ELECTRONOTES
1016 Hanshaw Road
Ithaca, NY 14850

APPLICATION NOTE NO. 404

Feb 13, 2014

SAVITZKY-GOLAY SMOOTHING

"Savitzky-Golay: or Least Square Polynomial Smoothing" *

The area of Digital Signal Processing (DSP) is extremely broad and rich. Anyone who has ever organized a course in DSP at any level needs to select from among myriad topics and examples. At the same time, we often marvel at the fact that most of the ideas we use can be understood in at least several ways, and many of the procedures employed are found to be reinventions, developed as needed, or in new forms. [For example, the relatively "recent" techniques of Fast Fourier Transform and of discrete-time exponential modeling trace back to work done by mathematicians (Gauss and Prony respectively) roughly 200 years ago.

The present topic of this note is the Savitzky-Golay (S-G) filter, also found listed as least square polynomial smoothing. Nearly every DSP engineer has heard the term S-G, and for periods of time, may even have known a bit about them. The present author first heard about S-G from Tom Parks at Cornell, and remembers that the recommended reference was part of a chapter by Schuessler in a somewhat neglected book edited by Lim and Oppenheim [1]. Previous to that (late 1980's) Allan Steinhardt (also then at Cornell) had suggested a problem for students that was essentially the problem of finding a filter by fitting a straight line to three points (minimizing squared error) which reappears below. This simple problem has also been outlined in our own publications [2] and has been a recurring student exercise.

More recently in finding topics for student projects, Tom Parks included a general reference to S-G, and this in turn has led to some interesting findings, historically. First, the original paper by Savitzky and Golay is apparently in the journal ***Analytical Chemistry*** [3] which is not a place one would think to look for notions about DSP (although, evidently, about data smoothing). Secondly, the popular program Matlab has a S-G function (***sgolay.m***) which was not immediately decipherable, but it did yield an important reference: S-G had been lurking all the time in the popular DSP text by Orfanidies [4]. In addition, Hamming [5,6] had the material covered concisely under least-squares smoothing.

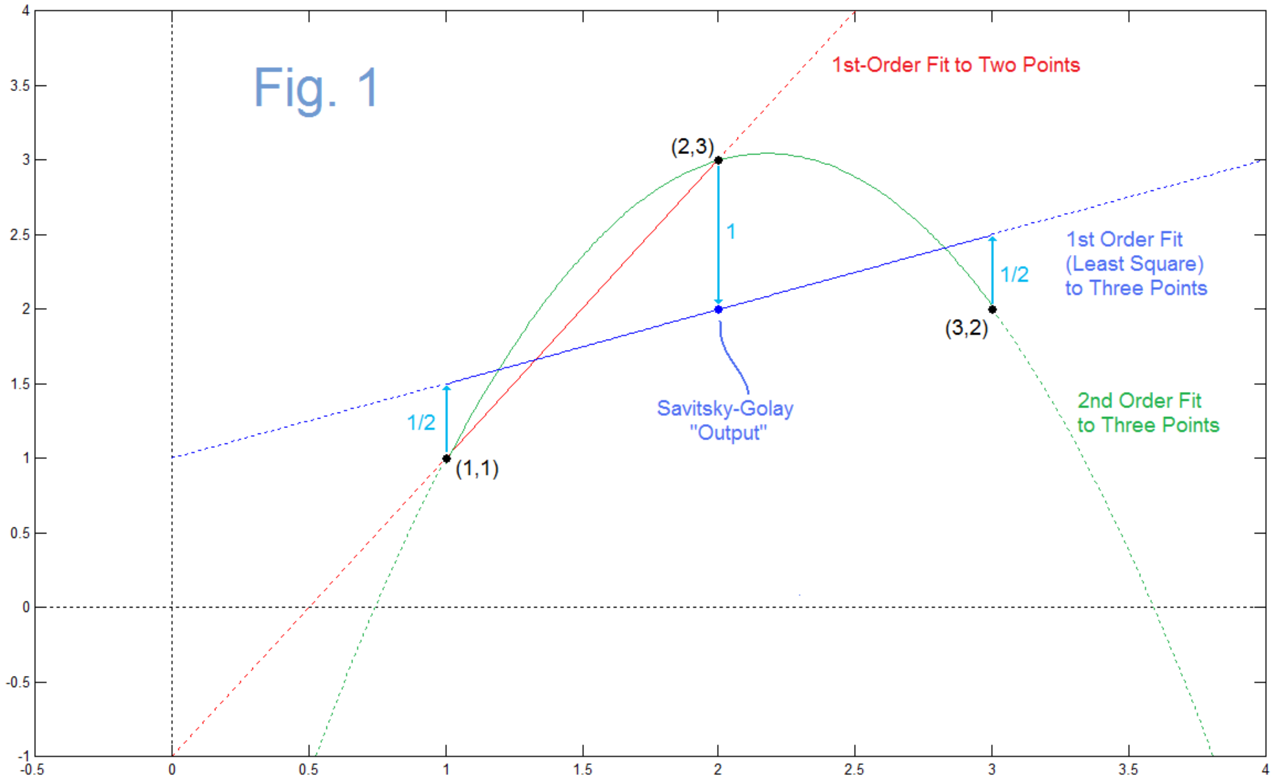
- - - - - - - - - -

* This text was begun in 2002 and I recently found when I was looking for Savitzky-Golay work I thought I had published somewhere! It was on a memory stick that condensed some 258 floppy disks! Fortunately the machine did the search.

Filtering by Fitting a Straight Line to Three Points:

In as much as S-G is polynomial fitting, it seems closely related to our use of polynomials for interpolation [7]. Indeed this is so, perhaps contributing to the confusion as I remember wondering again about S-G and thinking – that's just polynomials. The difference, as we hope to show here, is in the use of the output. In the case of interpolators, we are (obviously) computing samples between existing samples. In the case of the smoothing filters we consider here, the output is a replacement sequence that is smoothed. The simple moving average [2, 11] is the simplest S-G filter possible. It fits a straight line to data points.

Here we will establish a useful perspective, Consider that we have three points x_1, x_2, and x_3. Initially I will assign values of 1, 3, and 2 to these samples. (Another possible set are 0, 1, and 0 if we want to look at some sort of impulse response – see Fig. 2). In Fig. 1 we show the three initial points. The red "curve" is a straight line fit to the first two of the three points. We can only exactly fit two of three points. The green curve is a 2^{nd}-order (parabolic or quadratic) fit exactly to all three points. The two cases (red and green) are useful for interpolation. For example, we might want to estimate a point as interpolated at 1.75. Neither of these is useful for smoothing as they just give back the original points.

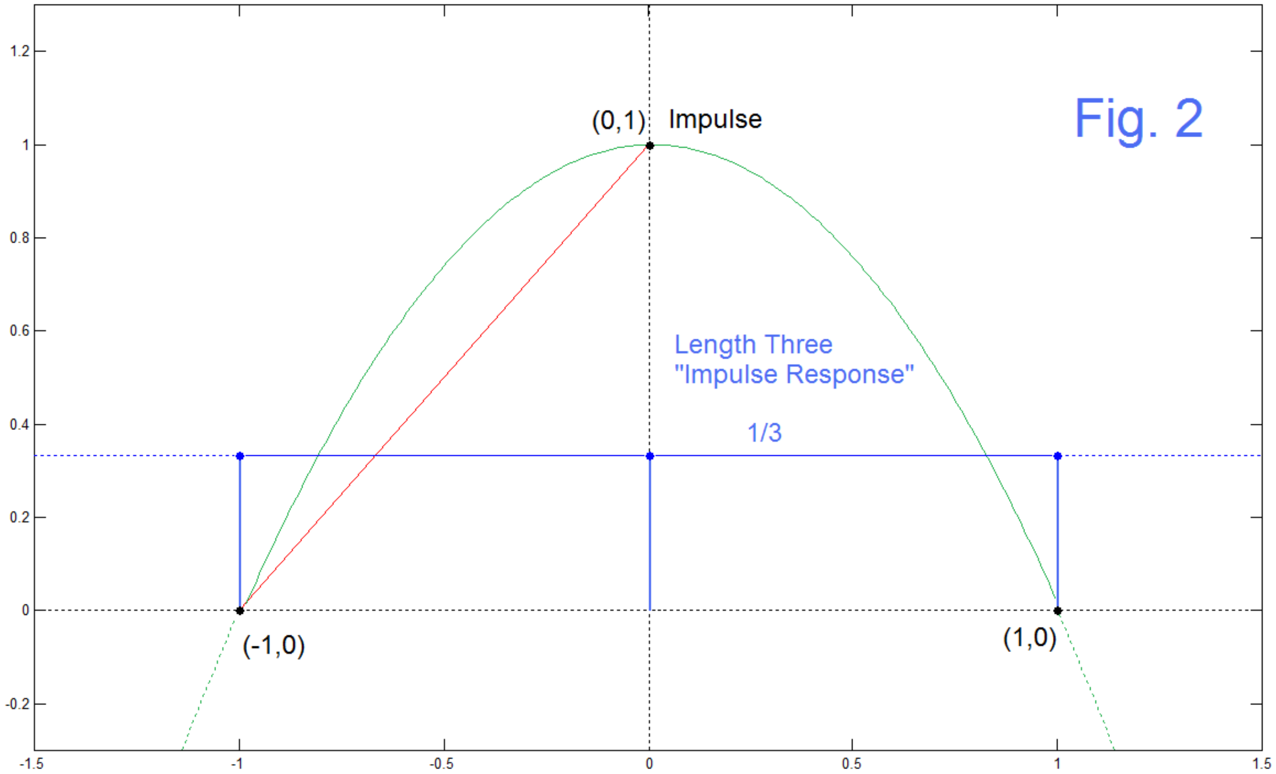


The blue curve (straight line) is in fact an S-G smoother. Here we have fit a straight line to three points by minimizing the squared error as in [2]. The Matlab code that generates Fig. 1 is at the end here as Program 1. There we see that fitting the blue line is a matter of using the "pseudo inverse", ***pinv,*** in Matlab applied to three equations in two unknowns (two parameters of the line to three given points) instead of three parameters for three points (the green parabola) from Matlab's "ordinary" matrix inerter, ***inv***.

For this particular example, the blue line has a total squared error of $(1/2)^2 + 1^2 + (1/2)^2 = 1.5$. One can become convinced that this is a minimum squared error by manipulating the line up/down or tilting it about (2,2) a bit (left as an exercise to the reader or see Program 1). What we evidently should learn from the blue curve is that it is telling us that in light of the fact that the two end points are lower, the midpoint (2,3) is perhaps a bit high, and vice versa, the "corrections" appearing as light-blue arrows. This "smoothing suggestion" applies only to the middle point (at horizontal axis value 2). If we want to know what value is suggested in a smoothed case at horizontal axis value 3, we might consider the blue straight line in Fig. 2 (suggesting 2.5 there) but we are better served by shifting one position to the right. Thus we consider the familiar case where we look for an FIR filter, as in [2]. But how do we get this to be a FIR filter?

The ploy here is a "trick question" that we have used in the case of interpolators and smoothers. We simply need to provide an answer to the question**:**

Q: What is the impulse response of a smoother that fits three points to a straight line?
A: It is the response of a smoother that fits three point to a straight line to an impulse.



Trick answer? Not at all. It's a method. All we need to do here is to replace the three values that were used in making Fig. 1 (1, 3, and 2) with a length-three impulse (a one and two zeros). This is shown in Fig. 2 (also from Program 1) where we have also shifted the input to have the impulse at the zero the horizontal axis. Note that the impulse response is "reconstructed" from the polynomial fit to **different** original points.

Least Squares Polynomial Fitting – Second Example

So far we haven't done much. We just fit polynomials to points (1st and 2nd order) and re-derived the moving average by trying to fit a straight line to three points. For the moment, we note that the impulse response of the length-3 moving average [1/3 1/3 1/3] is an example of S-G, but not one which in itself tells us what we need to do in a general case. The actual more general filters will be done in a bit, but let's be sure we understand the fitting of "too many" points first. Consider as a second example that we might have nine samples

$$y = [\ 1\ \ 2\ \ -1\ \ 0\ \ 2\ \ -3\ \ -1\ \ 2\ \ 1\ ] \qquad (1)$$

positioned at integers along the horizontal axis for x = 1 to 9, and we decide to fit a fifth-order polynomial to them. This is three too many points of course, so we will take our fit to just minimize the squared error on all nine points The most direct way to do this in Matlab is with the build-in ***polyfit*** function

polyfit([1 2 3 4 5 6 7 8 9],[1 2 -1 0 2 -3 -1 2 1],5) (2)

This is the right answer. But it is not difficult to just write your own code and this helps your understanding of course. Program 2 show the code developed for this. The procedure is as follows. We start with the form of the polynomial**:**

$$y = ax^5 + bx^4 + cx^3 + dx^2 + ex + f \qquad (3)$$

where we don't know a, b, c, d, e, or f. Since this is true for all y, it is true for each of the nine values of y at the values of x specified. This we have nine possible equations in 6 unknowns, which is "too many". These we can write in matrix form**:**

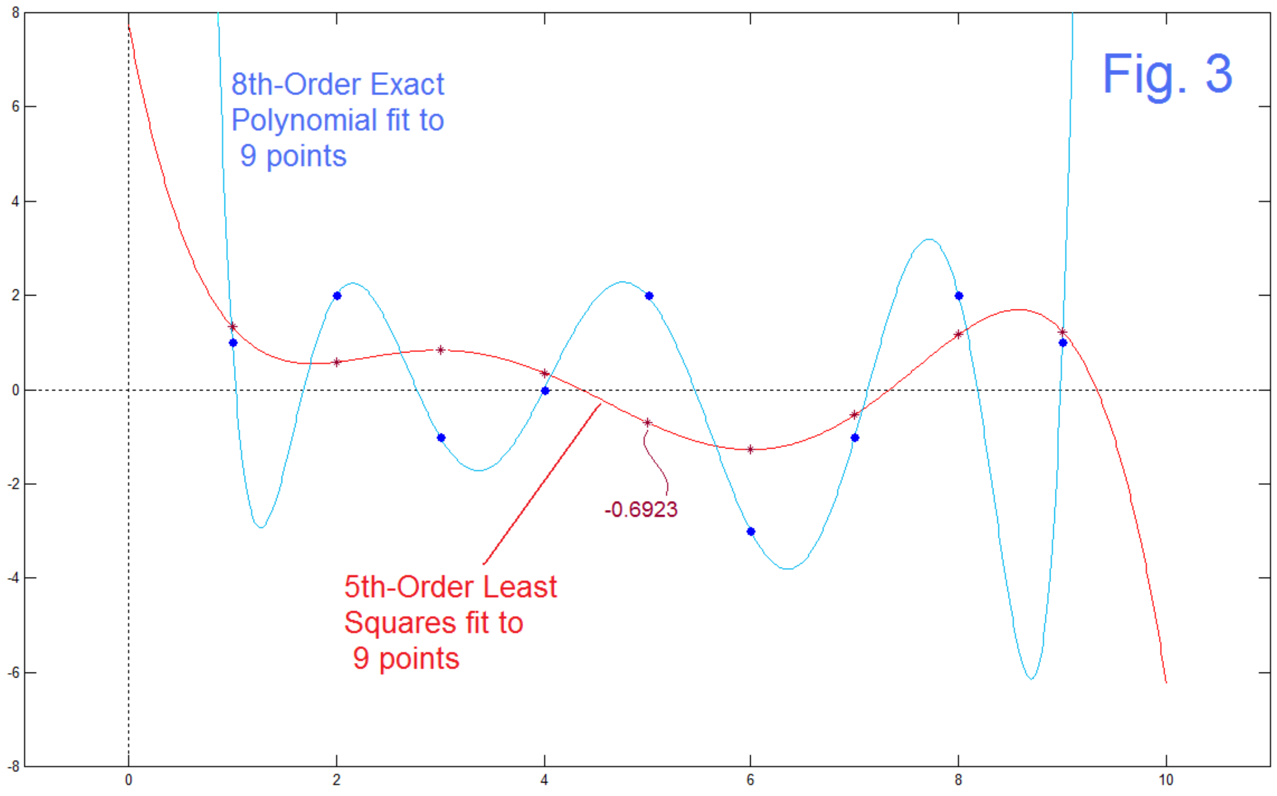
$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ y_9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 32 & 16 & 8 & 4 & 2 & 1 \\ 243 & 81 & 27 & 9 & 3 & 1 \\ 1024 & 256 & 64 & 16 & 4 & 1 \\ 3125 & 625 & 125 & 25 & 5 & 1 \\ 7776 & 1296 & 216 & 36 & 6 & 1 \\ 16807 & 2401 & 343 & 49 & 7 & 1 \\ 32768 & 4096 & 512 & 64 & 8 & 1 \\ 59049 & 6561 & 729 & 81 & 9 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} \qquad (4)$$

This 9x6 matrix can not be exactly inverted (it must be square), Instead we use the "pseudo-inverse" with Matlab's ***pinv*** function. This finds the minimum squared error solution. If your math language does not have a pseudo inverse, you can use the ordinary inverse by the method outlined at the beginning of the program code section at the end of this note. Inverting equation (4) gives:

$$\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \underbrace{\begin{bmatrix} -0.0013 & 0.0035 & -0.0013 & -0.0029 & 0.0000 & 0.0029 & 0.0013 & -0.0035 & 0.0013 \\ 0.0361 & -0.0943 & 0.0288 & 0.0747 & 0.0052 & -0.0695 & -0.0353 & 0.0820 & -0.0280 \\ -0.3875 & 0.9372 & -0.2191 & -0.7067 & -0.1049 & 0.6018 & 0.3473 & -0.6925 & 0.2244 \\ 1.9592 & -4.2181 & 0.6064 & 2.9346 & 0.6789 & -2.2708 & -1.4799 & 2.5986 & -0.8089 \\ -4.6159 & 8.1636 & -0.3111 & -5.0311 & -1.5443 & 3.6383 & 2.6362 & -4.2067 & 1.2709 \\ 4.0000 & -4.7500 & -0.1667 & 2.7500 & 1.0000 & -1.9167 & -1.5000 & 2.2500 & -0.6667 \end{bmatrix}}_{\text{Pseudo-Inverse}} \begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ y_9 \end{bmatrix} \qquad (5)$$

which solves for, with the y values of equation (1):

$$\begin{aligned} a &= -0.0087 \\ b &= 0.2091 \\ c &= -1.8172 \\ d &= 6.9553 \\ e &= -11.7598 \\ f &= 7.7500 \end{aligned} \qquad (6)$$



The Matlab command of equation (2) gives these exact same values for a, b, c, d, e, and f. We now know the polynomial of equation (3) and can compute its value on any range we want. Fig. 3 shows many interesting results. The blue dots are the original nine data points of equation (1). The red curve is the result of computing equation (3) from 0 to 10 (at intervals of 0.01 for plotting purposes). The nine red stars simply overplot this red curve at the integer positions. Note that this red 5th-order polynomial does not go through any of the blue data points (as we expected it would not) and like any polynomial, it runs off to ±∞ as we move out of range of the constraining points [8]. The red star at x=5 has value -0.6923, and would be the output of the S-G smoother at this particular shift. Note that it is quite different from the blue star at x=5 which is at +2. This makes the point that the values of the blue dots at x=3, 4, 6, and 7 are zero or negative and are pulling down the original value of two. In contrast here, we have also plotted the 8th-order polynomial in light blue. This one does go through all the nine original data points. This polynomial too runs off to ±∞ (+∞ in this case) outside the range constrained by the original blue samples. At this point, we can begin to understand how the least square polynomial is starting to smooth the data – the red has visibly less "ripple" than the blue..

So how do we get the filter – the impulse response? Well we just extend the "trick" suggested in the two lines above Fig. 2. We use the smoother on an impulse. We replace equation (1) with:

$$y = [\,0\ \ 0\ \ 0\ \ 0\ \ 1\ \ 0\ \ 0\ \ 0\ \ 0\,] \qquad (7)$$

and the result is shown in Fig. 4.

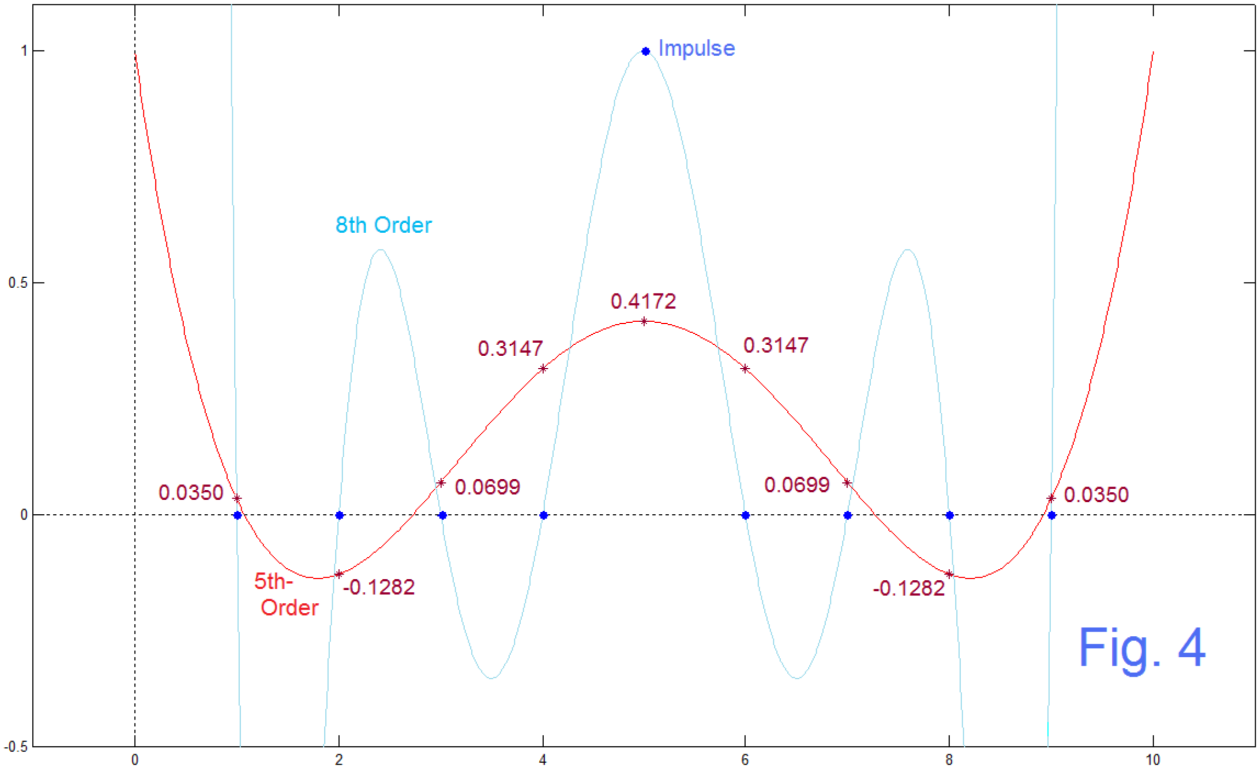


Fig. 4

In this case, we get the response to the impulse, the red curve, and the red stars in this case are exactly the impulse response of the S-G smoother we want. We note here that this agrees with Matlab's ***sgolay(5,9)*** as well as our own program ***sg(5,9***) using our own code that will be presented below.

Here we can verify one case of an FIR filter performing this smoothing. We just take the original 9 blue dot samples (Fig. 3) and multiply them point-by-point with the nine impulse response (red star) values of Fig. 4, and sum the results. It is indeed the value of -0.6923 we said we should get.

Getting to the Filters Faster – and Going from There

All of the above is not difficult, nor does it involve much computation time, particularly s we may be designing only a couple of filters. Right now we want to consider a more general approach where we can extract the S-G impulse response directly as the lowest row of the inverse matrix.

The general form for the equations is (corresponding Matlab code in red):

$$y = \mathbf{M}\,a \qquad [\ y = M*a\ ] \tag{8}$$

analogous to equation (4), which is inverted as:

$$a = \mathbf{M}^{-1}\,y \qquad [\ a = pinv(M)*y\ ] \tag{9}$$

analogous to equation (5). Here a is the *vector* of polynomial coefficients such as we have previously called abcdef, and here we are also considering y to be symmetrical about 0, and more specifically, being an impulse at zero (such as y=[0 0 0 0 1 0 0 0 0] for a length nine result).

Once a is computed, we can "back compute" the actual impulse response at integer values about zero. Thus, with MI = pinv(M) in Matlab:

$$yi = \mathbf{M}\ \mathbf{M}^{\mathbf{-1}}\,y^t \qquad [\ yi = M * MI * y'\ ] \tag{10}$$

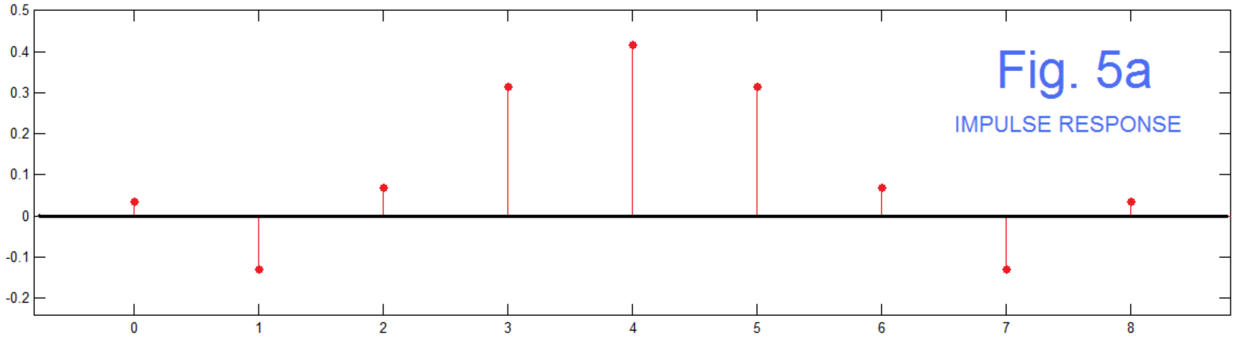
$$\mathbf{=}\ \mathbf{M}^{\mathbf{-t}}\ \mathbf{M}^{\mathbf{t}}\,y^t \qquad [\ yi = MI' * M' * y'\ ] \tag{11}$$

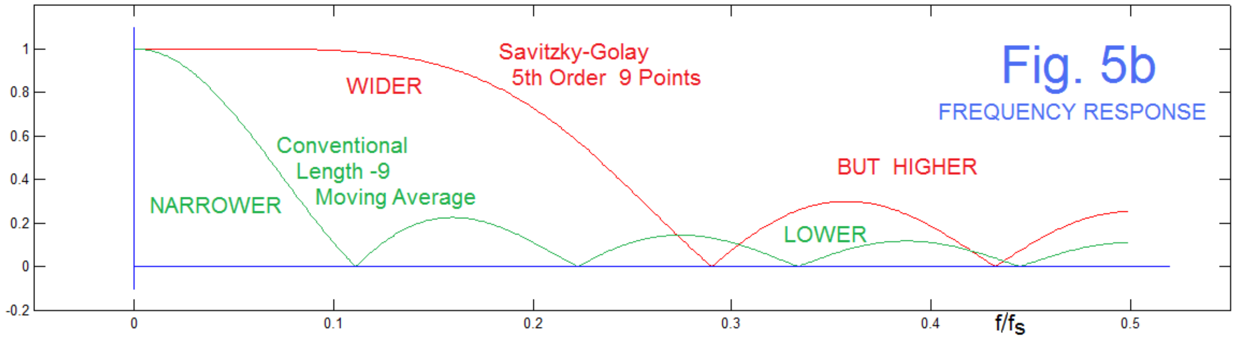
But $\mathbf{M}^{\mathbf{t}}y^t$ is here a column vector of zeros, except for the bottom element that is a 1. [Note that the superscript t is a transpose, -t is the inverse transposed. In Matlab, the **'** indicates the conjugate transpose, but this is just the ordinary transpose since everything is real here.] Thus . $\mathbf{M}^{\mathbf{t}}y^t$ selects the bottom row of the inverse matrix. A nice shortcut – but perhaps hard to explain! As I recall, this was based first on just an observation.

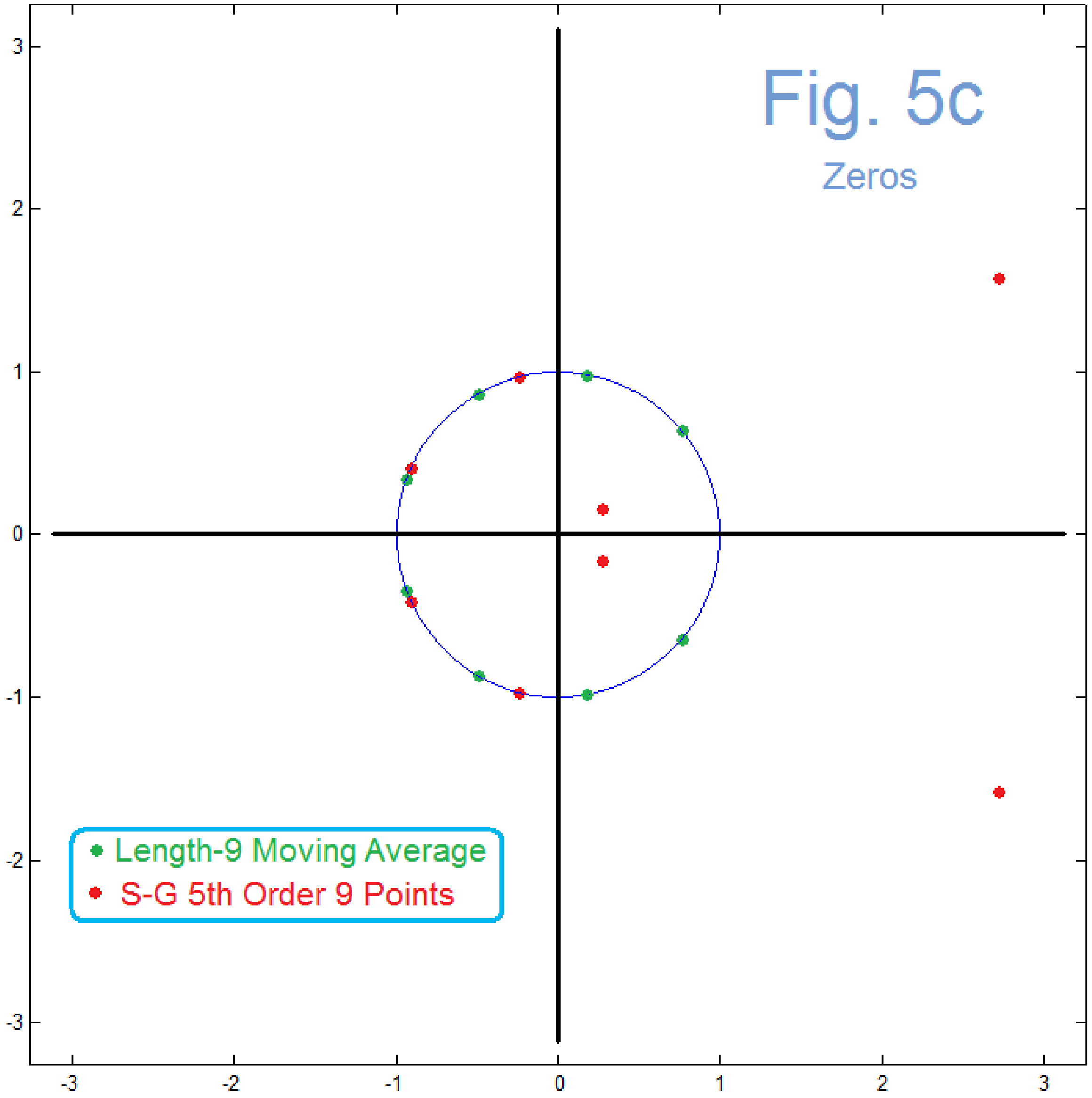
Once the impulse response is found, our usual practice is to look at the frequency response, the phase response, and the pole/zero plot. In this case, the impulse responses are all symmetric (hence linear phase) and are FIR (hence we have only zeros). The program ***sg*** (Program 3 at the end) computes the S-G impulse response and the follow-up frequency domain calculations. These appear to be very nicely behaved filters.

Examples Looked at as Filters

Using the ***sg*** program, we first consider again the previous result of a 5th-Order, 9-Point S-G smoother (red in Fig. 5a, 5b, and 5c), and will compare this to the conventional length-9 moving average (green in Fig. 5b and Fig. 5c – the moving average impulse response would just be 9 values of height 1/9). From Fig. 5b we see the major difference between the moving average and the S-G of the same length. In a sort of standard trade-off, the bandwidth of the S-G starting at zero is wider while it does not drop so low in the upper half. In as much as we often think of smoothing as the removal of higher frequencies without changing the lower ones too much, we see an advantage to the S-G with regard to the passband. We notice from the zeros in the z-plane (Fig. 5c) that the left side zeros of both filters are somewhat the same, while the right side zeros of the moving average are ON the unit circle (causing notches) , while those of the S-G back away. The flatness of the S-G about zero should be noted.

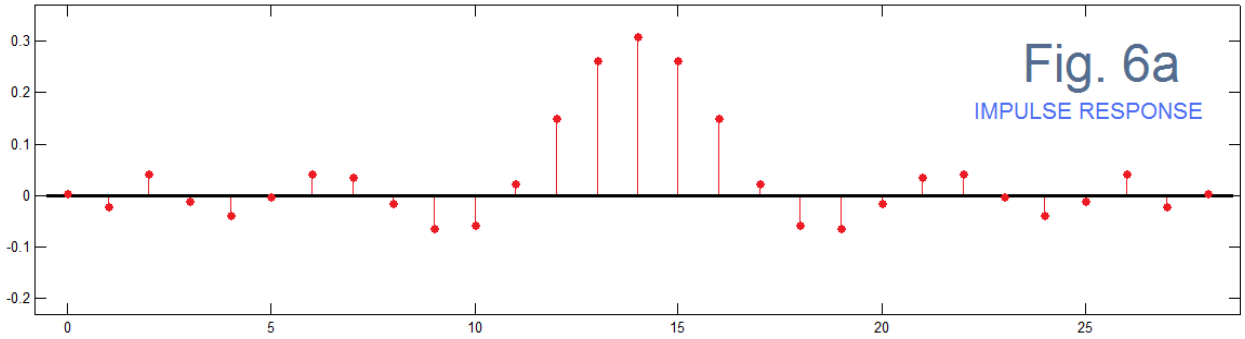


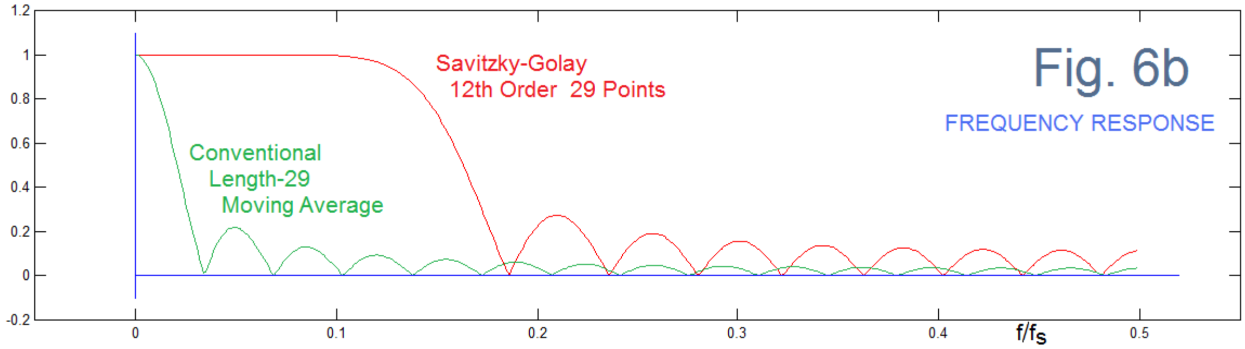




As a second example, we will (just as another example) consider a 12th –order polynomial fit to 29 points. Again we use the ***sg*** program which still plots the corresponding length-29 moving average for comparison. This example, shown in Fig. 6a, 6b, and 6c, is even better for showing the flat, wide passband, along with a somewhat less impressive stopband, relative to the moving average. Again we see in Fig. 6c the unit-circle zeros moving away to accommodate a flat passband.

Readers will likely at this point be inclined to just suppose we have introduced yet another design method for digital filters. Indeed the sinc-like impulse response and various frequency-domain ripples are familiar enough. The response of Fig. 6b looks a lot like the IIR design method using "Inverse Chebyshev" with no attempt to make the stopband exactly equiripple. But plenty of other FIR design procedures (such as

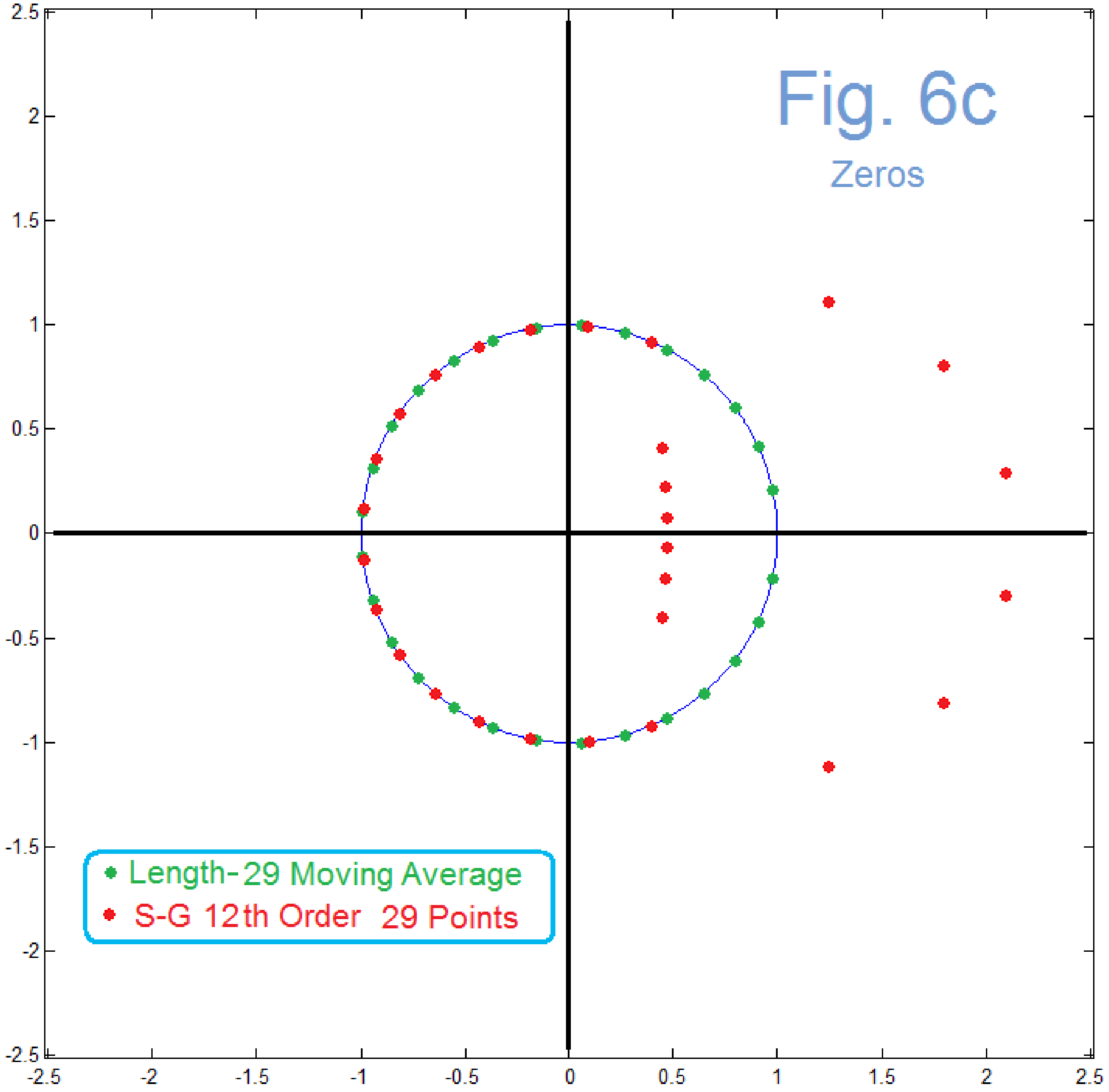




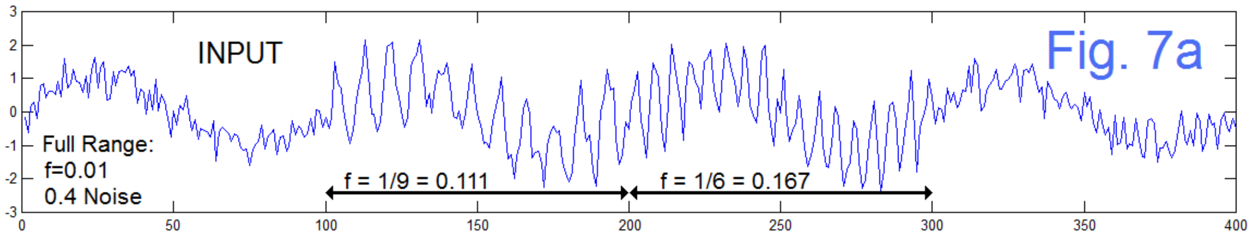
frequency-domain least squares, particularly as they are used with increased weighting on the passband, can be expected to give similar results [9]. While this is likely true, we have not examined it here. What we have found is a rational time-domain-based procedure that makes sense. So on to time-domain.

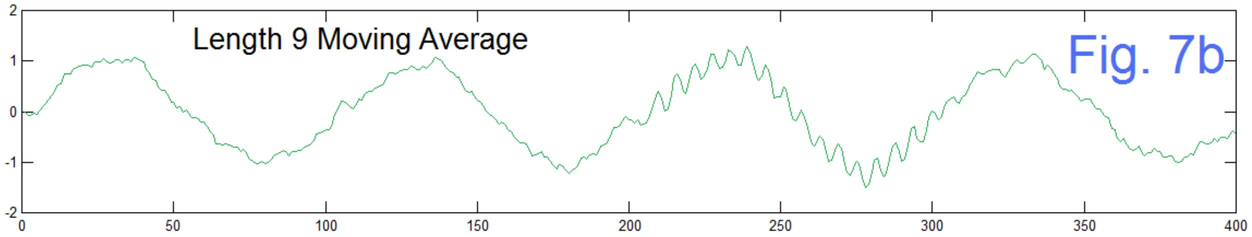
Time-Domain View of Smoothing

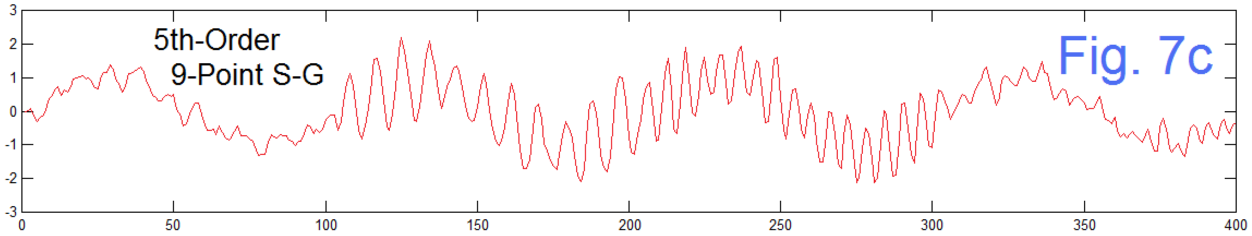
Here we are particularly anxious to see what happens with a time domain test. We have noted that the S-G procedure gives us a symmetric impulse response with certain frequency-domain properties. Since this is an FIR filter, the usual means of actually doing the filtering is a convolution process in the time domain. In fact, it is probably possible to understand most of what you need to know by considering what happens in the time domain. That is, we take our impulse response and position it relative to one end of the sequence to be filtered. Then we move the impulse one step at a time, and for each step we multiply the impulse response values by their overlapped sample value, and sum all the products. This is the output (smoothed) sequence, usually positioned with respect to the center of the impulse response. We then move the impulse response one step to the right and repeat. Of course, there can be significant end effects at the beginning and end of the sequence [10].

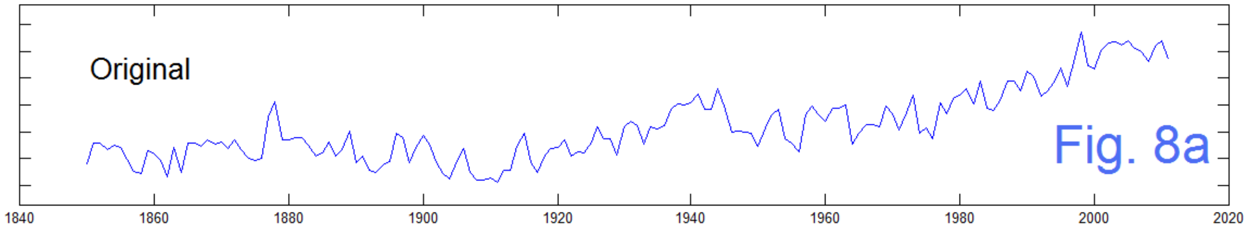


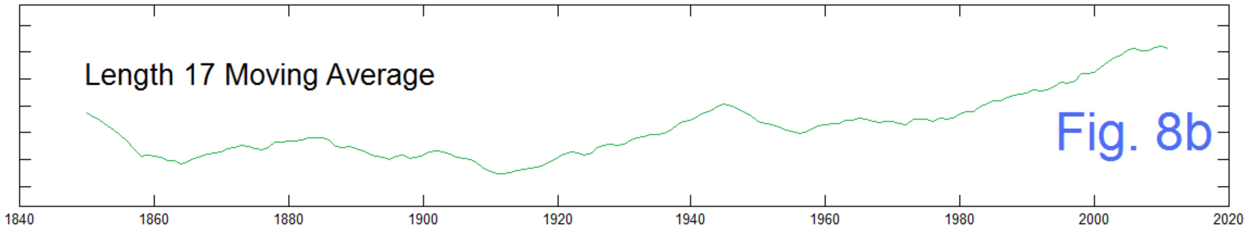
For this note, we will limit our study of test signals to two cases. The first is an artificially generated signal perhaps best understood from Fig. 7a. Here we have a sinewave of frequency 1/100, so there are 4 cycles in the 400 points shown. To this we have added a second sinewave of frequency 1/9 from samples 100 to 200, and a third sinewave of frequency 1/6 for samples 200 through 300. To the entire signal we then added Gaussians noise (Matlab's ***randn***) of amplitude 0.4. We will assume that the three sinewave components are of interest, but we would like to reduce the noise. Our filter is the 5th-order 9-Point S-G of Fig. 5. From Fig. 5b we see that the frequency 0.01 is low enough that the S-G filter (and the corresponding length 9 moving average) will pass it quite well. The other two sinewaves (1/9 and 1/6) which have higher frequency are

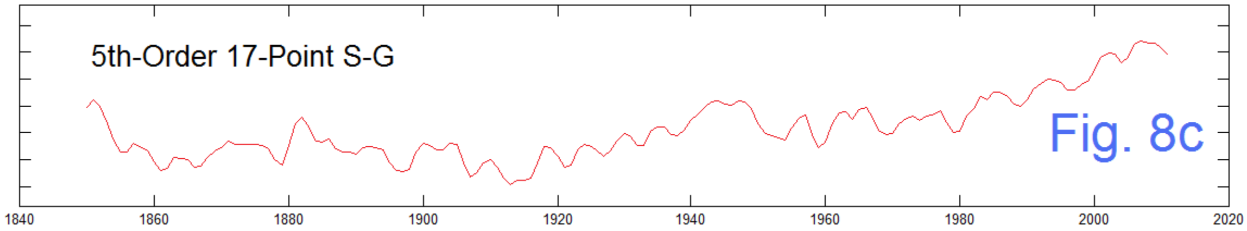

INPUT
Fig. 7a
Full Range:
f=0.01
0.4 Noise
f = 1/9 = 0.111
f = 1/6 = 0.167
3
2
1
0
-1
-2
-3
0
50
100
150
200
250
300
350
400


Length 9 Moving Average
Fig. 7b
2
1
0
-1
-2
0
50
100
150
200
250
300
350
400


5th-Order
9-Point S-G
Fig. 7c
3
2
1
0
-1
-2
-3
0
50
100
150
200
250
300
350
400


Original
Fig. 8a
1840
1860
1880
1900
1920
1940
1960
1980
2000
2020


Length 17 Moving Average
Fig. 8b
1840
1860
1880
1900
1920
1940
1960
1980
2000
2020


5th-Order 17-Point S-G
Fig. 8c
1840
1860
1880
1900
1920
1940
1960
1980
2000
2020

treated differently by the two filters. The S-G passes both well enough. The moving average attenuates both. In particular, the moving average, being length 9, completely blocks the frequency of 1/9 (that's why we put it in the test illustration). Looking at Fig. 7b (green) we may well be pleased to see much of the noise gone, and note that there was indeed a second component from time 200 to 300. The component from 100 to 200 is lost. And we do have a good feel for the sinewave at frequency 1/100 being there. But it could be a serious error if we did want that sinewave at 1/9 to show,

The S-G filter (Fig. 7c, red) quite happily passes both sinewaves, and the 1/100 low frequency, but achieves a less successful effort at removing the noise. So it is a trade-off like many other cases.

Note that it both cases of filtering there is a slight time shift (to the right) due to the linear phase delay. Correcting for delay is easy enough, This and the end effects noted above are refinements secondary to the more major concerns of frequency shaping that we have just illustrated here.

The second of our cases here uses actual data, a famous data series considered to present some measure of global temperature of the Earth over many years (going back to 1850 here – about the end of the "Little Ice Age"), as in Fig. 8. . This is the HadCRUT3 series we looked at a few note back [12, link to data there]. Traditionally, the data has a lot of "noise" and is smoothed over a period of several years or longer. This makes sense.

Although quite arbitrarily chosen, many folks choose a length of 17 years as being significant, so we have chosen here smoothing with a length 17 moving average (Fig. 8b) and with a 5th-Order, 17-Point S-G. The moving average is quite successful in showing the temperature increases over the approximately 30 year spans following 1850, 1910, and 1970, with the flatter (or slight declines) starting at 1910, 1940, and 2000, along with the overall increase since 1850. More detail remains with the S-G filtering, which may or may not be significant, or of interest to us.

The lesson from these time studies seems to suggest that hard-and-fast rules aren't going to be too useful. We try something, and see if it helps or hurts,

REFERENCES

[1] J.S. Lim and A.V. Oppenheim, eds. ***Advanced Topics in Signal Processing***. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988 .

[2] B. Hutchins, "Time Domain Least Square Low-Pass Filters," Electronotes Application Note No. 318, February 1992 http://electronotes.netfirms.com/AN318.PDF

[3] Savitzky and Golay, Analytical Chemistry Vol. 36, 1627, (1964)

[4] Sophocles Orfanidis ***Introduction to Signal Processing***: Prentice Hall (1996) pp 434-462

[5] Richard Hamming, ***Digital Filters***, Prentice-Hall, Englewood Cliffs, NJ (1977), pp 37-56.

[6] Richard Hamming, ***Numerical Methods for Scientists and Engineers,*** Dover (1987) pg 468, pp 570-571 .

[7] B. Hutchins, "Polynomial Fitting for Sample-Rate Changing at Rational and Irrational Frequency Ratios", Electronotes Application Note AN-317, January 1992
http://electronotes.netfirms.com/AN317.PDF
See also Section 4b of our digital filter design series at:
http://electronotes.netfirms.com/EN198.pdf

[8] The point here is that signals are horizontal and polynomials are vertical! See B. Hutchins, "Models – Good, and Bad: and the (Mis-)Use of Engineering Ideas in Them," ***Electronotes***, Vol. 22, No. 211, July 2012. See in particular Fig. 1 on page 5. The issue is here: http://electronotes.netfirms.com/EN211.pdf

[9] See our digital filter design series in EN#198, page 11, Fig. 23b. The issue is here:
http://electronotes.netfirms.com/EN198.pdf

[10] B. Hutchins, "Averaging - and Endpoint Garbage," Electronotes Application Note No. 395, March 30, 2013
http://electronotes.netfirms.com/AN395.pdf

[11] B. Hutchins. "Yearly Moving Averages as FIR Filters," Electronotes Application Note No. 401, Dec 22, 2013
http://electronotes.netfirms.com/AN401.pdf

[12] B. Hutchins, "Fun Plotting Your Own Global Warming Curves," Electronotes Application Note No. 379, July 2012
http://electronotes.netfirms.com/AN379.pdf

PROGRAMS

PSEUDO-INVERSE FROM INVERSE

In many cases (like here) we can obtain the pseudo-inverse from just the inverse as:

$$MI = (M^tM)^{-1}M^t$$

where M^t is the usual transpose operation. So if your math package has an inverse, you can do the pseudo-inverse in this way. In Matlab, this would be MI =inv(M'*M)*M'. You can practice on the examples of equations (4) and (5)

Program 1- Produces Figs 1 and 2, Etc.

```
% AN404Fig1-2.m
x1=1
x2=3
x3=2

figure(1)
plot([1 2 3],[x1 x2 x3],'ko')
hold on
%
% fit line to first 2
plot([1 2],[x1 x2],'r')
plot([0 1],[-1 1],'r:')
plot([2 3],[3 5],'r:')
plot([0 0],[-1 5],'k:')
plot([-1 4],[0 0],'k:')
%
%
% Fit quad to first 3
abc=inv([1 1 1;4 2 1;9 3 1])*[1 3 2]';
x=0:.01:4;
y=abc(1)*x.^2 + abc(2)*x +abc(3);
plot(x(1:100),y(1:100),'g:')
plot(x(301:400),y(301:400),'g:')
plot([1:.01:3],y(100:300),'g')
%
%
% Fit line to three points
ab=pinv([1 1;2 1;3 1])*[1 3 2]';
ysl=ab(1)*x + ab(2);
plot(x(1:100),ysl(1:100),'b:')
plot(x(301:400),ysl(301:400),'b:')
plot([1:.01:3],ysl(100:300),'b')
%
ysl(201)   % center point
%
```

```
% code below monkeys with y-intercept to verify square error min
% can do similar for slope
% error increases from 1.5 to 1.5003 in both cases
e1=ysl(101)-1
e2=ysl(201)-3
e3=ysl(301)-2
e1sq=e1^2+e2^2+e3^2
e1=ysl(101)-.01-1
e2=ysl(201)-.01-3
e3=ysl(301)-.01-2
e11sq=e1^2+e2^2+e3^2
e1=ysl(101)+.01-1
e2=ysl(201)+.01-3
e3=ysl(301)+.01-2
e12sq=e1^2+e2^2+e3^2
%
plot(2,2,'ob')
hold off
axis([-.5 4.0 -1 4])
figure(1)
%
% Now do Impulse Response
%
x1=0
x2=1
x3=0
figure(2)
plot([-1 0 1],[x1 x2 x3],'ko')
hold on
%
% fit line to first 2
plot([-1 0],[x1 x2],'r')
plot([0 0],[-1 5],'k:')
plot([-4 4],[0 0],'k:')
%
% Fit quad to first 3
abc=inv([1 -1 1;0 0 1;1 1 1])*[x1 x2 x3]';
x=-2:.01:2;
y=abc(1)*x.^2 + abc(2)*x +abc(3);
plot(x(1:100),y(1:100),'g:')
plot(x(300:400),y(300:400),'g:')
plot([-1:.01:1],y(100:300),'g')
%
% Fit line to three points
ab=pinv([1 1;2 1;3 1])*[x1 x2 x3]';
ysl=ab(1)*x+ab(2);
plot(x,ysl,'b:')
plot([-1:.01:1],ysl(100:300),'b')
%
% plot impulse response
plot(-1,1/3,'bo')
plot( 0,1/3,'bo')
plot( 1,1/3,'bo')
hold off
axis([-1.5 1.5 -.3 1.3])
figure(2)
```

Program 2 – for Fig. 3 (and 4)

```
% AN404Fig3.m
clear
for x=1:9
   for m=5:-1:0
      M(x,m+1)=x^(5-m);
   end
end
M

x=1:9
y=[1 2 -1 0 2 -3 -1 2 1]
%y=[0 0 0 0 1 0 0 0 0]   % for impulse Fig. 4
xx=0:.01:10;
abcdef=pinv(M)*y'
a=abcdef(1);
b=abcdef(2);
c=abcdef(3);
d=abcdef(4);
e=abcdef(5);
f=abcdef(6);
yy=a*xx.^5 + b*xx.^4 + c*xx.^3 + d*xx.^2 + e*xx + f;

figure(1)
plot([1:9],y,'o')
hold on
plot(xx,yy,'r')

plot([0 0],[-10 10],'k:')
plot([-5 15],[0 0],'k:')

yi=yy(101:100:901)
plot([1:9],yi,'*r')

%Matlab for complete fit, 8th order - could also do as above
abcdefghi=polyfit([1 2 3 4 5 6 7 8 9],y,8);
a=abcdefghi(1);
b=abcdefghi(2);
c=abcdefghi(3);
d=abcdefghi(4);
e=abcdefghi(5);
f=abcdefghi(6);
g=abcdefghi(7);
h=abcdefghi(8);
i=abcdefghi(9);
yy8=a*xx.^8 + b*xx.^7 + c*xx.^6 + d*xx.^5 + e*xx.^4 + f*xx.^3+ g*xx.^2 + h*xx +
i;
plot(xx,yy8,'c')
% end comparison plot for 8th order
hold off
axis([-1 11 -0.5 1.1])
figure(1)
```

Program 3 – for Fig. 5 and 6, etc.

```
function [h,M,MI]=sg(order,points)          % Nov. 16, 2002 (Revised Feb. 2014)
clf
pt=points;
or=order;
M=zeros(pt,or+1);
n=-(pt-1)/2:(pt-1)/2;
for k=0:or
     c=(n.^k)';
     for r=1:pt
         M(r,or+1-k)=c(r);
     end
end
M
MI=pinv(M)
%
for k=1:pt
   h(k)=MI(or+1,k);
end
h
% Main computation done at this point -we have h
%
% Continue with Plots and Reference Plot (mov. avg.)
figure(1)
subplot(211)
hmax=max(h);
hmin=min(h);
if hmin>0; hmin=0; end
stem([0:pt-1],h)
hold on
plot([0 pt+1],[0,0],'b');
axis([-.8  pt-1+.8  -0.1*hmax-0.2 1.2*hmax]);
hold off
%
href=(1/pt)*ones(1,pt);
HREF=abs(freqz(href , 1, 500));
%
subplot(212)
H=abs(freqz(h,1,500));
Hmax=max(H);
plot([0:.001:.499],H,'r');
hold on
plot([0:.001:.499],HREF,'g')
plot([0 .52],[0 0],'b')


plot([0,0],[-.1 1.1*Hmax],'b')
axis([-.05 .55 -.2 1.2*Hmax])
hold off
% continue for plot of zeros
zer=roots(h)
zerref=roots(href);
```

```
xmax=max(abs(real(zer)));
ymax=max(abs(imag(zer)));
plotmax=max([xmax ymax])
c=exp(j*2*pi*[0:1000]/1000);
figure(2)
plot(c)
hold on
for k=1:length(h)-1;
   plot(real(zer(k)),imag(zer(k)),'or')
   plot(real(zerref(k)),imag(zerref(k)),'og')
end
axis('equal')
pm=1.2*plotmax;
axis([-pm pm -pm pm])
hold off
figure(2)
% end plotting
%
% code below for verification only
abcdef=MI*[0 0 0 0 1 0 0 0 0]'
b=abcdef(2)
d=abcdef(4)
f=abcdef(6)
for k=-4:4
   yi(k+5)=b*k^4+d*k^2+f;
end
yi
% yi same as h
yiM = M*abcdef
% yiM same as h
MIy=MI*[0 0 0 0 1 0 0 0 0]'
yii=M*MIy
% yii same as h
yiii=M*MI*[0 0 0 0 1 0 0 0 0]'
% yiii same as h
Mtx=M'*[0 0 0 0 1 0 0 0 0]'
% column vector is transpose of [0 0 0 0 0 1]
MIMtx=MI'*Mtx
% selecting bottom row of MI
yiiii=MI'*M'*[0 0 0 0 1 0 0 0 0]'
% yiii same as h, equation (11) of text
```

Program 4 – for Fig. 7 and 8, etc.

```
function sgt(h)
% h is S-G impulse response of other test filter
ln=length(h)
% hma is moving average for reference
hma=(1/ln)*ones(1,ln);
% create test signal
s=[zeros(1,100),sin(2*pi*[0:99]/9),sin(2*pi*[0:99]/6),zeros(1,100)
]+0.4*randn(1,400);

s=s+sin(2*pi*[0:399]/100);
y=0:399;
% remove two comment markers (%) below to override
%   load gt;    % see AN-379 for data
%   s=A;
%
sma=filter(hma,1,s);
ssg=filter(h,1,s);
mas=1.3*max(abs(s));
%
figure(3)
ymin=min(y)-10;
ymax=max(y)+10;
subplot(311)
plot(y,s)
axis([ymin ymax -mas mas])
subplot(312)

plot(y,sma)
axis([ymin ymax -mas mas])
subplot(313)
plot(y,ssg)
axis([ymin ymax -mas mas])
figure(3)
```

In the programs above, much is redundant. We have included these programs for the usual reason of unambiguous documentation, and as a means of giving example code with a number of different approaches.