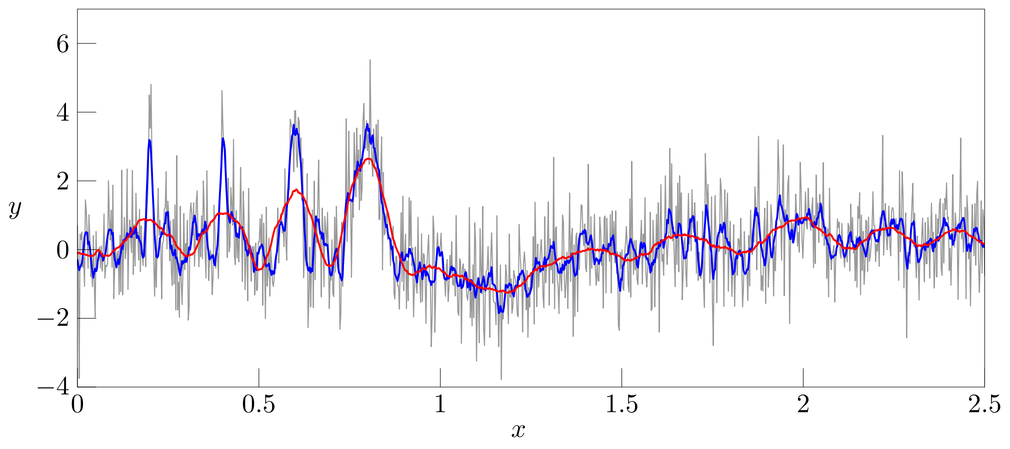
17 December 2011 at 00:56

1. Introduction.

SGFilter

A stand-alone implementation of the Savitzky–Golay smoothing filter.
(Version 1.6 of December 4, 2011)

Written by Fredrik Jonsson



This document was automatically extracted from the CWEB master source code for the SGFILTER program and typeset in the Computer Modern typeface using plain TEX and METAPOST. The source code and documentation of this program is electronically available at `http://jonsson.eu/programs/cweb/sgfilter/`.

Copyright © Fredrik Jonsson 2011

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photo-copying, recording, or otherwise, without the prior consent of the author. Non-commercial copying welcome.

Printed on 17 December 2011, at 00:56

TEX is a trademark of the American Mathematical Society

2. The Savitzky–Golay smoothing filter.

The Savitzky–Golay smoothing filter was originally presented in 1964 by Abraham Savitzky† and Marcel J. E. Golay‡ in their paper "Smoothing and Differentiation of Data by Simplified Least Squares Procedures", Anal. Chem., **36**, 1627–1639 (1964). Being chemists and physicists, at the time of publishing associated with the Perkin–Elmer Corporation (still today a reputable manufacturer of equipment for spectroscopy), they found themselves often encountering noisy spectra where simple noise-reduction techniques, such as running averages, simply were not good enough for extracting well-determined characteristica of spectral peaks. In particular, any running averaging tend to flatten and widening peaks in a spectrum, and as the peak width is an important parameter when determining relaxation times in molecular systems, such noise-reduction techniques are clearly non-desirable.

The main idea presented by Savitzky and Golay was a work-around avoiding the problems encountered with running averages, while still maintaining the smoothing of data and preserving features of the distribution such as relative maxima, minima and width. To quote the original paper on the target and purpose:

> "This paper is concerned with computational methods for the removal of the random noise from such information, and with the simple evaluation of the first few derivatives of the information with respect to the graph abscissa. [...] The objective here is to present specific methods for handling current problems in the processing of such tables of analytical data. The methods apply as well to the desk calculator, or to simple paper and pencil operations for small amounts of data, as they do to the digital computer for large amounts of data, since their major utility is to simplify and speed up the processing of data."

† Abraham Savitzky (1919–1999) was an American analytical chemist. (Wikipedia)
‡ Marcel J. E. Golay (1902–1989) was a Swiss-born mathematician, physicist, and information theorist, who applied mathematics to real-world military and industrial problems. (Wikipedia)

3. The work-around presented by Savitzky and Golay for avoiding distortion of peaks or features in their spectral data is essentially based on the idea to perform a linear regression of some polynomial *individually for each sample*, followed by the *evaluation of that polynomial exactly at the very position of the sample*. While this may seem a plausible idea, the actual task of performing a separate regression for each point easily becomes a very time-consuming task unless we make a few observations about this problem. However (and this is the key point in the method), for the regression of a polynomial of a finite power, say of an order below 10, the coefficients involved in the actual regression may be computed once and for all in an early stage, followed by performing a *convolution* of the discretely sampled input data with the coefficient vector. As the coefficient vector is significantly shorter than the data vector, this convolution is fast and straightforward to implement.

The starting point in deriving the algorithm for the Savitzky–Golay smoothing filter is to consider a smoothing method in which an equidistantly spaced set of M samples f_n, $n = 1, \ldots, M$ are linearly combined to form a filtered value h_k according to†

$$h_k = \sum_{j=-n_L}^{n_R} c_j f_{k+j}, \tag{1}$$

where n_L is the number of samples "to the left" and n_R the number of samples "to the right" of the centre index k. Notice that the running average smoothing corresponds to the case where all coefficients c_n are equal, with $c_n = 1/(n_L + n_R + 1)$. The idea of the Savitzky–Golay filter is, however, to find the set of c_n which better preserves the shape of features present in the sampled profile. The approach is to make a linear regression of a polynomial to the $n_L + n_R + 1$ samples in the window around sample k, and then evaluating this polynomial at that very sample, for all k from 1 to M.

We consider the regression of an m:th degree polynomial

$$p(x) = a_0 + a_1(x - x_k) + a_2(x - x_k)^2 + \ldots + a_m(x - x_k)^m \tag{2}$$

to the set of n_L samples to the left and n_R to the right of sample f_k, including the sample inbetween. Evaluating this polynomial at $x = x_k$ is particularly easy, as we at that point simply have $p(x_k) = a_0$. The linear regression of this polynomial to the samples f_{k+j}, with $-n_L \le j \le n_R$, means that we look for the least-square approximated solution to the linear system of $n_L + n_R + 1$ equations

$$\begin{aligned}
p(x_{k-n_L}) &= a_0 + a_1(x_{k-n_L} - x_k) + a_2(x_{k-n_L} - x_k)^2 + \ldots + a_m(x_{k-n_L} - x_k)^m \approx f_{k-n_L}, \\
&\ \ \vdots \qquad\qquad\qquad\qquad \vdots \\
p(x_{k-1}) &= a_0 + a_1(x_{k-1} - x_k) + a_2(x_{k-1} - x_k)^2 + \ldots + a_m(x_{k-1} - x_k)^m \approx f_{k-1}, \\
p(x_k) &= a_0 \approx f_k \\
p(x_{k+1}) &= a_0 + a_1(x_{k+1} - x_k) + a_2(x_{k+1} - x_k)^2 + \ldots + a_m(x_{k+1} - x_k)^m \approx f_{k+1}, \\
&\ \ \vdots \qquad\qquad\qquad\qquad \vdots \\
p(x_{k+n_R}) &= a_0 + a_1(x_{k+n_R} - x_k) + a_2(x_{k+n_R} - x_k)^2 + \ldots + a_m(x_{k+n_R} - x_k)^m \approx f_{k+n_R},
\end{aligned} \tag{3}$$

† More generally, Eq. (`'eq:10'`) can be interpreted as a discretized version of the convolution between a kernel $c(x)$ and a function $f(x)$,

$$h(x) = \int_{-\Delta_L}^{\Delta_R} c(s) f(x + s)\, ds \to \sum_{j=-n_L}^{n_R} c_j f_{k+j}.$$

Notice, however, the different sign of s as compared to the standard form of the convolution integral in mathematics, where the argument of the function usually yields "$f(x - s)$".

which (under the assumption that $n_L + n_R + 1 > m + 1$) provides an overdetermined system for the $m + 1$ coefficients a_j which can be expressed in matrix form as

$$\mathbf{A}\cdot\mathbf{a} \equiv \underbrace{\begin{pmatrix} 1 & (x_{k-n_L} - x_k) & (x_{k-n_L} - x_k)^2 & \cdots & (x_{k-n_L} - x_k)^m \\ \vdots & & \vdots & & \vdots \\ 1 & (x_{k-1} - x_k) & (x_{k-1} - x_k)^2 & \cdots & (x_{k-1} - x_k)^m \\ 1 & 0 & 0 & \cdots & 0 \\ 1 & (x_{k+1} - x_k) & (x_{k+1} - x_k)^2 & \cdots & (x_{k+1} - x_k)^m \\ \vdots & & \vdots & & \vdots \\ 1 & (x_{k+n_R} - x_k) & (x_{k+n_R} - x_k)^2 & \cdots & (x_{k+n_R} - x_k)^m \end{pmatrix}}_{[(n_L+n_R+1)\times(m+1)]} \underbrace{\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix}}_{[(m+1)\times 1]} = \underbrace{\begin{pmatrix} f_{k-n_L} \\ \vdots \\ f_{k-1} \\ f_k \\ f_{k+1} \\ \vdots \\ f_{k+n_R} \end{pmatrix}}_{[(m+1)\times 1]} \equiv \mathbf{f}. \tag{4}$$

The least squares solution to Eq. (4) is obtained by multiplying its left- and right-hand sides by the transpose of the system matrix $\mathbf{A}$, followed by solving the resulting $[(m + 1) \times (m + 1)]$-system of linear equations for $\mathbf{a}$,

$$\mathbf{A}^{\mathrm{T}} \cdot (\mathbf{A} \cdot \mathbf{a}) = (\mathbf{A}^{\mathrm{T}} \cdot \mathbf{A}) \cdot \mathbf{a} = \mathbf{A}^{\mathrm{T}} \cdot \mathbf{f} \qquad \Leftrightarrow \qquad \mathbf{a} = (\mathbf{A}^{\mathrm{T}} \cdot \mathbf{A})^{-1} \cdot (\mathbf{A}^{\mathrm{T}} \cdot \mathbf{f}) \tag{5}$$

Recapitulate that what we here target is the evaluation of $p(x_k) = a_0$, which according to Eqs. (2) and (5) is equivalent to evaluating the first ("zero:th") row of the solution for $\mathbf{a}$, or

$$p(x_k) = a_0 = \Big[\underbrace{(\mathbf{A}^{\mathrm{T}} \cdot \mathbf{A})^{-1}}_{[(m+1)\times(m+1)]} \cdot \underbrace{(\mathbf{A}^{\mathrm{T}} \cdot \mathbf{f})}_{[(m+1)\times 1]} \Big]_{\{\text{row } 0\}} \tag{6}$$

So, having arrived at this result for the regression, we clearly have a solution for a_0 depending on the actual function values f_{k+j} in the vicinity of sample f_k. Doesn't this mean that we nevertheless need to repeat the regression for every single sample to be included in the smoothing? Moreover, how does the result presented in Eq. (6) relate to the original mission, which we recapitulate was to find a general way of computing the coefficients c_j in the kernel of the convolution in Eq. (1)?

The answer to these questions lies in expanding Eq. (6) to yield the expression for the first ("zero:th") row as

$$\begin{aligned} p(x_k) = a_0 &= \big[(\mathbf{A}^{\mathrm{T}} \cdot \mathbf{A})^{-1} \cdot (\mathbf{A}^{\mathrm{T}} \cdot \mathbf{f})\big]_{\{\text{row } 0\}} \\ &= \sum_{j=1}^{m+1} \big[(\mathbf{A}^{\mathrm{T}} \cdot \mathbf{A})^{-1}\big]_{0j} \big[\mathbf{A}^{\mathrm{T}} \cdot \mathbf{f}\big]_j \\ &= \sum_{j=1}^{m+1} \big[(\mathbf{A}^{\mathrm{T}} \cdot \mathbf{A})^{-1}\big]_{0j} \sum_{k=1}^{m+1} \big[\mathbf{A}^{\mathrm{T}}\big]_{jk} \big[\mathbf{f}\big]_k \end{aligned} \tag{7}$$

and observing that the n:th coefficient c_n is obtained as equal to the coefficient a_0 whenever $\mathbf{f}$ is replaced by the unit vector $\mathbf{e}_n$ (with all elements zero except for the unitary n:th element). Hence

$$\begin{aligned} c_n &= \sum_{j=1}^{m+1} \big[(\mathbf{A}^{\mathrm{T}} \cdot \mathbf{A})^{-1}\big]_{0j} \sum_{k=1}^{m+1} \big[\mathbf{A}^{\mathrm{T}}\big]_{jk} \big[\mathbf{e}_n\big]_k \\ &= \sum_{j=1}^{m+1} \big[(\mathbf{A}^{\mathrm{T}} \cdot \mathbf{A})^{-1}\big]_{0j} \sum_{k=1}^{m+1} \big[\mathbf{A}^{\mathrm{T}}\big]_{jk} \delta_{nk} \\ &= \sum_{j=1}^{m+1} \big[(\mathbf{A}^{\mathrm{T}} \cdot \mathbf{A})^{-1}\big]_{0j} \big[\mathbf{A}^{\mathrm{T}}\big]_{jn} \\ &= \sum_{j=1}^{m+1} \big[(\mathbf{A}^{\mathrm{T}} \cdot \mathbf{A})^{-1}\big]_{0j} A_{nj} \end{aligned} \tag{8}$$

This concludes the derivation of the coefficients in the Savitzky–Golay filter, which may be computed once and for all in the beginning and afterwards used as the (for any practical purposes short) kernel c_j in the convolution described by Eq. (1).

4. In 2000, editors James Riordon, Elizabeth Zubritsky, and Alan Newman of *Analytical Chemistry* made a review article of what they had identified as the top-ten seminal papers in the history of the journal, based on the number of citations. Among the listed papers, of which some were written by Nobel-laureates-to-be, the original paper by Savitzky and Golay makes a somewhat odd appearance, as it not only concerns mainly numerical analysis, but also because it actually includes Fortran code for the implementation. The review article† concludes the discussion of the Savitzky–Golay smoothing filter with a reminiscence by Abraham Savitzky about the work:

> "In thinking about why the technique has been so widely used, I've come to the following conclusions. First, it solves a common problem–the reduction of random noise by the well-recognized least-squares technique. Second, the method was spelled out in detail in the paper, including tables and a sample computer subroutine. Third, the mathematical basis for the technique, although explicitly and rigorously stated in the article, was separated from a completely nonmathematical explanation and justification. Finally, the technique itself is simple and easy to use, and it works."

5. As a final remark on the Savitzky–Golay filtering algorithm, a few points on the actual implementation of the convolution need to be made. While the SGFILTER program relies on the method for computation of the Savitzky–Golay coefficients as presented in *Numerical Recipes in C*, 2nd Edn (Cambridge University Press, New York, 1994), it must be emphasized that the suggestion there presented for the convolution, which is to apply the *convlv* routine of *Numerical Recipes in C*, is significantly increasing the complexity and memory consumption in the filtering. In particular, the *convlv* routine in turn relies on consistent calls to the *twofft* routine, which in order to deliver proper data needs to be supplied with a return vector of *twice* the size of the input vector. In addition, *convlv* requires the size of the input vector to be of an integer power of two (say, $M = 1024$, 4096, etc.), which may be acceptable for one-off tests but is a rather inconvenient limitation for any more general applications.

Whether the SGFILTER program should employ the convolution engine supplied by the *convlv* routine (recommended in *Numerical Recipes in C*) or the direct convolution as implemented in the *sgfilter* routine (recommended by me) is controlled by the `CONVOLVE_WITH_NR_CONVLV` definition in the `sgfilter.h` header file. With reference to the above issues with *convlv*, I strongly advise keeping the default (`0`) setting for `CONVOLVE_WITH_NR_CONVLV`.

† Available at `http://pubs.acs.org/doi/pdf/10.1021/ac002801q`

6. Revision history of the program.

2006-01-18 [v.1.0] `<fj@phys.soton.ac.uk>`
First properly working version of the SGFILTER program.

2006-01-20 [v.1.1] `<fj@phys.soton.ac.uk>`
Added the test case for Savitzky–Golay filtering, modeling an underlying function $g(x)$ with superimposed Gaussian noise as

$$f(x) = \underbrace{\cos(3x)\sin^2(x^3) + 4\sum_{k=1}^{4}\exp(-(x-x_k)^2/w_k^2)}_{g(x)} + \underbrace{Vu(x)}_{\text{noise}}$$

where $u(x)$ is a normally distributed stochastic variable of mean zero and unit variance, V is the local variance as specified arbitrarily, and the remaining parameters (x_k, w_k) are the positions and widths of four Gaussian peaks superimposed onto the otherwise trigonometric expression for the underlying function.

2006-01-21 [v.1.2] `<fj@phys.soton.ac.uk>`
Changed the streaming of output (filtered) data so that the *stdout* stream now is directed to file whenever `-o` or `-outputfile` options are present at the calling command line.

2006-05-06 [v.1.3] `<fj@phys.soton.ac.uk>`
Added an introductory section documenting the derivation of the Savitzky–Golay filter as such. Always nice to have at hand when it comes to actually understanding why certain parameters in the filtering need to be in certain ranges. Also added automatic support for extracting the number of input samples automatically from the input file, hence making the `-M` and `--num_samples` option obsolete.

2006-05-06 [v.1.4] `<fj@phys.soton.ac.uk>`
Replaced the convolution from the previously used *convlv* routine to a brute-force but more economical one, which now does not rely on the input data being a set of 2^N samples for some N. However, I have chosen to keep the old implementation, which can be re-applied simply by changing the definition of `CONVOLVE_WITH_NR_CONVLV` to "(`1`)."

2009-11-01 [v.1.5] `<http://jonsson.eu>`
Included the `example.c` file in the CWEB source of SGFILTER, for automatically is generated from the master CWEB source when passed through CTANGLE. A very nifty way indeed for keeping updated test cases.

2011-12-04 [v.1.6] `<http://jonsson.eu>`
In the block concerned with the redirection of *stdout* when delivering filtered data, I found strange behaviour when executing SGFILTER under Windows. What happens is that any redirection of *stdout* back to terminal output in Windows naturally must be done in a different way than with the *freopen*(`"/dev/tty"`, `"a"`, *stdout*) which is accepted by OS X (Free BSD), Linux, or any other UNIX-like platforms. Hence, I simply added a (primitive) check on the platform type in the header file.

7. Compiling the source code. The program is written in CWEB, generating ANSI C (ISO C99) conforming source code and documentation as plain TEX-source, and is to be compiled using the sequences as outlined in the `Makefile` listed below. For general information on literate programming, CTANGLE, or CWEAVE, see `http://www.literateprogramming.com`.

```
#
# Makefile designed for use with ctangle, cweave, gcc, and plain TeX.
#
# Copyright (C) 2002-2011, Fredrik Jonsson <http://jonsson.eu>
#
# The CTANGLE program converts a CWEB source document into a C program
# which may be compiled in the usual way.  The output file includes #line
# specifications so that debugging can be done in terms of the CWEB source
# file.
#
# The CWEAVE program converts the same CWEB file into a TeX file that may
# be formatted and printed in the usual way.  It takes appropriate care of
# typographic details like page layout and the use of indentation, italics,
# boldface, etc., and it supplies extensive cross-index information that it
# gathers automatically.
#
# CWEB allows you to prepare a single document containing all the informa-
# tion that is needed both to produce a compilable C program and to produce
# a well-formatted document describing the program in as much detail as the
# writer may desire.    The user of CWEB ought to be familiar with TeX as well
# as C.
#
PROJECT  = sgfilter
CTANGLE  = ctangle
CWEAVE   = cweave
CC       = gcc
CCOPTS   = -O2 -Wall -ansi -std=iso9899:1999 -pedantic
LNOPTS   = -lm
TEX      = tex
DVIPS    = dvips
DVIPSOPT = -ta4 -D1200
PS2PDF   = ps2pdf
METAPOST = mpost

all:  $(PROJECT) $(PROJECT).pdf

$(PROJECT): $(PROJECT).o
	$(CC) $(CCOPTS) -o $(PROJECT) $(PROJECT).o $(LNOPTS)

$(PROJECT).o:  $(PROJECT).c
	$(CC) $(CCOPTS) -c $(PROJECT).c

$(PROJECT).c:  $(PROJECT).w
	$(CTANGLE) $(PROJECT) $(PROJECT).c
```

```
$(PROJECT).pdf:  $(PROJECT).ps
	$(PS2PDF) $(PROJECT).ps $(PROJECT).pdf

$(PROJECT).ps:  $(PROJECT).dvi
	$(DVIPS) $(DVIPSOPT) $(PROJECT).dvi -o $(PROJECT).ps

$(PROJECT).dvi:  $(PROJECT).tex
	$(TEX) $(PROJECT).tex

$(PROJECT).tex:  $(PROJECT).w
	$(CWEAVE) $(PROJECT)

clean:
	-rm -Rf $(PROJECT) *  *.c *.h *.o *.exe *.dat *.pdf *.mp *.trj *.mpx
	-rm -Rf *.tex *.aux *.log *.toc *.idx *.scn *.dvi *.ps *.1 *.eps

archive:
	make -ik clean
	tar --gzip --directory=../ -cf ../$(PROJECT).tar.gz $(PROJECT)
```

This `Makefile` essentially executes two major calls. First, the CTANGLE program parses the CWEB source file `sgfilter.w` to extract a C source file `sgfilter.c` which may be compiled into an executable program using any ANSI C conformant compiler. The output source file `sgfilter.c` includes `#line` specifications so that any debugging conveniently can be done in terms of line numbers in the original CWEB source file `sgfilter.w`. Second, the CWEAVE program parses the same CWEB source file `sgfilter.w` to extract a plain TEX file `sgfilter.tex` which may be compiled into a PostScript or PDF document. The document file `sgfilter.tex` takes appropriate care of typographic details like page layout and text formatting, and supplies extensive cross-indexing information which is gathered automatically. In addition to extracting the documentary text, CWEAVE also includes the source code in cross-referenced blocks corresponding to the descriptors as entered in the CWEB source code.

Having executed `make` (or `gmake` for the GNU enthusiast) in the same directory where the files `sgfilter.w` and `Makefile` are located, one is left with the executable file `sgfilter`, being the ready-to-use compiled program, and the PostScript file `sgfilter.ps` (or PDF file `sgfilter.pdf`) which contains the full documentation of the program, that is to say the document you currently are reading. Notice that on platforms running any operating system by Microsoft, the executable file will instead automatically be named `sgfilter.exe`. This convention also applies to programs compiled under the UNIX-like environment CYGWIN.

8. Running the program. The program is entirely controlled by the command line options supplied when invoking the program. The syntax for executing the program is

```
sgfilter [options]
```

where `options` include the following, given in their long as well as their short forms (with prefixes '`--`' and '`-`', respectively):

`--inputfile`, `-i` ⟨*input filename*⟩
: Specifies the raw, unfiltered input data to be crunched. The input file should describe the input as two columns containing x- and y-coordinates of the samples.

`--outputfile`, `-o` ⟨*output filename*⟩
: Specifies the output file to which the program should write the filtered profile, again in a two-column format containing x- and y-coordinates of the filtered samples. If this option is omitted, the generated filtered data will instead be written to the console (terminal).

`-nl` $\langle n_L \rangle$
: Specifies the number of samples n_L to use to the "left" of the basis sample in the regression window (kernel). The total number of samples in the window will be $nL + nR + 1$.

`-nr` $\langle n_R \rangle$
: Specifies the number of samples n_R to use to the "right" of the basis sample in the regression window (kernel). The total number of samples in the window will be $nL+nR+1$.

`-m` $\langle m \rangle$
: Specifies the order m of the polynomial $p(x) = a_0 + a_1(x - x_k) + a_2(x - x_k)^2 + \ldots + a_m(x - x_k)^m$ to use in the regression analysis leading to the Savitzky–Golay coefficients. Typical values are between $m = 2$ and $m = 6$. Beware of too high values, which easily makes the regression too sensitive, with an oscillatory result.

`-ld` $\langle l_D \rangle$
: Specifies the order of the derivative to extract from the Savitzky–Golay smoothing algorithm. For regular Savitzky–Golay smoothing of the input data as such, use $l_D = 0$. For the Savitzky–Golay smoothing and extraction of derivatives, set l_D to the order of the desired derivative and make sure that you correctly interpret the scaling parameters as described in *Numerical Recipes in C*, 2nd Edn (Cambridge University Press, New York, 1994).

`--help`, `-h`
: Displays a brief help message and terminates the SGFILTER program clean from any error codes.

`--verbose`, `-v`
: Toggle verbose mode. (Default: Off.) This option should always be omitted whenever no output file has been specified (that is to say, omit any `--verbose` or `-v` option whenever `--outputfile` or `-o` has been omitted), as the verbose logging otherwise will contaminate the filtered data stream written to the console (terminal).

9. Example of Savitzky–Golay filtering with the SGFILTER program. It is always a good idea to create a rather "nasty" test case for an algorithm, and in this case it also provides the reader of this code with a (hopefully) clear example of usage of the SGFILTER program.

We start off with generating a test suite of noisy data, in this case modeling an underlying function $g(x)$ with superimposed Gaussian noise as

$$f(x) = \underbrace{\cos(3x)\sin^2(x^3) + 4\sum_{k=1}^{4}\exp(-(x-x_k)^2/w_k^2)}_{g(x)} + \underbrace{Vu(x)}_{\text{noise}}$$

where $u(x)$ is a normally distributed stochastic variable of mean zero and unit variance, V is the local variance as specified arbitrarily, and the remaining parameters (x_k, w_k) are the positions and widths of four Gaussian peaks superimposed onto the otherwise trigonometric expression for the underlying function. For the current test suite we use $(x_1, w_1) = (0.2, 0.007)$, $(x_2, w_2) = (0.4, 0.01)$, $(x_3, w_3) = (0.6, 0.02)$, and $(x_4, w_4) = (0.8, 0.04)$. These Gaussian peaks serve as to provide various degrees of rapidly varying data, to check the performance in finding maxima. Meanwhile, the less rapidly varying domains which are dominated by the trigonometric expression serves as a test for the capability of the filter to handle rather moderate variations of low amplitude.

The underlying test function $g(x)$ is shown† in Fig. 1.

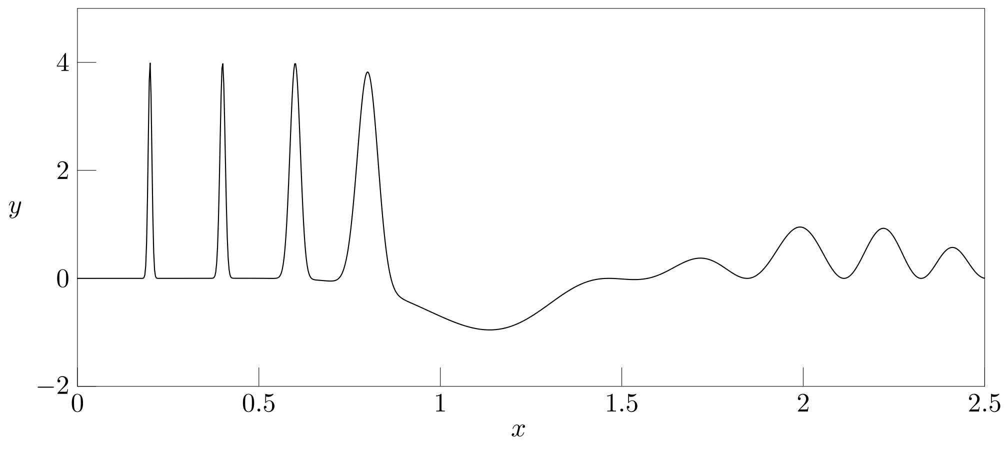


Figure 1. The underlying test function $g(x) = \cos(3x)\sin^2(x^3) + 4\sum_{k=1}^{4}\exp(-(x - x_k)^2/w_k^2)$ without any added noise. Here the positions and widths of the Gaussian peaks are $(x_1, w_1) = (0.2, 0.007)$, $(x_2, w_2) = (0.4, 0.01)$, $(x_3, w_3) = (0.6, 0.02)$, and $(x_4, w_4) = (0.8, 0.04)$.

The generator of artificial test data to be tested by the Savitsky–Golay filtering algorithm is here included as a simple C program, which automatically is generated from the master CWEB source when passed through CTANGLE.

```
⟨ example.c 9 ⟩ ≡
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#define TWOPI (2.0 * 3.141592653589793)
  float gauss(float x, float w, float xa)
```

† See the `*.eps` blocks in the enclosed `Makefile` for details on how METAPOSTwas used in the generation of the encapsulated PostScript images shown in Figs. 1–6.

```
{
  return (exp(−pow(((x − xa)/w), 2)));
}

float func(float x)
{
  float retval = gauss(x, 0.007, 0.2);    /* x1 = 0.2, w1 = 0.007 */
  retval += gauss(x, 0.01, 0.4);    /* x2 = 0.4, w2 = 0.01 */
  retval += gauss(x, 0.02, 0.6);    /* x3 = 0.6, w3 = 0.02 */
  retval += gauss(x, 0.04, 0.8);    /* x4 = 0.8, w4 = 0.04 */
  retval *= 4.0;
  retval += cos(3.0 * x) * pow(sin(pow(x, 3)), 2);
  return (retval);
}

int main(int argc, char *argv[ ])
{
  int k, mm = 1024;
  float var = 1.0, xmax = 2.5, x1, x2, u, v, f, z;
  if (argc > 1) sscanf(argv[1], "%f", &var);    /* Read first argument as variance */
  srand((unsigned) time(Λ));    /* Initialize random number generator */
  for (k = 0; k < mm − 1; k += 2) {
    x1 = xmax * k/((double) mm − 1);
    x2 = xmax * (k + 1)/((double) mm − 1);
    u = ((float) rand( ))/RAND_MAX;    /* Uniformly distributed over [0, 1] */
    v = ((float) rand( ))/RAND_MAX;    /* Uniformly distributed over [0, 1] */
    if (u > 0.0) {    /* Apply the Box–Muller algorithm on u and v */
      f = sqrt(−2 * log(u));
      z = TWOPI * v;
      u = f * cos(z);    /* Normally distributed with E(u)=0 and Var(u)=1 */
      v = f * sin(z);    /* Normally distributed with E(u)=0 and Var(u)=1 */
      fprintf(stdout, "%1.8f␣%1.8f\n", x1, func(x1) + var * u);    /* f(x1) */
      fprintf(stdout, "%1.8f␣%1.8f\n", x2, func(x2) + var * v);    /* f(x2) */
    }
  }
  return (0);
}
```

10. After having compiled the above code `example.c`, simply run `./example` ⟨*noise variance*⟩ `>` ⟨*file name*⟩ in order to generate the test function with a superimposed normally distributed (gaussian) noise of desired variance. In particular, we will in this test suite consider variances of 0 (that is to say, the underlying function without any noise), 0.5, 1.0, and 2.0. Such data files are simply generated by executing†

```
./example 0.0 > example-0.0.dat
./example 0.5 > example-0.5.dat
./example 1.0 > example-1.0.dat
./example 2.0 > example-2.0.dat
```

The resulting "noisified" suite of test data are in Figs. 2–4 shown for the respective noise variances of $V = 0.5$, $V = 1.0$, and $V = 2.0$, respectively.

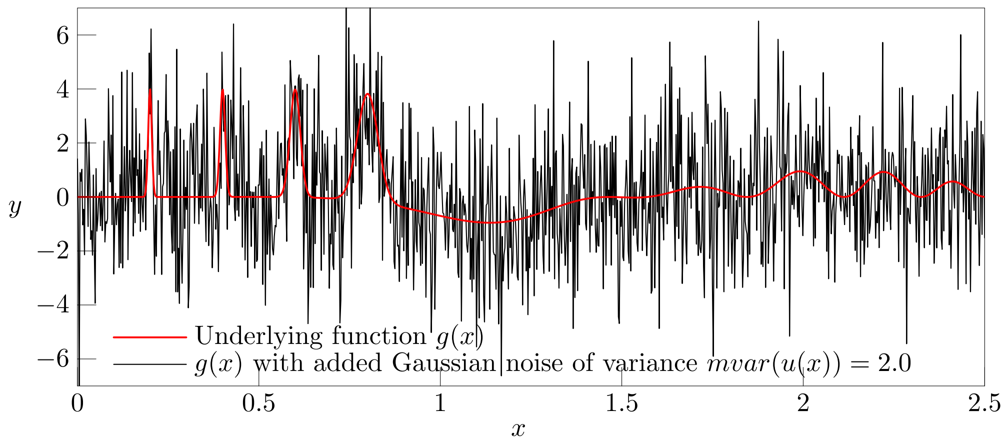


Figure 2. The test function $g(x)$ with added Gaussian noise of variance $V = 2.0$, as stored in file `example-2.0.dat` in the test suite.

† The resulting test data, which so far has not been subject to any filtering, may easily be viewed by running the following script in Octave/Matlab:

```
clear all; close all;
hold on;
u=load('example-2.0.dat'); plot(u(:,1),u(:,2),'-b');
u=load('example-1.0.dat'); plot(u(:,1),u(:,2),'-c');
u=load('example-0.5.dat'); plot(u(:,1),u(:,2),'-r');
u=load('example-0.0.dat'); plot(u(:,1),u(:,2),'-k');
legend('var(f(x))=2.0','var(f(x))=1.0','var(f(x))=0.5','var(f(x))=0.0');
hold off;
title('Artificial data generated for tests of Savitzky-Golay filtering');
xlabel('x');
ylabel('f(x)');
```

11. Applying the Savitzky–Golay filter to the test data is now a straightforward task. Say that we wish to test filtering with polynomial degree $m = 4$ and $ld = 0$ (which is the default value of ld, for regular smoothing with a delivered "zero:th order derivative", that is to say the smoothed non-differentiated function), for the two cases $nl = nr = 10$ (in total 21 points in the regression kernel) and $nl = nr = 60$ (in total 121 points in the regression kernel). Using the previously generated test suite of noisy data, the filtering is then easily accomplished by executing:

```
./sgfilter -m 4 -nl 60 -nr 60 -i example-0.0.dat -o example-0.0-f-60.dat
./sgfilter -m 4 -nl 10 -nr 10 -i example-0.0.dat -o example-0.0-f-10.dat
./sgfilter -m 4 -nl 60 -nr 60 -i example-0.5.dat -o example-0.5-f-60.dat
./sgfilter -m 4 -nl 10 -nr 10 -i example-0.5.dat -o example-0.5-f-10.dat
./sgfilter -m 4 -nl 60 -nr 60 -i example-1.0.dat -o example-1.0-f-60.dat
./sgfilter -m 4 -nl 10 -nr 10 -i example-1.0.dat -o example-1.0-f-10.dat
./sgfilter -m 4 -nl 60 -nr 60 -i example-2.0.dat -o example-2.0-f-60.dat
./sgfilter -m 4 -nl 10 -nr 10 -i example-2.0.dat -o example-2.0-f-10.dat
```

The resulting filtered data sets are shown in Figs. 3–6 for noise variances of $V = 2.0$, $V = 1.0$, $V = 0.5$, and $V = 0$, respectively. The final case corresponds to the interesting case of filtering the underlying function $g(x)$ without any added noise whatsoever, which corresponds to performing a local regression of the regression polynomial to the analytical trigonometric and exponential functions being terms of $g(x)$, a regression where we by no means should expect a perfect match.

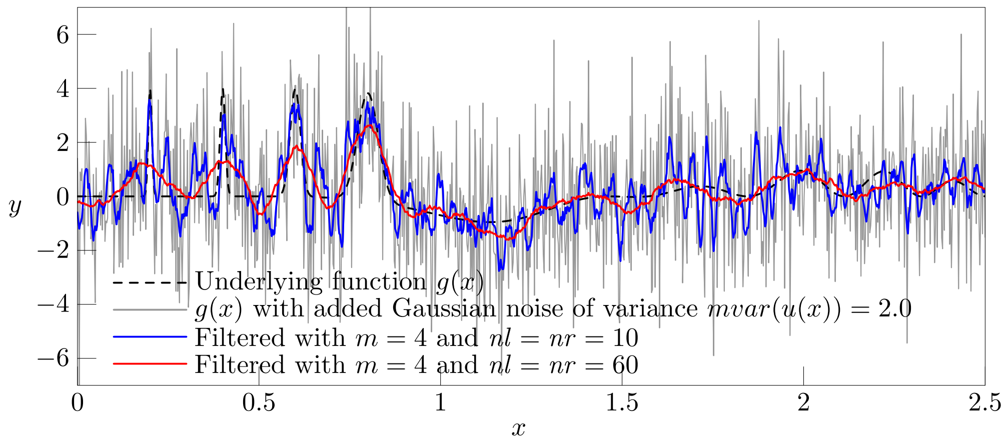


Figure 3. The profiles resulting from Savitzky–Golay-filtering of the test function $g(x)$ with added Gaussian noise of variance $V = 2.0$.

As can be seen in Fig. 3, the results from filtering the "worst-case" set (with noise variance $V = 2.0$) with $nl = nr = 10$ ($nl + nr + 1 = 21$ samples in the regression kernel) and $m = 4$ (curve in blue) yield a rather good tracking of the narrow Gaussian peaks, meanwhile performing rather poor in the low-amplitude and rather slowly varying trigonometric hills in the right-hand side of the graph. As a reference, the underlying function $g(x)$ is mapped in dashed black. On the other hand, with $nl = nr = 60$ ($nl + nr + 1 = 121$ samples in the regression kernel) and keeping the same degree of the regression polynomial (curve in red), the narrow peaks are barely followed, meanwhile having a rather poor performance in the slowly varying hills as well (albeit of a very poor signal-to-noise ratio).

However, with a lower variance of the superimposed noise, we at $V = 1$ have a nice tracking of the slowly varying hills by the 121-sample regression kernel, as shown in Fig. 4, meanwhile having a good tracking of the narrow Gaussian peaks by the 21-sample regression kernel. That the $nl+ = 21$-sample window is noisier than the 121-sample one should not really be surprising, as the higher number of samples in the regression window tend to smoothen out the regression even further.

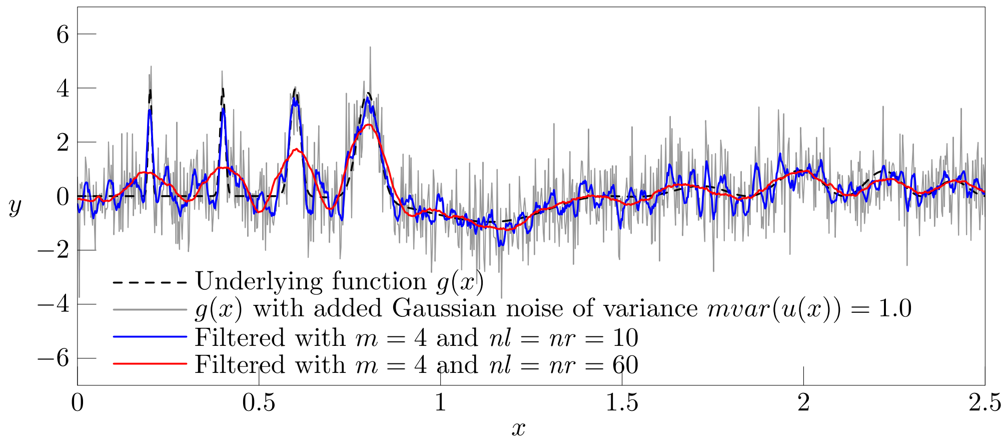


Figure 4. The profiles resulting from Savitzky–Golay-filtering of the test function $g(x)$ with added Gaussian noise of variance $V = 1.0$.

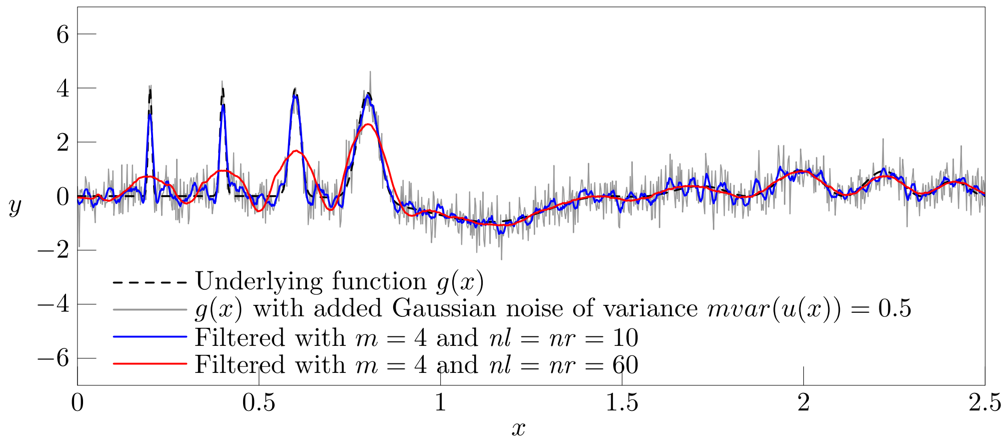


Figure 5. The profiles resulting from Savitzky–Golay-filtering of the test function $g(x)$ with added Gaussian noise of variance $V = 0.5$.

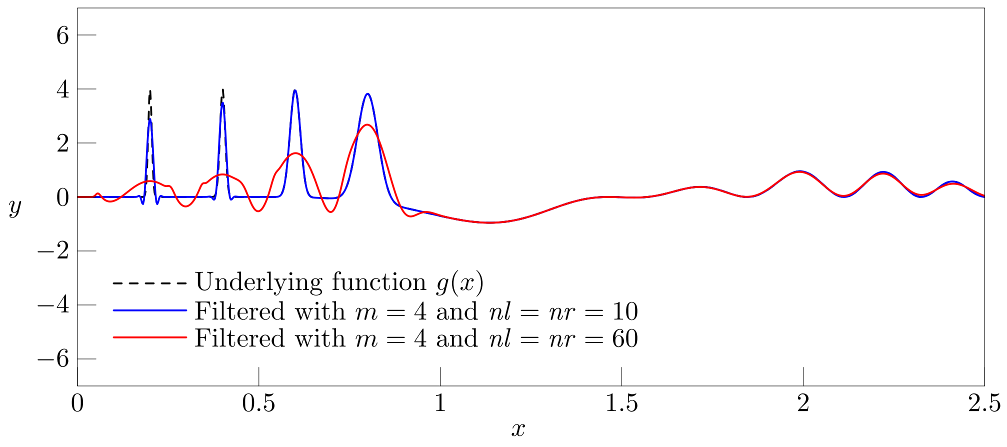


Figure 6. The profiles resulting from Savitzky–Golay-filtering of the test function $g(x)$ with added Gaussian noise of variance $V = 0$, that is to say a direct regression against the underlying function $g(x)$. This figure illustrates the limitations of the attempts of linear regression of a polynomial to an underlying function which clearly cannot be approximated by a simple polynomial expressin in certain domains. One should always keep this limitation in mind before accepting (or discarding) data filtered by the Savitzky–Golay algorithm, despite the many cases in which it performs exceptionally well.

12. Header files. We put the interface part of our routines in the header file `sgfilter.h`, and we restrict ourselves to a lowercase file name to maintain portability among operating systems with case-insensitive file names. There are just a few global definitions present in the SGFILTER program:

`VERSION`	The current program revision number.
`COPYRIGHT`	The copyright banner.
`DEFAULT_NL`	The default value to use for the *nl* variable unless stated otherwise at the command line during startup of the program. This parameter specifies the number of samples n_L to use to the "left" of the basis sample in the regression window (kernel). The total number of samples in the window will be $nL+nR+1$.
`DEFAULT_NR`	The default value to use for the *nr* variable unless stated otherwise at the command line during startup of the program. This parameter specifies the number of samples n_R to use to the "right" of the basis sample in the regression window (kernel). The total number of samples in the window will be $nL+nR+1$.
`DEFAULT_M`	The default value to use for the m variable unless stated otherwise at the command line during startup of the program. This parameter specifies the order m of the polynomial $p(x) = a_0 + a_1(x - x_k) + a_2(x - x_k)^2 + \ldots + a_m(x - x_k)^m$ to use in the regression analysis leading to the Savitzky–Golay coefficients. Typical values are between $m = 2$ and $m = 6$. Beware of too high values, which easily makes the regression too sensitive, with an oscillatory result.
`DEFAULT_LD`	The default value to use for the *ld* variable unless stated otherwise at the command line during startup of the program. This parameter specifies the order of the derivative to extract from the Savitzky–Golay smoothing algorithm. For regular Savitzky–Golay smoothing of the input data as such, use $l_D = 0$. For the Savitzky–Golay smoothing and extraction of derivatives, set l_D to the order of the desired derivative and make sure that you correctly interpret the scaling parameters as described in *Numerical Recipes in C*, 2nd Edn (Cambridge University Press, New York, 1994).
`NCHMAX`	The maximum number of characters allowed in strings for storing file names, including path.
log(...)	The *log*( ) macro forms the user interface to run-time error messaging and console logging activities, by invoking the *log_printf*( ) routine. The *log*( ) macro is preferrably used rather than direct calls to *log_printf*( ), as it automatizes the extraction of the calling routine, via the *__func__* macro. Notice that the clean construction of the macro using variable-length arguments (via `__VA_ARGS__`) *only* is supported at ISO 9899:1999 (ISO C99) standard level or above (see, for example, `http://en.wikipedia.org/wiki/C99`), which is the default compliance level used in the enclosed `Makefile`. (In fact, this is here *the* very reason for using ISO 9899:1999 rather than ISO 9899:1990).

```
⟨ sgfilter.h 12 ⟩ ≡
#define VERSION "1.6"
#define COPYRIGHT "Copyright␣(C)␣2006-2011,␣Fredrik␣Jonsson"
#define DEFAULT_NL (15)
#define DEFAULT_NR (15)
#define DEFAULT_M (4)
#define DEFAULT_LD (0)
#define EPSILON ((double)(1.0 · 10^−20))
#define NCHMAX (256)
#define CONVOLVE_WITH_NR_CONVLV (0)
#define log(...)log_printf (__func__, __VA_ARGS__)
#if defined (_CYGWIN_SIGNAL_H) ∨ defined (__APPLE__) ∨ defined (__unix__) ∨ defined (__linux)
```

```
#define UNIX_LIKE_OS (1)
#endif
  void log_printf(const char *function_name, const char *format, ...);
  int *ivector(long nl, long nh);
  double *dvector(long nl, long nh);
  double **dmatrix(long nrl, long nrh, long ncl, long nch);
  void free_ivector(int *v, long nl, long nh);
  void free_dvector(double *v, long nl, long nh);
  void free_dmatrix(double **m, long nrl, long nrh, long ncl, long nch);
  void lubksb(double **a, int n, int *indx, double b[]);
  void ludcmp(double **a, int n, int *indx, double *d);
  void four1(double data[], unsigned long nn, int isign);
  void twofft(double data1[], double data2[], double fft1[], double fft2[], unsigned long n);
  void realft(double data[], unsigned long n, int isign);
  char convlv(double data[], unsigned long n, double respns[], unsigned long m, int isign, double
      ans[]);
  char sgcoeff(double c[], int np, int nl, int nr, int ld, int m);
  char sgfilter(double yr[], double yf[], int mm, int nl, int nr, int ld, int m);
  char *strip_away_path(char filename[]);
  long int num_coordinate_pairs(FILE *file);
```

13. The main program. Here follows the general outline of the *main* routine of the SGFILTER program. This is where it all starts.

```
⟨ Library inclusions 14 ⟩
⟨ Global variables 15 ⟩
⟨ Definitions of routines 22 ⟩
int main(int argc, char *argv[ ])
{
  ⟨ Definition of variables 16 ⟩
  ⟨ Parse command line for options and parameters 17 ⟩
  ⟨ Allocate memory and read M samples of unfiltered data from file 18 ⟩
  ⟨ Filter raw data through the Savitzky–Golay smoothing filter 19 ⟩
  ⟨ Write filtered data to file or terminal output 20 ⟩
  ⟨ Deallocate memory 21 ⟩
  return (EXIT_SUCCESS);
}
```

14. Library dependencies. The standard ANSI C libraries included in this program are:

`math.h`	For access to common mathematical functions.
`stdio.h`	For file access and any block involving *fprintf*.
`stdlib.h`	For access to the *exit* function.
`stdarg.h`	For access to *vsprintf*, *vfprintf* etc.
`string.h`	For string manipulation, *strcpy*, *strcmp* etc.
`ctype.h`	For access to the *isalnum* function.
`time.h`	For access to *ctime*, *clock* and time stamps.
`sys/time.h`	For access to millisecond-resolved timer.

```
⟨ Library inclusions 14 ⟩ ≡
#include <math.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
#include <sys/time.h>
#include "sgfilter.h"
```

This code is used in section 13.

15. Declaration of global variables. The only global variables allowed in my programs are *optarg*, which is a pointer to the the string of characters that specified the call from the command line, and *progname*, which simply is a pointer to the string containing the name of the program, as it was invoked from the command line. Typically, the *progname* string is the result remaining after having passed *argv*[0] through the *strip_away_path* routine, just before parsing the command line for parameters.

```
⟨ Global variables 15 ⟩ ≡
  extern char *optarg;
  char *progname;
```

This code is used in section 13.

16. Declaration of local variables of the *main* program. In CWEB one has the option of adding variables along the program, for example by locally adding temporary variables related to a given sub-block of code. However, the philosophy in the SGFILTER program is to keep all variables of the *main* section collected together, so as to simplify tasks as, for example, tracking down a given variable type definition. The local variables of the program are as follows:

$*x$, $*yr$, $*yf$	Pointers to the double precision vectors keeping the abscissa, unfiltered (raw) ordinata, and the and the resulting filtered ordinata, respectively.
mm	Keeps track of the number of samples to analyze, $mm \equiv M$.
$*$*file*	Dummy file pointer used for scanning through input data and for writing data to an output file.
input_filename	String for keeping the file name of the input data.
output_filename	Ditto for the output data.
verbose	Determines whether the program should run in verbose mode (logging various activities along the execution) or not. Default is off.

⟨ Definition of variables 16 ⟩ ≡
```
int no_arg;
int nl = DEFAULT_NL;
int nr = DEFAULT_NR;
int ld = DEFAULT_LD;
int m = DEFAULT_M;
long int k, mm = 0;
double *x, *yr, *yf;
char input_filename[NCHMAX] = "", output_filename[NCHMAX] = "";
char verbose = 0;
FILE *file;
```
This code is used in section 13.

17. Parsing command line options. All input parameters are passed to the program through command line options to the SGFILTER program. The syntax of the accepted command line options is listed whenever the program is invoked without any options, or whenever any of the `--help` or `-h` options are specified at startup.

```
⟨ Parse command line for options and parameters 17 ⟩ ≡
  progname = strip_away_path(argv[0]);
  no_arg = argc;
  while (−− argc) {
    if (¬strcmp(argv[no_arg − argc], "-o") ∨ ¬strcmp(argv[no_arg − argc], "--outputfile")) {
      −− argc;
      strcpy(output_filename, argv[no_arg − argc]);
    }
    else if (¬strcmp(argv[no_arg − argc], "-i") ∨ ¬strcmp(argv[no_arg − argc], "--inputfile")) {
      −− argc;
      strcpy(input_filename, argv[no_arg − argc]);
    }
    else if (¬strcmp(argv[no_arg − argc], "-h") ∨ ¬strcmp(argv[no_arg − argc], "--help")) {
      showsomehelp( );
      exit(0);
    }
    else if (¬strcmp(argv[no_arg − argc], "-v") ∨ ¬strcmp(argv[no_arg − argc], "--verbose")) {
      verbose = (verbose ? 0 : 1);
    }
    else if (¬strcmp(argv[no_arg − argc], "-nl")) {
      −− argc;
      if (¬sscanf(argv[no_arg − argc], "%d", &nl)) {
        log("Error␣in␣'-nl'␣option.");
        exit(1);
      }
    }
    else if (¬strcmp(argv[no_arg − argc], "-nr")) {
      −− argc;
      if (¬sscanf(argv[no_arg − argc], "%d", &nr)) {
        log("Error␣in␣'-nr'␣option.");
        exit(1);
      }
    }
    else if (¬strcmp(argv[no_arg − argc], "-ld")) {
      −− argc;
      if (¬sscanf(argv[no_arg − argc], "%d", &ld)) {
        log("Error␣in␣'-ld'␣option.");
        exit(1);
      }
    }
    else if (¬strcmp(argv[no_arg − argc], "-m")) {
      −− argc;
      if (¬sscanf(argv[no_arg − argc], "%d", &m)) {
        log("Error␣in␣'-m'␣option.");
        exit(1);
      }
    }
    else {
      log("Unrecognized␣option␣'%s'.", argv[no_arg − argc]);
      showsomehelp();
      exit(1);
    }
  }
  if (verbose) fprintf(stdout, "This␣is␣%s␣v.%s.␣%s\n", progname, VERSION, COPYRIGHT);
```

This code is used in section 13.

18. This is the where the raw (unfiltered) data is loaded into the computer memory. This block assumes that the number of data points, M, has *not* previously been determined, neither by analysis of the input file nor explicitly stated via command line options, and hence the number of samples is automatically extracted using the *num_coordinate_pairs* routine.

⟨ Allocate memory and read M samples of unfiltered data from file 18 ⟩ ≡

```
  if (¬strcmp(input_filename, "")) {
    log("No␣input␣file␣specified!␣(Please␣use␣the␣'--inputfile'␣option.)");
    log("Execute␣'%s␣--help'␣for␣help.", progname);
    exit(1);
  }
  if ((file = fopen(input_filename, "r")) ≡ Λ) {
    log("Could␣not␣open␣%s␣for␣loading␣raw␣data!", input_filename);
    exit(1);
  }
  mm = num_coordinate_pairs(file);
  if (mm < nl + nr + 1) {
    log("Error:␣The␣number␣M=%ld␣of␣data␣points␣must␣be␣at␣least␣nl+nr+1=%d", mm, nl + nr + 1);
    log("Please␣check␣your␣-nl␣or␣-nr␣options.");
    exit(1);
  }
  if (verbose) {
    log("Loading␣%ld␣unfiltered␣samples␣from␣%s...", mm, input_filename);
    log("␣...␣allocating␣memory␣for␣storage␣...");
  }
  x = dvector(1, mm);
  yr = dvector(1, mm);
#if CONVOLVE_WITH_NR_CONVLV
  yf = dvector(1, 2 ∗ mm);
#else
  yf = dvector(1, mm);
#endif
  if (verbose) log("␣...␣scanning␣%s␣for␣input␣data␣...", input_filename);
  for (k = 1; k ≤ mm; k++) {
    fscanf(file, "%lf", &x[k]);     /∗ Scan x-coordinate ∗/
    fscanf(file, "%lf", &yr[k]);     /∗ Scan unfiltered y-coordinate ∗/
  }
  fclose(file);
  if (verbose) log("␣...␣done.␣Input␣now␣residing␣in␣RAM.");
```

This code is used in section 13.

19. Filter data. This is simple. One single call to the *sgfilter* routine and we are done.

```
⟨ Filter raw data through the Savitzky–Golay smoothing filter 19 ⟩ ≡
  if (¬sgfilter(yr, yf, mm, nl, nr, ld, m)) {
    if (verbose) log("Successfully␣performed␣Savitzky-Golay␣filtering.");
  }
  else {
    if (verbose) log("Error:␣Could␣not␣perform␣Savitzky-Golay␣filtering.");
  }
```

This code is used in section 13.

20. Write the filtered data to file or *stdout*, depending on whether or not the command line options `-o` or `-outputfile` were present when starting the SGFILTER program. We here use a redirection of the *stdout* stream using *freopen*(*output_filename*, `"w"`, *stdout*)) ≡ Λ) and (in the case of unix-like systems) a re-redirection back to terminal output again with *freopen*(`"/dev/tty"`, `"a"`, *stdout*).

```
⟨ Write filtered data to file or terminal output 20 ⟩ ≡
  if (¬strcmp(output_filename, "")) {    /* No filename specified */
    if (verbose) log("Writing␣%ld␣filtered␣samples␣to␣console...", mm);
  }
  else {    /* If file name specified ⇒ redirect stdout to file */
    if (verbose) log("Writing␣%ld␣filtered␣samples␣to␣%s...", mm, output_filename);
    if ((file = freopen(output_filename, "w", stdout)) ≡ Λ) {
      log("Error:␣Unable␣to␣redirect␣stdout␣stream␣to␣file␣%s.", output_filename);
      exit(1);
    }
  }
  for (k = 1; k ≤ mm; k++) fprintf(stdout, "%1.8f␣%1.8f\n", x[k], yf[k]);
#ifdef UNIX_LIKE_OS
  freopen("/dev/tty", "a", stdout);    /* Redirect stdout back to console output */
#endif
  if (verbose) log("␣...␣done.");
```

This code is used in section 13.

21. Deallocate memory.

```
⟨ Deallocate memory 21 ⟩ ≡
  free_dvector(x, 1, mm);
  free_dvector(yr, 1, mm);
#if CONVOLVE_WITH_NR_CONVLV
  free_dvector(yf, 1, 2 * mm);
#else
  free_dvector(yf, 1, mm);
#endif
```

This code is used in section 13.

22. Routines used by the program.

```
⟨ Definitions of routines 22 ⟩ ≡
  ⟨ Routine for error messaging 23 ⟩
  ⟨ Routine for allocation of integer precision vectors 24 ⟩
  ⟨ Routine for allocation of double floating-point precision vectors 25 ⟩
  ⟨ Routine for allocation of double floating-point precision matrices 26 ⟩
  ⟨ Routine for deallocation of integer precision vectors 27 ⟩
  ⟨ Routine for deallocation of double floating-point precision vectors 28 ⟩
  ⟨ Routine for deallocation of double floating-point precision matrices 29 ⟩
  ⟨ Routine for solving systems of linear equations by LU decomposition 30 ⟩
  ⟨ Routine for performing LU decomposition of a matrix 31 ⟩
  ⟨ Routine for discrete Fourier transformation of real-valued data 32 ⟩
  ⟨ Routine for simultaneous fast Fourier transformation of two data sets 33 ⟩
  ⟨ Routine for Fourier transformation of real-valued data 34 ⟩
  ⟨ Routine for numerical convolution 35 ⟩
  ⟨ Routine for computation of coefficients for Savitzky–Golay filtering 36 ⟩
  ⟨ Routine for Savitzky–Golay filtering 37 ⟩
  ⟨ Routine for removing preceding path of filenames 38 ⟩
  ⟨ Routine for displaying a brief help message on usage 40 ⟩
  ⟨ Routine for obtaining the number of coordinate pairs in a file 42 ⟩
```

This code is used in section 13.

23. The **void** *log_printf* (**const char** ∗*function_name*, **const char** ∗*format*, ...) routine writes formatted entries to standard output, displaying time and calling routine in a coherent manner. Notice that although the *log_printf* ( ) routine is the one which performs the actual messaging, the *log*( ) macro (defined in the header file) is the preferred way of accessing this routine, as it provides a more compact notation and automatically takes care of supplying the reference to the name of the calling function.

Also notice that the **const char** type of the last two input pointer arguments here is absolutely essential in order to pass strict pedantic compilation with GCC.

The routine accepts two input parameters. First, *function_name* which should be the name of the calling function. This is to ensure that any displayed error messages are properly matched to the issuing routines. Notice, however, that the *log*( ) macro (which is the preferred way of displaying error messages) automatically takes care of supplying the proper function name. Second, *format*, which simply is the format and message string to be displayed, formatted in the C-standard *printf* ( ) or *fprintf* ( ) syntax.

```
⟨ Routine for error messaging 23 ⟩ ≡
  void log_printf(const char *function_name, const char *format, ...)
  {
    va_list args;
    time_t time0;
    struct tm lt;
    struct timeval tv;
    char logentry[1024];

    gettimeofday(&tv, Λ);
    time(&time0);
    lt = *localtime(&time0);
    sprintf(logentry, "%02u%02u%02u␣%02u:%02u:%02u.%03d␣", lt.tm_year − 100, lt.tm_mon + 1,
          lt.tm_mday, lt.tm_hour, lt.tm_min, lt.tm_sec, tv.tv_usec/1000);
    sprintf(logentry + strlen(logentry), "(%s)␣", function_name);
    va_start(args, format);      /* Initialize args by the va_start( ) macro */
    vsprintf(logentry + strlen(logentry), format, args);
    va_end(args);      /* Terminate the use of args by the va_end( ) macro */
    sprintf(logentry + strlen(logentry), "\n");      /* Always append newline */
    fprintf(stdout, "%s", logentry);
    return;
  }
```

This code is used in section 22.

24. The **int** ∗*ivector*(**long** *nl*, **long** *nh*) routine allocates a real-valued vector of integer precision, with vector index ranging from *nl* to *nh*.

```
⟨ Routine for allocation of integer precision vectors 24 ⟩ ≡
  int *ivector(long nl, long nh)
  {
    int *v;

    v = (int *) malloc((size_t)((nh − nl + 2) * sizeof(int)));
    if (¬v) {
      log("Error:␣Allocation␣failure.");
      exit(1);
    }
    return v − nl + 1;
  }
```

This code is used in section 22.

25. The **double** $*dvector$(**long** nl, **long** nh) routine allocates a real-valued vector of double precision, with vector index ranging from nl to nh.

```
⟨ Routine for allocation of double floating-point precision vectors 25 ⟩ ≡
  double *dvector(long nl, long nh)
  {
    double *v;
    long k;
    v = (double *) malloc((size_t)((nh − nl + 2) * sizeof(double)));
    if (¬v) {
      log("Error:␣Allocation␣failure.");
      exit(1);
    }
    for (k = nl; k ≤ nh; k++) v[k] = 0.0;
    return v − nl + 1;
  }
```

This code is used in section 22.

26. The **double** $**dmatrix$(**long** nrl, **long** nrh, **long** ncl, **long** nch) routine allocates an array of double floating-point precision, with row index ranging from nrl to nrh and column index ranging from ncl to nch.

```
⟨ Routine for allocation of double floating-point precision matrices 26 ⟩ ≡
  double **dmatrix(long nrl, long nrh, long ncl, long nch)
  {
    long i, nrow = nrh − nrl + 1, ncol = nch − ncl + 1;
    double **m;
    m = (double **) malloc((size_t)((nrow + 1) * sizeof(double *)));
    if (¬m) {
      log("Allocation␣failure␣1␣occurred.");
      exit(1);
    }
    m += 1;
    m −= nrl;
    m[nrl] = (double *) malloc((size_t)((nrow * ncol + 1) * sizeof(double)));
    if (¬m[nrl]) {
      log("Allocation␣failure␣2␣occurred.");
      exit(1);
    }
    m[nrl] += 1;
    m[nrl] −= ncl;
    for (i = nrl + 1; i ≤ nrh; i++) m[i] = m[i − 1] + ncol;
    return m;
  }
```

This code is used in section 22.

27. The **void** $free\_ivector$(**int** $*v$, **long** nl, **long** nh) routine release the memory occupied by the real-valued vector v [nl ... nh].

```
⟨ Routine for deallocation of integer precision vectors 27 ⟩ ≡
  void free_ivector(int *v, long nl, long nh)
  {
    free((char *)(v + nl − 1));
  }
```

This code is used in section 22.

28. The *free_dvector* routine release the memory occupied by the real-valued vector $v\ [\ nl\ \ldots\ nh\ ]$.

⟨ Routine for deallocation of double floating-point precision vectors 28 ⟩ ≡

```
void free_dvector(double *v, long nl, long nh)
{
  free((char *)(v + nl − 1));
}
```

This code is used in section 22.

29. The **void** *free_dmatrix*(**double** ∗∗*m*, **long** *nrl*, **long** *nrh*, **long** *ncl*, **long** *nch*) routine releases the memory occupied by the double floating-point precision matrix $v\ [\ nl\ \ldots\ nh\ ]$, as allocated by *dmatrix*( ).

⟨ Routine for deallocation of double floating-point precision matrices 29 ⟩ ≡

```
void free_dmatrix(double **m, long nrl, long nrh, long ncl, long nch)
{
  free((char *)(m[nrl] + ncl − 1));
  free((char *)(m + nrl − 1));
}
```

This code is used in section 22.

30. The *lubksb*(**double** ∗∗*a*, **int** *n*, **int** ∗*indx*, **double** *b*[ ]) routine solves the set of n linear equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ where $\mathbf{A}$ is a real-valued $[n \times n]$-matrix and $\mathbf{x}$ and $\mathbf{b}$ are real-valued $[n \times 1]$-vectors. Here $a[1...n][1...n]$ is input, however *not* as the matrix $\mathbf{A}$ but rather as its corresponding LU decomposition as determined by the *ludcmp* routine. Here *indx*[1...*n*] is input as the permutation vector returned by *ludcmp*, $b[1...n]$ is input as the right-hand side vector $\mathbf{b}$, and returns with the solution vector $\mathbf{x}$. The parameters a, n, and *indx* are not modified by this routine and can be left in place for successive calls with different right-hand sides $\mathbf{b}$. This routine takes into account the possibility that $\mathbf{b}$ will begin with many zero elements, so it is efficient for use in matrix inversion. The *lubksb* routine is adopted from *Numerical Recipes in C*, 2nd Edn (Cambridge University Press, New York, 1994).

⟨ Routine for solving systems of linear equations by LU decomposition 30 ⟩ ≡

```
void lubksb(double **a, int n, int *indx, double b[ ])
{
  int i, ii = 0, ip, j;
  double sum;

  for (i = 1; i ≤ n; i++) {
    ip = indx[i];
    sum = b[ip];
    b[ip] = b[i];
    if (ii)
      for (j = ii; j ≤ i − 1; j++) sum −= a[i][j] ∗ b[j];
    else if (sum) ii = i;
    b[i] = sum;
  }
  for (i = n; i ≥ 1; i−−) {
    sum = b[i];
    for (j = i + 1; j ≤ n; j++) sum −= a[i][j] ∗ b[j];
    b[i] = sum/a[i][i];
  }
}
```

This code is used in section 22.

31. Given a square and real-valued matrix $a[1...n][1...n]$, the *ludcmp*(**float** $**a$, **int** n, **int** $*indx$, **float** $*d$) routine replaces it by the corresponding LU decomposition of a rowwise permutation of itself. Entering the routine, the square matrix a and its number of columns (or rows) n are inputs. On return, a is arranged as in Eq. (2.3.14) of *Numerical Recipes in C*, 2nd edition, while $indx[1...n]$ is an output vector that records the row permutation effected by the partial pivoting, and d is output as ±1 depending on whether the number of row interchanges was even or odd, respectively. This routine is commonly used in combination with *lubksb* to solve linear equations or invert a matrix. The *ludcmp* routine is adopted from *Numerical Recipes in C*, 2nd Edn (Cambridge University Press, New York, 1994).

⟨ Routine for performing LU decomposition of a matrix 31 ⟩ ≡

```
void ludcmp(double **a, int n, int *indx, double *d)
{
  int i, imax = 0, j, k;
  double big, dum, sum, temp;
  double *vv;

  vv = dvector(1, n);
  *d = 1.0;
  for (i = 1; i ≤ n; i++) {
    big = 0.0;
    for (j = 1; j ≤ n; j++)
      if ((temp = fabs(a[i][j])) > big) big = temp;
    if (big ≡ 0.0) {
      log("Error:␣Singular␣matrix␣found␣in␣routine␣ludcmp()");
      exit(1);
    }
    vv[i] = 1.0/big;
  }
  for (j = 1; j ≤ n; j++) {
    for (i = 1; i < j; i++) {
      sum = a[i][j];
      for (k = 1; k < i; k++) sum −= a[i][k] * a[k][j];
      a[i][j] = sum;
    }
    big = 0.0;
    for (i = j; i ≤ n; i++) {
      sum = a[i][j];
      for (k = 1; k < j; k++) sum −= a[i][k] * a[k][j];
      a[i][j] = sum;
      if ((dum = vv[i] * fabs(sum)) ≥ big) {
        big = dum;
        imax = i;
      }
    }
    if (j ≠ imax) {
      for (k = 1; k ≤ n; k++) {
        dum = a[imax][k];
        a[imax][k] = a[j][k];
        a[j][k] = dum;
      }
      *d = −(*d);
      vv[imax] = vv[j];
    }
    indx[j] = imax;
```

```
      **if** $(a[j][j] \equiv 0.0)$ $a[j][j] = \texttt{EPSILON}$;
      **if** $(j \neq n)$ {
        $dum = 1.0/(a[j][j])$;
        **for** $(i = j + 1;\ i \le n;\ i{+}{+})$ $a[i][j] \mathrel{*}= dum$;
      }
    }
    $free\_dvector(vv, 1, n)$;
  }
```

This code is used in section 22.

32. The *four1* (**double** *data*[ ], **unsigned long** *nn*, **int** *isign*) routine replaces *data*[1...2∗*nn*] by its discrete Fourier transform, if *isign* is input as +1; or replaces *data*[1..2 ∗ *nn*] by *nn* times its inverse discrete Fourier transform, if *isign* is input as −1. Here *data* is a complex-valued array of length *nn* or, equivalently, a real array of length 2 ∗ *nn*, wher *nn* *must* be an integer power of 2 (this is not checked for). The *four1* routine is adopted from *Numerical Recipes in C*, 2nd Edn (Cambridge University Press, New York, 1994).

```
⟨ Routine for discrete Fourier transformation of real-valued data 32 ⟩ ≡
#include <math.h>
#define SWAP(a, b) tempr = (a);
  (a) = (b); (b) = tempr
  void four1(double data[ ], unsigned long nn, int isign)
  {
    unsigned long n, mmax, m, j, istep, i;
    double wtemp, wr, wpr, wpi, wi, theta;
    double tempr, tempi;

    n = nn ≪ 1;
    j = 1;
    for (i = 1; i < n; i += 2) {
      if (j > i) {
        SWAP(data[j], data[i]);
        SWAP(data[j + 1], data[i + 1]);
      }
      m = n ≫ 1;
      while (m ≥ 2 ∧ j > m) {
        j −= m;
        m ≫= 1;
      }
      j += m;
    }
    mmax = 2;
    while (n > mmax) {
      istep = mmax ≪ 1;
      theta = isign ∗ (6.28318530717959/mmax);
      wtemp = sin(0.5 ∗ theta);
      wpr = −2.0 ∗ wtemp ∗ wtemp;
      wpi = sin(theta);
      wr = 1.0;
      wi = 0.0;
      for (m = 1; m < mmax; m += 2) {
        for (i = m; i ≤ n; i += istep) {
          j = i + mmax;
          tempr = wr ∗ data[j] − wi ∗ data[j + 1];
          tempi = wr ∗ data[j + 1] + wi ∗ data[j];
          data[j] = data[i] − tempr;
          data[j + 1] = data[i + 1] − tempi;
          data[i] += tempr;
          data[i + 1] += tempi;
        }
        wr = (wtemp = wr) ∗ wpr − wi ∗ wpi + wr;
        wi = wi ∗ wpr + wtemp ∗ wpi + wi;
      }
      mmax = istep;
    }
  }
#undef SWAP
```
This code is used in section 22.

33. The *twofft*(**double** *data1*[ ], **double** *data2*[ ], **double** *fft1*[ ], **double** *fft2*[ ], **unsigned long** *n*) routine takes two real-valued arrays *data1*[1...*n*] and *data2*[1...*n*] as input, makes a call to the *four1* routine and returns two complex-valued output arrays *fft1*[1...2 ∗ *n*] and *fft2*[1...2 ∗ *n*], each of complex length *n* (that is to say, a real length of 2 ∗ *n* elements), which contain the discrete Fourier transforms of the respective data arrays. Here *n* *must* be an integer power of 2. The *twofft* routine is adopted from *Numerical Recipes in C*, 2nd Edn (Cambridge University Press, New York, 1994).

⟨ Routine for simultaneous fast Fourier transformation of two data sets 33 ⟩ ≡
```
  void twofft(double data1[], double data2[], double fft1[], double fft2[], unsigned long n)
  {
    void four1(double data[], unsigned long nn, int isign);
    unsigned long nn3, nn2, jj, j;
    double rep, rem, aip, aim;
    nn3 = 1 + (nn2 = 2 + n + n);
    for (j = 1, jj = 2; j ≤ n; j++, jj += 2) {
      fft1[jj − 1] = data1[j];
      fft1[jj] = data2[j];
    }
    four1(fft1, n, 1);
    fft2[1] = fft1[2];
    fft1[2] = fft2[2] = 0.0;
    for (j = 3; j ≤ n + 1; j += 2) {
      rep = 0.5 ∗ (fft1[j] + fft1[nn2 − j]);
      rem = 0.5 ∗ (fft1[j] − fft1[nn2 − j]);
      aip = 0.5 ∗ (fft1[j + 1] + fft1[nn3 − j]);
      aim = 0.5 ∗ (fft1[j + 1] − fft1[nn3 − j]);
      fft1[j] = rep;
      fft1[j + 1] = aim;
      fft1[nn2 − j] = rep;
      fft1[nn3 − j] = −aim;
      fft2[j] = aip;
      fft2[j + 1] = −rem;
      fft2[nn2 − j] = aip;
      fft2[nn3 − j] = rem;
    }
  }
```
This code is used in section 22.

34. The *realft*(**double** *data*[ ], **unsigned long** *n*, **int** *isign*) routine calculates the Fourier transform of a set of *n* real-valued data points. On return the routine replaces this data (which is stored in array *data*[1...*n*]) by the positive frequency half of its complex Fourier transform. The real-valued first and last components of the complex transform are returned in elements *data*[1] and *data*[2], respectively. Here *n* must be a power of 2. This routine also calculates the inverse transform of a complex data array if it is the transform of real data. (In this case, the result must be multiplied by $2/n$.) The *realft* routine is adopted from *Numerical Recipes in C*, 2nd Edn (Cambridge University Press, New York, 1994).

⟨ Routine for Fourier transformation of real-valued data 34 ⟩ ≡

```
void realft(double data[], unsigned long n, int isign)
{
  void four1(double data[], unsigned long nn, int isign);
  unsigned long i, i1, i2, i3, i4, np3;
  double c1 = 0.5, c2, h1r, h1i, h2r, h2i;
  double wr, wi, wpr, wpi, wtemp, theta;
  theta = 3.141592653589793/(double)(n ≫ 1);
  if (isign ≡ 1) {
    c2 = −0.5;
    four1(data, n ≫ 1, 1);
  }
  else {
    c2 = 0.5;
    theta = −theta;
  }
  wtemp = sin(0.5 ∗ theta);
  wpr = −2.0 ∗ wtemp ∗ wtemp;
  wpi = sin(theta);
  wr = 1.0 + wpr;
  wi = wpi;
  np3 = n + 3;
  for (i = 2; i ≤ (n ≫ 2); i++) {
    i4 = 1 + (i3 = np3 − (i2 = 1 + (i1 = i + i − 1)));
    h1r = c1 ∗ (data[i1] + data[i3]);
    h1i = c1 ∗ (data[i2] − data[i4]);
    h2r = −c2 ∗ (data[i2] + data[i4]);
    h2i = c2 ∗ (data[i1] − data[i3]);
    data[i1] = h1r + wr ∗ h2r − wi ∗ h2i;
    data[i2] = h1i + wr ∗ h2i + wi ∗ h2r;
    data[i3] = h1r − wr ∗ h2r + wi ∗ h2i;
    data[i4] = −h1i + wr ∗ h2i + wi ∗ h2r;
    wr = (wtemp = wr) ∗ wpr − wi ∗ wpi + wr;
    wi = wi ∗ wpr + wtemp ∗ wpi + wi;
  }
  if (isign ≡ 1) {
    data[1] = (h1r = data[1]) + data[2];
    data[2] = h1r − data[2];
  }
  else {
    data[1] = c1 ∗ ((h1r = data[1]) + data[2]);
    data[2] = c1 ∗ (h1r − data[2]);
    four1(data, n ≫ 1, −1);
  }
}
```

This code is used in section 22.

35. The *convlv*(**double** *data*[ ], **unsigned long** *n*, **double** *respns*[ ], **unsigned long** *m*, **int** *isign*, **double** *ans*[ ]) routine convolves or deconvolves a real data set *data*[1...*n*] (including any user-supplied zero padding) with a response function *respns*[1...*n*]. The response function must be stored in wrap-around order in the first *m* elements of respns, where *m* is an odd integer less than or equal to *n*. Wrap-around order means that the first half of the array respns contains the impulse response function at positive times, while the second half of the array contains the impulse response function at negative times, counting down from the highest element *respns*[*m*]. On input, *isign* is +1 for convolution, and −1 for deconvolution. The answer is returned in the first *n* components of *ans*. However, *ans must* be supplied in the calling program with dimensions [1...2 ∗ *n*], for consistency with the *twofft* routine. Here *n must* be an integer power of two. The *convlv* routine is adopted from *Numerical Recipes in C*, 2nd Edn (Cambridge University Press, New York, 1994).

```
⟨ Routine for numerical convolution 35 ⟩ ≡
  char convlv(double data[ ], unsigned long n, double respns[ ], unsigned long m, int isign, double
        ans[ ])
  {
    void realft(double data[ ], unsigned long n, int isign);
    void twofft(double data1[ ], double data2[ ], double fft1[ ], double fft2[ ], unsigned long n);
    unsigned long i, no2;
    double dum, mag2, ∗fft;

    fft = dvector(1, n ≪ 1);
    for (i = 1; i ≤ (m − 1)/2; i++) respns[n + 1 − i] = respns[m + 1 − i];
    for (i = (m + 3)/2; i ≤ n − (m − 1)/2; i++) respns[i] = 0.0;
    twofft(data, respns, fft, ans, n);
    no2 = n ≫ 1;
    for (i = 2; i ≤ n + 2; i += 2) {
      if (isign ≡ 1) {
        ans[i − 1] = (fft[i − 1] ∗ (dum = ans[i − 1]) − fft[i] ∗ ans[i])/no2;
        ans[i] = (fft[i] ∗ dum + fft[i − 1] ∗ ans[i])/no2;
      }
      else if (isign ≡ −1) {
        if ((mag2 = ans[i − 1] ∗ ans[i − 1] + ans[i] ∗ ans[i]) ≡ 0.0) {
          log("Attempt␣of␣deconvolving␣at␣zero␣response␣in␣convlv().");
          return (1);
        }
        ans[i − 1] = (fft[i − 1] ∗ (dum = ans[i − 1]) + fft[i] ∗ ans[i])/mag2/no2;
        ans[i] = (fft[i] ∗ dum − fft[i − 1] ∗ ans[i])/mag2/no2;
      }
      else {
        log("No␣meaning␣for␣isign␣in␣convlv().");
        return (1);
      }
    }
    ans[2] = ans[n + 1];
    realft(ans, n, −1);
    free_dvector(fft, 1, n ≪ 1);
    return (0);
  }
```

This code is used in section 22.

36. The **void** *sgcoeff*(**double** $c[\,]$, **int** *np*, **int** *nl*, **int** *nr*, **int** *ld*, **int** *m*) routine computes the coefficients $c[1...np]$ for Savitzky–Golay filtering. The coefficient vector $c[1...np]$ is returned in *wrap-around order* consistent with the argument *respns* in the *Numerical Recipes in C* routine *convlv*. "Wrap-around order" means that the first half of the array *respns* contains the impulse response function at positive times, while the second half of the array contains the impulse response function at negative times, counting down from the highest element *respns*[*m*]. The Savitzky–Golay filter coefficients are computed for *nl* leftward (past) data points and *nr* rightward (future) data points, making the total number of data points used in the window as $np = nl + nr + 1$. *ld* is the order of the derivative desired (for example, $ld = 0$ for smoothed function). Here *m* is the order of the smoothing polynomial, also equal to the highest conserved moment; usual values are $m = 2$ or $m = 4$. The *sgcoeff* routine is adopted from *Numerical Recipes in C*, 2nd Edn (Cambridge University Press, New York, 1994).

```
⟨ Routine for computation of coefficients for Savitzky–Golay filtering 36 ⟩ ≡
  char sgcoeff(double c[], int np, int nl, int nr, int ld, int m)
  {
    void lubksb(double **a, int n, int *indx, double b[]);
    void ludcmp(double **a, int n, int *indx, double *d);
    int imj, ipj, j, k, kk, mm, *indx;
    double d, fac, sum, **a, *b;
    if (np < nl + nr + 1 ∨ nl < 0 ∨ nr < 0 ∨ ld > m ∨ nl + nr < m) {
      log("Inconsistent␣arguments␣detected␣in␣routine␣sgcoeff.");
      return (1);
    }
    indx = ivector(1, m + 1);
    a = dmatrix(1, m + 1, 1, m + 1);
    b = dvector(1, m + 1);
    for (ipj = 0; ipj ≤ (m ≪ 1); ipj++) {
      sum = (ipj ? 0.0 : 1.0);
      for (k = 1; k ≤ nr; k++) sum += pow((double) k, (double) ipj);
      for (k = 1; k ≤ nl; k++) sum += pow((double) −k, (double) ipj);
      mm = (ipj < 2 ∗ m − ipj ? ipj : 2 ∗ m − ipj);
      for (imj = −mm; imj ≤ mm; imj += 2) a[1 + (ipj + imj)/2][1 + (ipj − imj)/2] = sum;
    }
    ludcmp(a, m + 1, indx, &d);
    for (j = 1; j ≤ m + 1; j++) b[j] = 0.0;
    b[ld + 1] = 1.0;
    lubksb(a, m + 1, indx, b);
    for (kk = 1; kk ≤ np; kk++) c[kk] = 0.0;
    for (k = −nl; k ≤ nr; k++) {
      sum = b[1];
      fac = 1.0;
      for (mm = 1; mm ≤ m; mm++) sum += b[mm + 1] ∗ (fac ∗= k);
      kk = ((np − k) % np) + 1;
      c[kk] = sum;
    }
    free_dvector(b, 1, m + 1);
    free_dmatrix(a, 1, m + 1, 1, m + 1);
    free_ivector(indx, 1, m + 1);
    return (0);
  }
```

This code is used in section 22.

37. The *sgfilter*(**double** *yr*[], **double** *yf*[], **int** *mm*, **int** *nl*, **int** *nr*, **int** *ld*, **int** *m*) routine provides the interface for the actual Savitzky–Golay filtering of data. As input, this routine takes the following parameters:

yr[1...*mm*]	A vector containing the raw, unfiltered data
mm	The number of data points in the input vector
nl	The number of samples n_L to use to the "left" of the basis sample in the regression window (kernel). The total number of samples in the window will be $nL+nR+1$.
nr	The number of samples n_R to use to the "right" of the basis sample in the regression window (kernel). The total number of samples in the window will be $nL + nR + 1$.
m	The order m of the polynomial $p(x) = a_0 + a_1(x - x_k) + a_2(x - x_k)^2 + \ldots + a_m(x - x_k)^m$ to use in the regression analysis leading to the Savitzky–Golay coefficients. Typical values are between $m = 2$ and $m = 6$. Beware of too high values, which easily makes the regression too sensitive, with an oscillatory result.
ld	The order of the derivative to extract from the Savitzky–Golay smoothing algorithm. For regular Savitzky–Golay smoothing of the input data as such, use $l_D = 0$. For the Savitzky–Golay smoothing and extraction of derivatives, set l_D to the order of the desired derivative and make sure that you correctly interpret the scaling parameters as described in *Numerical Recipes in C*, 2nd Edn (Cambridge University Press, New York, 1994).

On return, the Savitzky–Golay-filtered profile is contained in the vector *yf*[1...*mm*]. Notice the somewhat peculiar accessing of the coefficients c_j via the $c[(j \ge 0\ ?\ j+1 : nr + nl + 2 + j)]$ construction in this routine. This reflects the *wrap-around storage* of the coefficients, where $c[1] = c_0$, $c[2] = c_1$, …, $c[nr + 1] = c_{n_R}$, $c[nr + 2] = c_{-n_L}$, $c[nr + 3] = c_{-n_L+1}$, …, $c[nr + nl + 1] = c_{-1}$.

⟨ Routine for Savitzky–Golay filtering 37 ⟩ ≡

```
  char sgfilter(double yr[], double yf[], int mm, int nl, int nr, int ld, int m)
  {
    int np = nl + 1 + nr;
    double *c;
    char retval;
#if CONVOLVE_WITH_NR_CONVLV    /* Please do not use this ... */
    c = dvector(1, mm);    /* Size required by the NR in C convlv routine */
    retval = sgcoeff(c, np, nl, nr, ld, m);
    if (retval ≡ 0) convlv(yr, mm, c, np, 1, yf);
    free_dvector(c, 1, mm);
#else    /* ... use this instead. (Strongly recommended.) */
    int j;
    long int k;
    c = dvector(1, nl + nr + 1);    /* Size required by direct convolution */
    retval = sgcoeff(c, np, nl, nr, ld, m);
    if (retval ≡ 0) {
      for (k = 1; k ≤ nl; k++) {    /* The first nl samples */
        for (yf[k] = 0.0, j = −nl; j ≤ nr; j++) {
          if (k + j ≥ 1) {
            yf[k] += c[(j ≥ 0 ? j + 1 : nr + nl + 2 + j)] * yr[k + j];
          }
        }
      }
      for (k = nl + 1; k ≤ mm − nr; k++) {    /* Samples nl + 1 ... mm − nr */
        for (yf[k] = 0.0, j = −nl; j ≤ nr; j++) {
          yf[k] += c[(j ≥ 0 ? j + 1 : nr + nl + 2 + j)] ∗ yr[k + j];
        }
      }
      for (k = mm − nr + 1; k ≤ mm; k++) {      /∗ The last nr samples ∗/
        for (yf[k] = 0.0, j = −nl; j ≤ nr; j++) {
          if (k + j ≤ mm) {
            yf[k] += c[(j ≥ 0 ? j + 1 : nr + nl + 2 + j)] ∗ yr[k + j];
          }
        }
      }
    }
    free_dvector(c, 1, nr + nl + 1);
#endif
    return (retval);      /∗ Returning 0 if successful filtering ∗/
  }
```

This code is used in section 22.

38. Routines for removing preceding path of filenames. In this block all routines related to removing preceding path strings go. Not really fancy programming, and no contribution to any increase of numerical efficiency or precision; just for the sake of keeping a tidy terminal output of the program. The *strip_away_path*( ) routine is typically called when initializing the program name string *progname* from the command line string *argv*[0], and is typically located in the blocks related to parsing of the command line options. The *strip_away_path*( ) routine takes a character string *filename* as argument, and returns a pointer to the same string but without any preceding path segments.

```
⟨ Routine for removing preceding path of filenames 38 ⟩ ≡
  ⟨ Routine for checking for a valid path character 39 ⟩
  char ∗strip_away_path(char filename[ ])
  {
    int j, k = 0;
    while (pathcharacter(filename[k])) k++;
    j = (−−k);      /∗ this is the uppermost index of the full path+file string ∗/
    while (isalnum((int)(filename[j]))) j−−;
    j++;      /∗ this is the lowermost index of the stripped file name ∗/
    return (&filename[j]);
  }
```

This code is used in section 22.

39. In this program, valid path characters are any alphanumeric character or '.', '/', '\', '_', '-', or '+'.

```
⟨ Routine for checking for a valid path character 39 ⟩ ≡
  short pathcharacter(int ch)
  {
    return (isalnum(ch) ∨ (ch ≡ '.') ∨ (ch ≡ '/') ∨ (ch ≡ '\\') ∨ (ch ≡ '_') ∨ (ch ≡ '-') ∨ (ch ≡ '+'));
  }
```

This code is used in section 38.

40. The **void** *showsomehelp*(**void**) displays a brief help message to standard terminal output. This is where the caller always should end up in case anything is wrong in the input parameters.

⟨ Routine for displaying a brief help message on usage 40 ⟩ ≡

```
⟨ Routine for displaying a single line of the help message 41 ⟩
void showsomehelp(void)
{
  hl("Usage:␣%s␣[options]", progname);
  hl("Options:");
  hl("␣-h,␣--help");
  hl("␣␣␣␣Display␣this␣help␣message␣and␣exit␣clean.");
  hl("␣-i,␣--inputfile␣<str>");
  hl("␣␣␣␣Specifies␣the␣file␣name␣from␣which␣unfiltered␣data␣is␣to␣be␣read.");
  hl("␣␣␣␣The␣input␣file␣should␣describe␣the␣input␣as␣two␣columns␣contain-");
  hl("␣␣␣␣ing␣$x$-␣and␣$y$-coordinates␣of␣the␣samples.");
  hl("␣-o,␣--outputfile␣<str>");
  hl("␣␣␣␣Specifies␣the␣file␣name␣to␣which␣filtered␣data␣is␣to␣be␣written,");
  hl("␣␣␣␣again␣in␣a␣two-column␣format␣containing␣$x$-␣and␣$y$-coordinates");
  hl("␣␣␣␣of␣the␣filtered␣samples.␣If␣this␣option␣is␣omitted,␣the␣generated");
  hl("␣␣␣␣filtered␣data␣will␣instead␣be␣written␣to␣the␣console␣(terminal).");
  hl("␣-nl␣<nl>");
  hl("␣␣␣␣Specifies␣the␣number␣of␣samples␣nl␣to␣use␣to␣the␣'left'␣of␣the");
  hl("␣␣␣␣basis␣sample␣in␣the␣regression␣window␣(kernel).␣The␣total␣number");
  hl("␣␣␣␣of␣samples␣in␣the␣window␣will␣be␣nL+nR+1.");
  hl("␣-nr␣<nr>");
  hl("␣␣␣␣Specifies␣the␣number␣of␣samples␣nr␣to␣use␣to␣the␣'right'␣of␣the");
  hl("␣␣␣␣basis␣sample␣in␣the␣regression␣window␣(kernel).␣The␣total␣number");
  hl("␣␣␣␣of␣samples␣in␣the␣window␣will␣be␣nL+nR+1.");
  hl("␣-m␣<m>");
  hl("␣␣␣␣Specifies␣the␣order␣m␣of␣the␣polynomial␣to␣use␣in␣the␣regression");
  hl("␣␣␣␣analysis␣leading␣to␣the␣Savitzky-Golay␣coefficients.␣Typical");
  hl("␣␣␣␣values␣are␣between␣m=2␣and␣m=6.␣Beware␣of␣too␣high␣values,␣which");
  hl("␣␣␣␣easily␣makes␣the␣regression␣too␣sensitive,␣with␣an␣oscillatory");
  hl("␣␣␣␣result.");
  hl("␣-ld␣<ld>");
  hl("␣␣␣␣Specifies␣the␣order␣of␣the␣derivative␣to␣extract␣from␣the␣");
  hl("␣␣␣␣Savitzky--Golay␣smoothing␣algorithm.␣For␣regular␣Savitzky-Golay");
  hl("␣␣␣␣smoothing␣of␣the␣input␣data␣as␣such,␣use␣ld=0.␣For␣the␣Savitzky-");
  hl("␣␣␣␣Golay␣smoothing␣and␣extraction␣of␣derivatives,␣set␣ld␣to␣the");
  hl("␣␣␣␣order␣of␣the␣desired␣derivative␣and␣make␣sure␣that␣you␣correctly");
  hl("␣␣␣␣interpret␣the␣scaling␣parameters␣as␣described␣in␣'Numerical");
  hl("␣␣␣␣Recipes␣in␣C',␣2nd␣Edn␣(Cambridge␣University␣Press,␣New␣York,");
  hl("␣␣␣␣1994).");
  hl("␣-v,␣--verbose");
  hl("␣␣␣␣Toggle␣verbose␣mode.␣(Default:␣Off.)␣␣This␣option␣should␣always");
  hl("␣␣␣␣be␣omitted␣whenever␣no␣␣output␣file␣has␣been␣specified␣(that␣is");
  hl("␣␣␣␣to␣say,␣omit␣any␣--verbose␣or␣-v␣option␣whenever␣--outputfile␣or");
  hl("␣␣␣␣-o␣has␣been␣omitted),␣as␣the␣verbose␣logging␣otherwise␣will");
  hl("␣␣␣␣contaminate␣the␣filtered␣data␣stream␣written␣to␣the␣console");
  hl("␣␣␣␣(terminal).");
}
```

This code is used in section 22.

41. In order to simplify the messaging, the *hl*(**const char** ∗*format*, . . .) routine acts as a simple front-end merely for compactifying the code by successive calls to *hl*(. . .) rather than the full *fprintf* (*stderr*, . . .), still maintaining all the functionality of string formatting in the regular *printf* ( ) or *fprintf* ( ) syntax.

⟨ Routine for displaying a single line of the help message 41 ⟩ ≡

```
void hl(const char *format, ...){ va_list args; char
      line [1024] ;
      va_start(args, format);      /* Initialize args by the va_start( ) macro */
      vsprintf ( line , format, args ) ;
      va_end(args);      /* Terminate the use of args by the va_end( ) macro */
      sprintf ( line +strlen ( line ) , "\n" ) ;      /* Always append newline */
      fprintf (stdout, "%s", line ) ;
      return; }
```

This code is used in section 40.

42. Routine for obtaining the number of coordinate pairs in a file. This routine is called prior to loading the input data, in order to automatically extract the size needed for allocating the memory for the storage.

⟨ Routine for obtaining the number of coordinate pairs in a file 42 ⟩ ≡

```
long int num_coordinate_pairs(FILE *file)
{
   double tmp;
   int tmpch;
   long int mm = 0;
   fseek(file, 0L, SEEK_SET);      /* rewind file to beginning */
   while ((tmpch = getc(file)) ≠ EOF) {
      ungetc(tmpch, file);
      fscanf(file, "%lf", &tmp);      /* Read away the x coordinate */
      fscanf(file, "%lf", &tmp);      /* Read away the y coordinate */
      mm++;
      tmpch = getc(file);      /* Read away any blanks or linefeeds */
      while ((tmpch ≠ EOF) ∧ (¬isdigit(tmpch))) tmpch = getc(file);
      if (tmpch ≠ EOF) ungetc(tmpch, file);
   }
   fseek(file, 0L, SEEK_SET);      /* rewind file to beginning */
   return (mm);
}
```

This code is used in section 22.

43. Index.

SGFILTER