Matlab/Octave script simulation of modulation and synchronous detection, demonstrating the noise reduction capability. See [page 281.](#)

[DerivativeNumericalPrecisionDemo.m](#). Self-contained function that demonstrates how the *numerical precision limits* of the computer effects 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. Smoothing with a Gaussian (three passes of a sliding-average) smooth with a smooth ratio of 0.2 removes most of the noise. With real experimental data, even the tiniest amounts of noise in the original data would be much greater than this. Used in [page 302.](#)

[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 302.](#)

[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 302.](#)

Smoothing

[fastsmooth](#), versatile function for fast data smoothing. The syntax is `SmoothY=fastsmooth(Y,w,type, ends)`. See [page 34](#). 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 a number of 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 34.

[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=pseudo Gaussian. 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 47.

[smoothdemo.m](#), self-contained function that compares the performance and speed of four types of [smooth operations](#): (1) sliding-average, (2) triangular, (3) pseudo-Gaussian (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 47.

[SmoothOptimization.m](#), script that shows why you don't 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 197.

[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 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.

[iSignal](#) (page 337) 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 "iSignal6.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.](#)

Differentiation and peak sharpening

[deriv](#), [deriv2](#), [deriv3](#), [deriv4](#), [derivxy](#) and [secdervxy](#), simple functions for computing the derivatives of time-series data. See page [65](#).

[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.

[enhance](#), peak sharpening by the even-derivative method. Syntax is Enhancedsignal= enhance(signal, factor1, factor2, SmoothWidth). See page 76. 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 [fastsmooth arguments](#). The technique is demonstrated for the 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](#). Type "help symmetrize" for examples.

[ProcessSignal](#), a Matlab/Octave command-line function that performs smoothing, See differentiation, peak sharpening, and median filtering on the time-series data set x,y (column or row vectors). Similar

to [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= ProcessSignal(x, y, DerivativeMode, w, type, ends, Sharpen, factor1, factor2, SlewRate, MedianWidth).

[derivdemo1.m](#), a function that demonstrates the basic shapes of derivatives. See page 54.

[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 54.

[derivdemo3.m](#), a function that demonstrates the effect of smoothing on the *first* derivative of a noisy signal. See page 54.

[derivdemo4.m](#), a function that demonstrates the effect of smoothing on the *second* derivative of a noisy signal. See page 54.

[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 effected 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, despite the fact that 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 65.

[iSignal](#) (page 337) is an interactive function for Matlab that performs *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. [Click here to download the ZIP file "iSignal6.zip".](#) [Click for animated example.](#)

[demoisignal.m](#) for Matlab is a self-running script that demonstrates 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 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 of these scripts pre-calculate the simulated data in line 28 and then accesses 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 an simulated signal for testing . Must have [PlotFrequencySpectrum.m](#) in the path.

[iSignalDeltaTest](#) is a Matlab/Octave script that demonstrates the *frequency response* (power spectrum) 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.

[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](#) (page 337) is a multi-purpose interactive signal processing tool (for Matlab only) 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 80 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 that 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's 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 $y_b = \text{ExpBroaden}(y, t)$. Convolutes the vector y with an exponential decay of time constant t . Mentioned on pages 30 and 361.

[GaussConvDemo.m](#), a script that demonstrates that a Gaussian of unit height, Fourier 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 2 shows an attempt to recover the original y from the convoluted result by using the `deconvgauss` function. You can optionally add noise in line 9 to show how convolution smooths the noise and how Fourier deconvolution restores it. Requires gaussian.m, peak-fit.m and deconvgauss.m in path.

[CombinedDerivativesAndSmooths.txt](#). Convolution coefficients for computing the first through fourth derivatives, with rectangular, triangular and Gaussian 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 100.

[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 268. 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.

[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 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.

[iSignal](#) (page 337) 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 asks you for the width or the time constant (in X units). [Click here to download the ZIP file "iSignal6.zip"](#)

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 107.

[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. [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).

[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, in an attempt 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 1.58 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 338. Click for [graphic](#).

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 (line 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.

Peak area measurement

[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).

[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 R2 values, which should ideally be 1,0, and 1, if the processing has no effect at all on peak area.

[iSignal](#) (page 337) is a 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](#) of 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 [PeakShapeAnalytical-Curve.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 298.

[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. [Power-MethodCalibrationCurve](#) 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 [Power-MethodDemo.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 145. 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 R2 (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 [testnum-peaks.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 131.

[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 131.

[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 138.

[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 138.

[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 288.

[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 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 effects the accuracy. This function needs the gaussian.m and gaussfit.m functions in the path. [Iterative methods](#) work much better in such cases.

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.

[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 a noisy data sets. It's 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 198.

[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 used called twice, with two different values of SlopeThreshold to demonstrate.

[iPeak](#) (page 216) 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 [ipeak7.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 295.

[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 similar to [autofindpeaks.m](#), the peak detection parameters SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, 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 the 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 337) 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 [Test-
FindpeaksG2d.m](#). Syntax: P = findpeaksG2d(x, y, SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, smoothtype)

[measurepeaks.m](#) automatically detects peaks in a signal, similar to [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 of 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](#).

[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, AUTOZERO). The

first seven input arguments are exactly the same as for the [findpeaksG.m](#) function; if you have been using findpeaks or iPeak (page 216) 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, $P = \text{findpeaksb}(x, y, \text{SlopeThreshold}, \text{AmpThreshold}, \text{smoothwidth}, \text{peakgroup}, \text{smoothtype}, \text{window}, \text{PeakShape}, \text{extra}, \text{NumTrials}, \text{AUTOZERO})$, except that the input arguments SlopeThreshold, AmpThreshold, smoothwidth, peakgroup, window, width, PeakShape, extra, NumTrials, autozero, 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 295.

[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 function $P = \text{findpeaksb3}(x, y, \text{SlopeThreshold}, \text{AmpThreshold}, \text{smoothwidth}, \text{peakgroup}, \text{smoothtype}, \text{PeakShape}, \text{extra}, \text{NumTrials}, \text{AUTOZERO}, \text{ShowPlots})$. The first seven input arguments are exactly the same as for the [findpeaksG.m](#) function; if you have been using findpeaks or iPeak (page 216) 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 $[P, \text{FitResults}, \text{LowestError}, \text{BestStart}, xi, yi] = \text{findpeaksfit}(x, y, \text{SlopeThreshold}, \text{AmpThreshold}, \text{smoothwidth}, \text{peakgroup}, \text{smoothtype}, \text{peakshape}, \text{extra}, \text{NumTrials}, \text{autozero}, \text{fixedparameters}, \text{plots})$. The first seven input arguments are exactly the same as for the [findpeaksG.m](#) function; if you have been using findpeaks or iPeak (page 216) 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 361) 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 198. [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 198. 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. [findstepspplot.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 215.

[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 an 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 216) must be downloaded and placed in the Matlab path.

[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 198.

[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 the [TestFindpeaksG2d](#) 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 mod- elpeaks.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. (Make the signal similar to 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.

[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 significantly improved.

[ipeakdata.mat](#), data set for demonstrating idpeaks.m or the peak identification function of iPeak; includes a high-resolution atomic spectrum and a table of known atomic emission wavelengths. See page 198.

Which to use: iPeak or Peakfit? Try these Matlab demo functions that compare iPeak.m (page 216) with peakfit.m (page 198) 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 to measure spikes (very narrow peaks) in the presence of serious interfering signals. See page 266.

[PowerTransformTest.m](#) is a simple script that demonstrates the [power method](#) of peak sharpening to aid in reducing 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 shoulder peaks ([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 198](#) 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](#)).

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 198) 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 152.

[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 198) 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 137. 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 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 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 expression 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.

[peakfit](#) (page 198) 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, AUTOZERO, fixedwidth, plots, bipolar, minwidth). Type "help peakfit". See page 343. 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's doing and the results, checking to see if the fitting error is greater than expected and printing out a WARNING if it is. See the [version history](#), page 343, 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, its 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 138.

[ipf.m](#) (page 361) 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 198) 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 285.

[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 171.

[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 171.

[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 171.

[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 167.

[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 167.

[BootstrapIterativeFit.m](#), a function that demonstrates bootstrap estimation of the variability of an iterative least-squares fit to a single noisy Gaussian peak. Form
is: `BootstrapIterativeFit(TrueHeight, TruePosition, TrueWidth, NumPoints, Noise, NumTrials)`. See page 134.

[BootstrapIterativeFit2.m](#), a function that demonstrates bootstrap estimation of the variability of an iterative least-squares fit to two noisy Gaussian peaks. Form
is: `BootstrapIterativeFit2(TrueHeight1, TruePosition1, TrueWidth1, TrueHeight2, TruePosition2, TrueWidth2, NumPoints, Noise, NumTrials)`. See page 134.

[DemoPeakfitBootstrap.m](#). Self-contained demonstration function for peakfit.m (page 198), with built-in signal generator. Demonstrates bootstrap error estimation. See page 134.

[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 359.

[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. For shape 50, the 10th input argument fixedparameters must be a *matrix* listing the peak shape (column 1), position (column 2), and width (column 3) of each peak, one row per peak. [peakfit9demoL](#) is similar but uses Lorentzian peaks (specified in the fixedparameters matrix and in the PeakShape vector).

[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). 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), displays the signals and the fits graphically with time as a kind of animation ([click to play animation](#)), then plots the measured peak position of the two peaks vs time on line 12.

[isignal](#) (page 337) 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 361. Use for testing that peakfit and related functions are present in the path. Updated for [peakfit 6](#). [autotestpeakfit.m](#) does the same without pausing between functions and waiting for a keypress.

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 R2 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. For example, a multiple Gaussians model is likely to fit best because it has more degrees of freedom and can "fit the noise", even if the *actual* peak shape is something other than a Gaussian. (The function peakfit.m has many more built-in shapes to choose from, but still it is 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 single mathematical function).

[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 376.

[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 another, more detailed, exploration of the effect of exponential broadening on a Gaussian peak (requires path: gaussian.m, expgaussian.m, halfwidth.m, val2ind.m, and peakfit.m in the Matlab/Octave path).

Keystroke-operated *interactive* signal processing tools (for Matlab only)

The three interactive functions described above, **iPeak**, **iSignal**, and **ipf**, all have several keystroke commands in common: all share the same set of *pan and zoom keys* (cursor arrow keys, < and > keys, [and] keys, etc.) to adjust the portion of the signal that is displayed in the upper panel. All use the **K** key to display the list of keystroke commands. All use the **T** key to cycle through the four baseline connection 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** functions.

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 (for Matlab only), 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 237. [Click for animated example](#).

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 192.

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 192.

udx.txt: a text file containing the 2 x 1091 matrix that consists of two Gaussian peaks with different sampling intervals. It's used as an example in [Smoothing](#) and in [Curve Fitting](#).

TimeTrial.txt, a text file comparing the speed of signal processing tasks running on Windows 10, 64 bit, 3 GHz core i5, with 12 GBytes RAM computer, using the following software configurations:

- (a) Matlab 7.8 R2009a
- (b) Matlab 2017b Home
- (c) Matlab Online, R2018b, in Chrome
- (d) Matlab Mobile (iPad)
- (e) Octave 3.6.4

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.

[Readability.txt](#). 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 may be stored anywhere it is convenient for you. They are self-contained and so not rely on external files. You may transfer your data to them by 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 22). 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 22. 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 35). 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 Gaussian 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, that uses the spreadsheet AVERAGE and INDIRECT functions (page 313). It allows you to change the smooth width simply by changing the value of a single cell. See page 45 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 53). [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 53.

Peak sharpening (page 69). 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 demonstration version with adjustable simulated peaks which you can experiment with ([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 peak sharpening template ([GradientPeakSharpeningDeriv.xlsx](#)) and an example with data ([GradientPeakSharpeningDerivExample.xlsx](#)); you need only set the starting and ending peak widths and the spreadsheet will apply the required sharpening factors K1 and K2.

[PeakSymmetrizationDemo.xlsxm](#) ([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 [PeakSymmetrizationTemplate.xlsxm](#) ([graphic](#)) and an example application with sample data already typed in: [PeakSymmetrizationExample.xlsxm](#).

[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 performs this method and [PowerSharpeningExample.xlsx](#) is the same with example data.

Convolution (page 93). 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 109). [EffectOfDx.xlsx](#) demonstrates that the simple equation $\text{sum}(y) \cdot dx$ 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 than the same amount of random noise. [PeakSharpeningAreaMeasurementDemo.xlsxm](#) ([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.

Curve Fitting (page 126). [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 152). [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 313) 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 138). 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 198). The spreadsheet [PeakDetection.xls](#) and [PeakDetection.xlsx](#) implement the simple derivative zero-crossing peak detection method. The input x,y data are contained in Sheet1, column **A** and **B**, rows 9 to 1200. (You can 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 [DerivativeSmoothing.xls](#) [described previously](#). 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. 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 data are noisy) 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). 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 sinusoidal signal 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 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, 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 234. 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 sharpened 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 237): 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 278) 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. 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), enable macros, then press **Ctrl-f** (or click **Developer** tab, click **Macros**, select **macro2**, and click **Run**). This macro constructs and plots the calibration curve for both the TFit (blue dots) and conventional (red dots) methods and computes the R² value for the TFit calibration curve, in the upper right corner of graph. (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 be a straight flat line with zero slope).

Advanced spreadsheet techniques (page 313): “[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 array functions such as INV and LINEST.

Afterword

How this book came to be.

During my career 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 one 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. When the Web became available to the academic community on the early 90s, I put up a syllabus, experiments, and other reading material for this and for my other courses online for students to access.

When I retired from the University in 1999, after 30 years of service, I noticed that I was getting a lot of pageviews on that course site from outside the University, especially on the lab experiment in digital data processing that I had developed in the 80's. I was getting an increasing number of emails with questions, suggestions, and comments, from people in widely varied scientific fields. So I began to broaden this beyond chemistry and my specific course. Ultimately I decided to make this a long-term retirement project. My aim is to help science workers learn and apply computer-based mathematical data processing techniques, by producing free tutorial material, coding examples, software, and guidance/consulting on specific projects.

Who needs this software?

Isn't software included in every scientific instrument hardware purchase? This is true, for those who are using conventional instruments in standard ways. But many scientists are working in completely new areas for which there are no commercial instruments, or they are building completely new types of instruments, or they are using modifications of existing systems for which there is no software. 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 don't understand the science.

Organization.

My project has five parts:

A book, entitled "A Pragmatic Introduction to Signal Processing", available in both [paper](#) and in [printable on-line formats](#);

- A [Web site](#) (.edu domain), with essentially the same material as the book. No sign-in or registration required.
- Downloadable free software in several different forms; on the [web site](#) and page 400.
- Help and consulting via email (optionally with data attachments);
- A [Facebook group](#) for announcements and public discussion.

The paper book is [sold through Amazon](#) ([ISBN](#) 978-1792916595), but the on-line materials, software, help, and consulting are all free. Open-source software alternatives are available, namely Octave and OpenOffice/LibreOffice. Although the complete book is available freely in PDF format, several readers have found it too long to print themselves and have requested a pre-printed version.

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 *Skype*. Requests for direct real-time voice/video is politely deflected. This is done to allow extended conversations between time zones, to avoid language problems and my own age-related hearing difficulties (readers come from at least 162 different countries) and to allow 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 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 do not know the age, gender, level of education, experience, or employment of clients unless they tell me. I have to look for clues in their writing in order to gauge their level of knowledge and experience and to avoid insulting them on the one hand or confusing them on the other.

Influence of the Internet

The Internet spans boundaries. 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. Google (or any search engine) looks at (almost) the entire internet, irrespective of the 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 a linguist, or a music scholar, know anything about my work? They would not, if I published only in the scientific journals of my specialty. But in fact all those type of researchers, and hundreds more like them, have found and cited my work by "stumbling across it" in a search engine query, in particular Google searches, rather than reading scholarly publications. A Google search is more likely to lead to a web site that is explicitly tutorial, rather than a research publication for specialists. In contrast, in my own academic career, I published research only in analytical chemistry journals, which are read mostly by other analytical chemists, whereas 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, medical, engineering, earth science, space, military, financial, agriculture, communications, and even language and musicology.

Writing

I intended my writing to be instructional, not especially scholarly or rigorous. It is pragmatic, meaning "Relating to matters of fact or practical affairs, often to the exclusion of intellectual or artistic matters; practical as opposed to idealistic." 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](#) (Gunning Fog index; Coleman-Liau index; Flesch-Kincaid Grade level; ARI ; SMOG; Flesch Reading Ease; ATOS Level). I have tried to minimize slang and obscure idioms and figures of speech that might confuse translators (machine and human), and I try to minimize the use of the passive voice. I often explain the same concept more than once in different contexts, because I believe it can help to make some ideas “stick” better. An important part of my writing process is *feedback from users*, by email (including data attachments), social media, search engine terms, questions, corrections, etc. Moreover, I also regularly re-read older sections with “fresh eyes”, correcting errors and making continuous improvements in phrasing. Questions from readers, and even search terms in Google searches, suggests areas where improvements are possible.

In order 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 search); ODT (editable Open Document Text); DOCX (editable Microsoft Word); PDF (Portable Document Format), and print (through Amazon's [Kindle Direct Publishing](#) program). All except for the web version have a detailed table of contents. All except the printed paper version 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 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 only partly reduced by including, on each Web page, "hot" links to the table of contents and to related previous and following material. The print version has an average of 3 page references per page and a table of contents with over 200 sections. Also, to facilitate communication, I have added a "mail to" link to each page in the Web version 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 reading.

Software platform selection criteria

As for software platforms, I chose two types: spreadsheets (page 15) and Matlab (page 16)/Octave (page 20). These 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. Octave on Raspberry Pi, page 304). Both are popular development environments that have a large user communities with multiple contributors, and both are widely used in science applications. These platforms have the advantage that they avoid secret algorithms, that is, their algorithms can be viewed in detail by any user. They are "open source" and "open document" formats that are either in plain text format (such as Matlab ".m" files) or in a format that can be opened and inspected using only free software (such as opening .xls and .xlsx spreadsheets with OpenOffice or LibreOffice). For those who cannot afford expensive software, OpenOffice Calc (page 15) and Octave (page 20) can be downloaded without cost.

Most of my Matlab/Octave programs are “functions”, which are essentially modular bits of code that fit together in different ways, much like high-tech Lego bricks, rather than self-contained stand-alone programs with elaborate graphic user interfaces, like commercial programs. Functions can be useful on

their own, but can also be used as components to construct something bigger. They have well-defined standard inputs and outputs, analogous to standard AC power outlets and plugs, USB and HDMI ports and cables, or Bluetooth connections between smartphone/tablets/computers and printers/earphones/speakers, etc., allowing you to mix and match to construct custom systems. I chose Matlab/Octave because of its very wide popularity and its similarity to other languages that have often been used by scientists, such as Fortran, Basic, and Pascal. Still, other languages, such as R, Python, Mathematica, Julia, and SciLab, have their champions and would have been valid alternatives.

I try to strike a balance between cost, speed, ease of use, and learning curve. In particular, I attempt to make my software usable even to those who do not read all the documentation, by providing lots and lots of examples and demos, including animated GIFs that will play on any web browser. Every script or function has *built-in* help that is internal to the software. In Matlab/Octave; you can display this built-in help simply by typing “help” followed by the name of the script or function. These help files contain not only instructions but also *examples of use* and in many cases include references to other similar functions. You can always look at (and even modify) my code if you wish, by opening it in the editor, but it’s not necessary if the existing action and inputs and outputs provide what you need. The spreadsheet templates and their examples and demos also have built-in instructions.

Outcomes

My website has received over 2 million pageviews and over 100,000 downloads of my software programs, both from my web site and from the [Matlab File Exchange](#). I have received many hundreds 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, these comments are so “over the top” that one wonders: *why such enthusiasm for such a nerdy topic?* After all, most people don’t take the time to write to the authors of web sites, especially to compliment them. One factor is surely that the number of users of the global Internet is so huge, even highly specialized topics are able to gather a substantial audience; as they say, “A wide net catches even the rarest fish”. According to the National Science Foundation, there are 11,000,000 graduate students in science worldwide, and UNESCO estimates that there are 7,000,000 full-time equivalents in research and development worldwide, so my users are still only a tiny fraction of that. But I also think that part of the reason for the enthusiasm is that software documentation is often badly written and hard to understand, so more effort is needed in better explaining software and how it works and where it can’t be expected to work. I try to be responsive, responding to each email, and acting on their suggestions and corrections. The growth in social media is also a contributing factor; for a specific example from the [Matlab File Exchange](#), see <https://blogs.mathworks.com/pick/2016/09/09/most-activeinteractive-file-exchange-entry/>.

Positive comments and lots of downloads are nice, but not everyone who downloads something actually uses in their work, and not everyone who does use it finds it valuable, and not all of those cite it in their publications. Most gratifyingly, as of December 2018, *over 370 publications have cited my website and programs*, based on Google Scholar searches (as listed on page 440).

References

1. Douglas A. Skoog, *Principles of Instrumental Analysis*, Third 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 D. Wentzell and Christopher D. Brown, Signal Processing in Analytical Chemistry, in *Encyclopedia of Analytical Chemistry*, R.A. Meyers (Ed.), p. 9764–9800, John Wiley & Sons Ltd, Chichester, 2000
(<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.124.2407&rep=rep1&type=pdf>)
7. Constantinos E. Efstatou, 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 edition, 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* March-April, **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://wmbiggs.com/blog/?p=195>
44. Nate Silver, *The Signal and the Noise: Why So Many Predictions Fail-but Some Don't*, 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>
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. ([164-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, 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, Signal Processing project on ResearchGate.
https://www.researchgate.net/profile/Joseph_Dubrovkin
-

Publications that cite the use of these programs and documentation

Last updated December, 2018

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. Chow, C. W. K. (1995). Optimisation, evaluation and chemometric studies of potentiometric stripping analysis. PDF link.
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 NO_x 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.ingen.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 Electroglossographic 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 Electroglossography and Audio Spectral Tilt", L Enflo, Proceedings of Fonetik, **2010**. Full text.
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 Chatnaballi. "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 tonotopic organization of glomeruli in the mouse olfactory bulb" Limei Maa, Qiang Qiua, Stephen Gradwohl, Aaron Scotta, Elden Q. Yua, Richard Alexandra, 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/RossDNAEnhancement_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 Schirrmeyer, 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 algorithmsfor 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. Valadares¹, 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. Hoerndl, 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. Žáčík, 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. "Corticospinal 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 a 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 Emiliос 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 $In_xGa_{1-x}N$ ($0.10 < x < 0.40$).” 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. "Corticocomuscular 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 reclamer," 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 ^{129}Xe 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 ŽMAN. "MEASUREMENT OF THE LIGHTWEIGHT ROTOR EIGENFREQUENCIES AND TUNING OF ITS MODEL PARAMETERS . Transactions of the VSB – Technical University of Ostrava, Mechanical Series №. 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, Imbabay 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, Imbabay 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 a steady 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. Bleeker, J. V. (2015). Relating phase separation and thickness mismatch in model lipid membranes (Doctoral dissertation).
205. Ilewickz, 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 H₂O, CO₂ 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. Bocaeghe, 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 SO_x and NO_x 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 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 an 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. (**2016**). 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. 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. (2017). 3D Moving Sound Source Localization via Conventional Microphones. *Elektronika ir Elektrotechnika*, 23(4), 63-69.
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 Valadanzej. "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 C* 121.35 (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 ^{13}C 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. 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.
317. Smith, Brad C., Bachana Lomsadze, and Steven T. Cundiff. "Optimum repetition rates for dual-comb spectroscopy." *Optics express* **26.9** **(2018)**: 12049-12056.
318. Butler, C. W., et al. "Neurons Specifically Activated by Fear Learning in Lateral Amygdala Display Increased Synaptic Strength." *eNeuro* **5.3** **(2018)**.
319. 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.
320. Cheng, Jie. *Peak Detection to Count Gold Nanoparticles Translocations in Nanopipette*. Diss. UC Santa Cruz, **2018**.
321. 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**.

322. Paige, Cristen, et al. "Characterizing the Normative Voice Tremor Frequency in Essential Vocal Tremor." *JAMA Otolaryngology–Head & Neck Surgery* (2018).
323. 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.
324. Taraszewski, Michal, and Janusz Ewertowski. "Small-Caliber Grenade Projectile Applicable to Individual Grenade Launchers." *Defence Science Journal* 68.5 (2018).
325. Trinh, N. D., et al. "Double differential neutron spectra generated by the interaction of a 12 MeV/nucleon ^{36}S 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.
326. 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.
327. Reynes, Julien, et al. "Experimental constraints on hydrogen diffusion in garnet." *Contributions to Mineralogy and Petrology* 173.9 (2018): 69.
328. 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.
329. 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).
330. 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.
331. 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.
332. Fratini, Marta, et al. "Surface Immobilization of Viruses and Nanoparticles Elucidates Early Events in Clathrin-Mediated Endocytosis." *ACS infectious diseases* (2018).
333. Siliņš, Kaspars. *Plasma Enhanced Chemical-and Physical-Vapor Depositions Using Hollow Cathodes*. Diss. Acta Universitatis Upsaliensis, 2018.
334. 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.
335. 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](#).
336. 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).
337. 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.
340. Roy, Daipayan, et al. "Frontiers in Ultrafast Chiral Chromatography." *LC• GC Europe* (2018): 308.
341. 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).
342. 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).
343. Li, Yuanlu, and Min Jiang. "Spatial-fractional order diffusion filtering." *Journal of Mathematical Chemistry* 56.1 (2018): 257-267.
344. 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.
345. 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.
346. 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.
347. Zhao, Chenjiang. *Signal Processing: Peak Detection*. Diss. UC Santa Cruz, 2018.
348. Coelho, Alan A. "Deconvolution of instrument and K_α2 contributions from X-ray powder diffraction patterns using nonlinear least squares with penalties." *Journal of Applied Crystallography* 51.1 (2018): 112-123.
349. Al-gawwam, Sarmad, and Mohammed Benissa. "Robust Eye Blink Detection Based on Eye Landmarks and Savitzky–Golay Filtering." *Information* 9.4 (2018): 93.
350. 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.
351. Gou, Yonggang, et al. "Motion parameter estimation and measured data correction derived from blast-induced vibration: new insights." *Measurement* (2018).
352. 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).
353. Mihálik, A., R. Ďuríkovič, 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.
354. 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.
355. 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.
356. Simon, David M., and Mark T. Wallace. "Integration and Temporal Processing of Asynchronous Audiovisual Speech" *Journal of cognitive neuroscience* 30.3 (2018): 319-337.

357. 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.
358. Dinç, Erdal, and Zehra Yazan. "Wavelet transform-based UV spectroscopy for pharmaceutical analysis" *Frontiers in Chemistry* 6 (2018).
359. Nocco, Mallika A., Matthew D. Ruark, and Christopher J. Kucharik. "Apparent electrical conductivity predicts physical properties of coarse soils." *Geoderma* 335 (2019): 1-11.
360. Oeltzschnner, Georg, et al. "Hadamard editing of glutathione and macromolecule-suppressed GABA." *NMR in Biomedicine* 31.1 (2018): e3844.
361. 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](#).
362. Лубов, Д. П., М. В. Катков, and Ю. В. Першин. "Вольт–амперные характеристики коммерческих сегнетоэлектрических конденсаторов: отклонения от модели Прейзаха." <<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).
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.
364. Humera Tariq, Abdul Muqeet, S.M.Aqil Burney, Humera Azam, "Otsu's Segmentation....", *J. Theoretical and Applied Information Technology*, Vol.95. No 22, 2017
365. 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.
366. 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(II), 2018.
367. Muhammad Mustafa and Nick Parziale, Single-Laser Krypton Tagging Velocimetry (KTV) Investigation of Air and N2 Boundary-Layer Flows Over a Hollow Cylinder in the Stevens Shock Tube, Conference: AIAA Scitech 2019 Forum, January 2019, DOI: 10.2514/6.2019-1820
368. 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
369. 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
370. 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/ijspes.7.1.7-13