5.10 Analysis of the tides (AE3 pp. 246-253)

So far in this chapter we have seen that Fourier transformation can be competitive with least squares analysis in, say, differentiation and integration. Below we will give an extensive analysis of a particular data set to illustrate how we can often *combine* Fourier transformation and least squares analysis for efficient data fitting of periodic phenomena. Each method has its own strengths and weaknesses: Fourier transformation can show us many simultaneous frequency components, but has limited frequency resolution, which may lead to leakage. Least squares fitting is more flexible in what it can fit, but needs more extensive guidance. Because the two methods often *complement* each other, their combined use can make a very powerful data analysis tool.

The tides have been understood quantitatively through the work of such scientific giants as Newton, Euler, Daniel Bernoulli, Laplace, and Kelvin as due to the combined effects of lunar and solar attraction. What we experience as tides is the differential effect of the attractive forces on the solid earth and those on the more mobile surface water, modulated by the shape (size and depth profile) of the particular body of water and by the cohesive forces that produce drag to water movement, and further modified by wind, barometric pressure, and local currents (as where rivers meet oceans). We need not look here into its detailed mathematical description, but merely consider the tidal record as a signal that should have as its principal frequency components the lunar and solar periods, and take it from there. Fortunately, tidal records are readily available on the web from NOAA, and we will use one such record. You are of course welcome to select a record from another location, and/or pick a different time period. Since arbitrarily chosen data sets seldom contain precisely $2n$ data points, we will deliberately take a record that does not fit that restriction, and then select a subset of it whenever we need to use Fourier transformation.

Exercise 5.10:

(1) Go to the website co-ops.nos.noaa.gov/, and under Observations select Verified/Historical Water Level Data: U.S. and Global Coastal Stations.

(2) Select a station; in the example given below we have used 8410140 Eastport, Passamaquoddy Bay, ME, but you can of course pick another.

(3) Specify a time interval (we have used W2, hourly heights), a Begin Data (here: 20010601 for 2001, June 1) and an End Data (here: 20010831, for August 31 of that same year, yielding a 2208-hour period).

(4) Take a preview of the data in ViewPlot.

(5) Select the data with View Data, highlight them all with Edit ⇨ Select All, and copy them to the clipboard with Ctrl∪c. Minimize or close the web site.

(6) Start Word, then click Open, Look in: Windows, select Notepad.exe, and paste the file into it with Ctrl∪v. Save the file as a Notepad file using any name that suits your fancy. As you will see in the next few steps, Notepad triggers Excel to open its Text Import Wizard, which is useful to format the data properly.

(7) Open Excel, Select Open, then specify Files of type: as All Files (*.*) so that you will see the just-saved Notepad .txt file, and select it.

(8) You will now see Step 1 of the Text Import Wizard, in which you specify that the data are of Fixed width, i.e., they are tab-delimited. Preview the file to see where the file header (containing all the explanatory text) ends, and then specify the row at which to start importing the data. (In our example, that would be at row 23.) Move to the next Step.

(9) In the Data preview of Step 2 of the Text Import Wizard, enter lines to define the columns you want (in our example, at lines 8, 12, 13, 15, 16, 19, 21, 27, 32, 35, and 40. You can use fewer columns, but then you will have more cleanup to do. Click Finish.

(10) You will now have all the data in your spreadsheet, in columns, starting in cell A1. In the first column replace the station number (8410140) by a row counter: 0 in the top row, 1 in the next row, etc. Delete all peripheral columns, such as the one containing the year (2001), a slant (/), minutes (:00).

(11) You can also delete the right-most columns, except the column between 35 and 40 that had been labeled Sigma, which you may want to save for the end of the exercise. Regardless of whether or not you save this column, first place the instruction `=STDEV(F3:F2210)` (or whatever appropriate range) at its top to compute the standard deviation of the fit between the observations and the predicted data. In our example it is only 0.006 m, or 6 mm, out of an average tidal swing of several meters!

(12) Insert two rows at the top, and use the higher one of these to enter the labels time, month, data, hour, and height (after having made sure that these labels are indeed appropriate). Also label the next two columns Hcalc and residuals. These labels and data will occupy columns A through G.

(13) Plot the water heights versus time t, in hours.

Figure 5.10.1 illustrates the 2208 data points so imported, as a function of time. It clearly shows a periodic oscillation, with a somewhat variable amplitude that is slightly more pronounced and alternating at its tops than at its bottom values.

For Fourier analysis we take the last 2048 data points, thereby leaving some space near the top of the spreadsheet. After their Fourier transformation we calculate and plot the magnitude of the response as a function of frequency.

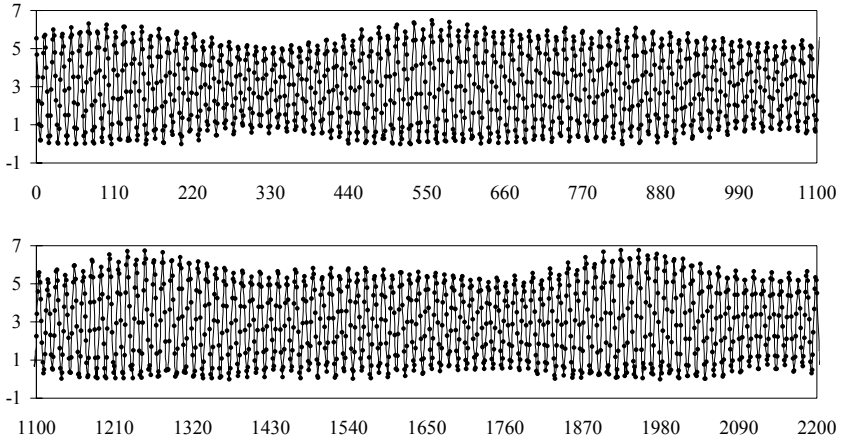


Fig. 5.10.1: The height of the water (as measured in meters vs. the 'average lowest low water level') at Eastport, ME, as a function of time (in hours) during the period from June 1 through Aug. 3, 2001.

Exercise 5.10 (continued):

(14) In row 163 (or wherever you find $t = 160$) copy the water level in, say, column J, and in column I enter the shifted time $t - 160$. Copy both down to the end of the data file. You should now have 2048 data in columns I and J. Highlight these, extend the highlighted area to include column K, and call the forward Fourier transform macro.

(15) In column O calculate the square root of the sum of the squares of the real and imaginary components so obtained, and plot these versus the frequency (in column L).

The result is illustrated in Fig. 5.10.2 at three different vertical scales. The top panel shows a large contribution, of value 2.991, at zero frequency. This component merely reflects the average value of the signal, which is measured versus a 'average lowest low level' in order to make most data values positive quantities. Indeed, by using the function =AVERAGE(*range*) to calculate the average we likewise obtain 2.991.

The largest peak at a non-zero frequency is found at $f = 0.0806\ \mathrm{h}^{-1}$, a value that roughly corresponds with half a moon day of 24 h 50 min 28.32 s or $1/12.4206\ \mathrm{h}^{-1} = 0.0805114\ \mathrm{h}^{-1}$. This peak has a rather wide base, suggesting that it may be broadened by multiple components and/or leakage. In addition, there are two series of minor peaks, one at integer multiples of $0.08\ \mathrm{h}^{-1}$, i.e., at 0.16, 0.32, 0.40 and $0.48\ \mathrm{h}^{-1}$, the other at half-integer multiples of the same value, at 0.04, 0.12, 0.20, 0.28, 0.36, and $0.44\ \mathrm{h}^{-1}$. Neither series has quite died out at $f = 0.5$, and one can therefore assume that there will be higher-order terms as well, which can only be observed using longer data records.

We can either fit these data on a purely empirical basis, or try to identify signals with known astronomical time constants, as we did in the above paragraph. The latter approach, which introduces independently obtainable information into the data analysis, is usually the more powerful, and will be pursued here. We therefore fit the data to an adjustable constant a_0 plus a sine wave of adjustable amplitude a_1 and phase shift p_1 but with a fixed frequency f_1 of $0.0805114\ \mathrm{h}^{-1}$, i.e., to $h = a_0 + a_1 \sin(2\pi f_1 t + p_1)$ where t is time in hours, starting with 0 at the first data point. We then calculate the residuals, and Fourier-transform them in order to find the next-largest term(s), etc.

Exercise 5.10 (continued):

(16) Arrange labels and values for the adjustable parameters *a*0, *a*1, and *p*1, in one column, and in another (leaving at least one space in-between) the fixed parameter *f*1. Specify *a*0, *a*1, and *p*1 as zero, and *f*1 as 0.0805114.

(17) In column F compute the water height *hcalc* using the assumed parameters $a0$, $a1$, and $p1$, and in column G calculate the difference between the measured and calculated water heights.

(18) Also deposit a label and cell for the computation of SSR as =SUMXMY2 (E3:E2210,F3:F2210) or for whatever the appropriate ranges are.

(19) Call Solver to minimize SSR by changing the values of $a0$, $a1$, and $p1$.

(20) In cell R163 repeat the count of $t - 160$ that you already used in cell I163, and in cell S163 copy the residual from G163. Copy these down to row 2210. Highlight R163:T2210, apply the forward Fourier transformation, in row X calculate the corresponding magnitude (i.e., the square root of the sum of squares of the real and imaginary components) of the Fourier transform, and plot these.

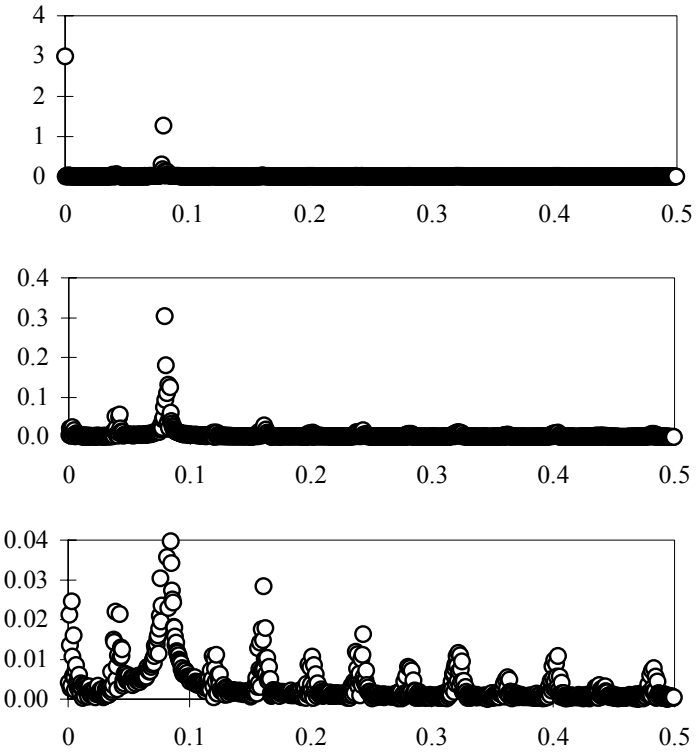


Fig. 5.10.2: Results of the Fourier analysis of 2048 data from Fig. 5.10.1, shown here as the magnitudes of the resulting frequency components, at three different vertical scales, in m. The horizontal scale shows the frequency, in h^{-1}. For a more compact representation of these data see Fig. 1.3.6.

The next most important term, clearly visible in Fig. 5.10.3, is at 0.079 h^{-1}, and is due to the ellipticity of the lunar orbit, which has a period of 27.55 days. As the moon travels from its perigee (at the shortest moon-earth distance) to its apogee (farthest away) and back, the gravitational attraction changes, and in our linear analysis this translates as a difference frequency. Indeed, the corresponding first-order correction term has a frequency of $0.0805114 - 1 / (24 \times 27.55) = 0.0805114 - 0.0015124 = 0.078999\ h^{-1}$.

Exercise 5.10 (continued):

(21) Extend the parameter lists to accommodate $a2$, and $p2$ as well as $f2 = 0.078999$, and add a corresponding, second sine wave to the instructions in column F. In order to facilitate later use of SolverAid, place all adjustable coefficients (i.e., the amplitudes and phase shifts) in a single, contiguous column, one below the other.

(22) Rerun Solver, now simultaneously adjusting the five coefficients $a0$, $a1$, $p1$, $a2$, and $p2$.

(23) Rerun the Fourier transform of the residuals, and look at the updated plot of these residuals.

(24) The next-highest peak in the residual plot is at 0.083 h^{-1}, close to the frequency of $2/24 = 0.083333\ h^{-1}$ associated with half the solar day.

(25) After you include this frequency and repeat the protocol sketched in points (18) through (20) you will find that there is yet another frequency near 0.08 h^{-1}, viz. at about 0.082 h^{-1}, which can be identified with the *sum* frequency $0.0805114 + 1 / (24 \times 27.55) = 0.0805114 + 0.0015124 = 0.082024\ h^{-1}$.

(26) Also incorporate this frequency, call Solver to adjust the nine resulting coefficients $a0$ through $a4$ and $p1$ through $p4$, Fourier-transform the residuals, and plot them.

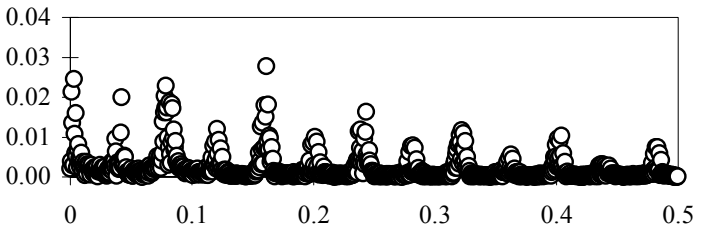


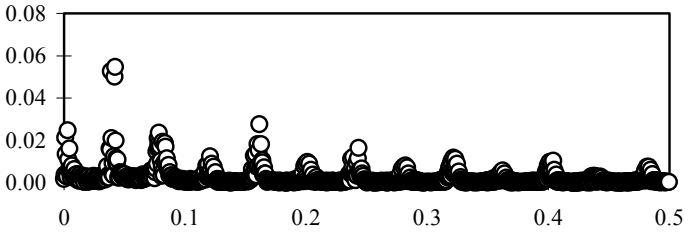
Fig. 5.10.3: The magnitudes of the residual frequency components, after accounting for the average and the leading sinusoidal component at $f1 = 0.0805114\ h^{-1}$.

(27) Extend the parameter lists to accommodate four new frequencies, amplitudes, and phase angles, and include them in the instruction for the calculated heights in column F.

(28) Set the frequencies at $f1/2$, $f2/2$, $f3/2$, and $f4/2$, and subsequently let Solver adjust the amplitudes $a0$ through $a8$ and $p1$ through $p8$.

(29) After you have done this, run SolverAid (which requires that $a0$ through $a8$ and $p1$ through $p8$ form one contiguous column) to calculate the standard deviations of the coefficients.

Fig. 5.10.4: The magnitudes of the residual frequency components, after



accounting for the average and four sinusoidal component near $0.08\ h^{-1}$.

The resulting Fig. 5.10.4 shows that we finally have accounted for the *four* major frequency components near $0.08\ h^{-1}$. Even though the Fourier analysis showed only one peak around $0.08\ h^{-1}$, we used astronomical information to resolve this into four different signals, exploiting least squares analysis to find their amplitudes and phase angles. The *combination* of different methods is often more powerful than each method by itself.

The next-largest contributions are around $0.04\ h^{-1}$. We therefore extend the analysis with four more frequencies, each one-half of the corresponding values near $0.08\ h^{-1}$, and subsequently use Solver to adjust the coefficients, which now number 17. As can be seen in Fig. 5.10.5, with the four frequencies we have found so far we can indeed represent the general envelope of the tidal curve, but not its alternating amplitudes or other details.

Table 10.5.1 lists the results so obtained for the (absolute values of the) various amplitudes; the phase angles are needed for the analysis but have no physical meaning because they are tied to the particular starting time chosen. We see that we can represent most of the signal in terms of predictable periodic functions, so that tide tables can indeed anticipate the tides. Such tables are, of course, based on much longer data sets (so as to include the length of the moon's *node*, a period of about 18.6 years) and on using more frequencies.

frequency	***amplitude***	***frequency***	***amplitude***
0	2.993 ± 0.004	standard deviation of the fit: 0.19	
0.03950	0.023 ± 0.006	0.078999	0.568 ± 0.006
0.040256	0.034 ± 0.006	0.080511	2.620 ± 0.006
0.041012	0.006 ± 0.006	0.082024	0.215 ± 0.006
0.041667	0.158 ± 0.006	0.083333	0.286 ± 0.006

Table 5.10.1: The amplitudes found, with their standard deviations as provided by SolverAid, for the nine frequencies considered so far.

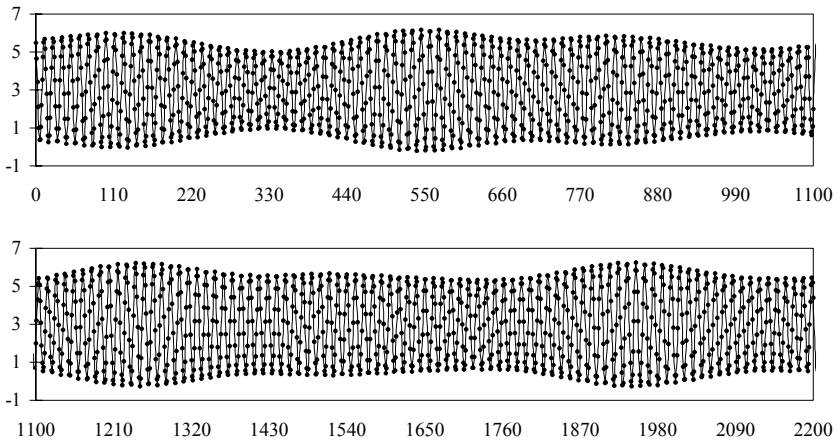


Fig. 5.10.5: The tides recalculated using the four principal frequency components near 0.08 h^{-1}. Comparison with Fig. 5.10.1 shows that this indeed represents the dominant longer-term features of the experimental data.

We see that only one of the four half-frequency components is important, and that (using three times the standard deviation as our criterion) one of them is not even statistically significant. However, the Fourier transform shows that not all frequency components around 0.04 h^{-1} have been accounted for, since there is a remaining signal at about 0.0386 h^{-1}, which we can tentatively associate with the difference frequency 0.0805114 / 2 – 0.0015128 = 0.038743 h^{-1}. Indeed, if we replace the non-significant frequency 0.041012 by 0.038743, run Solver again, and then Fourier-transform the residuals, we find that that all remaining components have amplitudes smaller than 0.03 m, see Fig. 5.10.6.

By comparing the data in Tables 5.10.1 and 5.10.2 we see that changing one frequency can alter the amplitudes of the neighboring frequencies, and we therefore look into the mutual dependence of these results. SolverAid can provide the corresponding array of linear correlation coefficients, in this case an array of 17 by 17 = 289 numbers. Below we show how we can quickly screen them for significant correlations.

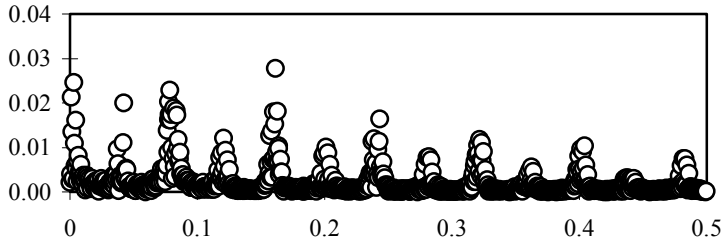


Fig. 5.10.6: The magnitudes of the residual frequency components, after accounting for the average and eight sinusoidal components near 0.08 and 0.04 h^{-1}.

frequency	***amplitude***	***frequency***	***amplitude***
0	2.994 ± 0.004	standard deviation of the fit: 0.17	
0.03950	0.007 ± 0.005	0.078999	0.568 ± 0.005
0.040256	0.020 ± 0.005	0.080511	2.620 ± 0.005
0.038743	0.114 ± 0.005	0.082024	0.216 ± 0.005
0.041667	0.156 ± 0.005	0.083333	0.286 ± 0.005

Table 5.10.2: The same results after one frequency near 0.04 h^{-1} has been redefined.

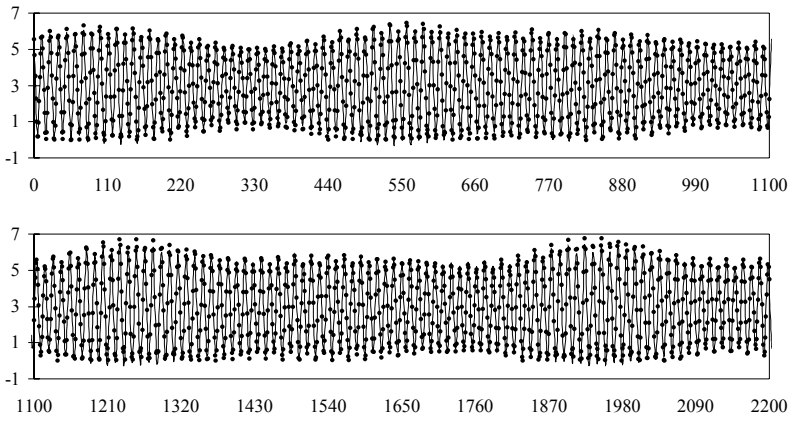


Fig. 5.10.7: The original data (solid points) and the fitted curve (drawn line) based on the average and eight sinusoidal components near 0.08 and 0.04 h^{-1}.

Exercise 5.10 (continued):

(30) Run SolverAid (again) and let it provide the matrix of linear correlation coefficients. Say that you have placed it in AA1:AQ17. Deposit in cell AS1 the instruction `=IF(ABS(AA1)>0.9,ABS(AA1)," ")`, and copy this instruction to the entire block AS1:BI17. Any linear correlation coefficient with an absolute value larger than 0.9 will show, whereas all other cells will remain empty because they will contain the 'empty' string between the two quotation marks in the IF statement. You can of course set the bar lower, at 0.8 or wherever, since in this particular case none of the 17 adjusted parameters has a very pronounced dependence on any other. In fact, the largest linear correlation coefficients (apart from the 1's on the main diagonal) are smaller than 0.2!

(31) To get an idea of how well you can represent the observed tidal data with just eight frequencies, plot the original and calculated curves in one graph, using different symbols and/or colors, as in Fig. 5.10.7. If you want to see how far you still would have to go, plot the residuals, as in Fig. 5.10.8. And if you want to see what is possible by harmonic analysis (using a longer database and many more harmonic terms), plot the data in the 'Sigma' column you may have set aside under point (11). This plot is shown in Fig. 5.10.9, and indicates that there is very little 'noise' on this signal. Such 'noise' may still be deterministic, when caused by, e.g., effects of earthquakes or storms, but could only be recognized as such in retrospect, by comparison with geological and meteorological records, and certainly would not be predictable.

It is clear that we can continue this process and, by including more and more frequencies, make the fit better and better. This is indeed how tidal tables are made. Remember that the standard deviation between the observed and the predicted heights listed in the NOS-NOAA table was a mere 6 mm, see under step (11) in the exercise. The corresponding value for our fit so far is 174 mm, more than 30 times larger. Still, you get the idea; in this case, with a large signal and apparently relatively little 'noise' from earthquakes, storms etc., the prediction can be extremely reliable, and the more so the longer the experimental record on which it is based.

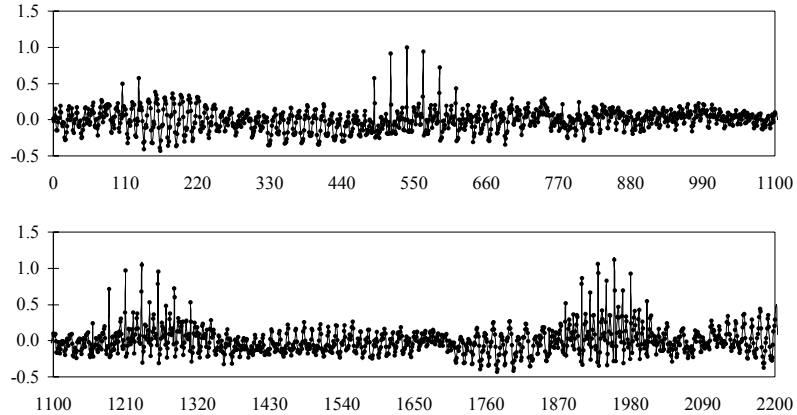


Fig. 5.10.8: The residuals after accounting for the average and eight sinusoidal components near 0.08 and 0.04 h^{-1}.

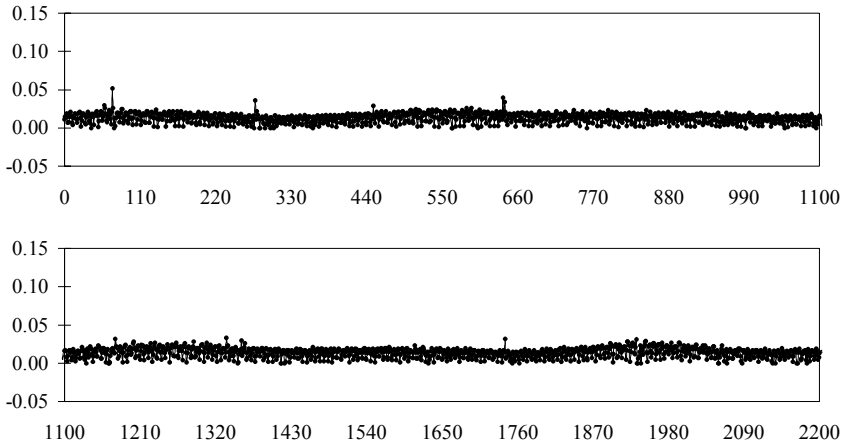


Fig. 5.10.9: The residuals in the NOS/NOAA harmonic analysis same data set. Note the ten times enlarged vertical scale.

6.7 Iterative deconvolution using Solver (AE3 pp. 289-291)

The van Cittert deconvolution method is general but quite sensitive to noise. An alternative approach was introduced by Grinvald & Steinberg in Anal. Biochem. 59 (1974) 583. It uses reverse engineering based on an assumed analytical (and therefore noise-free) function for the undistorted model signal $\boldsymbol{s}_m = f(a_i)$ in terms of one or more model parameters a_i. We convolve the model signal $\boldsymbol{s}_m$ based on guessed parameters a_i with the experimental transfer function $\boldsymbol{t}$ to obtain $\boldsymbol{r}_m = \boldsymbol{s}_m \otimes \boldsymbol{t}$. We then use Solver to adjust the a_i by minimizing the sum of squares of the residuals between $\boldsymbol{r}_m$ and the experimental (or, in our example, simulated) $\boldsymbol{r}_{exp}$. The requirement that $\boldsymbol{s}_m$ be describable as an explicit analytical function is somewhat restrictive but, when applicable, can greatly reduce noise.

Because macros do not self-update, Solver cannot respond automatically to the effect of parameter changes that involve macros. This means that, for a non-manual program, we can either rewrite Solver so that it can accommodate macros, or (much simpler) perform the convolution using a function rather than a macro. The latter approach is illustrated in exercise 6.7.1.

***Exercise 6.7.1**:*

(1) Our example will be modeled after exercise 6.2.2 and fig. 6.2.3, i.e., based on a single exponential decay. In cells A1:D1 deposit the column labels #, s, t, and r for the rank number # (which can represent time, wavelength, etc), original (undistorted) signal ***s***, filter or transfer function ***t***, and result ***r***.

(2) In A3 place the value –100, in A4 the instruction =A3+1, and copy this down to cell A303.

(3) In B103 insert the instruction `=$D$97*EXP(-$D$98*A103)` for the transfer function ***t***, and copy this down to row 303. In cell D97 enter an amplitude value (such as 1) and in cell D98 a value for a rate constant (e.g., 0.03), with accompanying labels in column C.

(4) For the transfer function ***t***, in cell C103 place the instruction `=EXP(-1*(LN(1+(A103-$D$100)/$D$101))^2)` and copy this down to row 303.

(5) Highlight A103:C303 and call the macro Convolve, which will write the convolution ***r*** in D103:D303.

(6) Deposit Gaussian ('normal') noise (with mean 0 and standard deviation 1) in N103:O303, and supply corresponding scale values, such as 0.01 in cell N100 and 0.02 in cell O100.

(7) In cell E103 place the instruction `=C103+$N$100*N103` to simulate a noisy transfer function ***texp***, in cell F103 use `=D103+$O$100*O103` for a noisy response signal ***rexp***, and copy both instructions down to row 303. In row 1 place appropriate labels. You now have a set of simulated, noisy data to try the deconvolution. In a real application these simulated values should of course to be replaced by experimental data, as anticipated by their labels.

(8) In G103 place a model function, such as `=$I$97*EXP(-$I$98*A103)`. Copy it down to row 303, and label it in G1 as s model. Place a guess value for the amplitude in I97, and an initial estimate for the rate constant in I98, with accompanying labels in column H. Do not place any numbers or text in G3:G102, but instead fill it with, e.g., yellow, to remind yourself to keep it clear.

(9) In cell H103 deposit the function `=Convol(G103:G202,$E$103: $E$202,100)` and copy it all the way to row 303. In cell H1 label the column as r model.

(10) Go to the VBA module and enter the following code for this function:

```
Function Convol(Array1, Array2, N)

Dim i As Integer
Dim Sum1 As Double, Sum2 As Double
Dim Array3 As Variant
ReDim Array3(1 To 2 * N)

Sum2 = 0
For i = 1 To N
  Sum2 = Sum2 + Array2(i)
Next i

For i = 1 To N
  Array3(i) = Array2(N + 1 - i)
Next i

Sum1 = 0
For i = 1 To N
  Sum1 = Sum1 + Array1(i - N + 1) * Array3(i)
Next i

Convol = Sum1 / Sum2

End Function
```

(11) In cell I100 deposit the function `=SUMXMY2(F103:F303,H103:H303)` and place a corresponding label such as SSR= in H100.

(12) Call Solver, and Set Target Cell to I100, Equal to Min, By Changing Cells I97:I98. Then engage SolverAid to find the corresponding uncertainties.

(13) Compare your results with those in fig. 6.7.1 which shows them before and after using Solver.

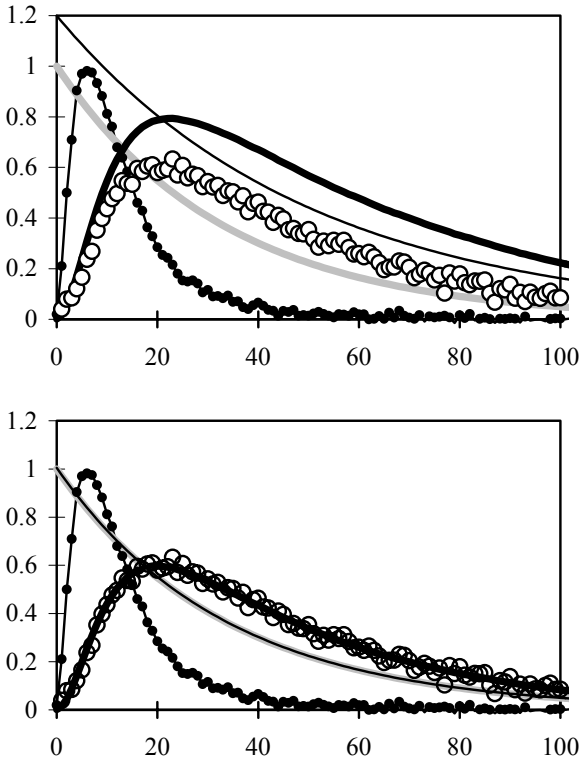


Fig. 6.7.1: The assumed signal $\boldsymbol{s}$ (gray band), the noisy transfer function $\boldsymbol{t}_{exp}$ (line with small solid points), the result $\boldsymbol{r}_{exp}$ obtained by convolving $\boldsymbol{s}$ with the (noise-free) transfer function and then adding noise (large open circles), the assumed model function $\boldsymbol{s}_m$ (line) and the resulting function $\boldsymbol{r}_m$ after convolving $\boldsymbol{s}_m$ with $\boldsymbol{t}_{exp}$ (heavy line). The top panel shows the situation just before calling Solver, the bottom panel that after Solver has been used.

This exercise demonstrates the principle of the method. We started with amplitude $a = 1$ and rate constant $k = 0.03$, used as initial guess values $a_m = 1.5$ and $k_m = 0.02$, and then found $a_m = 1.005 \pm 0.009$ and $k_m = 0.0300 \pm 0.0004$, with a standard deviation $s_y = 0.02_0$ of the fit in $\boldsymbol{r}$.

In practice, try to keep the convolving custom function as simple as possible, and especially avoid IF statements which tend to slow it down. We have used a barebones custom function Convol() to keep it simple, even though it is wasteful of spreadsheet real estate.

When the data involve a shift in time, try to adjust its initial estimate by trial and error in the graph, using a critical region (such as where the response takes off) where it makes a significant difference, because the least squares criterion is often not very sensitive to horizontal shifts, but looks at vertical differences, and averages over the entire curve.

6.8 Deconvolution by parameterization (AE3 pp. 291-295)

Many approaches to deconvolution are quite sensitive to noise, and may therefore require filtering, which (when not carefully compensated in the deconvolution routine) can again lead to distortion and loss of detail. The method described in section 6.7 avoids this problem by fitting the experimental data to a noise-free analytical function. Below we will carry this approach to its logical conclusion by using a noise-free convolution as well. Even though this approach has a somewhat limited applicability, it may still work when earlier-described methods fail.

To illustrate its basic idea (Am. J. Phys. 72 (2004) 644), we will here consider only a relatively simple case in which a measured ***r***esult ***r*** of the convolution, say, a spectrum or a chromatogram, can be described as a sum of Gaussian curves, $\boldsymbol{r} = \Sigma \boldsymbol{g}_r$, while the ***t***ransfer function ***t*** is given by a single Gaussian, $\boldsymbol{t} = \boldsymbol{g}_t$. We have already seen in chapter 4 how we can use Solver to fit complicated functions in terms of sums of Gaussians, and we will now apply this to deconvolution. Again we use bold lowercase symbols to indicate time-dependent functions rather than constants.

The approach we will take here substitutes deconvolving ***r*** with ***t*** by instead deconvolving $\Sigma \boldsymbol{g}_r$ with $\boldsymbol{g}_t$. Because $\boldsymbol{g}_r$ and $\boldsymbol{g}_t$ are fitted, noise-free analytical functions, this greatly reduces the effect of noise on the deconvolution. Noise only affects the result insofar as it limits the proper assignment of the Gaussians $\boldsymbol{g}_r$ and $\boldsymbol{g}_t$. The actual calculation is straightforward, the most critical part of the procedure being the initial fitting of Gaussians to the experimental functions ***r*** and ***t***.

We will first deconvolve two single N-point Gaussians, $\boldsymbol{g}_r$ and $\boldsymbol{g}_t$ that are both functions of a common parameter t which can represent elution time, wavelength, wavenumber, etc. We therefore start with the mathematical functions

$$\boldsymbol{g}_r = a_r \exp\{-\tfrac{1}{2}\,[(t-c_r)/b_r]^2\} \tag{6.8.1}$$

and

$$\boldsymbol{g}_t = a_t \exp\{-\tfrac{1}{2}\,[(t-c_t)/b_t]^2\} \tag{6.8.2}$$

which, upon analytical Fourier transformation, yield

$$\begin{aligned}\boldsymbol{G}_r &= [(2\pi)^{1/2} a_r b_r/N] \exp[-2\pi j f c_r] \exp[-2(\pi b_r f)^2] \\ &= [(2\pi)^{1/2} a_r b_r/N] \exp[-2(\pi b_r f)^2]\,[\cos(2\pi f c_r) - j \sin(2\pi f c_r)]\end{aligned} \tag{6.8.3}$$

and

$$\begin{aligned}\boldsymbol{G}_t &= [(2\pi)^{1/2} a_t b_t/N] \exp[-2\pi j f c_t] \exp[-2(\pi b_t f)^2] \\ &= [(2\pi)^{1/2} a_t b_t/N] \exp[-2(\pi b_t f)^2]\,[\cos(2\pi f c_t) - j \sin(2\pi f ct_r)]\end{aligned} \tag{6.8.4}$$

respectively, where we have used Euler's relation $e^{-jx} = \cos(x) - j \sin(x)$. From these we obtain by division

$$\begin{aligned}\boldsymbol{G}_s &= \boldsymbol{G}_r / \boldsymbol{G}_t \\ &= [a_r b_r / a_t b_t] \exp[-2\pi j f (c_r - c_t)] \exp[-2\pi^2 (b_r^2 - b_t^2) f^2]\end{aligned} \tag{6.8.5}$$

so that the original, undistorted ***s***ignal is given by

$$\boldsymbol{g}_s = \boldsymbol{g}_r \varnothing \boldsymbol{g}_t = \frac{a_r b_r N}{a_t b_t \sqrt{2\pi(b_r^2 - b_t^2)}} \exp\frac{-(t - c_r + c_t)^2}{2(b_r^2 - b_t^2)^2} = a_s \exp\{-\tfrac{1}{2}[(t - c_s)/b_s]^2\} \tag{6.8.6}$$

where

$$a_s = \frac{a_r b_r N}{a_t b_t \sqrt{2\pi(b_r^2 - b_t^2)}} = \frac{a_r b_r N}{a_t b_t b_s \sqrt{2\pi}} \tag{6.8.7}$$

$$b_s = (b_r^2 - b_t^2)^{1/2} \tag{6.8.8}$$

and

$$c_s = c_r - c_t \tag{6.8.9}$$

In other words, once we have characterized the two Gaussian functionss $\boldsymbol{g}_r$ and $\boldsymbol{g}_t$ in terms of the constants a_r, b_r, c_r and a_t, b_t, and c_t respectively, we can simply *calculate* the deconvoluted Gaussian $\boldsymbol{g}_s$.

Note that the constants b in eqs. (6.8.1) and (6.8.2) have the dimensions and functions of standard deviations. Equation (6.8.8) shows that their squares, the corresponding variances, are additive in convolution, $b_r^2 = b_t^2 + b_t^2$, and subtractive in deconvolution, $b_s^2 = b_r^2 - b_t^2$. With Gaussians, it is therefore easy to predict how much convolution will broaden peaks, and how much deconvolution can possibly sharpen them.

Typically the experimental response $\boldsymbol{r}$ to be corrected by deconvolution is calibrated, in which case we will want to maintain that calibration by deconvolving with a function that has been scaled to have unit average. In the case of a Gaussian $\boldsymbol{g}_t$ that implies that we should use

$$a_t = \frac{N}{b_t\sqrt{2\pi}} \tag{6.8.10}$$

so that (6.8.7) reduces to

$$a_s = a_r\, b_r\, (b_r^2 - b_t^2)^{-½} = a_r\, b_r / b_s \tag{6.8.11}$$

Moreover, the value of c_t is usually arbitrary. If we simply set it to zero, (6.8.9) becomes

$$c_s = c_r \tag{6.8.12}$$

When r must be expressed as a *sum* of Gaussians, $\boldsymbol{r} = \Sigma\boldsymbol{g}_r$, the same approach can be used, because then $\boldsymbol{R} = \Sigma\boldsymbol{G}_r$ and $\boldsymbol{T} = \boldsymbol{G}_t$ so that

$$\boldsymbol{S} = \frac{\boldsymbol{R}}{\boldsymbol{T}} = \frac{\sum_{i=1}^{r} \boldsymbol{G}_{ri}}{\boldsymbol{G}_t} = \sum_{i=1}^{r} \frac{\boldsymbol{G}_{ri}}{\boldsymbol{G}_t} = \sum_{i=1}^{r} \boldsymbol{S}_i \tag{6.8.13}$$

Exercise 6.8.1 illustrates this procedure for the deconvolution of the data shown in Fig. 6.2.4.

***Exercise 6.8.1**:*

(1) Retrieve the spreadsheet used in exercise 6.2.3, or repeat that exercise.

(2) In cells E20:H20 deposit column headings for ***t**exp*, ***r**exp*, *tmodel*, *rmodel*, and the recovered value *srecov*, and in N20 and O20 place headings for noise *n*.

(3) Generate Gaussian noise of zero mean and unit standard deviation in N22:O321.

(4) Place appropriate noise amplitudes for ***t*** and ***r*** in cells C19 and D19 respectively, in E22:E321 compute the function ***t*** with added noise with, e.g., the instruction `=C22+$C$19*N22` in cell E22. Similarly compute a noisy version of ***r*** in column F, using noise from column O. Plot these noisy versions of ***t*** and ***s***, as in Fig. 6.8.1*c*.

(5) In H2:H16 place the labels ar1=, br1=, cr1=, ar2=, br2=, cr2=, ar3=, br3=, cr3=, ar4=, br4=, cr4=, at=, bt=, and ct=. Alternatively you can copy them from F2:F16, then modify them.

(6) In cell G22 deposit `=$I$14*EXP(-0.5*((A22-$I$16)/$I$15)^2)`, and copy this instruction down to row 321. Enter this curve in the just-made plot.

(7) Place numerical values in I14:I16 so that the resulting curve approximately fits curve ***t**exp*. (You may first want to color the data in G2:G16 white, so that you will not be tempted to look at the data originally taken for the simulation of ***s***. When you are done fitting the data, change their color back to black or whatever.)

(8) In cell F19 calculate *SSR* for ***t*** as `=SUMXMY2(E22:E321,G22:G321)`.

(9) Call Solver, and let it minimize *SSR* in F19 by adjusting the guessed parameter values in I14:I16.

(10) Likewise, in cell H22 place the instruction `=$I$2*EXP(-0.5*((A22-$I$4)/$I$3)^2)+ ... +$I$11*EXP(-0.5*((A22-$I$13)/$I$12)^2)`, copy this down to row 321, and enter this curve in the graph.

(11) Compute SSR for ***r*** as `=SUMXMY2(F22:F321,H22:H321)`.

(12) Use the curve made under point (9) to guess numerical values for *ar*1 through *cr*4 in I2:I13 so that the resulting curve approximately fits the data ***r**exp*.

(13) Call Solver to refine these values by minimizing SSR in cell H19. Do this adjustment groupwise: first let Solver adjust I2:I3, then call it again to adjust I5:I7, then I2:I7, then I8:I10, I11:I13, I8:I13, and finally I2:I13. The graph might now resemble Fig. 6.8.1*d*.

(14) In K2:K13 copy the labels as1=, bs1=, ... cs4= from F2:F13.

(15) In L4 calculate *cs*1 = *cr*1 – *ct*, i.e., as `=H3-$H$15`.

(16) In L3 compute $bs1 = \sqrt{(b_{r1}^2 - b_t^2)}$, or `=SQRT(H2^2-$H$14^2)`.

(17) In L2 calculate $as1 = ar1\ br1 / bs1$, with `=H1*H2/J2`.

(18) Copy the block L2:L4 to L5, L8, and L11.

(19) In cell H22 compute the reconstituted signal ***s****recov* with the instruction `=$L$2*EXP(-0.5*((A22-$L$4)/$L$3)^2)+ ... +$L$11*EXP(-0.5*((A22-$L$13)/$L$12)^2)`, and copy this down to row 321.

(20) Plot this curve, and compare it with Fig. 6.8.1*f*.

(21) In this graph also display the function ***s*** used as the starting point of this simulation from B22:B321, a repeat from Fig. 6.8.1*a*.

This method can indeed reconstitute most features of the original curve, at least in a favorable case such as shown Fig. 6.8.1*f*, where all peaks are Gaussian, and can be identified as such in ***r*** despite the noise.

Exercise 6.8.1 (continued):

(22) Call SolverAid, enter the Solver-determined parameters in I2:I13, the location of SSR for *r* (H19), and the column (H22:H321) in which *r* was calculated. Let SolverAid display the covariance matrix in N2:Y13. It will also deposit the standard deviations of the individual parameters in J2:J13.

(23) Once more call SolverAid, this time to find the uncertainty estimates for *t*. Therefore enter the location (I14:I16) of the Solver-determined parameters, the location (F19) of SSR for *t*, and that (G22:G321) of the column where *t* was computed. Place the covariance matrix in Z14:AB16, so that it shares its main diagonal with that in N2:Y13.

(24) Call Propagation, and give it I2:I16 as input parameters, N2:AB16 as covariance matrix, and L2 as function. It will then place the corresponding standard deviation in M2. Repeat this for the other 11 results in column L. Sorry, Propagation handles only one parameter at a time.

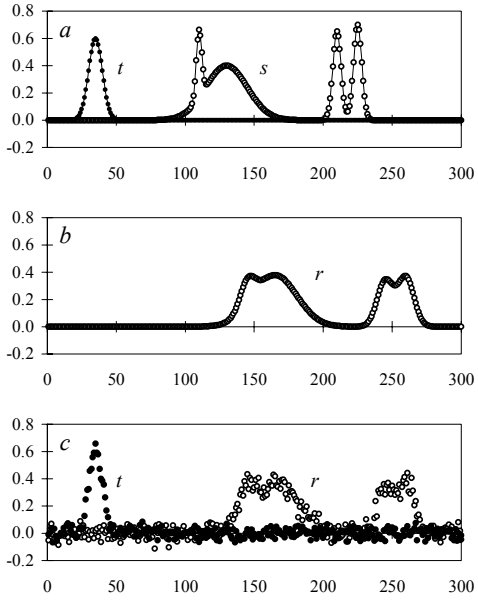


Fig. 6.8.1: Top and middle: replicas from Fig. 6.2.4, showing in panel *a* the original simulated spectrum ***s*** and the distorting transfer function ***t***, and in panel *b* its convolution leading to the result ***r***. Bottom panel, *c*: the same as in panel *b* after adding random noise.

The data in Table 6.8.1 illustrate the results obtained, and allow one to consider them quantitatively, because the method can provide its own uncertainty estimates. These indicate the level of agreement between the parameters used to simulate ***s*** and those recovered after convolution, noise addition, and deconvolution. For all 12 coefficients of ***s****recov* the agreement between the recovered value of ***s*** and that used in the simulation of Fig. 6.8.1*a* is within two standard deviations. Thus the original signal is retrieved within the uncertainty limits generated by the custom macros SolverAid and Propagation. The standard deviations depend, of course, on the amount of noise added.

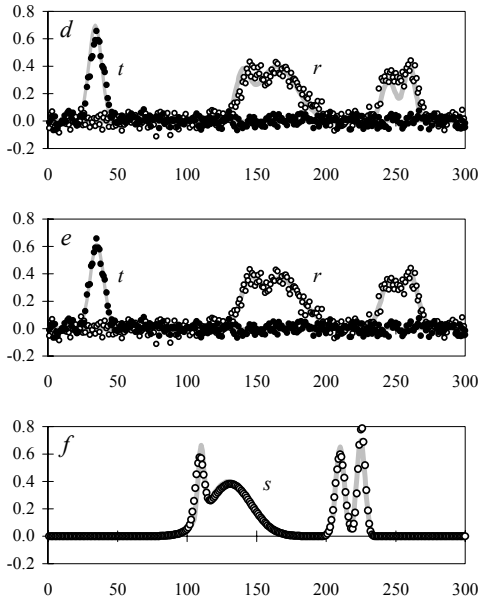


Fig. 6.8.1 (continued): The manually adjusted curves (with the parameters selected 'by eye') through the noisy data (gray curves in panel *d*), the same after Solver has refined the parameter estimates (panel *e*), and the resulting deconvoluted signal *s* (open circles in panel *f*). Panel *f* also displays, as a thick gray line, the original curve of the simulated function *s* repeated from panel *a*.

s_{taken}	$r_{guessed}$	r_{found}	$r_{st.dev.}$	s_{found}	$s_{st.dev.}$
*as*1=0.5	*ar*1=0.3	0.231	0.030	*as*1=0.44	0.14
*bs*1=2	*br*1=5	5.69	0.75	*bs*1=3.0	1.5
*cs*1=110	*cr*1=140	144.30	0.49	*cs*1=109.11	0.51
*as*2=0.4	*ar*2=0.4	0.364	0.010	*as*2=0.382	0.011
*bs*2=15	*br*2=14	15.95	0.96	*bs*2=15.2	1.0
*cs*2=130	*cr*2=165	166.3	1.2	*cs*2=131.1	1.2
*as*3=0.65	*ar*3=0.3	0.345	0.017	*as*3=0.60	0.13
*bs*3=3	*br*3=5	5.95	0.60	*bs*3=3.4	1.0
*cs*3=210	*cr*3=245	245.36	0.71	*cs*3=210.19	0.72
*as*4=0.7	*ar*4=0.4	0.377	0.019	*as*4=0.80	0.26
*bs*4=3	*br*4=4	5.50	0.50	*bs*4=2.6	1.1
*cs*4=225	*cr*4=260	260.75	0.61	*cs*4=225.57	0.62
t_{taken}	$t_{guessed}$	t_{found}	$t_{st.dev.}$		
at= 0.6	0.7	0.615	0.013		
bt= 5	4.5	4.85	0.12		
ct= 35	34	35.18	0.12		

Table 6.8.1: Some numerical results from exercise 6.8.1. The column labeled $\boldsymbol{s}_{taken}$ lists the values used for simulating the data in Fig. 6.8.1*a*. The columns $\boldsymbol{r}_{guessed}$ and $\boldsymbol{t}_{guessed}$ contain the initial guess values shown in Fig. 6.8.1*c*., and the next two columns the values obtained by Solver for the parameters, and those obtained by SolverAid for their standard deviations. Finally, the column $\boldsymbol{s}_{found}$ displays the deconvolved signal *s* as computed from $\boldsymbol{r}_{found}$ and $\boldsymbol{t}_{found}$, and the column $\boldsymbol{s}_{st.dev.}$ the corresponding uncertainty estimates. The added noise was Gaussian with zero mean and standard deviations of 0.04 and 0.03 for ***r*** and ***t*** respectively.

This method works, and does not lead to oscillatory instabilities. Its applicability depends, of course, on how well one can represent both the measured result ***R*** and the transfer function ***T*** in terms of model expressions with relatively simple Fourier transforms, so that the inverse Fourier transform of their quotient ***R***/***T*** can be expressed in analytical form. All baseline-separated peaks and peak aggregates can be treated individually. As a side benefit this method can furnish estimates of the standard deviations of the deconvolved signal ***s***.

6.9 Time-frequency analysis (AE3 pp. 295-298)

Fourier transformation presumes a steady state, because it considers the data set as one unit of an infinitely repeating sequence of identical units. Yet, there are many phenomena with frequency content that are not stationary, such as speech and music. In fact, music is an interesting example because its common form of notation, musical script, is really a graph of frequency (the pitch of the notes to be played) as a function of time, complete with grid lines for both time (vertical lines identifying the various measures) and frequency (the horizontal lines of the staff). It even has explicit time notation (for the lengths of notes and rests) and the corresponding scale factors (tempo indicators and/or metronome settings). Musical script is, of course, a set of instructions for the performer. We here address how, other than by ear, can we analyze and visualize sound (or any equivalent, non-auditory signal) as a function of time *and* frequency.

Time-frequency or Gabor transformation (D. Gabor, *J. Inst. Elect. Engin*. 93 (1946) 429) is an analysis in which a sliding time window moves stepwise along the data, dropping one or more points on one side, and gaining the same number of data points on the other. After each step, a Fourier transformation is applied. It is an inherently imprecise approach because the product of the resolutions in time and frequency is subject to the uncertainty relationship discussed in section 5.5. (That uncertainty is intrinsic to the problem of plotting frequency vs. time, and independent of the use of Fourier transformation or any other specific analysis method.) The uncertainty can be minimized with a Gaussian window function, which we therefore use. As a practical matter, we will exploit the fast Fourier transformation algorithm, and therefore require that the data be equidistant in time, as they usually are with sampled time-dependent signals.

The Gabor transform macro uses a Gaussian window function of N contiguous data points (with $N = 2^n$ where n is a positive integer) on a data set containing M data, where $M > N$. It starts with the first N data points in the set, multiplies these by the window function, and then performs a Fourier transformation on that product. It then moves the window function over by one point, and repeats this process $M–N+1$ times until it has reached the end of the data set. The results are returned to the spreadsheet as a function of time and frequency, and can then be plotted as, e.g., a 3-D plot of the absolute magnitude of the sound as a function of time and frequency. Such a plot is called a *sonogram*.

When the data set is so large that it would result in more than 250 columns (and therefore might exceed the 256-column width of the pre-2007 Excel spreadsheet), the macro will automatically move the window function each time by several data points, and the user can further restrict the size of the output file. If this presents a problem, modify the macro so that it stores rather than displays the data, or use rows instead of columns, since the spreadsheet contains many more rows than columns. In Excel 2007 and beyond, remove this condition from the code.

Exercise 6.9.1:

(1) Start a new spreadsheet. Leave the top 10 rows for graphs, and the next 4 rows for constants and column headings.

(2) Starting in cell A15 of column A deposit time t in constant increments Δt, such as $t = 0$ (1) 1000.

(3) In column B deposit as a trial function, such as with the instruction `=(SIN($B$11*A15))/(EXP(-0.1*(A15-200))+EXP(0.003*(A15-200)))` which has as frequency the value specified in B11 divided by 2π. Its amplitude, given by $1/\{\exp[-0.1(t-200)]+\exp[0.003(t-200)]\}$, quickly rises just before $t = 200$, and then slowly decays, somewhat like a note played on a piano.

(4) Plot the trial function; it should look like Fig. 6.9.1.

(5) Call the macro Gabor, and in its successive input boxes enter the time increments (here: 1), the location of the input data (here: B15:B1015), and the (optional) integer to restrict the number of samples to be analyzed (which you can leave at its default value of 5).

(6) The macro will now generate a data array, listing the frequency in its first column, and the rank number of the first data point used in each window in its top row. Inclusion of these parameters makes it easy to generate a labeled 3-D plot as well as a surface map.

(7) Make a 3-D plot of the result, and also a surface map with Mapper.

(8) Obviously, for such a simple trial function, you need not go through all this trouble. You may notice that the 3-D map for a sizable array is slow to rotate, and that its presence slows down the operation of the spreadsheet whenever it must be redrawn on the screen.

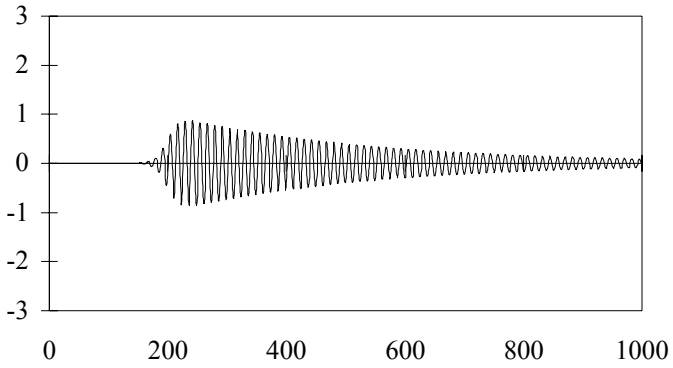


Fig. 6.9.1: The test function used, with the value 0.5 in cell B11.

(9) Now add some harmonics, as in a chord. Extend the instruction in cells B15:B1015 to include three additional terms, identical to the first one except that their frequencies are specified by cells C11, D11, and E11 respectively.

(10) In C11 deposit the instruction `=B11*2^(3/12)`, in D11 the instruction `=C11*2^(4/12)`, and in E11 the instruction `=D11*2^(5/12)`, for the harmonics of a major chord, such as C-E-G-C. On the Western, 'well-tempered' musical scale, all half-notes differ in frequency by a factor of 2^(1/12).

(11) The resulting signal is not so transparent any more, see Fig. 6.9.2.

(12) Now repeat the process of Gabor transformation and mapping. The map should now look similar to that of Fig. 6.9.3b.

(13) Now the surface map reveals very clearly the four different notes, starting at the same time but at different frequencies. The notes appear to start at about $t = 100$, whereas they really start only around $t = 200$. This time distortion results from the use of a Gaussian filter in the Gabor transformation macro.

(14) Modify the instruction in cells B15:B1015 to correspond with a broken chord, in which the various notes start one after the other, say, at $t = 200$, 300, 400, and 500 respectively. Fig. 6.9.4 illustrates such a signal, and Fig. 6.9.5 its Gabor transform.

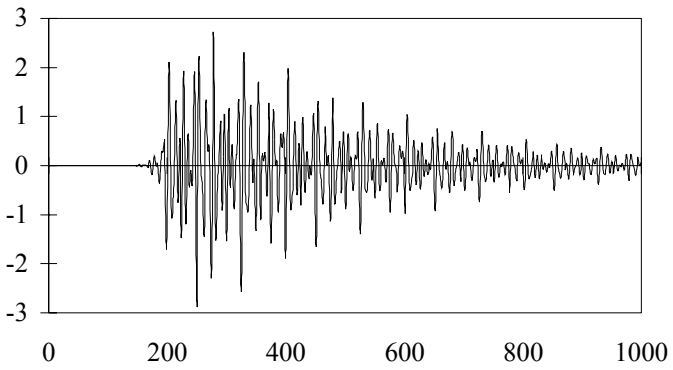


Fig. 6.9.2: The extended test function used, again with the value 0.5 in cell B11.

With such complicated signals we can readily appreciate the advantages of the Gabor transform and its representation as a 3-D graph or surface map. The different signal frequencies, and their time courses, are clearly displayed. This will become even more obvious when we consider more realistic musical signals, which may include short (staccato) and drawn-out (legato) notes, will have harmonics (characteristic for the musical instrument used), and may also exhibit gradually varying frequencies, as in a glissando.

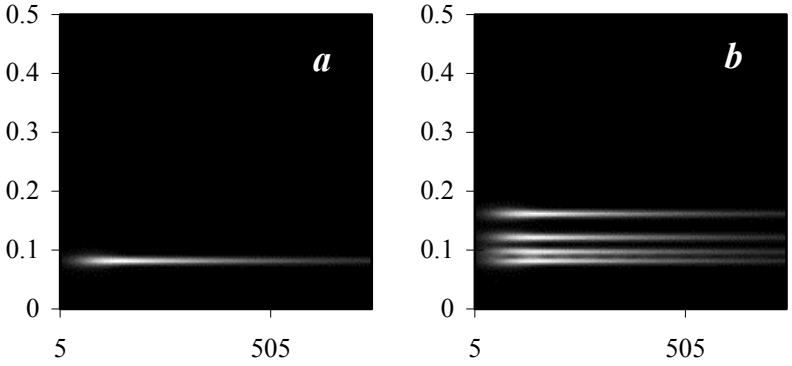


Fig. 6.9.3. Sonograms (i.e., surface maps of the Gabor transforms) of the functions shown in (*a*) Fig. 6.9.1 and (*b*) Fig. 6.9.2.

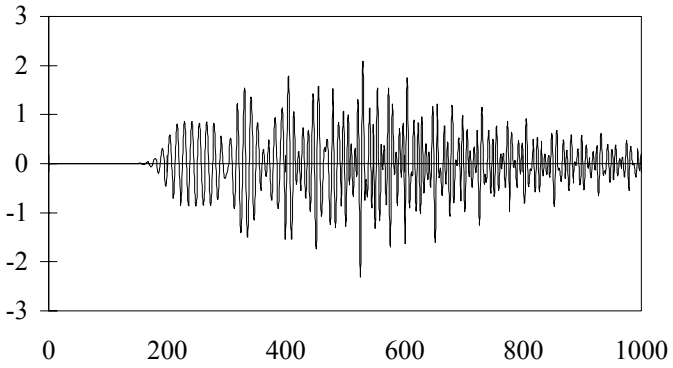


Fig. 6.9.4: The extended test function for a broken major chord.

The sonogram exhibits the three basic attributes of sound: time, frequency (pitch, tone-height), and amplitude (intensity, loudness, volume). In some respects it mimics musical notation, in that it uses the horizontal axis for time (indicating the duration of the various notes), while the vertical axis shows their pitch. In addition it displays their harmonics. In musical notation, amplitude (loudness) must be indicated separately, whereas the sonogram displays it as color or grayscale. We will analyze a real signal in the next section.

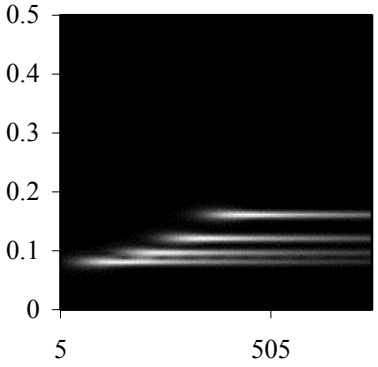


Fig. 6.9.5. The sonogram of the broken chord shown in Fig. 6.9.4.

6.10 The echolocation pulse of a bat (AE3 pp. 298-299)

Bats orient themselves at night by sending out short sound bursts of varying amplitude and frequency, and by analyzing the reflected sound. The echolocation pulses are short, so that they do not overlap with the reflected signals. A digitized echolocation pulse of a large brown bat (*Eptesicus fuscus*) can be downloaded from http://www-dsp.rice.edu/ software/TFA/RGK/BAT/bat.html, and can also be obtained by e-mail from, e.g., richb@rice.edu. The recorded pulse, courtesy of Curtis Condon, Ken White, and Al Feng of the Beckman Center at the University of Illinois, contains 400 equidistant data points taken at 7 μs intervals, and therefore covers a total time of less than 3 ms duration.

Exercise 6.10.1:

(1) Start a new spreadsheet, leaving the top rows for graphs. Import the bat data, and plot them.

(2) Apply the Gabor transform, and then map the results. The gray-scale of Fig. 6.10.1 illustrates what you might obtain. More subtle details can be discerned by using a full color palette, as with Mapper*n* for $n > 0$.

(3) The back cover of this book contains the original graph as well as the resulting sonogram, in color. The latter is also used on the front cover, with the time axis reversed to run from right to left.

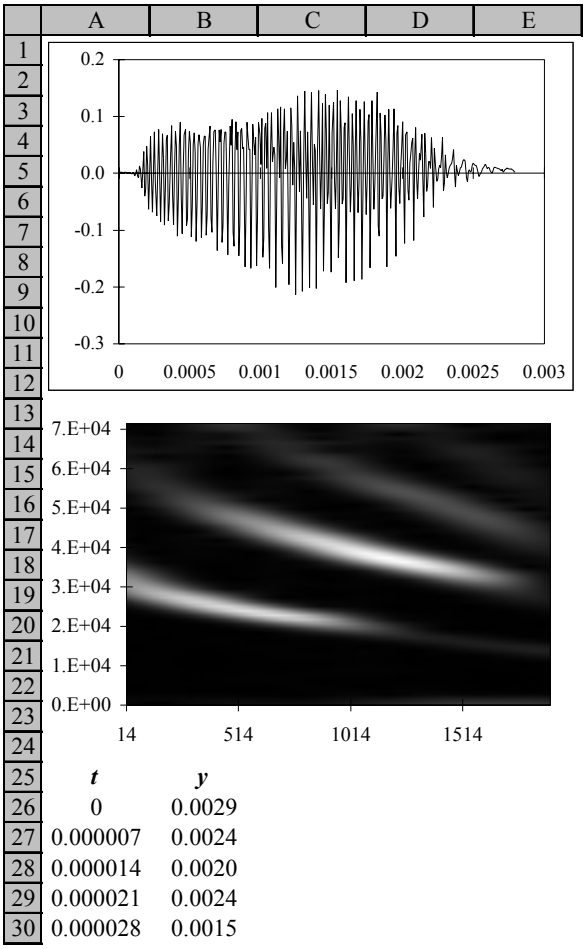


Fig. 6.10.1: The top of a spreadsheet for Gabor analysis of a bat chirp. Top graph: the echolocation signal as a function of time, in s. Bottom graph: the corresponding sonogram: frequency (in Hz) vs. time (in start-of-sequence number).

The signal in Fig. 6.10.1 starts out at about 30 kHz, descends to about 20 kHz, and after about 50 ms is joined by a second descending signal at its double frequency. The signal also contains weak higher harmonics at the triple and quadruple frequencies. The Gabor transform and its visualization make this much more transparent than the original data set.

The dominant sensory input of humans is visual. If we are trying to listen in on the meaning of whale songs, or on the communication between cawing crows, visualizing their sounds as Gabor plots may well help us to decode them.

7.2 The semi-implicit Euler method (for simulating the chemical reaction sequence A → B → C; AE3 pp. 306-308)

The procedure illustrated above uses the *initial* concentrations to compute the behavior during the interval Δt. We do not know how those concentrations are going to change, but instead of assuming a to remain constant we will now approximate the change in a as linear over a sufficiently small interval Δt. This leads to the *semi-implicit Euler method*, in which we replace, say, the concentration an during the interval from tn to $tn+1$ by its average value $(an+an+1)/2 = an + (an+1-an)/2 = an + \Delta a/2$. Upon replacing the concentrations a, b, and c in (7.1.6) through (7.1.8) by their initial values plus half their anticipated changes we have

$$\frac{\Delta a}{\Delta t} \approx -k_1 (a + \Delta a/2) \tag{7.2.1}$$

$$\frac{\Delta b}{\Delta t} \approx k_1 (a + \Delta a/2) - k_2 (b + \Delta b/2) \tag{7.2.2}$$

$$\frac{\Delta c}{\Delta t} \approx k_2 (b + \Delta b/2) \tag{7.2.3}$$

from which we obtain

$$\Delta a \approx \frac{-\,a\, k_1 \Delta t}{1 + k_1 \Delta t/2} \tag{7.2.4}$$

$$a_{t+\Delta t} = a_t + \Delta a \approx \frac{1 - k_1 \Delta t/2}{1 + k_1 \Delta t/2} a_t \tag{7.2.5}$$

$$\Delta b \approx \frac{a\, k_1 \Delta t}{(1 + k_1 \Delta t/2)\,(1 + k_2 \Delta t/2)} - \frac{b\, k_2 \Delta t}{(1 + k_2 \Delta t/2)} \tag{7.2.6}$$

$$b_{t+\Delta t} = b_t + \Delta b \approx \frac{a_t\, k_1 \Delta t}{(1 + k_1 \Delta t/2)\,(1 + k_2 \Delta t/2)} + \frac{(1 - k_2 \Delta t/2)\, b_t}{(1 + k_2 \Delta t/2)} \tag{7.2.7}$$

We need not compute c, because it follows directly from the mass balance (7.1.17). Still, for the sake of completeness, we list it here as

$$\Delta c \approx (b + \Delta b/2)\, k_2 \Delta t \tag{7.2.8}$$

$$c_{t+\Delta t} \approx c_t + (b_t + b_{t+\Delta t})\, k_2 \Delta t/2 \tag{7.2.9}$$

We see that equations such as (7.2.1), (7.2.2), and (7.2.3) cannot be used directly, but must first be solved for the concentration changes Δa, Δb, and Δc. This accounts for the *implicit* in the semi-implicit Euler method. It is only *semi*-implicit because $(an+an+1)/2 = an + \Delta a/2$ combines half of the known term an with half of the next unknown one, $an+1$. For linear systems, this is the best one can do and still retain an absolutely stable solution.

Exercise 7.2.1:

(1) Copy the spreadsheet of exercise 7.1 to a new page of the same workbook. In columns J and K of this copy, change the instructions to incorporate eq. (7.2.5) instead of (7.1.10), and (7.2.7) instead of (7.1.12). In column L you can either use (7.1.17) or (7.2.9).

(2) Click on the curves in your equivalent of Fig. 7.1.3 to the new page, then redirect their definitions in the formula box to the current worksheet. In doing so, be careful not to alter the general format of the argument: (‚*sheetname*!*Xn*:*Xm*, *sheetname*!*Yn*:*Ym*,*p*), where *Xn*:*Xm* and *Yn*:*Ym* specify the ranges, and *p* defines the relative precedence of the curves, with the highest number being shown on top of the other curves. All you need to change in the argument is the *sheetname*, which you find on the tab at the bottom of the spreadsheet. Incidentally, the equivalent of Fig. 7.1.2 is immaterial, because any differences are too small to be visible on this scale.

(3) The improvement in Fig. 7.2.1 over the results shown in Fig. 7.1.4 is immediate and dramatic: for the same step size ($\Delta t = 0.1$) the errors are now more than an order of magnitude smaller.

(4) Repeat the analysis of the simulated data set with added Gaussian noise. For the same noisy data as used in Fig. 7.1.3 we now find $a0 = 0.9921 \pm 0.0096$, $k1 = 0.9954 \pm 0.0070$, $k2 = 0.496 \pm 0.014$, a much better over-all fit to the assumed values of $a0 = 1$, $k1 = 1$, and $k2 = 0.5$ than obtained earlier.

(5) As suggested by comparing Figs. 7.1.3 and 7.2.1, the improvement is more obvious for data that contain less noise. For example, for $\Delta t = 0.1$ and the same Gaussian noise but now with s$n = 0.01$ the results of the explicit and semi-implicit Euler methods would be $a0 = 0.9983 \pm 0.0038$, $k1 = 0.944 \pm 0.058$, $k2 = 0.503 \pm 0.023$ and $a0 = 0.9973 \pm 0.0032$, $k1 = 0.9985 \pm 0.0023$, $k2 = 0.49$

Fig. 6.10.1: The top of a spreadsheet for Gabor analysis of a bat chirp. Top graph: the echolocation signal as a function of time, in s. Bottom graph: the corresponding sonogram: frequency (in Hz) vs. time (in start-of-sequence number).

87 ± 0.0047 respectively. Here the explicit method clearly shows its bias.

You may wonder what constitutes a *fully implicit* Euler method. Instead of the average value of the slope (as in the semi-implicit method), or its initial value (as in the explicit method), it uses the final value of the slope to evaluate the new value of $F(t)$. Since that is just as asymmetrical as using the initial value, the implicit Euler method has an inaccuracy proportional to Δt, i.e., comparable to that of the explicit Euler method, whereas the semi-implicit method has an inaccuracy $\propto (\Delta t)^2$.

In our example, the implicit Euler method would read

$$\frac{\Delta a}{\Delta t} \approx -k_1(a + \Delta a) \tag{7.2.10}$$

$$\frac{\Delta b}{\Delta t} \approx k_1(a + \Delta a) - k_2(b + \Delta b) \tag{7.2.11}$$

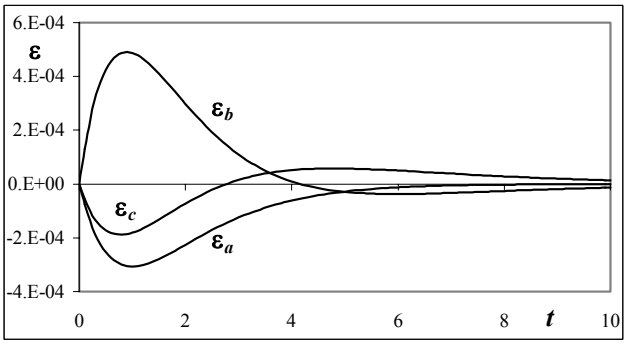


Fig. 7.2.1: The differences between the numerically integrated and exact solutions for the semi-implicit Euler method with $\Delta t = 0.1$.

from which we would obtain

$$\Delta a \approx \frac{-a\, k_1 \Delta t}{1 + k_1 \Delta t} \tag{7.2.12}$$

$$a_{t+\Delta t} = a_t + \Delta a \approx \frac{a_t}{1 + k_1 \Delta t} \tag{7.2.14}$$

$$\Delta b \approx \frac{a\, k_1 \Delta t}{(1 + k_1 \Delta t)\,(1 + k_2 \Delta t)} - \frac{b\, k_2 \Delta t}{(1 + k_2 \Delta t)} \tag{7.2.15}$$

$$b_{t+\Delta t} = b_t + \Delta b \;\approx \frac{a_t\, k_1 \Delta t}{(1 + k_1 \Delta t)\,(1 + k_2 \Delta t)} + \frac{b_t}{(1 + k_2 \Delta t)} = \frac{a_{t+\Delta t}\, k_1 \Delta t + b_t}{(1 + k_2 \Delta t)} \tag{7.2.16}$$

Upon comparing, e.g., (7.2.1) with (7.1.6) and (7.2.10), we verify that the semi-implicit method is indeed the average of the explicit and implicit Euler methods. It combines the absolute stability of the im-

plicit method with an accuracy that is higher than that of either the explicit or implicit Euler method, and is therefore often the method of choice for solving simple problems involving ordinary differential equations.

7.3 Using custom functions (AE3 pp. 308-311)

As indicated in the previous paragraph, the successful fitting of simulated, noisy data can be somewhat misleading, since a generous amount of noise may mask many inadequacies of the model. For fitting data with a high signal-to-noise ratio we may therefore need to improve the algorithm, as we will do in section 7.6. However, we can go a long way with the Euler methods by using the spreadsheet more intelligently.

Equations (7.1.6) through (7.1.8) clearly show that the simulation is based on replacing the differential quotients dy/dt by difference quotients $\Delta y/\Delta t$, a substitution that should become increasingly accurate as Δt becomes smaller. You can readily verify that the simulation errors shown in Figs. 7.1.3 and 7.2 indeed stem from the step size Δt. Upon reducing Δt by a factor of ten, the concentration differences Δc in Fig. 7.1.3 indeed also become smaller by an order of magnitude, and those in Fig. 7.2 by two orders of magnitude.

However, in order to cover the same total time (in the above example: from $t = 0$ to $t = 10$), we would have to lengthen the columns ten-fold, to 1000 rows. Further reductions in Δt would make the columns even longer. This will quickly lead to impracticably long columns. Moreover, it may be undesirable to lengthen the columns, e.g., because we may only have experimental data at given intervals Δt. Below we will indicate how we can improve the accuracy of our simulation *without* increasing the column length.

Exercise 7.3.1:

(1) Return to the spreadsheet of exercise 7.2.1, and set the values in K1:K3 back to the corresponding values in B1:B3.

(2) For the concentration a an elegant solution exists that does not require an increased column length. We saw in (7.2.5) that $at+\Delta t \approx at\,(1-k1\Delta t/2)/(1+k1\Delta t/2)$. Upon applying this n times with an n times smaller interval Δt we find $at+n\Delta t \approx at\,\{[1-k1\Delta t/(2n)]/[1-k1\Delta t/(2n)]\}^n$, so that we can replace the instruction in cell J9 for a by, say, `=J8*((1-$K$1*$D$3/20)/(1+$K$1*$D$3/ 20))^10` for $n = 10$. Copy this down through row 108. This will improve the precision of the simulated a-values another two orders of magnitude *without* lengthening the columns. Try it. Then change the value of n in these instructions from 10 to, say, 1000, and observe its effect.

Unfortunately, this trick does not work for the other concentrations, because (7.2.7) and (7.2.9) do not have such a simple recursivity. For those more general cases we will need to use some spreadsheet magic. Excel allows us to incorporate so-called *user-defined* or *custom* functions. These have much in common with small macros (to be discussed at length in chapter 8), except that they apply only to a *numerical value* in a *single* spreadsheet cell. On the other hand, custom functions update automatically, which in the present context is a significant advantage. Below we will use custom functions to compute the concentrations a, b, and c to higher accuracies by reducing the step size while keeping constant the number of spreadsheet cells used in the simulation. If writing computer code is new to you, you may first want to read sections 8.1 through 8.4 of the next chapter before continuing here.

Exercise 7.3.1 (continued):

(3) Return to the spreadsheet, and press Alt∪F11 (on the Mac: Opt∪F11). You will see a Microsoft Visual Basic screen appear, with its own menu bar. On *that* bar, select Insert ⇨ Module if the display does not show a white writing area to the right of the Project column; otherwise, if a page already exists, just move to the end of any text on it. Then enter (type, or copy from SampleMacros) the following instructions:

```
Function siEulerA(k1, oldT1, oldT2, n, oldA) As Double

'semi-implicit Euler method for A

Dim i As Integer
Dim A As Double, f As Double, step As Double

n = CInt(n)
A = oldA
step = (oldT2 - oldT1) / n
f = (1 - k1 * step / 2) / (1 + k1 * step / 2)

For i = 1 To n
  A = A * f
Next i

siEulerA = A

End Function
```

(4) A short explanation is in order. The top line specifies the name by which we can call this function, the parameters it will use (in exactly the same order as used here in the function *argument*, i.e., within the brackets following the function name), and (optionally) its precision, starting with an apostrophe. The next line contains a *comment* that will be ignored by the spreadsheet but reminds the user of the purpose of the function; the last line identifies its end.

(5) The next two lines of code (i.e., not counting empty lines inserted for better readability) define the types of constants used in the function; do *not* specify the dimensions of parameters (such as `k1`, `oldT1`, etc.) that are imported through the function argument. In general these lines are optional though very useful; they are mandatory if you use Option Explicit, an option that, when used, is listed at the very top of your module, and then applies to all procedures in that module.

(6) The fifth line (optional as well) makes sure that the method will work even if a non-integer value for *n* is used by mistake, by converting it to an integer *n* with the instruction `CInt` (for *c*onvert to *int*eger). This line will be executed from right to left, i.e., the computer takes the value of *n*, converts it to an integer (if it isn't already one), and then assigns that value to the variable to the left of the equal sign. We insert this line here merely to illustrate how you can make a function somewhat less error-prone by anticipating possible mistakes. This does not imply that the function is now immune to entry errors: using zero for *n* would certainly trip up the function when it tries to divide by 0 in the next line of code, and using a negative number, or a letter, would also give problems.

(7) Line 6 sets the concentration parameter `A` equal to the value of `oldA` imported through the function argument. The calculation starts in earnest on line 8 by defining the new step size, `step`. By letting oldT1 and oldT2 refer to relative addresses of cells containing *t* in successive rows of the spreadsheet, the time intervals in the spreadsheet need not be equidistant. Alternatively we can make the step size constant throughout the calculation by referring to absolute addresses for oldT1 and oldT2 respectively.

(8) Lines 9 through 11 contain the action part of the function, by *n* times repeating the computation of `A` for a time interval `step` that is *n* times smaller than the data spacing `oldT2 – oldT1`.

(9) Again, the equal sign here functions as an *assignment*. In other words, the line `A = A * f` should be read as if it were written as `A ⇐ A * f`, i.e., as "replace `A` by `A * f`."

(10) We calculate the value of `f` separately on line 8, rather than use, e.g., `A= A*(1-k1*step/2)/(1+k1*step/2)` directly in line 10, because line 8 is executed only once, whereas in line 10 the same calculation would be repeated *n* times. It is in such loops that we should be most careful to avoid busywork, because it can noticeably slow down the computation. Note that the line specifying `f` must follow the definition of `step`, because it uses its value which, otherwise, would not be defined.

(11) Finally, the output of the function is defined in its penultimate line. Incidentally, you will have noticed that a number of words you have entered (Function, As Double, Dim, As Integer, etc.) are displayed in blue after you have entered the line on which they appear. These are terms the VB Editor recognizes as instruction keywords, and seeing them in color therefore assures you that your instructions are being read.

(12) Now enter the corresponding instructions for siEulerB, or copy the instructions for siEulerA and then correct and amend that copy. For your convenience, the changes between the two sets of instructions are shown below in boldface.

```
Function siEuler B(k1, k2, oldT1, oldT2, n, oldA, oldB) As Double

'semi-implicit Euler method for B

Dim A As Double, B As Double, step As Double
Dim f As Double, fA As Double, fB As Double
Dim i As Integer

n = CInt(n)
A = oldA
B = oldB
step = (oldT2 - oldT1) / n
f = (1 - k1 * step / 2) / (1 + k1 * step / 2)
fA = k1 * step / ((1 + k1 * step / 2) * (1 + k2 * step / 2))
fB = (1 - k2 * step / 2) / (1 + k2 * step / 2)

For i = 1 To n
  B = A * fA + B * fB
  A = A * f
Next i

siEuler B = B

End Function
```

(13) Note the use of a space followed by an underscore at the end of line 1, in order to indicate a *line continuation*. This allows us to break up a long instruction so that it will be visible on the monitor screen (or the printed page) while being interpreted by the computer as a single line. There can be no text on that line beyond the continuation sign.

(14) In order to use the functions you have just entered, exit the editor with Alt∪F11 (Mac: Opt∪F11), which toggles you back to the spreadsheet. On the spreadsheet, in cell F2 place the label `n=`, and in G2 its value, which should be a positive integer larger than 0.

(15) Replace the instruction in J9 by `siEulerA($K$1,$A8,$A9,$G$2, J8)`, copy this instruction down to row 108. Likewise replace the instruction in K9 by `siEulerB($K$1,$K$2,$A8,$A9,$G$2,J8,K8)`, and see what happens with the concentration differences in columns N through P.

(16) Convert the instructions in columns N through P to the corresponding logarithms, so that you need not change the scale of the graph every time you change the value of n.

(17) Run the spreadsheet with $\Delta t = 0.1$ and various values for n, such as 1, 10, and 100. With $n = 100$, the plot of the concentration errors should look like Fig. 7.3.1. By using one-hundred times smaller steps, the error in the semi-implicit Euler method has been reduced ten-thousand-fold.

(18) Try $\Delta t = 0.1$ with $n = 1000$. Depending on the speed of your computer, the computation may now take its sweet time (after all, in each of the 100 cells you make the For … Next loop do 1000 complete calculations), but you get rewarded with absolute errors that are all smaller than 5×10^{-10}! That will be good enough for almost any experiment.

(19) Reset the values in K1:K3 to new guess values, and rerun Solver and SolverAid. For almost any realistic noise the accuracy of your results will now be limited by that noise, rather than by inadequacies in the model. And that is precisely where you want to be: the computation should not add any inaccuracies to your experimental results.

(20) Go back to exercise 7.1.1 and write the corresponding functions `eEulerA` and `eEulerB` for the *e*xplicit case. Then try them out, see how they run. For the same $\Delta t = 0.1$, what value of n do you need in order to get the errors down to the same order of magnitude as those shown in Fig. 7.3.1?

(21) Save the spreadsheet for further use in section 7.6.

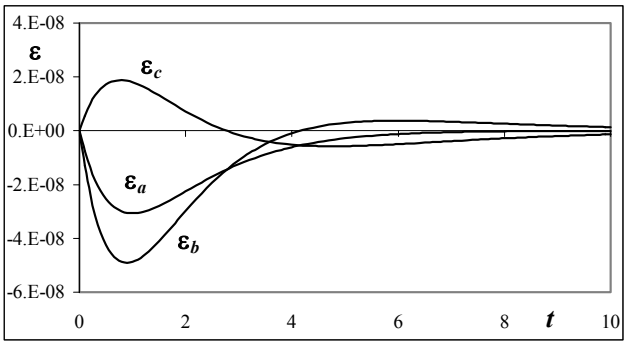


Fig. 7.3.1: The differences between the numerically integrated and exact solutions for the semi-implicit Euler method with $\Delta t = 0.1$ and $n = 100$ for an actual step size of 0.001.

7.4 The shelf life of medicinal solutions (AE3 pp. 311-314)

The rate of decomposition of pharmaceutical solutions depends on the kinetics involved, which are characterized by reaction order, rate constant, and activation energy. This typically requires multiple rate measurements at different temperatures, which can then be combined to compute the activation energy. Industrially, this approach is often considered too time-consuming, especially since only a value for the shelf life is needed. A popular shortcut is therefore to use a single, *nonisothermal* data analysis, introduced to pharmacology by A. R. Rogers in *J. Pharm. Pharmacol.* 15 *Suppl.* (1963) 101T. Rogers used an inverse-logarithmic temperature profile that made the analysis mathematically tractable; others have used a reciprocal or linear temperature profile, and methods based on numerical differentiation or numerical integration have also been used. Here we will use a combination of numerical integration and nonlinear least squares fitting to illustrate its versatility as a generally applicable tool for any temperature profile.

As our specific example we will consider the alkaline degradation of riboflavin, also known as vitamin B2, as reported by B. W. Madsen, R. A. Anderson, D. Herbison-Evans and W. Sneddon in *J. Pharmac. Sci.* 63 (1974) 777. They listed experimental results for a 10^{-4} M solution of riboflavin in 0.1 M aqueous NaOH, on which they took temperature readings and reaction samples while the temperature was gradually raised. The samples were quenched in acetic acid to halt the reaction, and assayed spectrometrically. The reported temperature readings are shown in Fig. 7.4.1, and are listed in A16:B28 of Fig. 7.4.3). They can be represented by the quadratic relation $T - 273.15 = (20.92 \pm 0.27) + (0.4456 \pm 0.0098)\ t - (0.000503 \pm 0.000073)\ t^2$, where T is the absolute temperature in ^{o}K, and t is time in minutes.

We assume first-order kinetics, as common for decomposition reactions of a dilute (10^{-4} M) species in a large excess (0.1 M) of reactant. We will approximate the continuously increasing temperature t as a se-

ries of sufficiently short isothermal steps of length Δt, each at a slightly higher temperature ΔT than its predecessor. Using the semi-implicit Euler method we then have, see (7.1.2), (7.2.1) and (7.2.5),

$$c + \Delta c = \frac{1 - k\,\Delta t/2}{1 + k\,\Delta t/2}\, c \tag{7.4.1}$$

where c is the riboflavin concentration at the start of an isothermal time interval Δt with rate constant k, and Δc is the (negative) riboflavin concentration change during that period Δt. For the dependence of the rate constant on temperature we will assume the Arrhenius equation

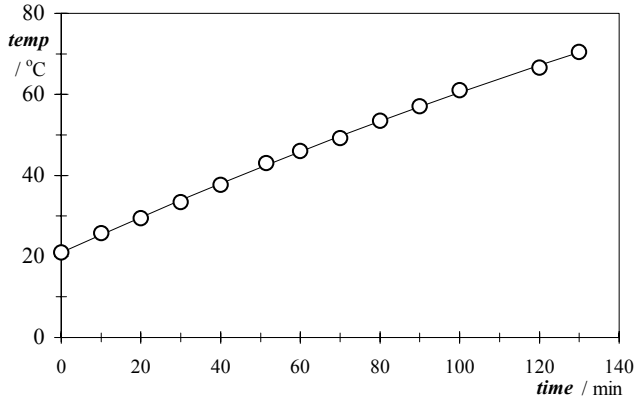


Fig. 7.4.1: The relationship between temperature and time reported by Madsen et al. (open circles) and a quadratic curve (drawn) fitting these data.

$$k = k'\exp\left[\frac{-E}{RT}\right] \tag{7.4.2}$$

where k' is a constant, and E is the activation energy of the reaction; the gas constant R has the value 1.9858775×10^{-3} kcal mol^{-1} K^{-1}. Madsen et al. used 20 °C = 293.15 °K as their reference temperature, and we will do likewise. Rewriting (7.4.2) for $k = k_{20}$ at $T = 293.15$ and combining this with (7.4.2) to eliminate k' yields

$$k = k_{20}\exp\left[\frac{E}{R}\left(\frac{1}{293.15} - \frac{1}{T}\right)\right] \tag{7.4.3}$$

Because Madsen et al. specified their times to the nearest 0.5 min, we write a custom function TStep to subdivide each time interval between successive data points into sub-intervals of 0.5, 0.05, or 0.005 min respectively, and use the earlier-established cubic relation between temperature and time to establish the corresponding temperatures during those sub-intervals. The rate constants at each of these sub-intervals then follow from (7.4.3), and the riboflavin concentrations c are computed from (7.4.1). TStep will then display that riboflavin concentration c at the end of each interval. Since the intervals between data points contain varying numbers of sub-intervals, a `Do ... Loop Until` is used.

```
Function TStep (StepSize, Time1, Time2, c1, k20, E)

' StepSize = length of sub-intervals
' Time1 = time at start of interval
' Time2 = time at end of interval
' k20 = rate constant at 20 C
' E = activation energy, in kcal/mol
' R = 0.0019858775 is the gas constant, in kcal/(mol*K)
' c = concentration
' c1 = ealier-found concentration at start of interval
' c2 = computed concentration at end of interval

' NOTE: Before using this function, make sure that the
' correct equation is used for Temp as a function of Time,
' and that the reference temperature is correct as well

Dim c As Double, Temp As Double, Time As Double
```

```
c = c1
Time = Time1 + StepSize

Do
  Temp = 273.15 + 20.92789 + 0.4455754 * Time - 0.0005030758 * Time ^ 2
  k = k20 * Exp((E / 0.0019858775) * (1 / 293.15 - 1 / Temp))
  c = c * (1 - k * StepSize / 2) / (1 + k * StepSize / 2)
  Time = Time + StepSize
Loop Until Time > Time2 + 0.001
TStep = c
End Function
```

Figure 7.4.2 compares the observed dependence of the riboflavin concentration on time (open circles) with that using a rather crude guess value (gray line) and with the (visually indistinguishable) fits (thin black line) obtained with the three different step sizes. As Table 7.4.1 shows, the results obtained for the three step sizes are quite similar, and in this case are optimal for a step size of 0.05 min (3 sec). While TStep updates within the blink of an eye, repeating it inside the iterative Solver routine will quite noticeably slow Solver at the 0.005 min step size.

The entire spreadsheet, after use of Solver and SolverAid, is shown in Fig. 7.4.3. The strong collinearity between E and k_{20} (see cells D10 and E9) is not surprising for this type of exponential expressions, see section 4.21. The corresponding error surface is displayed in Fig. 7.4.4.

	guess value	step = 0.5 min	step = 0.05 min	step = 0.005 min	
A_{init}	0.6	$0.640_4 \pm 0.001_3$	$0.640_4 \pm 0.001_3$	$0.640_4 \pm 0.001_3$	
E	22	$20.3_1 \pm 0.2_3$	$20.2_9 \pm 0.2_3$	$20.2_8 \pm 0.2_3$	kcal mol^{-1}
k_{20}	0.00015	$0.00031_5 \pm 0.00001_3$	$0.00031_9 \pm 0.00001_3$	$0.00031_9 \pm 0.00001_3$	min^{-1}
SSR	0.02434765	0.00006423	0.00006422	0.00006422	
s_f		0.00253	0.00253	0.00253	

Table 7.4.1: The results obtained with different step sizes. In this case, a step size of 0.05 min suffices.

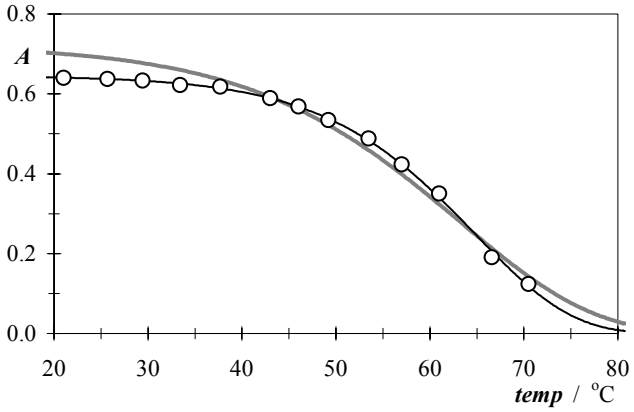


Fig. 7.4.2: The experimental data of Madsen et al. (open circles), a crude fit to them (broad gray line) with assumed values for c_{init}, E, and k_{init}, and the fit obtained by Solver (thin black line).

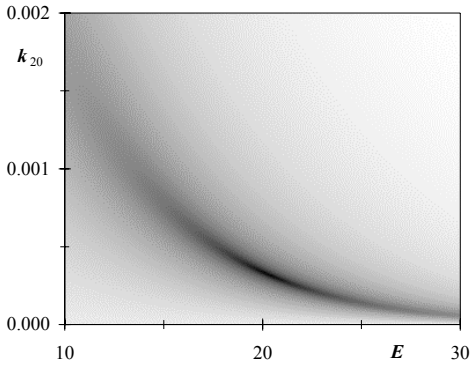


Fig. 7.4.4: The error surface of this problem

	A	B	C	D	E	F
1	**Analysis of data from B.W.Madsen et al., *J. Pharmac. Sci.* 63 (1974) 777-781.**					
2				A_{init} = **0.640385207**	*0.001279146*	
3				E = **20.28332305**	*0.225764006*	kcal mol^{-1}
4				k_{20} = **0.000319187**	*1.33015E-05*	min^{-1}
5			***CM:*** *1.63621E-06*	*-0.000178882*	*1.14645E-08*	
6			*-0.000178882*	*0.050969387*	*-2.96704E-06*	
7			*1.14645E-08*	*-2.96704E-06*	*1.76929E-10*	
8			***CC:*** 1	*-0.619428593*	*0.673809115*	
9			*-0.619428593*	1	*-0.988029796*	
10			*0.673809115*	*-0.988029796*	1	
11				***SSR*** = 7.9969E-05	*0.002534388*	
12						
13	***t***	***T*** -273.15	***T***	***k***	A_{exp}	A_{calc}
14	min	°C	°K	min^{-1}		
15						
16	0	21.0	294.15	3.593E-04	0.640	0.640
17	10	25.7	298.85	6.204E-04	0.637	0.637
18	20	29.4	302.55	9.423E-04	0.633	0.632
19	30	33.4	306.55	1.464E-03	0.622	0.625
20	40	37.7	310.85	2.321E-03	0.618	0.613
21	51.5	43.0	316.15	4.026E-03	0.589	0.591
22	60	46.0	319.15	5.454E-03	0.568	0.568
23	70	49.2	322.35	7.494E-03	0.534	0.532
24	80	53.5	326.65	1.137E-02	0.488	0.484
25	90	57.0	330.15	1.584E-02	0.423	0.423
26	100	61.0	334.15	2.294E-02	0.351	0.350
27	120	66.6	339.75	3.797E-02	0.191	0.191
28	130	70.5	343.65	5.341E-02	0.124	0.120
29	140					0.065
30	150					0.030
31	160	***cell:***	***instruction:***		***copied to:***	0.011
32	170					0.003
33	180	D11 = SUMXMY2(F16:F28,E16:E28)				0.001
34	190	F16 = D2				0.000
35	200	F17 = TStep2(0.05,A16,A17,F16,$D$4,$D$3)			F18:F35	0.000

Fig 7.4.3: The spreadsheet after using Solver and SolverAid. The extension of t beyond 130 min and the corresponding extension of A_{calc} is so that Figs. 7.4.2 can show the entire theoretical curve; cells C13:D28 are not strictly needed either.

7.7 The XN 4th order Runge-Kutta function AE3 pp. 320-322)

XN contains several ODE-solving functions that are all listed in the Paste Function box once XN is installed. Here we will merely illustrate one of them, ODE_RK4, which makes it very simple to use the 4th order Runge-Kutta method. In exercise 7.7.1 we will use the simple differential expression $dy/dx = -2xy$, with the known solution $y = e^{-x^2}$, and in exercise 7.7.2 we will apply the same method to the sequential chemical reactions A→B→C, which also has known, algebraic solutions, see (7.1.15) though (7.1.17).

Exercise 7.7.1:

(1) On a new spreadsheet deposit the quasi-algebraic expression `y'=-2*x*y` in cell C3, and in cell E3 put a value for the step size, say 0.1.

(2) In B4 place the label x, and in C4 the label y. These labels are needed so that the function can relate the variables in C3 with their numerical values, which must be listed immediately below their labels.

(3) In B5 write the initial value of x, and in C5 that of y. We will here use $x_0 = 0$ so that $y_0 = 1$.

(4) Highlight cells B6:C6, type in the function description, `=ODE_RK4($C$3,B5:C5,$E$3)`, and deposit this instruction with Ctrl◡Shift◡Enter.

(5) Now comes a neat trick: again highlight B6:C6, grab its handle (at the lower-right corner of cell C6), pull it down to row 35, and release. You will now have the 4th-order Runge-Kutta approximation of the differential equation in cell C4, computed from $x = 0$ to $x = 3$ at intervals of 0.1.

(6) In cell D4 place a label for y_{ref}, and in D5 the instruction `=EXP(-1*B5^2)`. Copy this instruction down to row 35.

(7) In cell E4 put a label for p*E*, in cell E6 the instruction `='IF(D6=E6," ! ",-LOG(ABS((D6-E6)/E6))`, and pull this also down to row 35.

While the absolute error ε = C35–D35 at $x = 3$ is only about 10^{-6}, the value in C35 is still almost 1% off from its correct value of y_{ref} in D35. Can we do better with a smaller step size? Answer that question for yourself, using steps of 0.01 and 0.001 respectively. Reaching $x = 3$ from $x = 0$ with smaller steps implies longer columns, but the results (as summarized in the right bottom of Fig. 7.7.1) speak for themselves. As expected for a 4th order method, a tenfold reduction in step size leads to a decrease in the relative error of the order of 10^4. Again, we can avoid the longer columns with a function that, in turn, exercises the function ODE_RK4 with *n* times smaller increments, but displays its results only after *n* such steps.

For the sequence of chemical reactions A → B → C with concentrations *a*, *b*, and *c*, we have the differential equations (7.1.2) through (7.1.4) with the initial conditions (7.1.5). Exercise 7.7.2 illustrates how we might approach this problem with the XN function ODE_RK4.

Exercise 7.7.2:

(1) On a new spreadsheet put symbols for elapsed time *t*, and for the concentrations *a*, *b*, and *c* in, say, B4:E4. Immediately below them place their initial conditions, for which we will here use those of section 7.1, i.e., $t_0 = 0$, $a_0 = 1$, $b_0 = 0$, and $c_0 = 0$.

(2) Above the labels for a, b, and c place the differential equations governing their behaviors, in quasi-algebraic form, i.e., $a' = -a$, $b' = a - b/2$, and $c' = b/2$, see (7.1.2) through (7.1.4) with $k_1 = 1$ and $k_2 = 0.5$.

(3) Place the value of the step size somewhere, say, 0.1 in cell E2.

(4) Highlight B6:E6, type the instruction `=ODE_RK4($C$3:$E$3,B5:E5,$E$2).`, and deposit it with the array instruction Ctrl◡Shift◡Enter.

(5) In B5 write the initial value of x, and in C5 that of y. We will here use $x_0 = 0$ so that $y_0 = 1$.

(6) Grab the handle of this array, and pull it down to, say, row 105.

(7) Highlight the block B5:E105, and plot it as an XY graph.

(8) Note how quick and easy ODE_RK4 makes it to solve a set of coupled differential equations.

(9) Highlight B105:E105, grab its handle, and pull it all the way down to row 1005. For longer times *t*, the concentrations *a* and *b* become quite negligible vs. *c*., which means that the simulated reaction completely ran its course.

(10) Change the value of the step size in E2 to, say, 0.01. The graph will show the initial tenth of the reaction in ten times greater detail and with systematic errors reduced by about a factor of ten thousand. Changing the step size to 1 does just the opposite. Try it. Simulating such solutions to coupled ODE's can hardly be made any easier.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1													
2		***4th order Runge-Kutta ODE-solver:***											
3		***ODE =***	**y'=-2*x*y**	***step =***	**0.1**	***ODE =***	**y'=-2*x*y**	***step =***	**0.01**	***ODE =***	**y'=-2*x*y**	***step =***	**0.001**
4		***x***	***y***	y_{ref}	p***E***	***x***	***y***	y_{ref}	p***E***	***x***	***y***	y_{ref}	p***E***
5		**0**	**1**	1.000000		**0**	**1**	1.000000		**0**	**1**	1.000000	
6		0.1	0.990050	0.990050	9.38	0.01	0.999900	0.999900	!	0.001	0.999999	0.999999	!
7		0.2	0.960789	0.960789	8.39	0.02	0.999600	0.999600	!	0.002	0.999996	0.999996	!
8		0.3	0.913931	0.913931	7.91	0.03	0.999100	0.999100	!	0.003	0.999991	0.999991	!
9		0.4	0.852144	0.852144	7.71	0.04	0.998401	0.998401	!	0.004	0.999984	0.999984	!
10		0.5	0.778801	0.778801	8.49	0.05	0.997503	0.997503	15.18	0.005	0.999975	0.999975	!
11		0.6	0.697676	0.697676	7.06	0.06	0.996406	0.996406	14.95	0.006	0.999964	0.999964	!
12		0.7	0.612627	0.612626	6.45	0.07	0.995112	0.995112	14.75	0.007	0.999951	0.999951	!
13		0.8	0.527293	0.527292	6.02	0.08	0.993620	0.993620	14.57	0.008	0.999936	0.999936	!
14		0.9	0.444859	0.444858	5.66	0.09	0.991933	0.991933	14.42	0.009	0.999919	0.999919	!
15		1.0	0.367881	0.367879	5.35	0.10	0.990050	0.990050	14.29	0.010	0.999900	0.999900	!
16		1.1	0.298200	0.298197	5.08	0.11	0.987973	0.987973	14.18	0.011	0.999879	0.999879	!
17		1.2	0.236931	0.236928	4.84	0.12	0.985703	0.985703	14.09	0.012	0.999856	0.999856	!
18		1.3	0.184524	0.184520	4.62	0.13	0.983242	0.983242	14.00	0.013	0.999831	0.999831	!
19		1.4	0.140864	0.140858	4.41	0.14	0.980591	0.980591	13.92	0.014	0.999804	0.999804	!
20		1.5	0.105406	0.105399	4.22	0.15	0.977751	0.977751	13.87	0.015	0.999775	0.999775	!
21		1.6	0.077312	0.077305	4.04	0.16	0.974725	0.974725	13.82	0.016	0.999744	0.999744	!
22		1.7	0.055584	0.055576	3.88	0.17	0.971514	0.971514	13.78	0.017	0.999711	0.999711	!
23		1.8	0.039171	0.039164	3.72	0.18	0.968119	0.968119	13.76	0.018	0.999676	0.999676	!
24		1.9	0.027059	0.027052	3.57	0.19	0.964544	0.964544	13.77	0.019	0.999639	0.999639	!
25		2.0	0.018322	0.018316	3.43	0.20	0.960789	0.960789	13.80	0.020	0.999600	0.999600	!
26		2.1	0.012161	0.012155	3.29	0.21	0.956858	0.956858	13.87	0.021	0.999559	0.999559	!
27		2.2	0.007912	0.007907	3.17	0.22	0.952753	0.952753	14.06	0.022	0.999516	0.999516	!
28		2.3	0.005046	0.005042	3.04	0.23	0.948475	0.948475	14.76	0.023	0.999471	0.999471	!
29		2.4	0.003155	0.003151	2.93	0.24	0.944027	0.944027	14.08	0.024	0.999424	0.999424	!
30		2.5	0.001933	0.001930	2.81	0.25	0.939413	0.939413	13.66	0.025	0.999375	0.999375	!
31		2.6	0.001162	0.001159	2.70	0.26	0.934634	0.934634	13.40	0.026	0.999324	0.999324	!
32		2.7	0.000684	0.000682	2.60	0.27	0.929694	0.929694	13.20	0.027	0.999271	0.999271	!
33		2.8	0.000395	0.000394	2.50	0.28	0.924595	0.924595	13.03	0.028	0.999216	0.999216	!
34		2.9	0.000224	0.000223	2.40	0.29	0.919339	0.919339	12.89	0.029	0.999159	0.999159	!
35		3.0	0.000124	0.000123	2.30	0.30	0.913931	0.913931	12.76	0.030	0.999100	0.999100	!

row:	***instruction:***	***pulled down to row:***	***h:***	***pE for x = 3:***
B6:C6 =	ODE_RK4($C$3,B5:C5,$E$3)	35	0.1	2.30
F6:G6 =	ODE_RK4($G$3,F5:G5,$I$3)	305	0.01	6.47
J6:K6 =	ODE_RK4($K$3,J5:K5,$M$3)	3005	0.001	10.50

		copied to row:
D5 =	EXP(-1*B5^2)	35
E6 =	IF(C6=D6," ! ",-LOG(ABS((C6-D6)/D6))	35
H5 =	EXP(-1*F5^2)	305
I6 =	IF(G6=H6," ! ",-LOG(ABS((G6-H6)/H6))	305
L5 =	EXP(-1*J5^2)	3005
M6 =	IF(K6=L6," ! ",-LOG(ABS((K6-L6)/L6))	3005

Fig. 7.7.1: The top part of the spreadsheet of exercise 7.7.1 for the Runge-Kutta analysis of $dy/dx = -2xy$ using the XN function ODE_RK4.

	A	B	C	D	E
1					
2		**A to B to C**		*step* =	**0.1**
3		*ODE* =	**a'=-a**	**b'=a-b/2**	**c'=b/2**
4		***t***	***a***	***b***	***c***
5	*InitVal* =	**0**	**1**	**0**	**0**
6		0.1	0.904838	0.092784	0.002379
7		0.2	0.818731	0.172213	0.009056
8		0.3	0.740818	0.239779	0.019402
9		0.4	0.67032	0.296821	0.032859
10		0.5	0.606531	0.34454	0.048929
11		0.6	0.548812	0.384013	0.067175
12		0.7	0.496586	0.416205	0.087209
13		0.8	0.449329	0.441982	0.108689
14		0.9	0.40657	0.462116	0.131314
15		1.0	0.36788	0.477302	0.154818
16		1.1	0.332871	0.488157	0.178972
17		1.2	0.301195	0.495234	0.203571
18		1.3	0.272532	0.499027	0.228441
19		1.4	0.246597	0.499976	0.253427
20		1.5	0.22313	0.498472	0.278397
21		1.6	0.201897	0.494864	0.303239
22		1.7	0.182684	0.489462	0.327854
23		1.8	0.165299	0.482541	0.35216
24		1.9	0.149569	0.474344	0.376087
25		2.0	0.135336	0.465088	0.399577

cells: instruction:
B6:E6 = ODE_RK4($C$3:$E$3,B5:E5,E

Fig. 7.7.2: The top of the spreadsheet simulating the chemical reaction A → B → C with the NX instruction ODE_RK4.

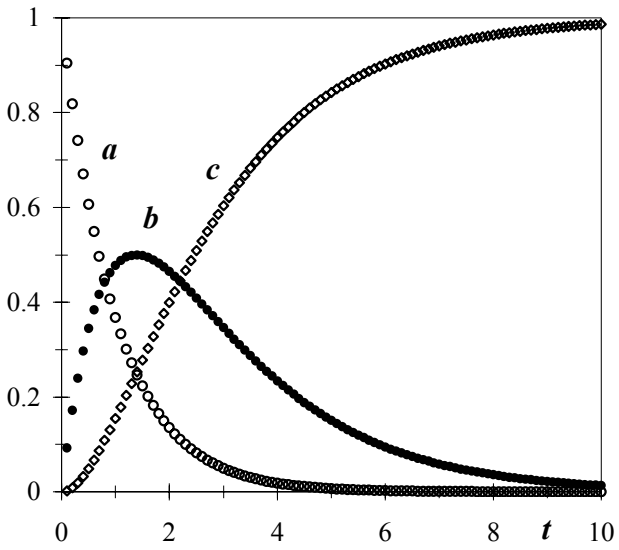


Fig. 7.7.3: The concentrations *a*, *b*, and *c* of the reacting chemical species A → B → C as a function of time *t*, as simulated with ODE_RK4. As in Fig. 7.6.1, the concentration errors are all smaller than $\pm 10^{-6}$.

8.6 Ranges & arrays (AE3 pp. 345-346)

A typical stand-alone computer language uses *arrays*, i.e., collections of data ordered by indices, and VBA is no exception. Each element of an array is defined by its indices, which typically start counting from 1 or 0. Such array indices are purely a matter internal to the array, regardless of the origin of the information stored in it. Moreover, each array element contains only one item of information: a number, a

text string, a date, etc. A VBA array must have at least two elements: `vA(1 To 1)` is not acceptable. Depending on its dimensionality, an array can use one or more indices to define its elements: `vA(1 To 7), vB(1 To 7, 1 To 3), vC(1 To 7, 1 To 3, 1 To 15)`, etc. Note that the number of array elements of multi-dimensional arrays can be quite large: as just defined, `vC` would have $7 \times 3 \times 15 = 315$ elements.

The integration of a computer language with a spreadsheet requires another structure, which is where *ranges* come in. VBA uses ranges to specify individual cells, cell blocks, or a collection of cells and/or cell blocks *on the spreadsheet*, including their locations and their contents. And those contents can have many aspects: each spreadsheet cell may not only specify, say, a formula, a resulting value, and an address, but also type font, color, and all kinds of other formatting properties.

A range contains its spreadsheet address(es), and therefore doesn't lose its connection with the spreadsheet. This allows the user to go back and forth between the spreadsheet and a VBA calculation. But because a range must hold so much information, much of it often peripheral to the computation, it is slow and tedious to use a range in complicated calculations. On the other hand, VBA arrays are strictly local to VBA, and are independent (and ignorant) of the location of the corresponding input information on the spreadsheet. In all but the simplest cases we will therefore read and write with ranges, then transfer the necessary information to arrays, and operate within VBA with those arrays. Finally, at the end of the calculation, we read the array(s) back into a range in order to make the result appear on the spreadsheet.

Other differences: a range can be defined by an explicit spreadsheet address or, as we have already seen, by the convenient `Selection`, in which case VBA associates the range address with that of the highlighted region. A range can refer to a single spreadsheet cell, as it did in the subroutines Read() of section 8.1. A range can be moved around on the spreadsheet with an instruction such as `rgName.Offset(3,-1).Select` which moves the range rgName 3 rows down and one column to the left. A range can also be resized with `rgName.Resize(5,4).Select`, in which its top left corner stays, but its number of rows and columns can either increase or decrease to 5 rows and 4 columns. Combining the offset and resize instructions allows complete freedom to move a range. No information is lost upon offsetting and/or resizing a range, see the MoveAndResizeARange MacroMorsel.

Because a range is tied to the spreadsheet, it is strictly one- or two-dimensional: one-dimensional for a single cell, row, or column, two-dimensional for any other collection of input cells. Excel does not support three- or higher-dimensional ranges.

An array can contain only one type of data, with a minimum of two elements. An array has an internal numbering system that starts with either index 1 or 0 (when Option Base 0 is used), and can have up to 60 dimensions. Because of the ambiguity of the base option used, it is best always to specify where you start and stop counting in an array.

For an array to be compatible with a range, that array must use two indices specifying (in this order) rows and columns. If the array contains only one row or one column, it must specify the other index as 1 in order to be compatible with a range.

As already mentioned, you can redimension an array, but all previous information in the array is lost, except when you enlarge the array with the combination `ReDim Preserve`, in which case you can add extra columns but no additional rows. This is because the array is actually stored as a linear sequence, first the first row, then the second, and so on. The data in the new columns can therefore simply be added at the end of this linear sequence.

In summary: ranges are tied to the Excel spreadsheet, while arrays exist only in either the spreadsheet, or VBA. For maximum efficiency, use a range to read information from the spreadsheet, extract the needed information into one or more specific arrays, use these arrays to manipulate the data in VBA, then write the results back into a range, and in that form return the results to the spreadsheet. In order to facilitate the translations from range to array and back, the array must be two-dimensional, even if it only represents data in a single row or column, in which case the unused dimension should be specified as 1 to 1. Conversion from spreadsheet array to VBA array via the corresponding range can often be achieved conveniently with an instruction such as `dataArray = Selection.Value` that extracts an array of values from `Selection`, the range highlighting the spreadsheet array. Conversely, writing the results back can often be done simply with `Selection.Value = dataArray`. We will revisit this matter in sections 8.9 and 8.10.

8.15.1 Invasive sampling (AE3 pp. 363-364)

It is sometimes convenient to change a parameter on the spreadsheet, and to exploit the consequences of that change. As our example, we will use this here to find the first derivative of a function $F(a)$ by changing a. The simplest approach is to read a range, change the contents of one of its cells, write that back onto the spreadsheet, and then read the resulting change(s) in other cells. In this manner we can, e.g., read values of a function $F(x)$ by changing x, without the need to code that function explicitly in the subroutine. Below we will illustrate this in order to compute the derivative of a function. Excel has two specific text functions that can be helpful in this respect: `SUBSTITUTE` and `REPLACE`.

The basis of numerical differentiation is that Excel can evaluate a function of one or more variables, and that, in order to find its (partial) derivative, we merely need to change one such variable by a known, relatively small amount Δa (i.e., a small *fraction* of the variable value a) and observe the resulting change ΔF in the function. After all, the derivative dF/da of a function F with respect to a (or, when F depends on more than one parameter, its partial derivative $\partial F/\partial a$) is defined as

$$\frac{dF}{da} = \lim_{\Delta a \to 0} \frac{F(a+\Delta a) - F(a)}{\Delta a} \tag{8.15.1}$$

which, for $\Delta a \ll a$, can be approximated by

$$\frac{dF}{da} \approx \frac{F(a+\Delta a) - F(a)}{\Delta a} \tag{8.15.2}$$

By letting the spreadsheet evaluate $F(a+\Delta a)$ and $F(a)$, we can find the value of dF/da of any function that Excel can compute. For a more detailed treatment see section 9.2.

In order to use (8.15.1) in a macro, place a in one spreadsheet cell, and in another cell deposit a formula for F that depends on a (and possibly on other factors, in which case we will compute the partial derivative). Note that we will only find the *numerical value* of the derivative of the function, *at its current value*, not a generally applicable formula.

For starters, just to illustrate how this macro works, try $a = 3$ with $F = 4 + 5\,a^2 = 49$, for which F_{deriv} should be $5 \times 2\,a = 30$, or with $F = 7 - 6\ln(a) = 0.408326268$, which should yield $F_{deriv} = -6/a = -2$. Then change the value of a and/or the formula for F to suit your own fancy.

```
Sub NumericalDifferentiation()

Dim A As Double, Amin As Double, Aplus As Double
Dim F As Double, Fmin As Double, Fplus As Double
Dim Fderiv As Double
Dim rgA As Range, rgF As Range

' Read the values of A and F

Set rgF = Application.InputBox(Prompt:= _
  "The function is located in ", Type:=8)
rgF.Select
F = rgF.Value

Set rgA = Application.InputBox(Prompt:= _
  "The variable a is located in ", Type:=8)
rgA.Select
A = rgA.Value

' Modify A to Aplus and read the corresponding Fplus

Aplus = A * (1 + 1 / 1048576)
Selection.Value = Aplus
rgF.Select
Fplus = rgF.Value

' Modify A to Aminus and read the corresponding Fminus

Aminus = A * (1 - 1 / 1048576)
rgA.Select
Selection.Value = Aminus
rgF.Select
Fminus = rgF.Value
```

```
' Clean up by restoring the initial value of A

rgA.Select
Selection.Value = A

' Compute and display the derivative

Fderiv = (Fplus - Fminus) / (A / 524288)
MsgBox "The first derivative of the function is " & Fderiv

End Sub
```

The above works when we change values, but is slightly more complicated when the cells to be invaded contain formulas but display their values, as in Figs. 2.14.1 and 2.14.2. To cover both values and formulas, we start the macro by copying their formulas, then temporarily replace these by their numerical values with an in-place Copy PasteSpecial as Values, operate on the resulting values, and finally restore the original formulas. This also works for numbers because asking the formula of a cell containing a number yields that number. For details, see how this approach is implemented in the custom macro Propagation. Since Solver only yields numbers, it is not necessary for SolverAid.

8.16 Using the XN equation parser (AE3 pp. 365-368)

Invasive sampling works in general, because anything that can be computed on the spreadsheet can be read. But because it requires time-consuming switching between spreadsheet and VBA, using ranges loaded with excess information, it is often an impractical choice. A much faster, though somewhat less general approach illustrated in section 8.15.2 is analogous to what was described in section 8.14 as deconstructing an address, viz. deconstructing an equation, then reassembling it in VBA. This is precisely what the XN equation parser does: it relieves you of the need to enter this code yourself. The parser takes an equation written in quasi-algebraic, Excel-like notation and converts it into VBA code, so that the rest of the calculation can all be done in VBA.

Take, e.g., the integration of an algebraic function $F(x)$ in the range $a \le x \le b$. In chapter 9 we will use macros such as RombergAuto to exploit the spreadsheet formula in computing the integral, but it is much faster to code the function in VBA. However, it is more user-friendly, and the computation is equally fast, when we write that code as an Excel-style text string that can be converted into VBA code. This requires a parser, which evaluates the string and reformulates it in VBA, as in `=Integr("1.3*exp(-x^2 /6)*sin(pi/4)/Sqr(3/x)",0,13)`. In this example you can see the structure of the instruction: the formula $F(x)$ is entered in code between quotation marks, followed by the limits of integration, here 0 and 13. Below we list the rules used for writing code that the XN parser can interpret.

(1) Numbers can be integer, decimal, or in scientific (exponential) notation: 1, 2.34, –5.6, 7.8E–9.

(2) Complex numbers can be indicated as an ordered pair: (a,b), (1,-0.1), (-0.12345, 6.78E-19), or as a compound number: (a+bj), (1-0.1j), (-0.12345+6.78E-19j). For nested formulas, use a multiplier before the symbol j, as in ((2+3*4)+(5/6)*j. You can use either i or j.

(3) Angles must be specified as rad, deg, or grad, where rad(pi/2) = deg(90) = grad(100). Angles can also be written in degrees, minutes and seconds, as in 20d 34m 56s, always in the Sumerian sexagesimal (base 60) system with 60s = 1m and 60m = 1d.

(4) Variables can be any alphanumeric string that starts with a letter: a, x, Alpha, b1, time_2. Capitals are accepted but make no difference: alpha, Alpha and ALPHA are interpreted as the same quantity.

(5) The usual algebraic expressions are accepted, such as +, -, *, /, \, ^, ||, as well as the logical expressions <, >, =, <=, >=, <>, or, nor, nxor (for exclusive nor), not.

(6) The mathematical constants pi (= 3.1416..) and e (= 2.718..) are accepted.

(7) Most common mathematical functions are accepted, such as: exp, ln, log, sin, cos, atan, sqr, !. As in Excel, they must be followed by their argument, within parentheses. Separate multiple arguments by commas, as in max(a,b). The parser also accepts some multivariable arguments, as in DSNormal(x,m,s) or HypGeom(x,a,b,c) for a normal (Gaussian) distribution or a hypergeometric series respectively, and functions with a variable number of arguments (up to 20), as in mean(x1, x2, …).

(8) Implicit multiplication is (with some exceptions) not acceptable: xy is read as a separate symbol name, not as the product of x and y, which should be formulated as x*y. Similarly, write 2*a rather than 2a.

(9) The complete (and impressive) list of all operations and functions recognized by the parser of the current version of XN is given in Table 17.1. For more details, see the Help file under the "Math formula string" heading.

Function	*Use*	*Function*	*Use*
+	R C M	–	R C M
!	R C M	<	R C M
%	R C M	<= , =<	R C M
*	R C M	<>	R C M
/	R C M	=	R C M
\	R C M	>	R C M
^	R C M	| |	R C M
abs(x)	R C M	DSBinom(k, n, p, [j])	R M
acos(x)	R C M	DSCauchy(x, m, s, n, [j])	R
acosh(x)	R C M	DSChi(x, r, [j])	R
acot(x)	R M	DSErlang(x, k, l, [j])	R
acoth(x)	R M	DSGamma(x, k, l, [j])	R
acsc(x)	R M	DSLevy(x, l, [j])	R
acsch(x)	R M	DSLogNormal(x, m, s, [j])	R M
AiryA(x)	R	DSLogistic(x, m, s, [j])	R M
AiryB(x)	R	DSMaxwell(x, a, [j])	R M
alog(z)	R C M	DSMises(x, k, [j])	R
and(a,b)	R C M	DSNormal(x, m, s, [j])	R M
arg(z)	C	DSPoisson(k, z, [j])	R
asec(x)	R M	DSRayleigh(x, s, [j])	R M
asech(x)	R M	DSRice(x, v, s, [j])	R
asin(x)	R C M	DSStudent(t, v, [j])	R
asinh(x)	R C M	DSWeibull(x, k, l, [j])	R M
atanh(x)	R C M	Ei(x)	R C
atn(x), atan(x)	R C M	Ein(x,n)	R
atan2(y,x)	R M	Elli1(f,k)	R
BesselI(x,n)	R	Elli2(f,k)	R
I0(x)	R	erf(x)	R C M
BesselJ(x,n)	R	erfc(x)	R C M
J0(x)	R	e#	R C M
BesselK(x,n)	R	eu#	R C M
K0(x)	R	exp(x)	R C M
BesselY(x,n)	R	exp(x)	R C M
Y0(x)	R	fact(x)	R C M
beta(x,y)	R C M	fix(x)	R C M
betaI(x,a,b)	R	FresnelC(x)	R
cbr(x)	R M	FresnelS(x)	R
Ci(x)	R	gamma(x)	R C M
clip(x,a,b)	R M	gammai(a,b)	R
comb(n,k)	R C M	gammaln(x)	R C
conj(x)	C	gcd(a,b)	R
cos(x)	R C M	grad(x)	R M
cosh(x)	R C M	hour(a)	R
cot(x)	R M	HypGeom(a,b,c,x)	R
coth(x)	R M	im(z)	C
csc(x)	R M	int(x)	R C M
csch(x)	R M	integral(f,z,a,b)	C
dateserial(a1,a2,a3)	R	inv(x)	R C M
day(a)	R	lcm(a,b)	R M
dec(x)	R M	ln(x)	R C M
deg(x)	R M	ln2#	R C M

Function	*Use*	*Function*	*Use*
digamma(x), psi(x)	R C	ln10#	R C M
DSBeta(x, a, b, [j])	R	log(x)	R C M
logn(a,b)	R M	rad(x)	R M
max(a,b,...)	R C M	rad5#	R C M
min(a,b,...)	R C M	re(z)	C
mcd(a,b,...)	R C M	rnd(x)	R C M
mcm(a,b,...)	R C M	root(x,n)	R C M
Mean(a,b,...)	R M	round(x,d)	R M
Meang(a,b,...)	R M	sec(x)	R M
Meanq(a,b,...)	R M	sech(x)	R M
minute(a)	R	second(a)	R
mod(a,b)	R C M	serie(….)	C
month(a)	R	sgn(x)	R C M
nand(a,b)	R M	Si(x)	R
neg(z)	R C M	sin(x)	R C M
nor(a,b)	R M	sinh(x)	R C M
not(a)	R C M	sq(x)	R C M
nxor(a,b)	R M	sqr(x), sqrt(x)	R C M
or(a,b)	R C M	Stdev(a,b,...)	R M
perm(a,b)	R M	Stdevp(a,b,...)	R M
pi	R C M	Step(x,a)	R M
pi2	R C M	sum(a1,a2,…)	R M
pi3	R C M	tan(x)	R C M
pi4	R C M	tanh(x)	R C M
pix2	R C M	timeserial(a1,a2,a3)	R
PolyCh(x,n)	R	Var(a,b,...)	R M
PolyHe(x,n)	R	Varp(a,b,...)	R M
PolyLa(x,n)	R	xor(a,b)	R C M
PolyLe(x,n)	R	year(a)	R
r2c(a,b)	C	zeta(x)	R C M

Table 8.16.1: The functions recognized by the MathParser. R = real, C = complex, M = multiprecision. Functions with M can be evaluated in double or multi-precision, those with only C can be used only with cplxEval, and those with only R with xEval or xEvall in standard precision (with DgtMax = 0).

You can use the parser in various ways, both as the explicit function Eval or implicitly, as in the above `=Integr("1.3*exp(-x^2/6)*sin(pi/4)/sqr(3/x)",0,13)`. You can also use it in writing your own custom functions. For extended precision, the functions xEval or xEvall are used instead. (The difference between xEval and xEvall is merely that xEvall first looks at the top of a column for its label, and is therefore somewhat slower.) The large range of functions and function names that the MathParser can handle can greatly simplify complicated calculations.

8.22 Case study 5: modifying Mapper's BitMap (AE3 pp. 382-386)

Mapper provides a convenient visualization of a monotonic function $z = F(x,y)$ of two variables, x and y, but there are many possible approaches as well as color schemes, and Exercise 8.22.1 will show you how to modify these. The examples will be for Mapper0, because we can show its gray-scale results here in print. The same principles apply to color images, some of which are illustrated on my Excellaneous website.

Exercise 8.22.1:

(1) On a new spreadsheet, place the number 0 in cell D20, the instruction `=D20+1` in E20, and copy this to cell CZ20, so that you have a horizontal axis from 0 to 100.

(2) Likewise, in cell C21 put the number 100, with the instruction `=C21-1` in cell C22, and copy this down to C121 to make your vertical scale.

(3) In cell D21 deposit the instruction =D$20+$C21, and copy this instruction to the entire block D4:CZ121. We will use this as our test pad.

(4) Highlight C20:CZ121, and call Mapper0, which will make a map like the one shown in Fig. 8.22.1*a*.

(5) Now go to the VBEditor (e.g., with Alt◡F11), find the code section for Mapper (near the end of the MacroBundle; it is easiest to step through the code using PageDown and PageUp, because then the name of the routine will show in the top-right window above the Module), find the subroutine BitMap0, highlight that bitmap, copy and paste it (e.g., with Ctrl◡c, Ctrl◡v, Ctrl◡v) so that you get two copies of it, one below the other, then rename one of them BitMap00. This is the spare copy of the original, which we will rename back to Bitmap0 once we are done with our experiment. Below we reproduce the code of BitMap0; the parts to be modified are bold-printed.

```
Private Sub BitMap0(hMax As Integer, wMax As Integer, pixelArray As Variant)

 ' Gray-scale, no color
Dim H As Integer, w As Integer, RedVal As Integer
Dim GreenVal As Integer, BlueVal As Integer
For H = hMax To 0 Step -1
  For w = 0 To wMax - 1
    RedVal = pixelArray(H, w)
    GreenVal = pixelArray(H, w)
    BlueVal = pixelArray(H, w)
    WriteAPixel RedVal Mod 256, GreenVal Mod 256, BlueVal Mod 256
  Next w
' Do not change the following, essential row padding
 w = wMax * 3
  Do While (w Mod 4) <> 0
    WriteAByte 0
    w = w + 1
  Loop
Next H
End Sub
```

(6) In BitMap0, now make the following substitution (again shown in boldface), which replaces the direct proportionality of the earlier gray scale with three distinct *bands* covering specific intervals, converting Mapper into a Bander. The resulting map is shown in Fig. 8.22.1*b*, which now has boundaries similar to those of a contour diagram. These boundaries are crude, having their resolution determined by the size of the underlying array, but their computation requires no further processing and is therefore very quick. It can therefore serve as a quick-and-dirty substitute for a contour diagram.

```
Private Sub BitMap0(hMax As Integer, wMax As Integer, pixelArray As Variant)

' Gray-scale, no color
Dim H As Integer, w As Integer, RedVal As Integer
Dim GreenVal As Integer, BlueVal As Integer
For H = hMax To 0 Step -1
  For w = 0 To wMax - 1
    If pixelArray(H, w) < 85 Then
      RedVal = 85
      GreenVal = 85
      BlueVal = 85
    ElseIf pixelArray(H, w) < 170 Then
      RedVal = 170
      GreenVal = 170
      BlueVal = 170
   ElseIf pixelArray(H, w) >= 170 Then
      RedVal = 255
      GreenVal = 255
      BlueVal = 255
    End If
    WriteAPixel RedVal Mod 256, GreenVal Mod 256, BlueVal Mod 256
  Next w
' Do not change the following, essential row padding
  w = wMax * 3
 Do While (w Mod 4) <> 0
    WriteAByte 0
    w = w + 1
  Loop
Next H
End Sub
```

(7) Now that we have established the principle of plotting bands, here is a more flexible example of coding, for a flexible number of bands (here: 15). Incorporate this in your BitMap0, and verify that the resulting banded map is similar to Fig. 8.22.1*c*.

```
Private Sub BitMap0(hMax As Integer, wMax As Integer, pixelArray As Variant)

' Gray-scale, no color

Dim H As Integer, i As Integer, iMax As Integer, w As Integer
Dim RedVal As Integer, GreenVal As Integer, BlueVal As Integer

iMax = 15
For H = hMax To 0 Step -1
  For w = 0 To wMax - 1
    If pixelArray(H, w) < Int(255 / iMax) Then
      RedVal = 0
      GreenVal = 0
      BlueVal = 0
    End If
    For i = 1 To iMax - 1
     If pixelArray(H, w) > i * Int(255 / iMax) And _
        pixelArray(H, w) <= (i + 1) * Int(255 / iMax) Then
        RedVal = i * Int(255 / iMax)
        GreenVal = i * Int(255 / iMax)
        BlueVal = i * Int(255 / iMax)    End If
      End If
    Next I
    WriteAPixel RedVal Mod 256, GreenVal Mod 256, BlueVal Mod 256
  Next w

' Do not change the following, essential row padding

  w = wMax * 3
  Do While (w Mod 4) <> 0
    WriteAByte 0
    w = w + 1
  Loop
Next H

End Sub
```

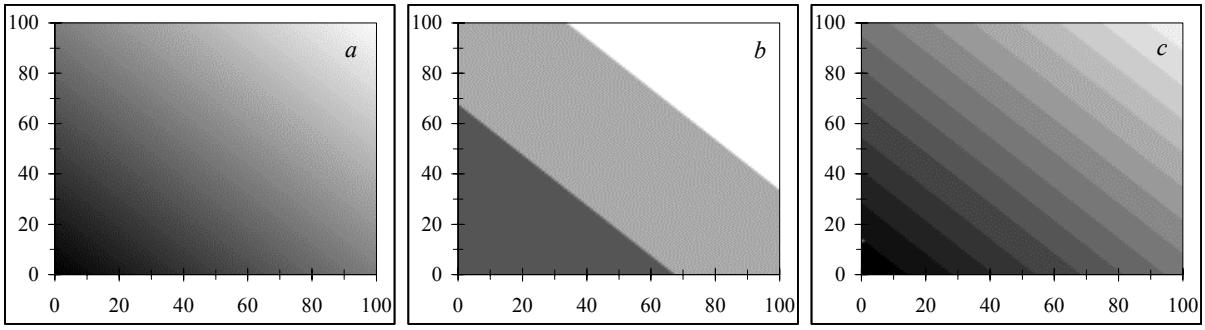


Fig. 8.22.1: Three maps of the same input data: *a*: with a gradual gray-scale, *b*: with three bands, and *c*: with a user-defined number of bands.

(8) Use the following code to emphasize the central portion of the graph, as was done, e.g., in Fig. 1.5.3. This leads to the banded map of Figs. 8.22.1*d* through 8.22.1*f* for various values of iMax.

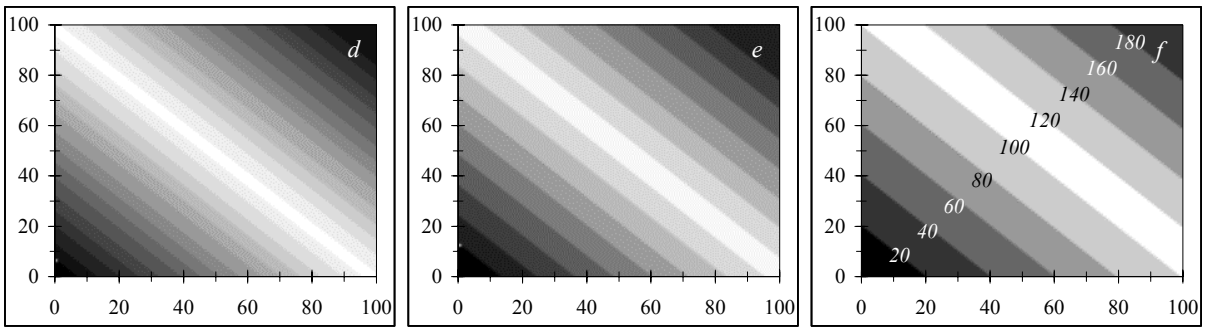


Fig. 8.22.1 cont'd: Three banded maps of the same input data, with emphasis on the central data values: *d*: with iMax = 15, *e*: with iMax = 8, and *f*: with iMax = 5. Panel *d* illustrates that too many bands will blur the effect, and approaches a gradual, non-banded display.

```
Private Sub BitMap0(hMax As Integer, wMax As Integer, pixelArray As Variant)

' Gray-scale, no color

Dim H As Integer, i As Integer, iMax As Integer, w As Integer
Dim RedVal As Integer, GreenVal As Integer, BlueVal As Integer

iMax = 15
For H = hMax To 0 Step -1
  For w = 0 To wMax - 1
    If pixelArray(H, w) < Int(127 / iMax) Then
      RedVal = 0
      GreenVal = 0
      BlueVal = 0
    End If
    For i = 1 To iMax - 1
      If pixelArray(H, w) > i * Int(127 / iMax) And _
        pixelArray(H, w) <= (i + 1) * Int(127 / iMax) Then
        RedVal = i * Int(255 / iMax)
        GreenVal = i * Int(255 / iMax)
        BlueVal = i * Int(255 / iMax)
      End If
    Next i
    For i = iMax To 2 * iMax - 1
      If pixelArray(H, w) > i * Int(127 / iMax) And _
        pixelArray(H, w) <= (i + 1) * Int(127 / iMax) Then
        RedVal = (2 * iMax - i) * Int(255 / iMax)
        GreenVal = (2 * iMax - i) * Int(255 / iMax)
        BlueVal = (2 * iMax - i) * Int(255 / iMax)
      End If    Next I
    WriteAPixel RedVal Mod 256, GreenVal Mod 256, BlueVal Mod 256
  Next w

' Do not change the following, essential row padding

  w = wMax * 3
  Do While (w Mod 4) <> 0
    WriteAByte 0
    w = w + 1
  Loop
Next H

End Sub
```

(9) You can still use a "1" in the top left-hand corner of the highlighted array to limit the range of input values over which the bands apply.

(10) By carefully selecting your minimum and maximum limits as well as the value of iMax, you can control the range and values at the boundary lines. You can then identify these boundary lines with overlaying numbers on the banded map, by highlighting the plot area, typing the number in the formula window, and using the Enter key to deposit it in the middle of the plot area. Then highlight the deposited number and modify its appearance, and also change the size and position of the surrounding box, to yield a result such as shown in Fig. 8.22.1*f*. Try it!

(11) Finally, when you are done exploring such alternatives, interchange the labels BitMap0 and BitMap00 to return to the original Mapper.

The new Mapper is an elaboration of the above approaches, which you can consult to see what the code for such a finished macro might look like. Using color (or gray) bands rather than a gradual scale is helped by the optical illusion that enhances our perception of contrast between two adjacent areas of different uniform grayness or color.

The main point of this case study is to show how easily you can test a possible extension of existing, open-access code, such as in going from Fig. 8.22.1*a* to 8.22.1*b*, and how, through a series of small improvements, you can then get it to do what *you* want.

9.1 A measure of error, pE (AE3 pp. 399-401)

So far we have dealt mostly with experimental uncertainty and the resulting *imprecision*. However, in this and the following chapters we will often encounter well-defined mathematical functions $f(x)$, which are not subject to the inherent vagaries of experimental variability (i.e., they are *precise*) but which, for a variety of reasons, can still be *inaccurate*.

Before delving into a detailed discussion of likely sources of numerical inaccuracy, which we will postpone until chapter 11, we must first define a useful criterion for the reliability of computed results or, if you will, a measure of their 'number of significant figures'. Several such criteria have been proposed, all relying on reference functions f_{ref} that are supposed to be above reproach, so that we can find the absolute error, $E_{abs} = f - f_{ref}$, the relative error, $E_{rel} = (f - f_{ref})/f_{ref}$ or, more usefully, the absolute values of these, $|E_{abs}| = |f - f_{ref}|$ and $|E_{rel}| = |(f - f_{ref})/f_{ref}|$ respectively.

Usually, the relative error is the more useful quantity, because Excel displays and stores its results as up to 15 decimals, regardless of the number of leading or trailing zeros. Unfortunately, the relative error $|E_{rel}|$ cannot be computed when $f_{ref} = 0$. Here we will therefore use a combined criterion, based on either $|E_{rel}|$ or $|E_{abs}|$, viz.,

$$\begin{aligned} \mathrm{p}E &= -\log|f| && \text{if } f \neq 0,\ f_{ref} = 0 \\ \mathrm{p}E &= -\log\left|\frac{f - f_{ref}}{f_{ref}}\right| && \text{if } f \neq f_{ref},\ f_{ref} \neq 0 \\ \mathrm{p}E &= \mathrm{p}E_{max} && \text{if } f = f_{ref} \end{aligned} \tag{9.1.1}$$

where p denotes the negative ten-based logarithm, as in pH. This definition specifies the magnitude (i.e., the absolute value) of the error in a way that approximates the number of unaffected, significant decimals. Furthermore we will use

$$\mathrm{p}E_{min} = 0 \qquad \mathrm{p}E_{max} = 14 \tag{9.1.2}$$

where a rather conservative definition for $\mathrm{p}E_{max}$ is used because reference data are often rounded from results with higher numerical accuracy, and the rules used for such rounding are not always evident, see section 9.6.4. It can make a difference of ±1 in the least significant digit whether one consistently rounds up (i.e., from 3.5 to 4 and from –3.5 to –3) or down, rounds away from zero (i.e., from 3.5 to 4 and from –3.5 to –4) or towards it, or uses some other rounding scheme, such as bankers' rounding (to the nearest even digit, i.e., round 2.5 down to 2, but round 3.5 up to 4) to avoid systematic bias in the above methods. When the last digit is uncertain to ±1, a 15-digit number starting with 1.001 will have a corresponding relative uncertainty of about 10^{-14}, whereas a 15-digit number starting with 0.999 with the same uncertainty of ±1 in its last digit will have a resulting relative uncertainty of 10^{-15}. Given that a common rounding error of ±1 in the least significant digit can lead to a p*E*-value between 14 and 15, we will consider p*E* = 14 to be as good as can be expected, and usually truncate the p*E* scale accordingly in displaying p*E*-values. Excel will still display a 15th digit, which may be considered a guard digit. The only time we will use the full $\mathrm{p}E_{max} = 15$ is when calibrating algorithms with NIST StRD tests or with equivalent results obtained with extended-precision XN. (The NIST data were obviously obtained with MPFun, the model for XN.)

A p*E*-value of 14 indicates that the first 13 decimals displayed will be reliable, and the 14th will be correct to within ±1; a p*E* of 6.4 that the first six decimals can be trusted; and a p*E* of 0 that none of them can be. In Excel, with its present limit of 15 decimals, little useful information is gained by extending the p*E*

scale below 0 or beyond 14. This range, incidentally, is like that of the pH, which chemists have used since 1909.

We note that pE depends on the error of the function f for a particular set of input parameters. Therefore, the pE of a function of n parameters will itself be a function of those n parameters. Especially for functions of a single parameter, pE is often best displayed graphically, as illustrated in sections 9.2, 11.6 and 11.7. When a single, fixed number characterizing the accuracy of a function or macro over its entire range is required, the most conservative measure would be to list its lowest value, pE_{min}. For many practical applications, however, such a number might scare users away from perfectly acceptable behavior in a more limited range of input parameters. A graphical display of pE as a function of x will then be more informative than a single number, as it allows users to draw their own conclusions.

The concept of pE used here is closely related to that of *LRE* (Logarithm of the Relative Error) introduced by B. D. McCullough in *Am*. *Statist*. 52 (1998) 358. The shorter expression pE explicitly denotes the *negative* ten-based logarithm of (the magnitude of) the *relative or absolute* error.

Other criteria are sometimes useful, such as the performance measure $P(x)$ used by Cook, Cox, Dainton & Harris in their *NPL Report CISE*27/ 99, downloadable from http://www.npl.co.uk/ssfm/ download/ documents/cise27_99.pdf, which highlights how many more significant decimals are lost by a particular function than would be lost by an optimally stable algorithm performing the same task. Such a criterion can be very helpful to designers of algorithms, but has little relevance for most end users, who will often not know (or may not even have access to) information on such an optimal algorithm. The latter is, anyway, a moving target.

The definition (9.1.1) depends on the availability of a reliable set of reference algorithms. Here we will use cases with known derivatives, and in chapter 11 we will use as reference those otherwise well-tested algorithms that, through the use of extended numberlengths, can move their unavoidable truncation errors to digits sufficiently far down to become insignificant. The higher accuracy functions described in sections 11.6 and 11.7 were indeed checked by comparison with their extended numberlength versions, as well as with standard tables. The latter are limited to specific input values, but a set of high-quality comparison values for a large number of scientific functions, for any input value, is available in Jan Myland's Equator software.

Please keep in mind that even a function $f(x)$ of a single variable x will typically cover an infinite number of values, so that testing it for all possible values x would still take an infinite time, an uninviting proposition. Instead, we will sample the function over its applicable range, and will thereby run the risk of missing some peculiar behaviors restricted to small patches in parameter space. This cannot be helped, other than by being vigilant about sampling with higher resolution in suspect areas, such as those near discontinuities and singularities.

Another benefit of a graphical display of pE is that it can highlight major discontinuities, and thereby may identify regions in parameter space where the function as provided might, e.g., cause problems when differentiated numerically. In Excel functions, such discontinuities are sometimes artifacts caused by piecemeal approximations, see Fig. 11.7.1.

*****

As calculators and computers replaced longhand calculations, the concept of *significant digits* (decimals, numbers, or figures) was introduced as a didactic device to indicate the inherent limitations of numerical answers. That measure, however, cannot withstand close scrutiny, and tends to disappear later in the curriculum, to be replaced by more specific estimates, such as standard deviations or confidence limits. Indeed, standard statistical textbooks and NIST standards don't even mention it.

In binary counting, each digit is either right or wrong, and is therefore, unambiguously, significant or not. But in decimal counting this is not the case: 8.3, when known to ± 0.4, is underspecified with one digit, and overspecified with two. This illustrates the problem with significant decimals: it implies an *integer* for answer. But once we realize this implied constraint, and drop it, the solution is obvious: the number of significant digits is then simply pE, which in this example is $-\log(0.4/8.3) = 1.3$ or, if you wish, 1.32, indeed *between* 1 and 2. (It is seldom useful to specify pE to more than one or two decimal places.) This simple generalization of the concept of significant digits can thus rescue its scientific significance.

9.2.9 A general model (for numerical differentiation; AE3 pp. 416-420)

In Excel, numbers without declared dimensions, and those dimensioned "As Double", are represented by one bit for their sign, 52 bits for their mantissa, and 11 bits for their exponent, for a total of 64 bits. As the mantissa ignores leading zeros, the leading bit must be a 1, which is therefore implied. This makes effectively 53 bits available for the mantissa, which consequently ranges from 0 to $2^{53}-1 \approx 9.00720\times10^{15}$. The relative uncertainty ε of these numbers is therefore one in in $2^{53}-1$ or $\pm1/(2^{53}-1) \approx \pm1.110223\times10^{-16}$, the more precise value for the number we called ε in section 9.2.2. Any bits beyond those that can be accommodated in the mantissa are simply ignored, i.e., the binary number is *truncated* to fit the available space. However, in what follows we will use the term *cancellation noise* to distinguish it from the error introduced by truncating Taylor expansions.

To illustrate cancellation noise we will first consider the arguments of the functions $-f_{-1}$ and f_1 in the simplest equation for central differencing, (9.2.4) or (9.2.9), where cancellation noise results from taking the relatively small difference between f_{-1} and f_1.

Lopping off all bits in the mantissa beyond the effectively available 53 results in relative errors that are randomly distributed between 0 and ε. These errors can be described as ε times a *uniform distribution* $U(0, 1)$, with a lower limit a of 0 and an upper limit b of 1 or, equivalently, as a bias of ε/2 plus a uniform distribution ε $U(-½, ½)$. A uniform distribution $U(a, b)$ has a mean value of $(a+b)/2$ and a variance of $(b-a)^2/12$, so that the mean ε/2 of ε $U(0, 1)$ replaces the just-mentioned bias plus the zero mean of ε $U(-½, ½)$. Both ε $U(0, 1)$ and ε $U(-½, ½)$ have a variance of 1/12 and, therefore, a standard deviation of $1/\sqrt{12} \approx 0.2887$.

The numerical computation of a difference such as $f_1 - f_{-1}$ in the numerator of (9.2.4) or (9.2.9) involves three distinct stages: we first generate the arguments $x+\delta$ of f_1 and $x-\delta$ of f_{-1}, then compute their formulas, f_1 and f_{-1}, and finally find their difference, f_1-f_{-1}. In the first stage we replace x by $x_0(1+\varepsilon/2\pm\varepsilon/\sqrt{12})$, and therefore compute the functions $f_1 = f(x+\delta)$ and $f_{-1} = f(x-\delta)$ as

$$f_1 = f(x_0 + \varepsilon x_0/2 \pm \varepsilon x_0/\sqrt{12} + \delta) \approx f_0 + (\varepsilon x_0/2 \pm \varepsilon x_0/\sqrt{12} + \delta) f_0^{\mathrm{I}} \tag{9.2.44}$$

and

$$f_{-1} = f(x_0 + \varepsilon x_0/2 \pm \varepsilon x_0/\sqrt{12} - \delta) \approx f_0 + (\varepsilon x_0/2 \pm \varepsilon x_0/\sqrt{12} - \delta) f_0^{\mathrm{I}} \tag{9.2.45}$$

where we approximate the function by the first two terms of its Taylor series, see (9.2.5).

Before they can be subtracted, these functions must be computed and then temporarily held or stored, at which point they will again be truncated. Inclusion of the corresponding errors will then lead to

$$\begin{aligned} f_1 &\approx (1+\varepsilon/2\pm\varepsilon/\sqrt{12}) f_0 + (\varepsilon x_0/2 \pm \varepsilon x_0/\sqrt{12}+\delta)(1+\varepsilon/2\pm\varepsilon/\sqrt{12}) f_0^{\mathrm{I}} \\ &\approx (1+\varepsilon/2\pm\varepsilon/\sqrt{12}) f_0 + (\varepsilon x_0/2 \pm \varepsilon x_0/\sqrt{12}+\delta) f_0^{\mathrm{I}} \end{aligned} \tag{9.2.46}$$

and a corresponding expression for f_{-1}. Taking their difference yields

$$f_{0\,calc}^{\mathrm{I}} = (f_1-f_{-1})/2\delta \approx f_0^{\mathrm{I}} \pm \varepsilon f_0/2\delta\sqrt{6} \pm \varepsilon x_0 f_0^{\mathrm{I}}/2\delta\sqrt{6} \approx f_0^{\mathrm{I}} \pm (\varepsilon/\delta\sqrt{24})(f_0+x_0 f_0^{\mathrm{I}}) \tag{9.2.47}$$

where the bias terms cancel each other, and the two noise terms come from computing x and $f(x)$ respectively, and are combined by adding their variances, i.e., as the square root of the sum of the squares of their standard deviations, in this case $\sqrt{(1/12+1/12)} = \sqrt{(1/6)} = 1/\sqrt{6}$.

For $j = 5$, see (9.2.12), we likewise find

$$\begin{aligned} f_{0,calc}^{\mathrm{I}} &= f_0^{\mathrm{I}} \pm \frac{(\varepsilon/\sqrt{12})\sqrt{1^2+8^2+8^2+1^2}\,(f_0+x_0 f_0^{\mathrm{I}})}{12\,\delta} \\ &\approx f_0^{\mathrm{I}} \pm 0.27428\ \varepsilon\ \frac{(f_0+x_0 f_0^{\mathrm{I}})}{\delta} \end{aligned} \tag{9.2.48}$$

and, in general, $f_{0\,calc}^{\mathrm{I}} = f_0^{\mathrm{I}} \pm c_j\,(f_0+x_0 f_0^{\mathrm{I}})/\delta$. Dependent on the nature of the function f_0 and on the magnitude of x_0, one of the two error terms in $(f_0+x_0 f_0^{\mathrm{I}})$ may be dominant.

It may be useful to consider the *magnitude* (i.e., absolute value) of the *relative* errors. For a value a, with an error Δ, the relative error is Δ/a, and its absolute value is $|\Delta/a|$, which can therefore be formulated for the expressions in Table 9.2.4 as

$$E_{canc} = \frac{c_j}{\delta} \left| \frac{f_0 + x_0 f_0^{\mathrm{I}}}{f_0^{\mathrm{I}}} \right| \tag{9.2.49}$$

Some values of c_j are listed in Table 9.2.6, as calculated from the integers that multiply the terms $f_{\pm i}$ in the equations of Table 9.2.4. Table 9.2.6 also contains some related coefficients.

j	b_j	c_j	d_j	$\varepsilon^{1/j}$
3	1.666667×10^{-1}	2.266233×10^{-17}	4.081395×10^{-6}	4.8062174×10^{-6}
5	3.333333×10^{-2}	3.045159×10^{-17}	7.442764×10^{-4}	6.4429097×10^{-4}
7	7.142857×10^{-3}	3.467493×10^{-17}	6.982307×10^{-3}	5.2574226×10^{-3}
9	1.587502×10^{-3}	3.741589×10^{-17}	2.429285×10^{-2}	1.6875933×10^{-2}
11	3.607504×10^{-4}	3.937709×10^{-17}	5.379360×10^{-2}	3.5447266×10^{-2}
13	8.325008×10^{-5}	4.086932×10^{-17}	9.335571×10^{-2}	5.9254845×10^{-2}
15	1.942502×10^{-5}	4.205382×10^{-17}	1.399440×10^{-1}	8.6369555×10^{-2}

Table 9.2.6: Approximate numerical values of the coefficients b_j, c_j, and d_j, in equations (9.2.49), (9.2.50), and (9.2.54) for the compact central differencing expressions of the first derivative f_0^{I} listed in Table 9.2.4 for $\varepsilon = 1/(2^{53}-1) \approx 1.110223\times10^{-16}$. The last column lists the values of $\varepsilon^{1/j}$.

In numerical differentiation by compact central differencing, we have control over only one parameter, the step size δ. How do we choose it so that our answer has the best possible accuracy? We will here focus on the first derivative, using the expressions listed in Table 9.2.4. Our answers will therefore contain both the cancellation noise of (9.2.49) and the systematic errors that can be summarized from section 9.2.5 as

$$E_{syst} = b_j \, \delta^{j-1} \left| f_0^{\mathrm{J}} / f_0^{\mathrm{I}} \right| \tag{9.2.50}$$

where b_j is the absolute value of the coefficient of the term $f_0^{\mathrm{J}} \delta^{j-1}$ on the right-hand side of Table 9.2.4 for a given value of j; $b_3 = 1/3!$, $b_5 = 4/5!$, $b_7 = 36/7!$, etc. In total we then have

$$E_{total} = E_{syst} + E_{canc} = b_j \, \delta^{j-1} \left| \frac{f_0^{\mathrm{J}}}{f_0^{\mathrm{I}}} \right| + \frac{c_j}{\delta} \left| \frac{f_0 + x_0 f_0^{\mathrm{I}}}{f_0^{\mathrm{I}}} \right| \tag{9.2.51}$$

where we again use the absolute values to ensure that the logarithms and/or roots can always be computed, regardless of the signs of f_0, f_0^{I}, and f_0^{J}. After all, we are only interested in minimizing the absolute magnitude of the total error, which at the optimal value δ_{opt} of δ follows from (9.2.51) as

$$\frac{d\,E_{total}}{d\,\delta} = (j-1)\, b_j \, \delta_{opt}^{j-2} \left| \frac{f_0^{\mathrm{J}}}{f_0^{\mathrm{I}}} \right| - \frac{c_j}{\delta_{opt}^2} \left| \frac{f_0 + x_0 f_0^{\mathrm{I}}}{f_0^{\mathrm{I}}} \right| = 0 \tag{9.2.52}$$

so that

$$\delta_{opt} = \left(\frac{c_j}{(j-1)\, b_j} \right)^{1/j} \left| \frac{f_0 + x_0 f_0^{\mathrm{I}}}{f_0^{\mathrm{J}}} \right|^{1/j} = d_j \left| \frac{f_0 + x_0 f_0^{\mathrm{I}}}{f_0^{\mathrm{J}}} \right|^{1/j} \tag{9.2.53}$$

with

$$d_j = \left(\frac{c_j}{(j-1)\, b_j} \right)^{1/j} \tag{9.2.54}$$

for which Table 9.2.6 includes some values.

Figure 9.2.6 illustrates the above relations with the function $f(x) = x^{20}$ for $x_0 = 1.234$, by varying the magnitude of δ over eight decades, from $\delta = 10^{-3}$ to $\delta = 10^{-11}$. In Fig. 9.2.7 we display equivalent results for two different values of x_0, one negative and one positive, as well as of different magnitudes, and in Fig. 9.2.8 for two larger values of j.

In Fig. 9.2.6 the asymptotes are well represented by the lines predicted by (9.2.49) and (9.2.50), while the maximum in pE is indeed found near the value δ_{opt} given by (9.2.53) as indicated by an arrow.

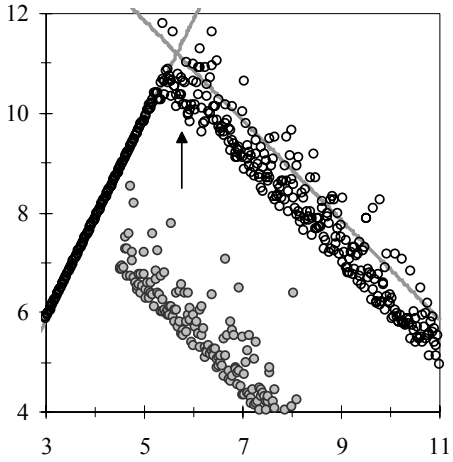


Fig. 9.2.6: Plot of pE vs. pδ for $f(x) = x^{20}$, and its numerical analysis for $j = 3$ with $x_0 = 1.234$ (open black circles). The gray straight line with slope –1 through the data is computed with (9.2.49), and the gray line with slope +2 with (9.2.50). The vertical arrow drawn at pδ = $p\delta_{opt}$ is calculated with (9.2.53). The gray-filled circles near the center-bottom represent uniform noise $U(-½, ½)$ added to a baseline with slope –1.

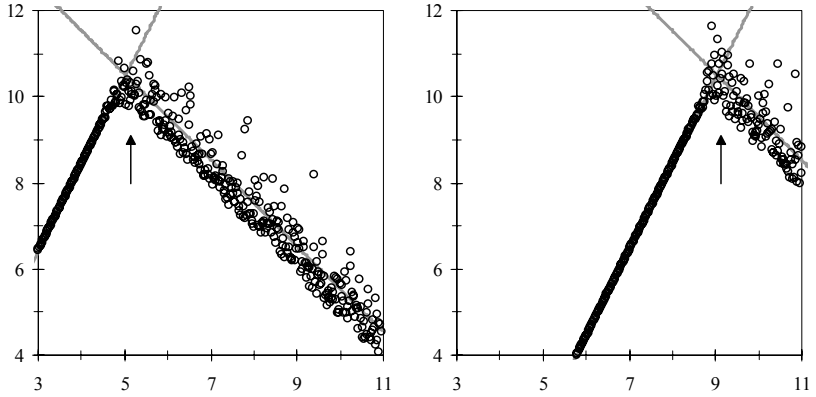


Fig. 9.2.7: Plot of pE vs. pδ for $f(x) = x^{20}$, and its numerical analysis for $j = 3$ with $x_0 = –12.34$ (left panel) and 0.001234 (right panel). The gray straight lines with slope –1 are based on (9.2.49), those with slope +2 on (9.2.50), and the vertical arrows are drawn for $p\delta_{opt}$ as found from (9.2.53).

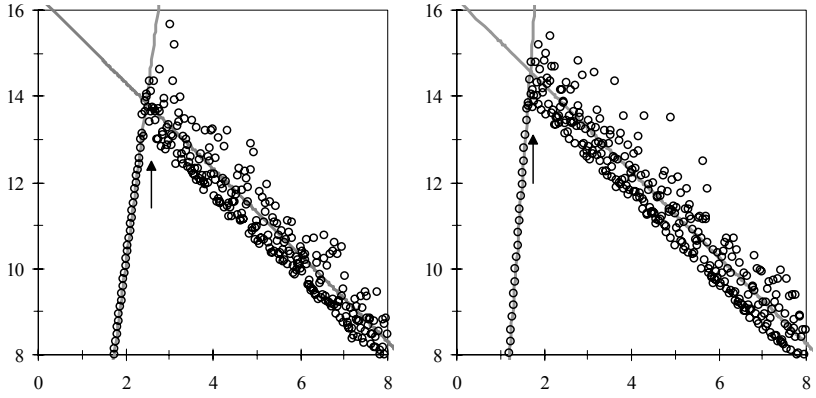


Fig. 9.2.8: Plot of pE vs. pδ for $f(x) = x^{20}$ with $x_0 = 1.234$, for $j = 9$ (left panel) and 15 (right panel). The gray straight lines with slope –1 are based on (9.2.49), those with slope +2 on (9.2.50), and the vertical arrows are drawn for $p\delta_{opt}$ as found from (9.2.53). Note that the pE and pδ scales are both shifted with respect to those in Figs. 9.2.6 and 9.2.7.

Different behavior can be expected depending on whether f_0 is larger or smaller than $x_0 f_0^{\mathrm{I}}$. We can see this with the exponential function $f(x) = e^x$, where $f_0^{\mathrm{I}} = f_0$, where merely changing x_0 from $x_0 \gg 1$ to $x_0 \ll 1$ can serve as our test. We illustrate in Figs. 9.2.9 and 9.2.10 that this model prediction indeed holds. For $x_0 \gg 1$, the location of the right-hand asymptote and the value of $\mathrm{p}\delta_{opt}$ change with x_0 as $-(1/j)\,\log(x_0)$, whereas both are essentially constant for $x_0 \ll 1$, confirming the presence of two different stages where cancellation errors can occur in central differencing, each with its own response.

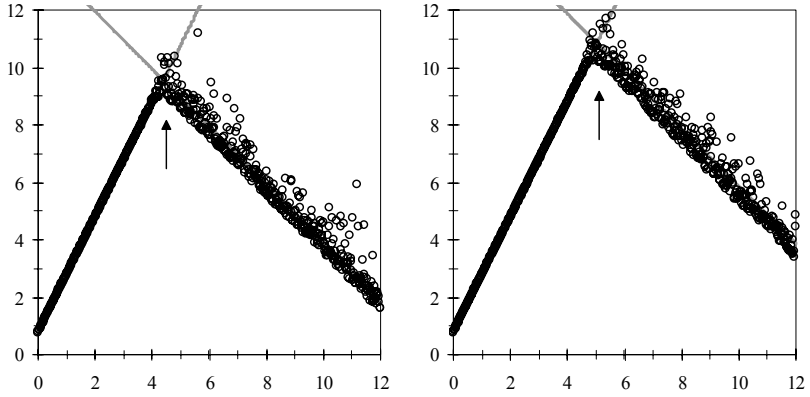


Fig. 9.2.9: Plot of pE vs. pδ for $f(x) = e^x$, and its numerical analysis for $j = 3$ with $x_0 = 500$ (left panel) and 5 (right panel). The gray straight lines are the model asymptotes, the vertical arrow is positioned at $\mathrm{p}\delta_{opt}$.

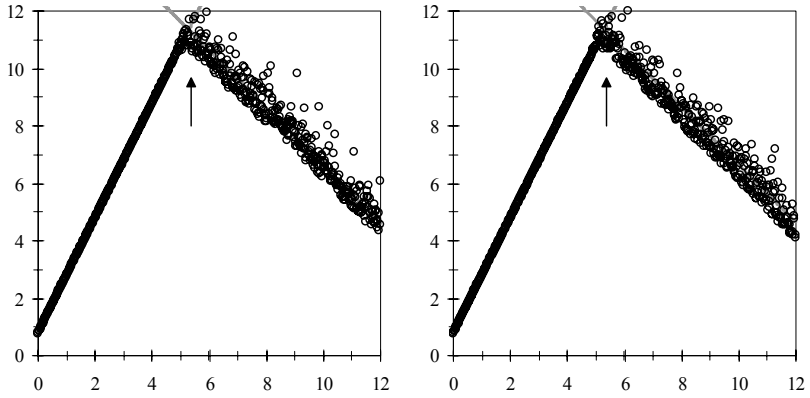


Fig. 9.2.10: Plot of pE vs. pδ for $f(x) = e^x$, and its numerical analysis for $j = 3$ with $x_0 = 0.05$ (left panel) and 0.0005 (right panel). The gray straight lines are the model asymptotes, the vertical arrow is positioned at $\mathrm{p}\delta_{opt}$.

9.2.10 Implementation (AE3 pp. 421-423)

How can we use the above to compute the first derivative when, according to (9.2.53), the crucial parameter, the step size δ_{opt}, not only requires the function f_0, but also its sought first derivative f_0^{I}, and even the higher derivative f_0^{J}? We solve this riddle by realizing that we can use *approximate* values for f_0^{I} and f_0^{J}, based on a first estimate, such as $\delta \approx |x|\,\varepsilon^{1/j}$, and follow where that leads us. This yields the following iterative algorithm:

(1) read the number j of equidistant samples (an odd integer), the function $f(x)$, and the particular value $x = x_0$ for which the derivative is sought;
(2) use $\delta \approx |x_0|\,\varepsilon^{1/j}$ to estimate the step spacing, where $\varepsilon = 2^{-52}$ for Excel's double precision;
(3) sample the function in j equidistant steps from $x = x_0 - (j+1)\,\delta/2$ to $x_0 + (j+1)\,\delta/2$;
(4) use these samples to estimate both f_0^{I} and f_0^{J};
(5) now estimate δ_{opt} with (9.2.53); and finally
(6) use this estimate for δ_{opt} to compute an improved value for f_0^{I}.

In a single iteration, this algorithm produces quite satisfactory results when tested with simple functions for which the first derivative f_0^{I} is known, as in Fig. 9.2.11, where we find $\mathrm{p}E \geq 9.5$ for $j = 3$, $\mathrm{p}E \geq 11$ for $j = 5$, $\mathrm{p}E \geq 12$ for $j = 7$, and $\mathrm{p}E \approx 13$ for $j = 9$. While there is no guarantee that such good values

will be observed with other functions or x-values, these are certainly encouraging results. This approach is implemented in the macro Deriv1 of the MacroBundle, and is more fully described in *J. Chem. Sci* 121 (2009) 935-950, see http://www.ias.ac.in/chemsci/Pdf-Sep2009/671.pdf. No significant further improvement is obtained with further iterations.

As can be seen in Figs. 9.2.7 through 9.2.11, random noise limits a more accurate single determination, but averaging over a small range of neighboring δ-values might yield a more precise result. However, a tenfold increase in noise reduction would require 100 repeats. It is therefore much easier to use XN's extended precision to obtain similar or better answers.

Since we here consider accuracies beyond the first six or seven digits, Fig. 9.2.11 does not display the values of the derivatives themselves, but instead their pE-values, i.e., their number of significant figures, using as reference their algebraic first derivatives as evaluated on Excel. In a few cases we had to deviate from this: for the error function erf(x) we instead used the function cErf(x) of section 11.8, but let Excel evaluate its analytical derivative, $(2/\sqrt{\pi})\exp(-x^2)$ for computing pE. For the Bessel functions, which in Excel appear to use single-precision algorithms, we used the corresponding double-precision functions from XN.xla for both the function $f(x_0)$ to be differentiated and as reference values $f^{\mathrm{I}}(x_0)_{ref}$ for determining pE. Comparison of Excel's $J_0(x)$, $J_1(x)$, $Y_0(x)$, and $Y_1(x)$ with their numerical values as listed in, e.g., the *Handbook of Mathematical Functions*, M. Abramowitz & I. A. Stegun, eds., Dover (1968) pp. 390-391, confirms the low accuracy of Excel's Bessel functions, which were part of the Data Analysis Toolkit that was fully integrated into Excel 07. It is beyond my comprehension why Microsoft in 2007, when they could (and should) have exploited the *quadruple* precision option made possible by the wide availability of x64 cpu's, decided to incorporate their old *single*-precision algorithms instead.

We see that the approximation $\delta \approx x_0\,\varepsilon^{1/j}$ is equivalent to approximating d_j by $\varepsilon^{1/j}$, and $x_0 f_0^{\mathrm{I}}/f_0^{\mathrm{J}}$ by x_0. This is appropriate for $|f_0| \ll |x_0 f_0^{\mathrm{I}}|$, while the alternative $\delta \approx \varepsilon^{1/j}$ is more appropriate when $|f_0| \gg |x_0 f_0^{\mathrm{I}}|$. At a practical level, in a double-precision environment, one should preferably use j = 5, 7, or 9 (for the most accurate results) when the sampling range or footprint $\pm(j-1)\,\delta/2$ is not important, and j = 3 when it is, choices that Deriv1 leaves to the user. In double precision, there appears to be no good reason to use j-values higher than 9.

Differentiation obviously requires special care when the function to be differentiated has discontinuities and/or singularities. Discontinuities occur in nature with, e.g., phase transitions. We can also encounter them with functions, especially with those computed using piecemeal approximations, as illustrated in Fig. 11.7.1. In the case of discontinuities, the function value changes suddenly, and so may its derivative, and it is therefore crucial that all data used in computing the derivative are taken on the same side of that discontinuity.

Exercise 9.2.5:

(1) To illustrate the use of Deriv1, on a new spreadsheet place labels for x, $f(x)$, $f'(x)$, and $f''(x)$ in cells B2:B5. Next to these, place a value for x in C2, and in C3, next to the label $f(x)$, deposit an explicit, *quasi-algebraic* function $f(x)$, i.e., written in the Excel-like parser formalism described in the Math Formula String section of the help file, and as text, i.e., *not* preceded by an equal sign, see Fig. 9.2.12. In this example, a truly algebraic expression would read x^2 instead of x^2. You can precede the formula by a (non-showing) apostrophe. (Such an apostrophe is not needed here, since Excel interprets this as text, but is a still good general practice, because it would be required if the expression were to start with, e.g., a minus sign.) Verify that C4 indeed yields the first derivative $f'(x) = [(x^2+1)-2x(x+3)]/(x^2+1)^2$ of the function $f(x) = (x+3)/(x^2+1)$ in cell C3 for the x-value in C2.

(2) In cell C4, next to the label for $f'(x)$, insert the instruction `=Diff1(C2,C3)`, which will yield $f'(x)$, the first derivative of the mathematical expression for $f(x)$ in C3 for the x-value in C2.

In such cases we may have to use the asymmetric expressions for lateral differencing. The approach is the same as for central differencing, but the pE-values will be somewhat lower. Incidentally, please keep in mind that a function at its discontinuity has two values, and is often represented on a spreadsheet by their average, but such an average value should *not* be used in asymmetric differencing.

Singularities are rather rare in nature, but are quite common in math. Consequently, our necessarily simplified models of natural phenomena may well include such singularities.

The approach used in this section is quite general, but has here been limited to equidistant data, because these yield simpler solutions. When f is a function of multiple variables, as in $f(x, y, z, \ldots)$, the above method yields estimates of the *partial* derivative, $\partial f/\partial x$.

x_0	$f(x)$	$f^{\mathrm{I}}(x)$	pE at j =						
			3	5	7	9	11	13	15
1.234	$4-3x+2x^2-x^3 = 1.46443110$	$-3+4x-3x^2 = -2.63226800$	10.62	14.16	13.82	14.36	14.52	15.77	15.47
1.234	$x^{20} = 67.0352439$	$20x^{19} = 1086.47073$	10.00	11.89	12.65	13.83	13.70	15.20	14.73
1.234	$x^{-20} = 0.01491753$	$-20x^{-21} = -0.24177514$	9.87	11.69	12.62	13.05	13.31	13.74	13.65
1.234	$x^{1/20} = 1.01056850$	$0.05x^{-19/20} = 0.04094686$	10.47	12.20	13.06	13.85	12.84	13.07	14.31
1.234	$x^{-1/20} = 0.98954202$	$-0.05x^{-21/20} = -0.04009490$	10.35	11.40	12.59	13.10	12.95	13.60	13.28
1.234	$\ln(x) = 0.21026093$	$1/x = 0.81037277$	11.70	13.69	13.98	13.72	14.86	14.12	15.86
1.234	$\exp(x) = 3.43494186$	$\exp(x) = 3.43494186$	10.78	12.99	14.01	14.93	14.71	14.47	15.04
1.234	$\sin(x) = 0.94381821$	$\cos(x) = 0.33046511$	10.97	12.93	13.96	14.08	15.77	14.50	15.77
1.234	$\sinh(x) = 1.57190806$	$\cosh(x) = 1.86303380$	11.13	13.01	15.02	14.29	14.53	14.56	14.88
0.567	$\arcsin(x) = 0.60285925$	$1/\sqrt{(1-x^2)} = 1.21400801$	11.52	12.73	14.01	14.23	14.59		
1.234	$\operatorname{arsinh}(x) = 1.03755875$	$1/\sqrt{(1+x^2)} = 0.62959660$	10.86	12.98	13.45	14.01	15.45	14.50	14.16
1.234	$\tan(x) = 2.85602984$	$1+\tan^2(x) = 9.15690644$	10.38	13.51	12.78	13.79	13.33	14.37	13.61
1.234	$\tanh(x) = 0.84373566$	$1-\tanh^2(x) = 0.28811013$	10.69	12.36	13.18	13.32	14.04	14.14	13.81
1.234	$\arctan(x) = 0.88976245$	$1/(1-x^2) = 0.39639188$	11.47	13.05	13.45	13.77	14.95	14.90	14.51
0.567	$\operatorname{artanh}(x) = 0.64309026$	$1/(1+x^2) = 1.47381546$	11.76	12.89	13.02	13.89	14.00	14.02	
1.234	$\operatorname{erf}(x) = 0.91903942$	$(2/\sqrt{\pi})\exp(-x^2) = 0.24611072$	10.25	12.10	12.62	13.36	13.07	13.52	13.78
1.234	$I_0(x) = 1.41848958$	$I_1(x) = 0.742135021$	11.39	12.58	13.20	13.85	15.13	14.38	14.32
1.234	$J_0(x) = 0.65404541$	$-J_1(x) = -0.50677701$	10.70	13.22	14.01	14.05	14.10	14.62	14.38
1.234	$K_0(x) = 0.30411714$	$-K_1(x) = -0.41218265$	10.45	12.75	12.86	13.60	14.83	14.29	14.33
1.234	$Y_0(x) = 0.24877388$	$-Y_1(x) = 0.596023514$	10.91	12.53	13.62	14.58	15.73	14.30	14.33

Fig. 9.2.11: The accuracy of the first derivative $f_{0\,calc}^{\mathrm{I}}$ obtained with Deriv1 for various values of j, for a number of simple functions $f(x)$ at $x = x_0$ for which the first derivative f_0^{I} is known. Because the values of $f_{0\,calc}^{\mathrm{I}}$ differ only in their less significant digits, we only show their pE-values.

Exercise 9.2.5:

(1) To illustrate the use of Deriv1, on a new spreadsheet place labels for x, $f(x)$, $f'(x)$, and $f''(x)$ in cells B2:B5. Next to these, place a value for x in C2, and in C3, next to the label $f(x)$, deposit an explicit, *quasi-algebraic* function $f(x)$, i.e., written in the Excel-like parser formalism described in the Math Formula String section of the help file, and as text, i.e., *not* preceded by an equal sign, see Fig. 9.2.12. In this example, a truly algebraic expression would read x^2 instead of x^2. You can precede the formula by a (non-showing) apostrophe. (Such an apostrophe is not needed here, since Excel interprets this as text, but is a still good general practice, because it would be required if the expression were to start with, e.g., a minus sign.) Verify that C4 indeed yields the first derivative $f'(x) = [(x^2+1)–2x(x+3)]/(x^2+1)^2$ of the function $f(x) = (x+3)/(x^2+1)$ in cell C3 for the x-value in C2.

(2) In cell C4, next to the label for $f'(x)$, insert the instruction `=Diff1(C2,C3)`, which will yield $f'(x)$, the first derivative of the mathematical expression for $f(x)$ in C3 for the x-value in C2.

In such cases we may have to use the asymmetric expressions for lateral differencing. The approach is the same as for central differencing, but the pE-values will be somewhat lower. Incidentally, please keep in mind that a function at its discontinuity has two values, and is often represented on a spreadsheet by their average, but such an average value should *not* be used in asymmetric differencing.

Singularities are rather rare in nature, but are quite common in math. Consequently, our necessarily simplified models of natural phenomena may well include such singularities.

The approach used in this section is quite general, but has here been limited to equidistant data, because these yield simpler solutions. When f is a function of multiple variables, as in $f(x, y, z, \ldots)$, the above method yields estimates of the *partial* derivative, $\partial f/\partial x$.

*****

In this section we have seen how, starting from the Taylor expansion, and with a little matrix algebra, we can find useful approximations to compute derivatives. At this point you may well ask why, in this book about scientific data analysis, so much space is devoted to differentiating *theoretical* expressions. One reason is to illustrate that mathematical conditions such as $\lim \delta \to 0$ do not translate as such into computer routines, because additional error sources can get involved, as cancellation errors do with differentiation. Another is the interesting interplay of systematic and random-like errors, i.e., of aspects resembling inaccuracy and imprecision. Moreover, we see that problems that are relatively straightforward in formal mathematics may sometimes be complicated numerically, while integration illustrates the opposite, see section 9.4.

But the most compelling reason for our emphasis on differentiation is that it is involved in virtually all optimization methods. Moreover, while most mathematical functions can be differentiated formally, in practice this can be quite cumbersome, and often requires the numerical evaluation of special functions anyway. Consequently, in a spreadsheet environment, a numerical differentiation is almost always more convenient. On the other hand, non-optimal numerical differentiation can limit the ultimate reliability of an algorithm. For instance, Excel's Solver only offers its users the choice between the simplest forms of forward and central differencing, i.e., between (9.2.2) and (9.2.4). Not surprisingly, when tested with demanding NIST reference data sets such as MGH10 or Bennett5, it only gets the first two decimals consistently right! As a result of such experiences, it is often stated that optimization methods need analytical derivatives because they are the only ones that have the required high accuracy. With the approach outlined here, and implemented in Deriv1, that argument is no longer so compelling.

10.10 Matrix inversion, once more (AE3 pp. 485-489)

Matrix inversion provides an elegant, formal way to solve many problems. However, not all square matrices can be inverted; those that cannot be inverted are called *singular*. Mathematically, there is a sharp distinction between singular and nonsingular matrices: a singular matrix has a determinant equal to zero, whereas a non-singular matrix doesn't. Excel has the corresponding instruction MDETERM. But such a seemingly well-defined difference can become rather fuzzy when matrices are evaluated on a computer with binary math, which cannot represent most fractional decimal numbers exactly. How should we interpret a finding that the determinant of a square matrix is about 10^{-17}, or 10^{-40}? Those values are not per se small for Excel, which can represent numbers smaller than 10^{-300}. But how do we know whether 10^{-17} or 10^{-40} is significantly different from 0, or is just a zero disguised by round-off errors?

In least squares analysis, only the matrix $\mathbf{X}$ is inverted, or a function thereof, such as $\mathbf{X}^T\,\mathbf{X}$ in (10.5.6), where $\mathbf{X}$ is usually assumed to be noise-free. Uncertainties in inverting such a matrix then have nothing to do with experimental noise, which would only appear in $\mathbf{y}$, but must be the result of algorithmic inaccuracy and/or numerical imprecision in the matrix algebra used.

As our illustration of the problem, we will use an example taken from Meyer's *Matrix Analysis and Applied Linear Algebra*, SIAM (2000) pp. 33 and 128. Consider the two simultaneous equations

$$\begin{aligned} 835\,x_1 + 667\,x_2 &= 168 \\ 333\,x_1 + 266\,x_2 &= \;\;67 \end{aligned} \qquad (10.10.1)$$

which have the solution

$$x_1 = 1; \qquad x_2 = -1 \qquad (10.10.2)$$

as you can readily verify by substituting these values for x_1 and x_2 back into (10.10.1).

Now let there be a small amount of noise in the constant on the right-hand side of the first expression in (10.10.1), changing its value from 168 to, say, 167. The solution then becomes

$$x_1 = 267; \qquad x_2 = -334 \qquad (10.10.3)$$

Likewise, replacing 67 in the second expression of (10.10.1) by 66 will modify the answer to

$$x_1 = -666; \qquad x_2 = 834 \qquad (10.10.4)$$

while changing 168 to 169 and, simultaneously, 67 to 66, yields

$$x_1 = -932; \qquad x_2 = 1167 \qquad (10.10.5)$$

When we write (10.10.1) in matrix form as

$$\mathbf{A}\,\mathbf{x} = \mathbf{b} \tag{10.10.6}$$

with

$$\mathbf{A} = \begin{bmatrix} 835 & 667 \\ 333 & 266 \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 168 \\ 67 \end{bmatrix} \tag{10.10.7}$$

we see that (10.10.1) through (10.10.5) show quite different answers for the coefficients x_1 and x_2 of **x** as the result of rather minor changes in b_1 and/or b_2.

Exercise 10.10.1:

(1) In rows 3 and 4 of a new spreadsheet, enter the numerical values of **A** of (10.10.6) in columns B and C, and the corresponding values of **b** in column E. Use row 2 for appropriate labels.

(2) Verify that (10.10.2) is, indeed, the solution of (10.10.1), by computing **x** in G3:G4 with the matrix instruction `=MMULT(MINVERSE(B3:C4),E3:E4)` because $\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{A}\,\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b}$.

(3) In cell B6 place the instruction `=$B$3`, put `=$B$4` in B7, `=$C$3` in C3, `=$C$4` in C7, `=$E$4` in E7, but `=$E$3-1` in E6. Then copy G3:G4 to G6. You should now find the result of (10.10.3).

(4) Copy B6:G7 to B8, then replace the minus sign in E8 by a plus sign. This will yield x_1 = -265 and x_2 = +332, the same distance 266 away from x_1 = 1and –333 x_2 = –1 but in the opposite direction. These are large effects in **x** for small changes in **b**, but they are symmetrical.

(5) To clearly offset B8:G9 from B6:G7, you might want to put some light background color in B8:C9, E8:E9, and G8:G9, and/or put frames around the areas containing the matrix **A** and the vectors **b** and **x**.

(6) Copy B6:G9 to B10, and then change the terms 1 in B10 and B12 to 10. The result is again a linear response to the change, now by ten times larger amounts. The spreadsheet you have made so far should resemble that in Fig. 10.10.1.

(7) When you get tired of this game, change the approach to use this spreadsheet more efficiently, as follows.

(8) Go to Tools ⇨ Data Analysis, select Random Number Generation, then set Distribution: Normal, Mean = 0, Standard Deviation =: 1, click on the round "radio button" to the left of Output Range:, then click on the associated window, enter I6:K105, and click OK. This will fill I6:K105 with random Gaussian numbers, our usual approximation of random noise.

(9) In cell I3 place a label for a noise amplitude, and in J3 a value, such as 1E-7.

(10) Modify the instruction in E6 to `=$E$3*(1+$J$3*K6)`, and that in E7 likewise to `=$E$4*(1+$J$3*K7)`.

(11) Copy B6:G7 to B8, repair any cell coloring and framing, then copy B6:G9 to B10, B14, B18, B22, … B102.

	A	B	C	D	E	F	G
1							
2			**A**		**b**		**x**
3		835	667		168		1
4		333	266		67		-1
5							
6		835	667		167		267
7		333	266		67		-334
8		835	667		169		-265
9		333	266		67		332
10		835	667		158		2661
11		333	266		67		-3331
12		835	667		178		-2659
13		333	266		67		3329

Fig. 10.10.1: A spreadsheet illustrating various solutions **x** = $\mathbf{A}^{-1}\,\mathbf{b}$ obtained with a near-singular matrix **A** by varying b_1.

(12) In order to plot the results, in M4 place labels for x_1, and in N4 for x_2, then place the instruction `=G6` in M6, and `=G7` in N6. Highlight M6:N:7, and copy it to M8, then copy M6:N9 to M10, and so on, to fill the entire column M6:N133. Delete M106:N133.

(13) It is useful to place the instruction `=MAX(M6:M105)` in cell M2, and then copy it to cell N2, where it will read `=MAX(N6:N105)`. Similarly, place the instruction =MIN(M6:M105) in cell M3, then copy it to cell O3. This will help you see what axis scales are needed to show all data.

(14) Highlight M6:N105 and plot x_2 vs. x_1. Because you cannot exactly reproduce random noise, you will get some other data than shown here, but their general behavior will be the same.

(15) Repeat with other values for the relative noise amplitude. Two such examples are shown in Fig. 10.10.2.

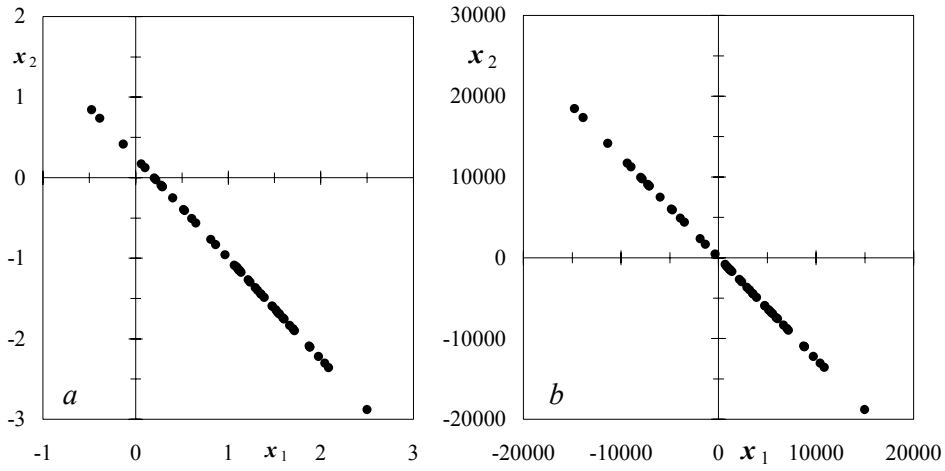


Fig. 10.10.2: Results obtained with the same spreadsheet after modification to compute and display the effect of noise in **b** on the computation. Relative noise amplitude in panel *a*: 1E–5, in panel *b*: 1E–1.

The data shown in Fig. 10.10.2 illustrate that the input noise added to **b** results in output noise in **x** that is directly proportional to its amplitude, i.e., the response of the matrix operation is linear. That is also to be expected: the inversion of **A** is unaffected by the noise in **x**, while the multiplication of $\mathbf{A}^{-1}$ and **x** is a strictly linear operation.

Incidentally, the points in the two plots lie on a straight line through $x_1 = 1$, $x_2 = -1$, with slope –1.251877, a number that is the average of the slopes mentioned later in this section, and their distribution along that line is indeed Gaussian. Each point on that line in panel *b* is exactly 10000 times more widely spaced on the line through (1, –1) than it is in panel *a*, just as the relative noise amplitude is also 10000 times larger.

Now we ask a different question: what will happen when we add such noise to the elements of **A**? Exercise 10.10.2 considers that issue, using the very same spreadsheet you have just made.

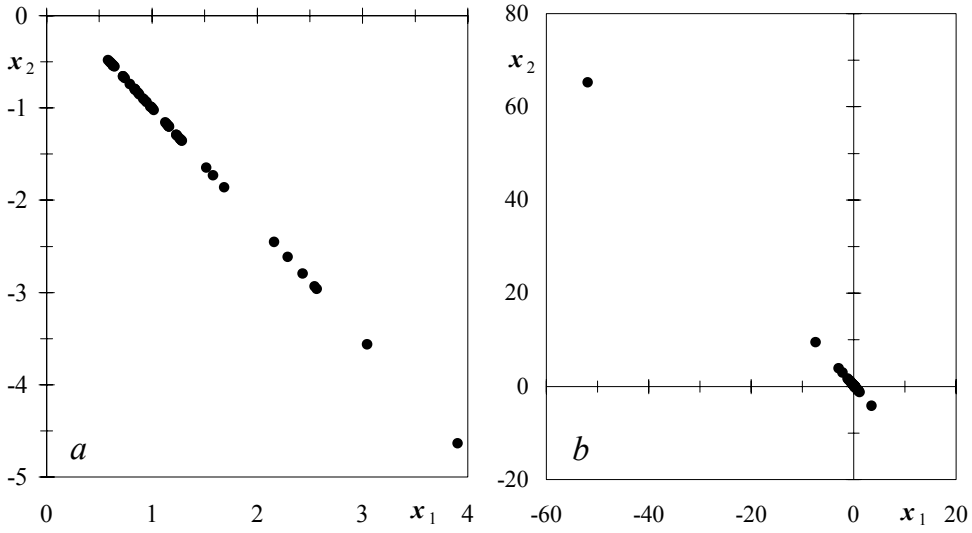


Fig. 10.10.2: The same spreadsheet after modification to compute and display the effect of noise in **A** on the computation. Same set of noise data, but with different relative amplitudes: 1E–6 in panel *a*, 1E–5 in panel *b*.

Exercise 10.10.2:

(1) Copy the spreadsheet of Exercise 10.10.1 into a new sheet. Then make the following changes.

(2) Replace the instruction in B6 by `=$B$3*(1+$J$3*I6)`, that in B7 by `=$B$4*(1+$J$3*I7)`, and likewise place `=$C$3*(1+$J$3*J6)` in C6, and `=$C$4*(1+$J$3*J7)` in C7. Moreover, to simplify matters, set E6 back to `=$E$3`, and E7 to `=$E$4`.

(3) Verify that, for a relative noise amplitude of 0, you get everywhere the same answers, identical to those in G3:G4. If not, check your equations.

(4) Now try various sets of random numbers at a given relative noise amplitude, and then experiment with different relative noise amplitudes. You will, again, find different answers than shown in Fig. 10.10.3, because your specific noise values will differ, but you will also get the general point: the effect of noise is now highly *nonlinear*. And the results look decidedly non-Gaussian.

What causes this great sensitivity of **x** to relatively small fluctuations in **A**? To find the answer, it is helpful to step away from the algebra, and to look instead at a physical interpretation of expressions such as (10.10.1), which we can consider as representing two straight lines, say,

$$x_2 = \frac{168}{667} - \frac{835}{667}x_1 \qquad \text{or, in general,} \qquad x_2 = \frac{b_1}{a_{12}} - \frac{a_{11}}{a_{12}}x_1 \tag{10.10.8}$$

and

$$x_2 = \frac{67}{266} - \frac{333}{266}x_1 \qquad \text{or} \qquad x_2 = \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}}x_1 \tag{10.10.9}$$

where the solution of these simultaneous equations defines the values of x_1 and x_2 at their intersection.

Look at the slopes of these lines, a_{11}/a_{12} and a_{21}/a_{22}. Their numerical values are 835/667 ≈ **1.2518**74 and 333/266 ≈ **1.2518**80 respectively, where we have used bold-facing to emphasize their common digits. These lines therefore have *near-identical slopes*. Consequently, the place where these two lines intersect is extremely sensitive to small changes in their intercepts, and therefore to changes in the coefficients b_i, but the real culprits are their almost identical slopes, a_{11}/a_{12} and a_{21}/a_{22} respectively, which these intercepts b_i do not affect, see (10.10.8) and (10.10.9). The basic problem, therefore, lies in the matrix **A** rather than in the vector **b**, as exercises 10.10.1 and 10.10.2 confirmed.

If the slopes a_{11}/a_{12} and a_{21}/a_{22} were identical, the matrix **A** would be singular, and we would not find any answer, because parallel lines have no (finite) intersection, but only intersect at "infinity". In our case, **A** is merely *very close* to singular. In the next section we will therefore consider how we can characterize matrices in terms of their near-singularity or *conditioning*. With a matrix that is almost singular, it is merely a matter of chance how added noise may occasionally bring it very close to that abyss of singularity. In fact, it is not a single singularity, but an infinite collection of them, because for every ratio a_{11}/a_{12} there will be a corresponding ratio a_{21}/a_{22} that will make the second line parallel to the first, and therefore create a singularity! And the closer one gets to a singularity, the more even very small fluctuations tend to become magnified to noticeable levels. Such a highly *nonlinear* effect in general will *not* yield a Gaussian output for a Gaussian input.

A prominent feature of Fig. 10.10.2*b* is what one might consider an outlier. (If you didn't find any, return to Tools ⇨ Data Analysis ⇨ Random Number Generation, and click OK; this will update the computations with a new set of Gaussian numbers, and also update the graph. You will soon, usually within a few tries, find some seemingly extreme points.) Of course, these simulations merely illustrate that, upon inverting a near-singular matrix, Gaussian noise *affecting the matrix elements* does not produce an outcome with a Gaussian distribution. If you were to make that latter (faulty) assumption, you might be tempted to reject the outliers because, were they Gaussian, they would be highly unlikely events indeed. But in this case there is no good reason for doing so, other than that such a point might not fit your (incorrect) expectation of a linear operation, with a consequently Gaussian output.

The above, geometric argument is convenient for 2×2 matrices. For larger matrices it is more difficult to visualize matters, although it can still be done for 3×3 matrices. There, the general intersection between two planar surfaces is a line, and this line in general intersects with the third plane at a point. The three space coordinates of that common point are the elements x_1 through x_3 of the solution, **x**. When at least two of these planes have a similar *tilt* (as, for lack of a better term, I will call the three-dimensional equivalent of the slope of a line), their intersection will again be very sensitive to small parameter changes; when those two tilts are identical, the planes are parallel and don't intersect, and the correspond-

ing matrix is singular. Again, there will be more than one combination of parameter values where this can occur. For larger matrices, things are harder to imagine, but you get the idea of what makes some matrices inherently difficult to work with.

10.11 Eigenvalues and eigenvectors (AE3 pp. 489-494)

A singular matrix occurs when the simultaneous equations to be solved are fully linearly dependent. We have seen in, e.g., sections 2.9, 8.14, and 10.10, that such dependency is not a black-and-white proposition, but instead can exhibit shades of gray. Near-singular matrices are often called *ill-conditioned*, and their numerical inversion may cause problems due to the finite numberlength used in the computations. One possible remedy would be to use a longer numberlength, an option we will indeed entertain in chapter 11. Another will be described in section 10.13. But first we will consider an important aspect of square matrices that we have not discussed so far, namely its eigenvalues and eigenvectors.

In section 10.10 we considered two simultaneous equations, and formulated them in matrix format as

$$\mathbf{A}\,\mathbf{x} = \mathbf{b} \tag{10.10.6}$$

where **x** and **b** were vectors, and **A** denoted a square matrix. We can interpret such a formalism as follows: the product of **A** with the vector **x** is another vector, **b**, of the same size as **x**. In the example of section 10.10 the vectors were of size 2×1 and can therefore be visualized easily in a graph as arrows pointing from the origin to points defined by the two coordinates x_1 and x_2 of **x**, or b_1 and b_2 of **b**. Forming the matrix product **A x** can then be seen as **A** transforming vector **x** into a new vector **b** of the same size but, in general, with a different direction and/or magnitude. And even where we cannot easily visualize vectors in four or more dimensions, this conceptual interpretation of the action of **A** as operating on **x** in (10.10.6) is readily generalized to more simultaneous equations in more unknowns.

We will from now on use different symbols in order to make some useful distinctions. We will denote ***s***quare matrices by **S**, and ***r***ectangular ones by **R**. In this section we will deal only with square matrices, and ask whether there are any special $n\times1$ vectors **x** which, after such action by an $n\times n$ matrix **S**, yield a new $n\times1$ vector **q** that points in the very same (n-dimensional) direction as **x**, and therefore is only changed in length. In that case we should be able to write

$$\mathbf{S}\,\mathbf{q} = \lambda\,\mathbf{q} \tag{10.11.1}$$

where λ is a *scalar*, i.e., a simple (real or complex) numerical multiplier. It turns out that nonsingular (i.e., invertible) square matrices **S** indeed have n such special scalars, which are called their *eigenvalues* λ, with n corresponding *eigenvectors* **q**. The German term *eigen* indicates that these characteristic properties as it were *belong to* **S**, and the term has stuck in English.

Eigenvalue problems are quite common in matrix algebra, and typically occur not so much when we consider simultaneous *algebraic* equations, as we have mostly done so far, but with simultaneous *differential* equations. A famous eigenvalue problem is the Schrödinger wave equation

$$\mathbf{H}\,\boldsymbol{\psi} = E\,\boldsymbol{\psi} \tag{10.11.2}$$

shown here in its simplest (time-independent) form, which relates the Hamiltonian operator (a differential equation in matrix form) **H** of a quantum-mechanical system and its wavefunctions (eigenvectors) **ψ** to its scalar eigenvalues, the energies E. The field of resulting *quantum mechanics* is heavily dependent on matrix algebra. Quantum mechanics is the basis of modern atomic-scale physics and chemistry, relativity theory applies mostly to very large-scale phenomena, and Newtonian physics to most everything in-between. Matrix algebra plays a huge role in Newtonian mechanics, and its importance in quantum mechanics and relativity is even larger.

For a 2×2 matrix, the eigenvalues can be found by solving a quadratic *characteristic polynomial*. If its roots are complex, these eigenvalues are each other's complex conjugates. For a 3×3 matrix, the eigenvalues are given by a cubic equation, i.e., the characteristic polynomial is of 3rd order, etc. Unfortunately, equations of fifth or higher order have no known exact solutions, and must therefore be found iteratively. General-purpose software for finding eigenvalues and eigenvectors therefore uses an iterative approach that approximates the eigenvalues and eigenvectors to within a desired accuracy.

Only square, nonsingular matrices have eigenfunctions. In general, a nonsingular $n\times n$ matrix will have n eigenvalues, which need not all be different. When two or more eigenvalues are identical, they are called *degenerate*.

Matrix.xla(m) has several functions for finding eigenvalues and eigenvectors, most with names starting with MEigen, as listed in Table 10.9.1. They differ in capability, complexity, and execution speed, and which one is best will depend on the particular matrix to which it is applied. For the small matrices we will use in our exercises, you will not notice any such differences. Some examples are illustrated in exercise 10.11.1.

Exercise 10.11.1:

(1) On a clean spreadsheet, place the real, symmetric matrix **S** shown in B3:C4, as in Fig. 10.11.1. A real (rather than complex) square matrix has only real matrix elements m_{ij}, and its symmetry (along its main diagonal) requires that it be square with $m_{ij} = m_{ji}$. Any real symmetric matrix, or any complex square matrix with complex elements m that are its own conjugate transpose (so that the matrix elements m_{ij} are the complex conjugates of the elements m_{ji}), is called *Hermitian*, and has real eigenvalues. Hermitian matrices are fairly common, as they automatically arise when we form the product of a matrix with its transpose (or, in the case of a complex matrix, with its conjugate transpose, which we will denote as $\mathbf{S}^H$ with the superscript H of Hermitian in order to distinguish it from the normal transpose $\mathbf{S}^T$). Matrix.xla has the instruction `=MTC()` for a complex transpose, and `=MTH()` for a Hermitian or complex conjugate transpose.

(2) Highlight E3:E4, type the instruction `=MEigenValQR(B3:C4)`, and enter it with Ctrl◡Shift◡Enter. You will indeed get two real eigenvalues, but they need not be positive integers as in this example.

(3) Use several other instructions instead, such as MEigenValQL which is meant for tridiagonal matrices (which any 2×2 matrix automatically is) or the general MEigenValPow. They should all yield the same two eigenvalues, although not necessarily in the same order, as you can see in Fig. 10.11.1.

(4) For each eigenvalue λ there should be one associated eigenvector **q**. In C7 place the instruction =E3, and find the corresponding eigenvector in E7:E8, e.g., with the instruction `=MEigenVec(B3:C4,C7)`.

(5) In H7:H8 compute the product of S and $\mathbf{q}_1$, and in J7:J8 calculate λ_1 times $\mathbf{q}_1$. Check that the two answers (in Fig. 10.11.1 connected by double-headed arrow) are identical, i.e., verify that (10.11.1) indeed applies in this example.

(6) Highlight B6:J8, and copy it to B10, then change the instruction in C11 to `=E4`, and adjust the references to **S** in E11:E12 and H11:H12 to B3:C4. You should again find an eigenvector that satisfies (10.11.1). Also adjust the labels.

(7) With the instruction `=MEigenVec(B3:C4,E3:E4)` in, e.g., F15:G16 you can simultaneously display both eigenvectors. Your spreadsheet should now resemble that of Fig. 10.11.1.

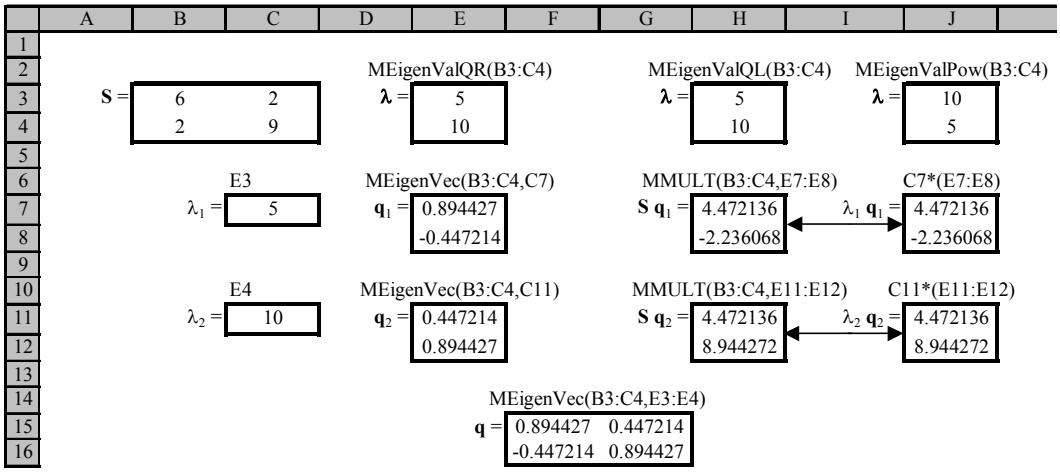


Fig. 10.11.1: An annotated spreadsheet of Exercise 10.11.1 for the eigenvalues and eigenvectors of a real Hermitian matrix.

So far we have glossed over two points. The first is that the eigen*values* are uniquely defined, but the corresponding eigen*vectors* are not, in the sense that multiplication of the vector **q** by any scalar affects the left- and right-hand sides of (10.11.1) in the same way, and therefore keeps that equation intact no matter what multiplier we use. It is therefore customary to normalize the eigenvectors by dividing them by their Euclidian vector length (or Frobenius norm) $\|\mathbf{q}\|$ which is the positive square root of the sum of squares of the individual vector elements. You can readily verify on the spreadsheet that, e.g., $0.894427^2 + (-0.447214)^2 = 1$ in E7:E8. In that way, the eigenvectors are standardized, except for their signs. Matrix.xla(m) has the convenient functions MNorm to compute the matrix or vector norm, and MNormalize and MNormalizeC to perform that normalization for you.

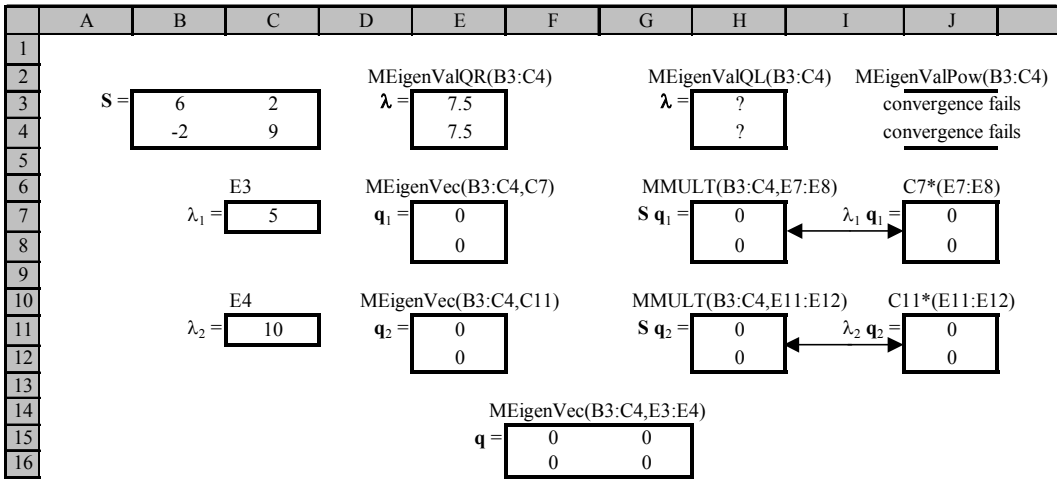


Fig. 10.11.2: A mere change from 2 to –2 in cell B4 yields an output full of warning signs, such as question marks in H3:H4, error messages in J3:J4, and zeroes in many derived results. Clearly, the software commands used are inadequate for a complex output.

The second point is that the eigenvalues and/or eigenvectors of a square matrix with real elements can be complex, just as the roots of a quadratic equation with real coefficients can be. We did not have to worry about that in exercise 10.11.1 because we used a Hermitian matrix **S**. But as Fig. 10.11.2 illustrates, as soon as we deviate from a Hermitian matrix, we run the risk of a nonsensical answer when we don't anticipate a complex result.

In general, therefore, we need instructions that can handle complex eigenanalysis, such as MEigenValQRC and MEigenVecC. In this book we will use the default (split) complex format, which assigns separate blocks to the real and imaginary components of **S** and **q** as illustrated in Fig. 10.11.2. For added clarity we have used a thin separator between the real and imaginary components. Alternatively you can use two different, light background colors, or two different shades of gray. Exercise 10.11.2 will use the equivalent spreadsheet for general, complex numbers

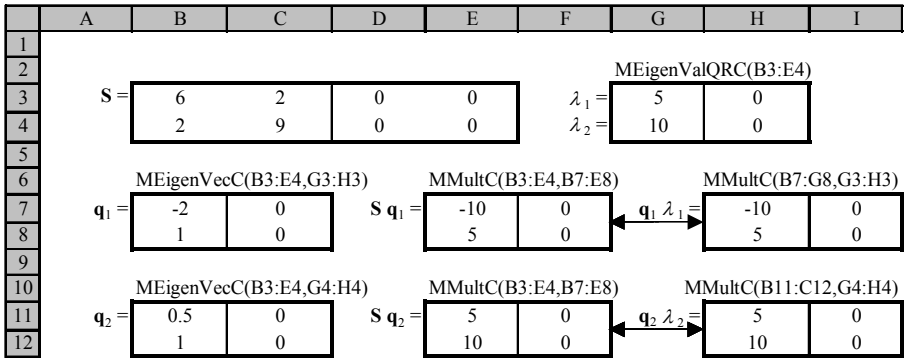


Fig. 10.11.3: A general spreadsheet for finding the eigenfunctions of a nonsingular 2×2 matrix, allowing for complex input and output which, in this case, is actually not needed since **S** is real and Hermitian.

Note that we consider S in B3:E4 as a square 2×2 matrix, even though its notation takes twice as much spreadsheet space because we *display* their real and imaginary components in separate cells. Likewise, λ in G3:H3 is a 1×1 scalar, whether or not it has a non-zero imaginary component and takes up one or two cells.

Exercise 10.11.2:

(1) Open a new spreadsheet, and model it after Fig. 10.11.3, i.e., with the same Hermitian matrix **S** as used there. You need not specify the zeroes in D3:E4; leaving those cells unspecified will be interpreted as zeroes.

(2) As you will see, the results are the same as those obtained in Fig. 10.11.1, except that MEigenVecC can only deal with one eigenvalue at a time, and therefore cannot generate **(3)** in a single instruction, as in F15:G16 of Fig. 10.11.1.

(3) Note that λ_1 is now no longer a scalar but a vector, so that vector dimensionality requires that λ_1 $\mathbf{q}_1$ actually be computed as $\mathbf{q}_1$ λ_1., because λ_1 is displayed as a 1×2 vector, and the sought product should be 2×1.

(4) Change the 2 in cell B4 into –2. It will show you why Fig. 10.11.2 did not work, because both the eigenvalues and eigenvectors are now complex.

(5) Note that the output of MEigenVecC is not normalized, because no single element in a normalized vector can ever be outside the range from –1 to +1, whereas there are many numbers exceeding that range in B7:C8 and B11:C12. To normalize the eigenvectors you can use the instruction MNormalizeC, as long as you use the block containing both its real and imaginary components as its argument.

(6) Here is the result for a complex Hermitian matrix **S**, in which case the top and bottom triangle of the imaginary part must have opposite signs. Its diagonal can only contain zeroes, the only value that is equal to its own negative.

(7) We see that this complex Hermitian input indeed has real eigenvalues, but its eigenvectors are complex.

(8) Finally, Fig. 10.11.6 shows the results for a general nonsingular square matrix. Play with it by varying the values of **S**.

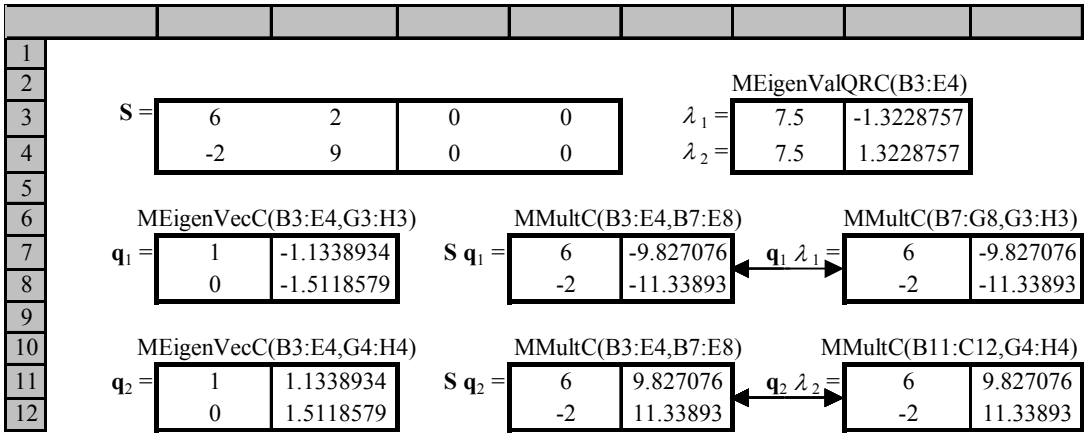


Fig. 10.11.4: The correct result for the problem shown in Fig. 10.11.2.

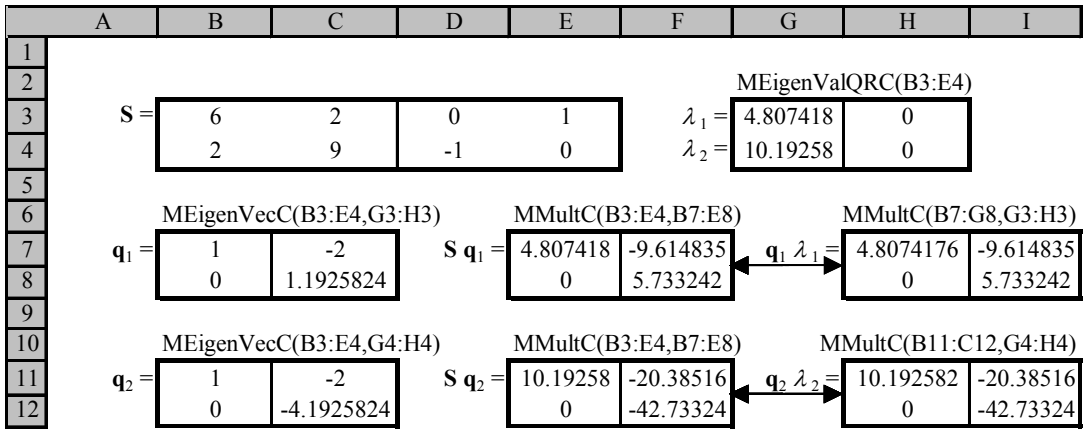


Fig. 10.11.5: A complex Hermitian matrix has real eigenvalues but complex eigenvectors.

So far we have used the instruction MEigenVecC which handles one eigenvector at a time. Often it is more efficient to use MEigenVecInvC to make one complex matrix; its underlying routine is also more robust, and it yields normalized eigenvectors where, for complex vectors, normalization of course includes both the real and imaginary components. The self-annotated Fig. 10.11.7 therefore shows how such an analysis is most efficiently achieved, using a minimum of instructions and spreadsheet real estate. Eigenfunction analysis can hardly be simpler.

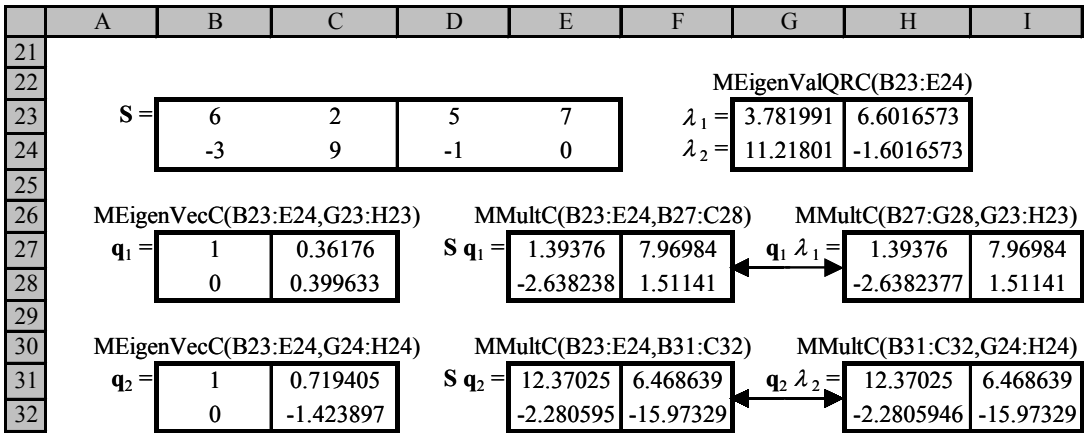


Fig. 10.11.6: An annotated spreadsheet of Exercise 10.11.2 for the eigenfunction analysis of a general complex matrix.

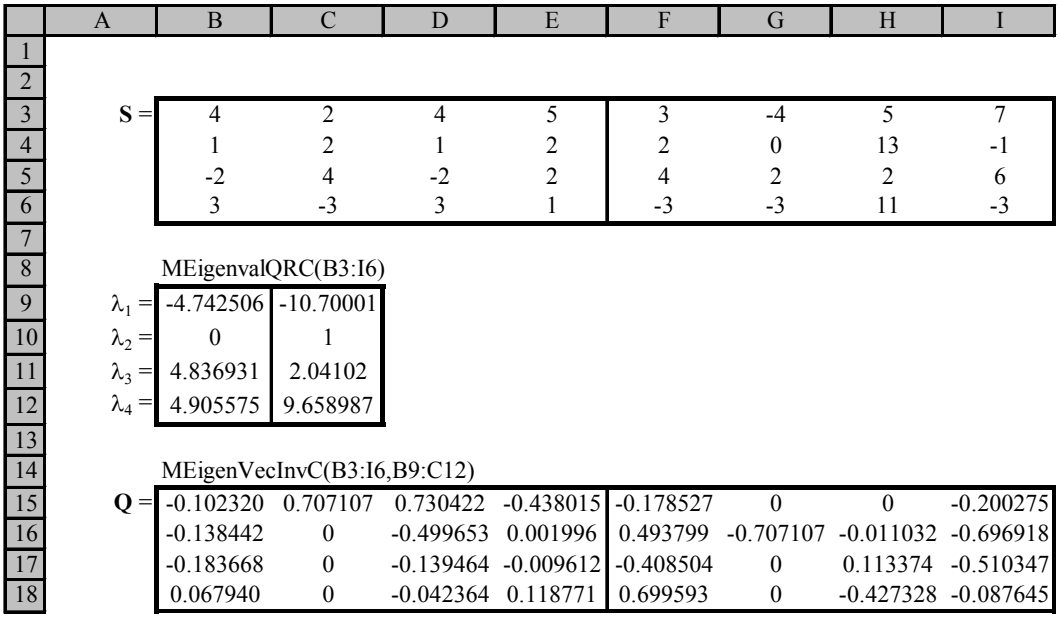


Fig. 10.11.7: An annotated spreadsheet for the eigenfunction analysis of a general complex matrix.

10.12 Eigenvalue decomposition (AE3 pp. 495-497)

Now that we now know how to find the eigenvalues λ and eigenvectors $\mathbf{q}$ of $\mathbf{S}$, we can ask whether we can travel this route in reverse, and compute $\mathbf{S}$ from its eigenvalues and eigenvalues. The answer is yes, and is in fact simple to derive as well as to implement. In general, an $m \times m$ matrix will have m eigenvalues λ (which, just like the roots of a higher-order polynomial, need not all be distinct), and each eigenvalue λ_i will have an associated eigenvector $\mathbf{q}_i$. Place the eigenvalues λ_i on the diagonal of a diagonal matrix $\mathbf{\Lambda}$, so that all off-diagonal elements are zero. Then arrange all eigenvectors $\mathbf{q}_i$ in a matrix $\mathbf{Q}$, by placing them one next to the other in the same order as that of the corresponding eigenvalues λ_i in $\mathbf{\Lambda}$, i.e., in our chosen format for complex notation, grouping the real components first and then, separately, the imaginary ones. We then have $\mathbf{S}\,\mathbf{q} = \mathbf{q}\,\lambda$. (Mathematically, it doesn't matter whether we write $\lambda\,\mathbf{q}$ or $\mathbf{q}\,\lambda$, because both merely mean that each element of $\mathbf{q}$ is multiplied by the scalar λ. But on the spreadsheet we treat vectors as either column or row vectors, and if we elect to write $\mathbf{q}$ as a column vector, as in Fig. 10.12.1, we obviously must use $\mathbf{q}_{r\times 1}$ times $\lambda_{1\times 1}$ in order to make their inner indices equal, as necessary for matrix multiplication, see section 10.1.3.) Application of the rules of matrix multiplication will then quickly convince you that we can condense the n equations $\mathbf{S}\,\mathbf{q} = \mathbf{q}\,\lambda$ and into the single matrix equation

$$\mathbf{S}\,\mathbf{Q} = \mathbf{Q}\,\mathbf{\Lambda} \qquad (10.12.1)$$

so that we can reconstruct **S** simply as

$$\mathbf{S} = \mathbf{S}\,\mathbf{Q}\,\mathbf{Q}^{-1} = \mathbf{Q}\,\mathbf{\Lambda}\,\mathbf{Q}^{-1} \tag{10.12.2}$$

as will be illustrated in the next two exercises. Alternatively, we can consider (10.12.2) as a way to decompose any nonsingular square matrix **S** into the product of three separate matrices of the same size $m \times m$ as **S**. The middle matrix of this trio, $\mathbf{\Lambda}$, is diagonal and contains only the eigenvalues of **S**, while its two neighbors derive from the corresponding eigenvectors. We will illustrate these relationships in exercise 10.12.1 for a square matrix (which need not be Hermitian) with real eigenfunctions, and in exercise 10.12.2 for a general square matrix.

Exercise 10.12.1:

(1) On a new spreadsheet enter a square matrix **S** such as shown in B3:D5 of Fig. 10.12.1, which was carefully selected to have only real eigenvalues and eigenvectors to keep the spreadsheet simple.

(2) Verify that it has only real eigenvalues by filling the block E3:G5 with zeroes, and in block I3:J5 use `=MEigenValQRC(B3:G5)` to find the corresponding eigenvalues. With both **S** and the λ's real, the **q**'s must also be real, but you need not take my word for it: test it yourself. As you can see from Fig. 10.12.1, this is indeed the case.

	A	B	C	D	E	F	G	H	I	J
1										
2									MEigenValQRC(B3:G5)	
3	**S** =	-1	6	7	0	0	0		-7.834209	0
4		2	2	8	0	0	0		-3.924174	0
5		5	4	-3	0	0	0		9.758383	0
6										
7	MEigenVecC(B3:G5,I3:J3)				MEigenVecC(B3:G5,I4:J4)				MEigenVecC(B3:G5,I5:J5)	
8	-0.377465	0			1.226825	0			1.431538	0
9	-0.736721	0			-1.764575	0			1.400173	0
10	1.000000	0			1.000000	0			1.000000	0

Fig. 10.12.1: Verifying that the square matrix **S** has real eigenvalues and real eigenfunctions.

(3) On a new spreadsheet, copy the real part of S in B2:D4, find its eigenvalues in F3:F5, and all eigenvectors in H3:J5.

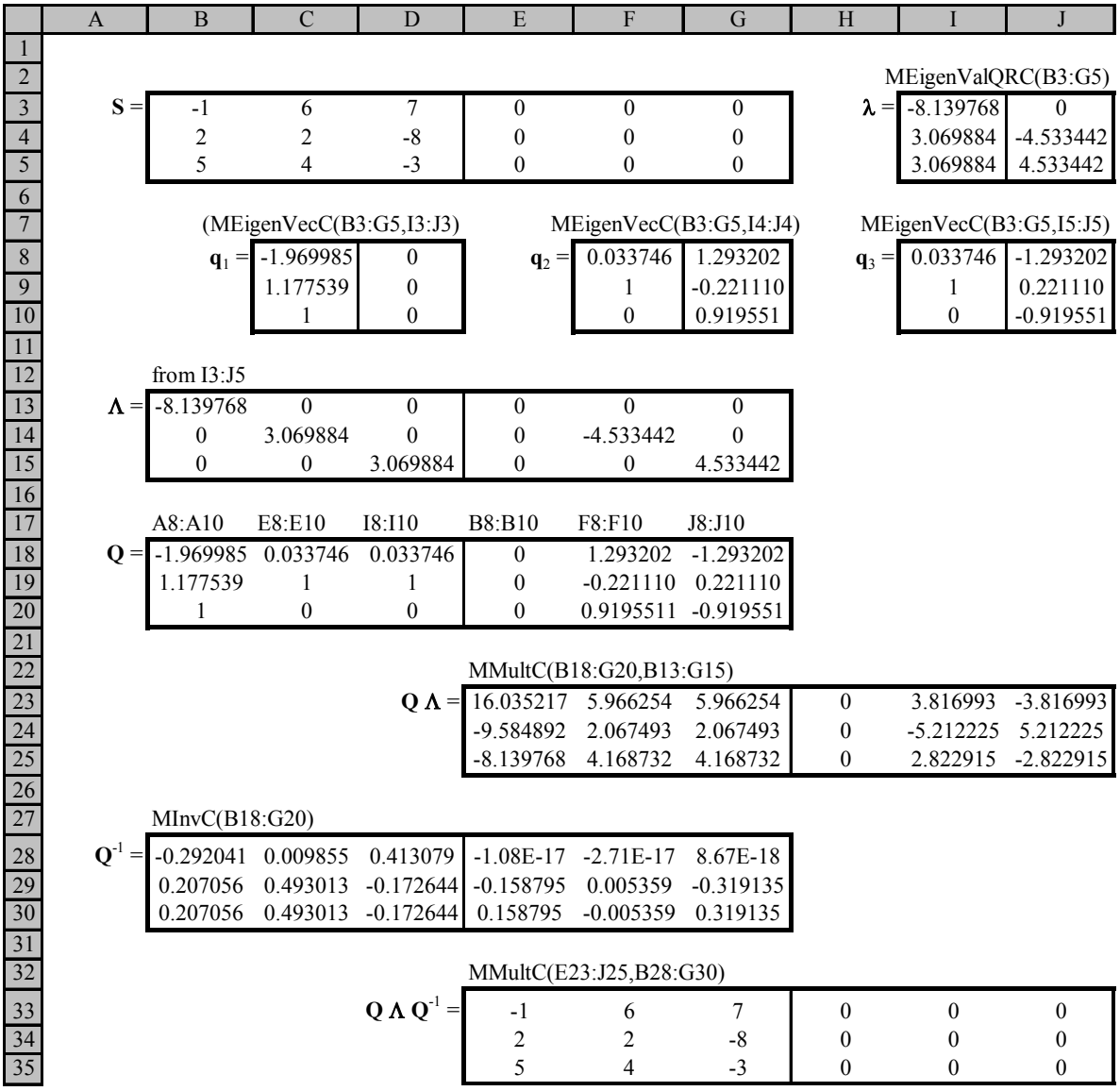
(4) In B8:D10 enter the eigenvalues on its main diagonal, and place zeroes in all off-diagonal positions. (In this case you cannot leave them blank.) This will generate $\mathbf{\Lambda}$.

	A	B	C	D	E	F	G	H	I	J
1										
2						MEigenValQR(B3:D5)		MEigenVec(B3:D5,F3:F5)		
3	**S** =	-1	6	7	λ_1 =	-7.834209	**Q** =	0.290767	0.517559	0.639578
4		2	2	8	λ_2 =	-3.924174		0.567509	-0.744419	0.625564
5		5	4	-3	λ_3 =	9.758383		-0.770317	0.421869	0.446777
6										
7		data from F5:F5					MMULT(H3:H5,F3)		MMULT(H3:J5,B8:B10)	
8	$\mathbf{\Lambda}$ =	-7.834209	0	0	$\mathbf{q}_1\,\lambda_1$ =		-2.277933		-2.277933	
9		0	-3.924174	0			-4.445982	⟷	-4.445982	
10		0	0	9.758383			6.034823		6.034823	
11										
12		MMULT(B3:D5,H3:J5)					MMULT(I3:I5,F4)		MMULT(H3:J5,C8:C10)	
13	**S Q** =	-2.277933	-2.030992	6.241245	$\mathbf{q}_2\,\lambda_2$ =		-2.030992		-2.030992	
14		-4.445982	2.921229	6.104497			2.921229	⟷	2.921229	
15		6.034823	-1.655486	4.359817			-1.655486		-1.655486	
16										
17	MMULT(H3:J5,MMULT(B8:D10,MINVERSE(H3:J5)))						MMULT(J3:J5,F5)		MMULT(H3:J5,D8:D10)	
18	$\mathbf{Q\Lambda Q}^{-1}$=	-1	6	7	$\mathbf{q}_3\,\lambda_3$ =		6.241245		6.241245	
19		2	2	8			6.104497	⟷	6.104497	
20		5	4	-3			4.359817		4.359817	

Fig. 10.12.2: Illustrating (10.12.1) and (10.12.2), the reconstruction of a square matrix **S** from its eigenvalues λ_i and its eigenvectors $\mathbf{q}_i$, for the special case where **S**, $\mathbf{\Lambda}$, and **Q** are all real.

(5) In B13:B16 calculate the matrix product **S Q**. In G8:G10 compute $\mathbf{q}_1\ \lambda_1$, in I8:I10 the product of Q and the first column in L, and compare these two with each other as well as with the first column in **S Q**. All should be the same. Do the same for the second and third eigenvalues and their eigenvectors, and compare with the second and third column of **S Q**.

(6) Finally, in B18:D20, construct $\mathbf{Q}\,\mathbf{\Lambda}\,\mathbf{Q}^{-1}$ and verify that it indeed regenerates **S**. In Fig. 10.12.2 this is done with one compound instruction (to keep the picture compact) but you may prefer to do this in several simpler steps, e.g., by first computing $\mathbf{Q}\,\mathbf{\Lambda}$ and $\mathbf{Q}^{-1}$ separately.



$\mathbf{S}$ =

-1	6	7	0	0	0
2	2	-8	0	0	0
5	4	-3	0	0	0

$\boldsymbol{\lambda}$ =

-8.139768	0
3.069884	-4.533442
3.069884	4.533442

$\mathbf{q}_1$ =

-1.969985	0
1.177539	0
1	0

$\mathbf{q}_2$ =

0.033746	1.293202
1	-0.221110
0	0.919551

$\mathbf{q}_3$ =

0.033746	-1.293202
1	0.221110
0	-0.919551

$\mathbf{\Lambda}$ =

-8.139768	0	0	0	0	0
0	3.069884	0	0	-4.533442	0
0	0	3.069884	0	0	4.533442

$\mathbf{Q}$ =

-1.969985	0.033746	0.033746	0	1.293202	-1.293202
1.177539	1	1	0	-0.221110	0.221110
1	0	0	0	0.9195511	-0.919551

$\mathbf{Q}\,\mathbf{\Lambda}$ =

16.035217	5.966254	5.966254	0	3.816993	-3.816993
-9.584892	2.067493	2.067493	0	-5.212225	5.212225
-8.139768	4.168732	4.168732	0	2.822915	-2.822915

$\mathbf{Q}^{-1}$ =

-0.292041	0.009855	0.413079	-1.08E-17	-2.71E-17	8.67E-18
0.207056	0.493013	-0.172644	-0.158795	0.005359	-0.319135
0.207056	0.493013	-0.172644	0.158795	-0.005359	0.319135

$\mathbf{Q}\,\mathbf{\Lambda}\,\mathbf{Q}^{-1}$ =

-1	6	7	0	0	0
2	2	-8	0	0	0
5	4	-3	0	0	0

Fig. 10.12.3: Illustrating (10.12.1) and (10.12.2), the reconstruction of a square matrix **S** from its eigenvalues λ_i and its eigenvectors $\mathbf{q}_i$ for the general case where **S**, $\mathbf{\Lambda}$, and **Q** are all potentially complex quantities.

In the above example we made sure that **S** only had real eigenvalues and real eigenvectors, but in general that will not be the case. Just play with the numbers in Fig. 10.12.1 and you will realize that this is the exception rather than the rule. Merely changing the sign of one of its elements generates a complex response for 7 of the 9 numbers in B2:D4. Here we will just change the 8 in D3 to –8, while keeping E2:G4 empty (or filled with zeroes). Of course, when **S** itself already contains imaginary components, some or all of the eigenvalues and/or eigenvectors must be complex.

Exercise 10.12.2:

(1) On a new spreadsheet enter a square matrix **S** such as shown in B3:D5 of Fig. 10.12.3.

(2) In I3:J5 use `=MEigenValQRC(B3:G5)` to find the corresponding eigenvalues.

(3) For each of the eigenvalues λ_i find the corresponding eigenvector $\mathbf{q}_i$ in C8:D10, F8:G10, and I8:J10.

(4) Construct the matrix $\mathbf{\Lambda}$ from two adjacent zero matrices, one using the real components I3:I5 of $\boldsymbol{\lambda}$, and the other its imaginary components from J3:J5.

(5) Likewise, in B13:G15, stitch together the matrix Q from its individual pieces in rows 8:10.

(6) Compute the matrix product $\mathbf{Q}\,\mathbf{A}$ in E23:J25, and the inverse $\mathbf{Q}^{-1}$ of $\mathbf{Q}$ in B28:G30.

(7) Finally calculate $\mathbf{Q}\,\mathbf{\Lambda}\,\mathbf{Q}^{-1}$ in E33:J35, which should reconstruct $\mathbf{S}$ from its eigenvalues and eigenvectors. Your spreadsheet should now resemble Fig. 10.12.3

Now play with this spreadsheet, by changing the values in B3:D5, and by inserting non-zero numbers into E2:G4, and see what happens. If your spreadsheet is built correctly, it will automatically reproduce its input $\mathbf{S}$ in its output $\mathbf{Q}\,\mathbf{\Lambda}\,\mathbf{Q}^{-1}$. That, of course, is the point of this exercise.

A final note: the eigenvectors in rows 8:10 are not normalized. However, do *not* try to normalize them, as the above check will not work. Normalization seems not to work properly with some of the complex number operations of Matrix.xla.

10.13 Singular value decomposition (AE3 pp. 498-501)

A singular matrix occurs when the simultaneous equations to be solved are not linearly independent. We have seen in sections 2.9, 8.14 and 10.10 that such independency is not an all-or-none proposition, but instead can exhibit some uncertainty, because round-off errors can make a singular matrix appear to be nonsingular. Near-singular matrices are often called *ill-conditioned*, and their numerical inversion may cause problems due to the finite numberlength used in floating-point calculations.

One possible remedy would be to use a longer numberlength, an option we will consider in chapter 11. There is, however, a powerful matrix method that can reduce some of the problems involving ill-conditioned matrices, even while remaining in the usual "double precision" mode. We will briefly describe this method, called *singular value decomposition* (SVD), which is based on the decomposition of any ***r***ectangular matrix $\mathbf{R}$ into three matrices, just as eigenvalue decomposition does for square matrices. This method constitutes a useful generalization of eigenvalue decomposition, because it applies to all rectangular matrices, including those that are square, and even to those that are both square and singular.

In the compact notation used by Matrix.xla(m) as well as the Numerical Recipes, singular value decomposition of a rectangular real matrix $\mathbf{R}$ of size $m\times n$ where $m \geq n$ yields

$$\mathbf{R} = \mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^{\mathrm{T}} \tag{10.13.1}$$

where $\mathbf{U}$ is an $m\times n$ orthogonal matrix (*not* an upper triangular one, as traditionally indicated by the same symbol), and $\mathbf{\Sigma}$ and $\mathbf{V}$ are square matrices of size $n\times n$. We may compare (10.13.1) with the somewhat simpler (10.12.2) for a square matrix. The corresponding pseudo-inverse $\mathbf{R}^{+}$ of $\mathbf{R}$ is

$$\mathbf{R}^{+} = (\mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^{\mathrm{T}})^{-1} = \mathbf{V}\,\mathbf{\Sigma}^{-1}\,\mathbf{U}^{\mathrm{T}} \tag{10.13.2}$$

because

$$\mathbf{R}^{+}\,\mathbf{R} = \mathbf{V}\,\mathbf{\Sigma}^{-1}\,\mathbf{U}^{\mathrm{T}}\,\mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^{\mathrm{T}} = \mathbf{V}\,\mathbf{\Sigma}^{-1}\,(\mathbf{U}^{\mathrm{T}}\,\mathbf{U})\,\mathbf{\Sigma}\,\mathbf{V}^{\mathrm{T}} = \mathbf{V}\,(\mathbf{\Sigma}^{-1}\,\mathbf{\Sigma})\,\mathbf{V}^{\mathrm{T}} = \mathbf{V}\,\mathbf{V}^{\mathrm{T}} = \mathbf{I} \tag{10.13.3}$$

where we have used the properties that $\mathbf{U}$ and $\mathbf{U}^{\mathrm{T}}$ as well as $\mathbf{V}$ and $\mathbf{V}^{\mathrm{T}}$ are orthogonal, i.e., $\mathbf{U}^{\mathrm{T}}\,\mathbf{U} = \mathbf{V}\,\mathbf{V}^{\mathrm{T}} = \mathbf{I}$. (In general, however, $\mathbf{U}\,\mathbf{U}^{\mathrm{T}} \neq \mathbf{I}$. and $\mathbf{U}^{\mathrm{T}}\,\mathbf{U} \neq \mathbf{I}$.) Because $\mathbf{\Sigma}$ is diagonal, with diagonal elements σ_i and off-diagonal zeroes, its inverse $\mathbf{\Sigma}^{-1}$ is found directly by replacing all non-zero terms σ_i by $1/\sigma_i$.

Several special properties of singular value decomposition are:

(1) Singular value decomposition is possible for both square and rectangular matrices, and for both singular and non-singular matrices.

(2) The diagonal elements σ_i of $\mathbf{\Sigma}$ are called the *singular values* of $\mathbf{R}$. These singular values are all non-negative, and are usually presented in order of decreasing magnitude: $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \ldots \geq \sigma_r \geq 0$, where the index r denotes the *rank* of $\mathbf{R}$, i.e., the number of columns in $\mathbf{R}$ that are not completely linearly dependent on one or more of the other columns.

(3) For a Hermitian matrix (i.e., either a symmetrical real square matrix for which $\mathbf{R} = \mathbf{R}^{\mathrm{T}}$, or a complex square matrix for which $\mathbf{R} = \mathbf{R}^{\mathrm{H}}$, its conjugate, Hermitian transpose) the singular values are the square roots of the absolute values of its eigenvalues, i.e., $\sigma_i^2 = |\lambda|$.

(4) The ratio of the largest to smallest singular value yields the *condition number* $\kappa = \sigma_1/\sigma_r \geq 1$. Singular value decomposition therefore provides a direct way to characterize a matrix in terms of its robustness against the effects of numerical errors. If the matrix $\mathbf{R}$ is singular, κ should go to infinity, but since

the spreadsheet cannot represent infinity, κ either becomes a very large number or shows a divide-by-zero error message. If **R** is near-singular, the condition number κ is very much larger than 1. In all such cases, the quantity pκ (pronounced as "pee-kappa") = –log(κ) provides an estimate (in terms of decimal places) of the maximum loss of precision in inverting a matrix, in terms of number of decimals. The corresponding decimal places are not lost, but their significance usually is. In exercise 10.10.1 we already encountered an example of an ill-conditioned matrix which, as you will see shortly, has a pκ of about –6, i.e., its last six decimals become statistically insignificant, because inverting that matrix can amplify the noise-to-signal ratio by a factor of the order of κ or, in the above example, about a million-fold.

Exercise 10.13.1 illustrates the following basic properties of singular value decomposition:
(1) *Any* rectangular matrix **R** can be written as $\mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^T$.
(2) The singular values on the diagonal of $\mathbf{\Sigma}$ are the positive square roots of the non-zero eigenvalues of $\mathbf{R}^T\mathbf{R}$ or $\mathbf{R}\,\mathbf{R}^T$.
(3) The columns of **U** are the eigenvectors of $\mathbf{R}\,\mathbf{R}^T$.
(4) The columns of **V** are the eigenvectors of $\mathbf{R}^T\mathbf{R}$.
These properties establish SVD as the most useful generalization to rectangular matrices of the eigen-analysis of square matrices. This is why singular value decomposition plays such a central role in applied matrix algebra.

Exercise 10.13.1:

(1) On a new spreadsheet place a small non-singular, non-square matrix **R** in B3:C5

(2) In E3:G4 place its transpose, $\mathbf{R}^T$.

(3) In B8:C10, F8:G9, and I9:J10 compute the SVD matrices **U**, $\mathbf{\Sigma}$, and **V** respectively. Note that Matrix.xla uses the symbol **D** for the ***d***iagonal matrix we here call $\mathbf{\Sigma}$ containing the ***s***ingular values σ_i in order to distinguish it from the diagonal matrix $\mathbf{\Lambda}$ containing the eigenfunctions λ_i.

(4) In J3 use `=MpCond(B3:C5)` to find pκ, the negative logarithm of the condition number κ, and in J6 verify that κ is the ratio of the largest to smallest singular value.

(5) In B13:D15 calculate the matrix product $\mathbf{R}\,\mathbf{R}^T$, which you can readily verify is a symmetrical square matrix.

(6) In F13:F15 find the three eigenvalues of the square matrix $\mathbf{R}\,\mathbf{R}^T$. You need not consider complex eigenvalues, because $\mathbf{R}\,\mathbf{R}^T$ is a symmetrical square matrix, and is therefore Hermitian.

(7) In B18:C19 calculate the matrix product $\mathbf{R}^T\mathbf{R}$, in E18:E19 its eigenvalues, and in G18:G19 their square root. (Here the eigenvalues are all positive. If not, compute the square roots of their absolute values.) Verify that the eigenvalues (in F14:F15 and E18:E19 respectively) and F14:F15 are the same.

(8) Verify that the *non-zero* eigenvalues of $\mathbf{R}\,\mathbf{R}^T$ and $\mathbf{R}^T\mathbf{R}$ in F14:F15 and E18:E19 respectively are the same, as they should be. This therefore also applies to their square roots. Verify that the latter are indeed the same as the significant values on the main diagonal of Σ in F8:G9. This illustrates point (2) just above this exercise.

(9) In I13:J15 compute the eigenvectors of $\mathbf{R}\,\mathbf{R}^T$, again using only the *non-zero* eigenvalues in F14:F15, and compare these eigenvectors with the columns of **U** in B8:C10. This illustrates point (3) given above his exercise. The signs of entire columns may differ, because normalized vectors still have an inherent sign ambiguity.

(10) Likewise, in I18:J19 find the eigenvectors of $\mathbf{R}^T\mathbf{R}$, which should be the same (but for their signs) as those in I9:J10 for **V**, and thereby illustrate point (4).

(11) In B22:C24 calculate the product $\mathbf{U}\,\mathbf{\Sigma}$, and in F22:G24 $\mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^T$. This should reconstitute **R**, as it indeed does, see point (1) above.

Note the ease of getting these results on the spreadsheet. Moreover, once you have made the spreadsheet, you can change one or more elements of **R** and immediately see how such changes affect the answers. You need to make a new spreadsheet only when the size of **R** changes.

Use the just-made spreadsheet to see what happens in Fig. 10.13.1 when you make the two columns in **R** linearly dependent, e.g., by putting 3, 5, 7 in C3:C5 so that the second column is twice the first + 1 (which in matrix parlance does *not* count as "fully linearly dependent" because of the additive constant 1) or 3, 6, and 9 (which does count as such because the second column is now a multiple of the first). In the latter case, there is only one nonzero singular value in $\mathbf{\Sigma}$, and only one nonzero eigenfunction in F15 or E19, and the condition number exceeds 16, yet **R** is still properly reproduced in F22:G24 by $\mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^T$.

The compact notation omits singular values that are zero, and the corresponding rows and columns in $\mathbf{\Sigma}$, **U** and **V**. Figure 10.13.2 shows what the corresponding "full" SVD would look like. It makes both **U** and **V** square, while $\mathbf{\Sigma}$ assumes the size of **R**. (We will use $\mathbf{U}^\wedge$ and $\mathbf{\Sigma}^\wedge$ to distinguish them from their compact values **U** and $\mathbf{\Sigma}$.) When R is $m\times n$, $\mathbf{U}^\wedge$ will have the size $m\times m$, and $\mathbf{\Sigma}^\wedge$ $m\times n$. For a typical least squares problem, where there are often many more data (m) than variables (n–1), the difference in space require-

ments can be significant, while the final results are identical. Both $\mathbf{U}^T\,\mathbf{U} = \mathbf{I}$ and $(\hat{\mathbf{U}})^T\,\hat{\mathbf{U}} = \mathbf{I}$ apply, but in general $\mathbf{U}\,\mathbf{U}^T \neq \mathbf{I}$ and $\hat{\mathbf{U}}\,(\hat{\mathbf{U}})^T \neq \mathbf{I}$.

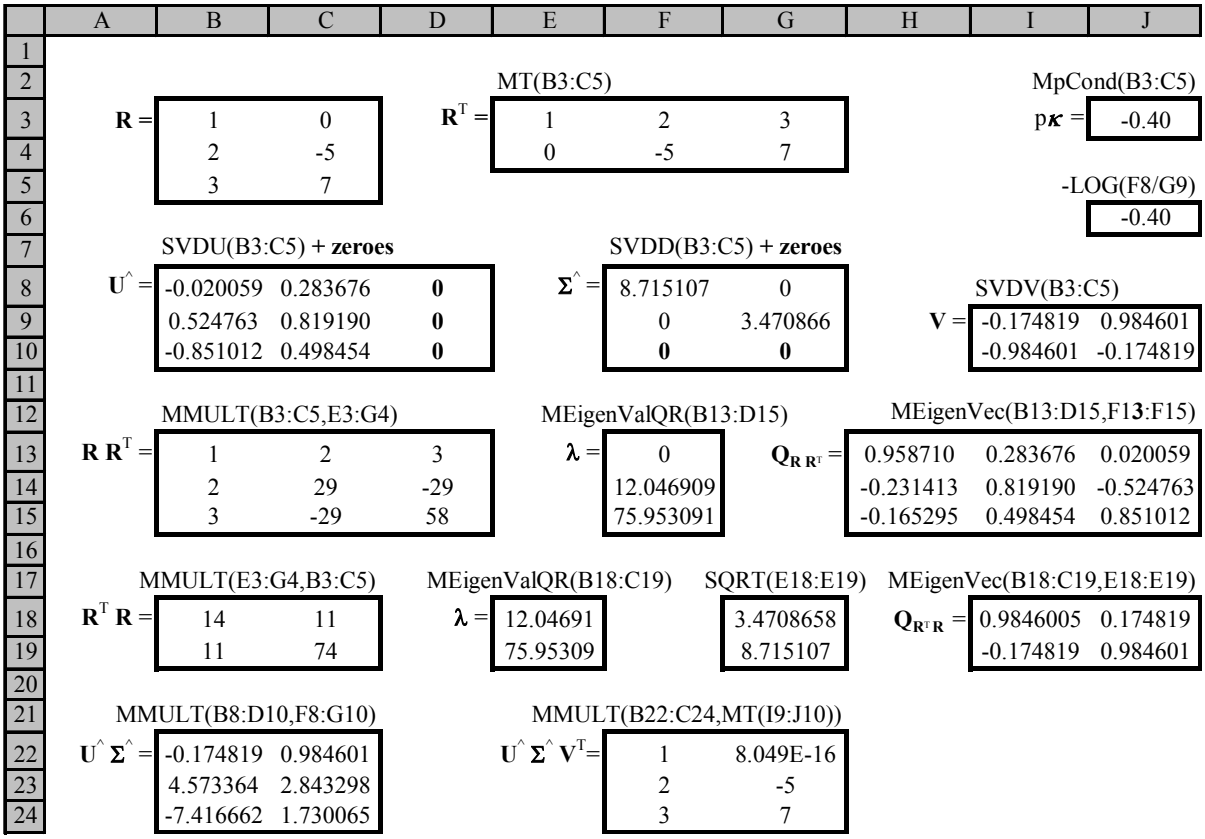


Fig. 10.13.1: A spreadsheet illustrating some properties of the singular value decomposition of a rectangular matrix **R**.

The above rules establishing the relation between the SVD of rectangular matrices (including square and/or singular ones) and the eigenanalysis of non-singular square matrices are readily generalized to complex matrices by replacing the transpose $\mathbf{R}^T$ by the conjugate (Hermitian) transpose $\mathbf{R}^H$. In this more general form, applicable to both real and complex matrices, these rules therefore are:

(1) Any rectangular matrix **R** can be written as $\mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^T$.
(2) The singular values on the diagonal of $\mathbf{\Sigma}$ are the positive square roots of the non-zero eigenvalues of $\mathbf{R}^H\,\mathbf{R}$ or $\mathbf{R}\,\mathbf{R}^H$.
(3) The columns of **U** are the eigenvectors of $\mathbf{R}^H\,\mathbf{R}$.
(4) The columns of **V** are the eigenvectors of $\mathbf{R}\,\mathbf{R}^H$.

Here is a simple demonstration of rule (2) in the above list. We start from a real rectangular matrix **R**, and form the square product $\mathbf{R}^T\,\mathbf{R}$, which is Hermitian. We now apply singular value decomposition to $\mathbf{R}^T$ and **R**, and combine these to

$$\begin{aligned}\mathbf{R}^T\,\mathbf{R} &= (\mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^T)^T\,(\mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^T) = (\mathbf{V}\,\mathbf{\Sigma}^T\,\mathbf{U}^T)\,(\mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^T)\\ &= \mathbf{V}\,\mathbf{\Sigma}^T\,(\mathbf{U}^T\,\mathbf{U})\,\mathbf{\Sigma}\,\mathbf{V}^T = \mathbf{V}\,\mathbf{\Sigma}^T\,\mathbf{\Sigma}\,\mathbf{V}^T = \mathbf{V}\,\mathbf{\Sigma}^2\,\mathbf{V}^T \qquad (10.13.3)\end{aligned}$$

where we have used the rule of matrix transposition that $(\mathbf{A}\,\mathbf{B})^T = \mathbf{B}^T\,\mathbf{A}^T$ and hence $(\mathbf{A}\,\mathbf{B}\,\mathbf{C})^T = \mathbf{C}^T\,\mathbf{B}^T\,\mathbf{A}^T$, together with the facts that $\mathbf{U}^T\,\mathbf{U} = \mathbf{I}$ because the columns of **U** contain orthogonal eigenvectors, and that $\mathbf{\Sigma}$ is diagonal, so that $\mathbf{\Sigma}^T = \mathbf{\Sigma}$ and $\mathbf{\Sigma}^T\,\mathbf{\Sigma} = \mathbf{\Sigma}^2$. An equivalent derivation applies to complex matrices, with the superscript T (for transposition) replaced by H (for Hermitean transposition).

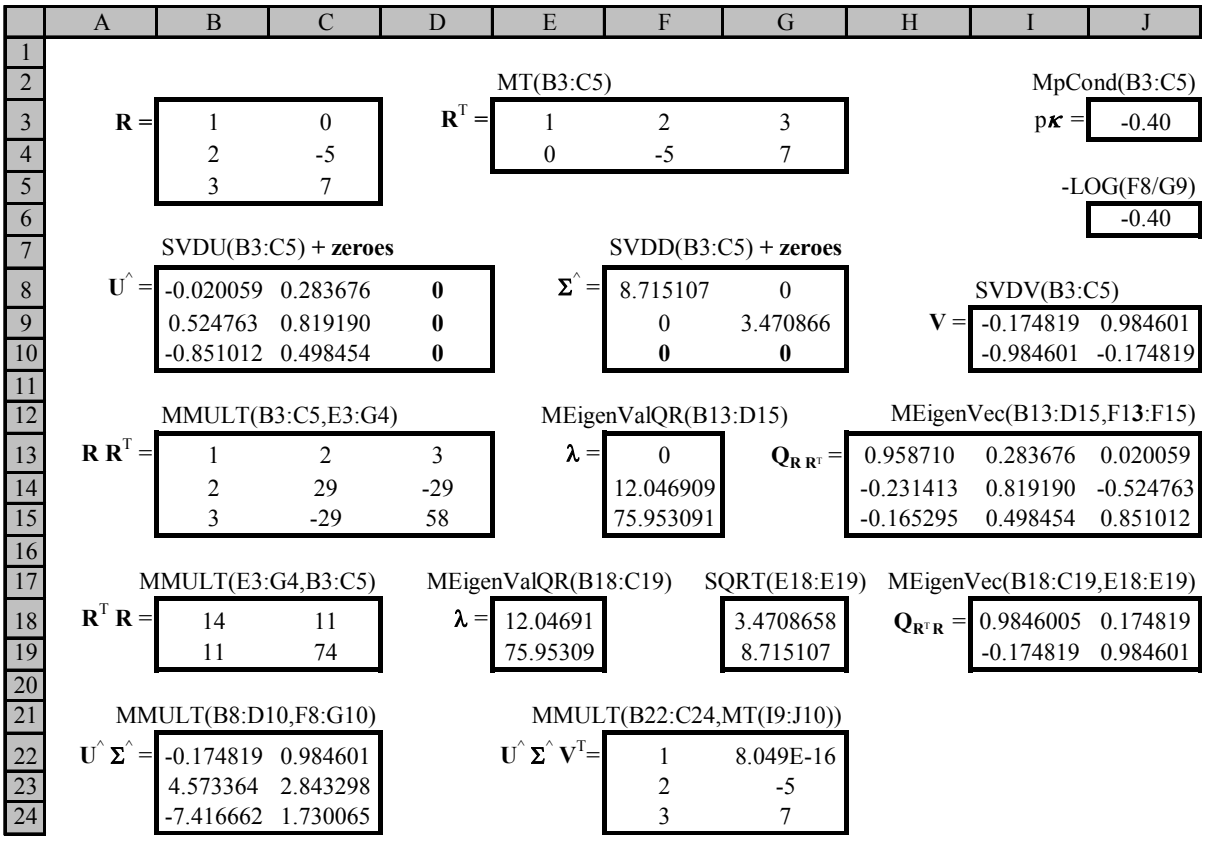


Fig. 10.13.2: The corresponding spreadsheet using the full format of the singular value decomposition. The added zeroes are printed in bold numbers.

Comparing (10.13.3) with the eigenvalue decomposition of the square matrix $\mathbf{R}^T\,\mathbf{R}$, see (10.12.1), yields

$$\mathbf{V}\,\mathbf{\Sigma}^2\,\mathbf{V}^T = \mathbf{Q}_{\mathbf{R}\,\mathbf{R}^T}\,\mathbf{\Lambda}\,(\mathbf{Q}_{\mathbf{R}\,\mathbf{R}^T})^{-1} \tag{10.13.4}$$

from which we see that $\mathbf{\Sigma}$ is indeed the square root of $\mathbf{\Lambda}$ and, because both $\mathbf{\Sigma}$ and $\mathbf{\Lambda}$ are diagonal, the individual singular values σ_i are the square roots of the eigenvalues λ_i. Moreover, $\mathbf{V}$ is the matrix containing the eigenvectors of $\mathbf{R}\,\mathbf{R}^T$, and $\mathbf{V}^T$ is its inverse. In general, for a complex rectangular matrix $\mathbf{R}$, replace the real transpose by its conjugate counterpart.

10.14 SVD and linear least squares (AE3 pp. 501-504)

The benefit of using singular value decomposition for least squares analysis is that it is much less sensitive to ill-conditioning than the traditional approach of section 10.5. Below we will first illustrate the use of singular value decomposition in solving least squares problems.

Exercise 10.14.1:

(1) We now use the same data set as in exercise 10.5.1, i.e., with the x,y coordinates (1,5), (2,8), (3,11), (4,14), and (5,17). These data fit the line $y = 2 + 3x$ exactly, and are so close to the origin that there is no need for centering.

(2) Take a new spreadsheet, and enter the values 5, 8, 11, 14, and 17 of the input data vector **y** in cells B2:B6, and the associated x-values in D2:D6.

(3) In F2:G6 place the matrix **Q**, with its first column of $x^0 = 1$ (in all five cells F2:F6), and with the x-values 1, 2, 3, 4, and 5 as its second column, G2:G6. (If we were forcing the line to pass through the origin, the first column of matrix **Q**, F2:F6, would contain zeros rather than ones.)

(4) In B9:C13 compute **U** with the instruction `=SVDU(F2:G6)`. Note that **U** must have the same size as **X**, in this case 5×2.

(5) In E9:F10 deposit the instruction for **Σ**, `=SVDD(F2:G6)`, and in H9:E10 likewise compute **V**.

(6) In F13:G14 calculate the product $\mathbf{V}\,\mathbf{\Sigma}^{-1}$.

(7) In B16:F17 compute the SVD pseudo-inverse $\mathbf{X}^+ = \mathbf{V}\,\mathbf{\Sigma}^{-1}\,\mathbf{U}^T$ as the product of $\mathbf{V}\,\mathbf{\Sigma}^{-1}\,\mathbf{U}^T$.

(8) Finally, in I16:I17, find **a** as the product $\mathbf{X}^+$ **b**. Your spreadsheet should now resemble Fig. 10.14.1. The results for $\mathbf{X}^+$ and **a** are, of course, the same as those for $\mathbf{X}^+$ and **a** in Fig. 10.5.1, although the steps leading from **X** to $\mathbf{X}^+$ are quite different. Figure 10.14.1.shows what, with some annotation, you might see on your monitor screen.

	A	B	C	D	E	F	G	H	I
1									
2	**y** =	5		**x** =	1	**X** =	1	1	
3		8			2		1	2	
4		11			3		1	3	
5		14			4		1	4	
6		17			5		1	5	
7									
8		SVDU(F2:G6)			SVDD(F2:G6)			SVDU(F2:G6)	
9	**U** =	0.160007	-0.757890	**Σ** =	7.691213	0	**V** =	0.266934	-0.963715
10		0.285308	-0.467546		0	0.91937		0.963715	0.266934
11		0.410609	-0.177202						
12		0.535909	0.113142		MMULT(H9:I10,MInv(E9:F10))				
13		0.661210	0.403486	**V** $\mathbf{\Sigma}^{-1}$ =	0.034706	-1.048234			
14					0.125301	0.290344			
15		MMULT(F13:G14,MT(B9:C13))						MMULT(B16:F17,B2:B6)	
16	$\mathbf{X}^+$ =	0.8	0.5	0.2	-0.1	-0.4		**a** =	2
17		-0.2	-0.1	2.78E-17	0.1	0.2			3

Fig. 10.14.1: A spreadsheet illustrating the individual steps in the solution of a least squares problem using singular value decomposition.

The above exercise illustrates the principle of SVD-based least squares, but for a problem that doesn't really need it. In exercise 10.14.2 we show another example, written with more compact instructions.

Exercise 10.14.2:

(1) Open a new spreadsheet, which we will use for this and the next three exercises. In order to keep the instructions compact, we will again specify locations of the various spreadsheet elements, so that they will also correspond to those in the accompanying figures, and will therefore be easy to compare. Feel free, however, to use your own layout, rather than one driven by the author's need to make compact figures. The same applies to matrix operations you may want to combine in a single instruction, and to labeling used to keep the spreadsheet readable. The best way to get acquainted with matrix operations is to play around with them, and to see what works for you.

(2) In cells all cells in B3:B9 deposit the values 1, in C3:C9 the *x*-values 1 (1) 7, and in D3:D9 compute the corresponding values for x^2. These are the elements of matrix **X** in B3:D9.

(3) In F3:H9 compute **U** with the matrix instruction `=SVDU(B3:D9)`.

(4) In B15:D17 calculate **Σ** with `=SVDD(B3:D9)`, and in B20:D22 find **V** with `=SVDV(B3:D9)`.

(5) In F14:H20 recover the matrix **X** as **U Σ** $\mathbf{V}^T$ through `=MProd(F3:H9,B15:D17,MT(B20:D22))`. Verify that you indeed have reconstructed the original matrix **X**. Using MProd yields more compact and better readable code than twice MMULT.

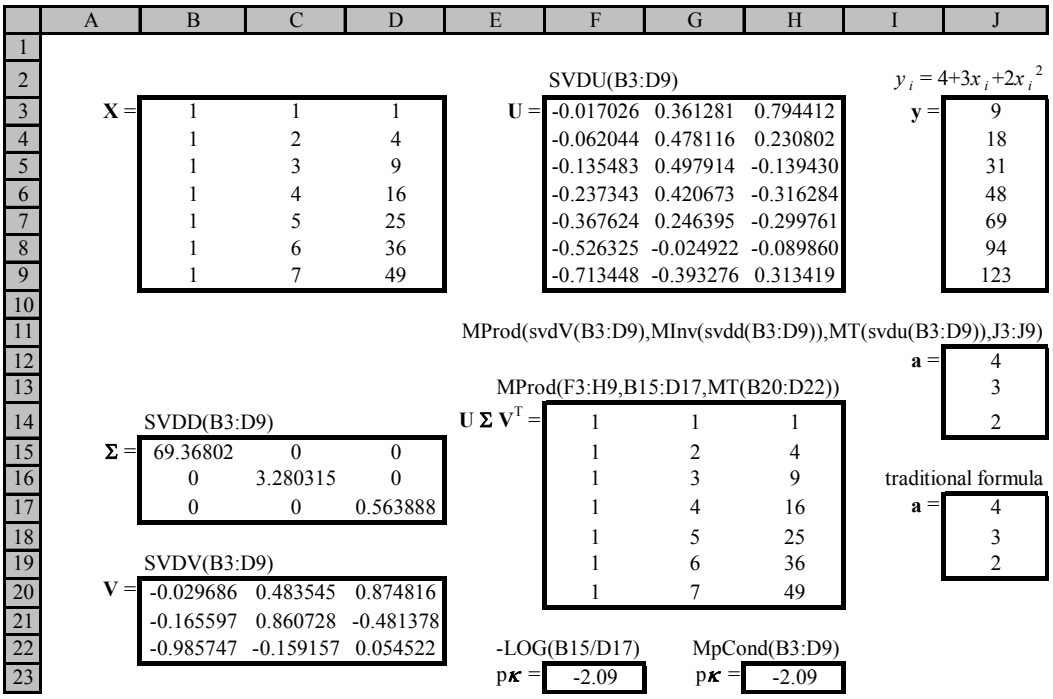
(6) If you want to make the spreadsheet even smaller, you could of course skip displaying **U**, **S**, and **V** entirely, and merely use the megaformula `=MProd(SVDU(B3:D9),SVDD(B3:D9),MT(SVDV(B3:D9)))`, or write a function that will do this automatically for you.

(7) In cell F23 calculate pκ as the negative ten-based logarithm of the ratio of the larger to the smaller of the singular values in **Σ**, and in H23 compute the same pκ as `=MCond(B3:D9)`.

(8) In J3:J9 compute **y** as $y_i = 4 + 3x_i + 2x_i^2$ using the values of x_i and x_i^2 in B2:D8. If you want to play with fitting a line through the origin, code it in J3 as `=4*B3+3*C3+2*D3`, and copy this down to J9.

(9) Just for kicks (because it is not really recommended to use such long, hard-to-read instructions) in J12:J14 compute the least-squares solution of the problem **X a** = **y**. Notice that you can do this indeed with one line of code, without any other input information than X in B3:D9 and y in J3:J9.

(10) For comparison, in J17:J20 compute a with the traditional formula (10.5.9) as $(\mathbf{X}^T\,\mathbf{X})^{-1}\,\mathbf{X}^T\,\mathbf{y}$. You could use the formula `=MProd(MInv(MMULT(MT(B3:D9),(B3:D9))),MT(B3:D9),J3:J9)`; the answers in J13:J15 and J17:J20 should of course be the same. Your annotated spreadsheet may now resemble Fig. 10.14.2. Save it for the next exercise.



X			
	1	1	1
	1	2	4
	1	3	9
	1	4	16
	1	5	25
	1	6	36
	1	7	49

U			
	-0.017026	0.361281	0.794412
	-0.062044	0.478116	0.230802
	-0.135483	0.497914	-0.139430
	-0.237343	0.420673	-0.316284
	-0.367624	0.246395	-0.299761
	-0.526325	-0.024922	-0.089860
	-0.713448	-0.393276	0.313419

y	
	9
	18
	31
	48
	69
	94
	123

a	
	4
	3
	2

U Σ V^T			
	1	1	1
	1	2	4
	1	3	9
	1	4	16
	1	5	25
	1	6	36
	1	7	49

Σ			
	69.36802	0	0
	0	3.280315	0
	0	0	0.563888

traditional formula a	
	4
	3
	2

V			
	-0.029686	0.483545	0.874816
	-0.165597	0.860728	-0.481378
	-0.985747	-0.159157	0.054522

-LOG(B15/D17)	MpCond(B3:D9)
pκ = -2.09	pκ = -2.09

Fig. 10.14.2: A spreadsheet illustrating the individual steps in the solution of a least squares problem using both singular value decomposition and the traditional approach.

The matrices **X** in exercises 10.14.1 and 10.14.2 are perfectly well behaved, which is also reflected in their pκ values: you should be able to get answers correct to 13 or 14 decimals respectively. Now we are going to tinker with **X**, in order to make it increasingly ill-conditioned, and observe what happens. One way to do that is to make the third column in **X**, representing the x^2-values, approach the value x+1, in which case this column no longer contains linearly independent parameters, because the third column is then the algebraic sum of the corresponding values in the first two columns.

Exercise 10.14.3:

(1) Leave cell D3 alone, and change the instruction in cell D**4** to `=B4+C4`. Notice that not much changes with pκ in H23, and nothing with the coefficients in J12:J14 and J17:J20.

(2) Copy the instruction from D3 to D4:D8. Still not much change, because y_1, the lone holdout in cell D2, does not fit the expression x+1, and the pκ in cell H23 certainly does not indicate a very ill-conditioned, near-singular matrix. Of course, things would change if we were to insert a 2 in cell D3.

(3) Change the value in D3 from 1 to 1.9, and notice any changes. Then add another nine to make it 1.99, see what happens, add another to yield 1.999, etc.

(4) You should observe that pκ in F23 and H23 decrease by about 1 each time the difference between y_1 and 2 is reduced by an order of magnitude by adding another nine.

(5) Table 10.14.1 lists what happens with the retrieved least squares coefficients in J17:J19 with the traditional approach, and in J12:J14 with SVD.

Table 10.14.1 illustrates that singular value decomposition is much more immune to ill-conditioning than the traditional matrix inversion approach, and this observation is generally valid. With y_1 in cell D3 equal to 1.999 999 99, pκ = –9.6, and the least squares coefficients a_0 through a_2 with the traditional method are all far off, while those found with SVD are still good to six significant figures. Even when y_1 differs from 2 by only 1×10^{-12}, the values obtained for a_0 through a_2 are still within 0.2% of their correct values, whereas the standard method already exhibits errors of that order of magnitude when the deviation of the value in cell D3 from 2 is five orders of magnitude larger! Moreover, as long as y_1 differs from 2 by only 1×10^{-14}, the smallest possible difference the spreadsheet can display, the coefficients in a remain of the correct order of magnitude, whereas those in the last three columns of Table 10.14.1 start to fluctuate wildly already when p$\kappa \approx -8$. Section 10.15 will reinforce this message with a further analysis of the NIST reference data set Filip.dat. Note that **y** contains no experimental noise in this exercise, so that we only probe the numerical sensitivity to ill-conditioning of the algorithm used.

		singular value decomposition			standard matrix inversion		
y_1	pκ	a_0	a_1	a_2	a_0	a_1	a_2
1	–1.6	4	3	2	4	3	2
1.9	–2.6	4	3	2	4	3	2
1.99	–3.6	4	3	2	4	3	2
1.999	–4.6	4	3	2	4	3	2
1.999 9	–5.6	4	3	2	3.99998	2.99997	2.00001
1.999 99	–6.6	4	3	2	3.99548	2.99619	2.00390
1.999 999	–7.6	4	3	2	3.68331	2.44161	2.24540
1.999 999 9	–8.6	4	3	2	7.66292	10.02230	3.11833
1.999 999 99	–9.6	4	3	2	6.10507	8.33944	–0.60507
1.999 999 999	–10.6	3.99999	2.99999	2.00001	6.54691	3.93753	–1.45316
1.999 999 999 9	–11.6	3.99997	3.00002	2	5.64844	8.51563	0.65625
1.999 999 999 99	–12.6	4.00024	3	1.99976	–7.62500	8.87500	7.18750
1.999 999 999 999	*–13.6*	*4.00195*	*3.00391*	*2.00195*	*3.14063*	*1.17188*	*3.67187*
1.999 999 999 999 9	–14.6	4.07813	3.10938	1.90625	1.43750	5.35937	2.30469
1.999 999 999 999 99	–15.6	4.50000	3.25000	1.5	9.02344	10.21094	–2.45688
2	–16.6	12	5	–3	#NUM!	#NUM!	#NUM!

Table 10.14.1: The results obtained, to within ±0.00001, with the spreadsheets built in exercises 10.11.3 through 10.11.5 when y_i for i = 2 through 7 (in D3:D8) is redefined as $y_i = 1+x_i$, while the value of y_1 (in D2) gradually approaches the value 2 that would make **X** singular. The penultimate value for y_1 contains 15 decimals, i.e., 14 behind the decimal point; Excel truncates after 15 decimals and therefore either ignores the last-added 9 or reads 1. followed by more than 14 nines as 2. Values for a_0 through a_2 that end in .00000 are shown as integers.

Table 10.14.1 illustrates that singular value decomposition is much more immune to ill-conditioning than the traditional matrix inversion approach, and this observation is generally valid. With y_1 in cell D3 equal to 1.999 999 99, pκ = –9.6, and the least squares coefficients a_0 through a_2 with the traditional method are all far off, while those found with SVD are still good to six significant figures. Even when y_1 differs from 2 by only 1×10^{-12}, the values obtained for a_0 through a_2 are still within 0.2% of their correct values, whereas the standard method already exhibits errors of that order of magnitude when the deviation of the value in cell D3 from 2 is five orders of magnitude larger! Moreover, as long as y_1 differs from 2 by only 1×10^{-14}, the smallest possible difference the spreadsheet can display, the coefficients in a remain of the correct order of magnitude, whereas those in the last three columns of Table 10.14.1 start to fluctuate wildly already when p$\kappa \approx -8$. Section 10.15 will reinforce this message with a further analysis of the NIST reference data set Filip.dat. Note that **y** contains no experimental noise in this exercise, so that we only probe the numerical sensitivity to ill-conditioning of the algorithm used.

The approach sketched here is mathematically sound, but computationally dangerous, because a near-singular matrix **R** will have a large spread of singular values σ_i. In that case the smaller singular values σ_i in **Σ**, which percentage-wise are most strongly affected by round-off errors and data uncertainty, will become the dominant terms $1/\sigma_i$ in $\mathbf{\Sigma}^{-1}$. That is a prescription for obtaining nonsense, and computationally stable algorithms for $\mathbf{R}^+$ therefore avoid formally inverting **Σ** by replacing all its singular values σ_i by their inverses, $1/\sigma_i$.

10.20 Summary (AE3 pp. 522-523)

In principle, a spreadsheet is a near-ideal platform for one- and two-dimensional matrix operations, and the increase in spreadsheet area in Excel 2007, especially its widening from 256 to 16K columns, promises to make it even more suitable for this purpose, although many operations on large matrices will be too slow on the spreadsheet except for occasional use. Unfortunately, Microsoft provides only a meager set of matrix operations, which it has not updated since its market introduction in 1985, now more than a quarter of a century ago. The resulting void was filled admirably by Volpi's wide selection of additional matrix functions and macros. Readers with a need for matrix operations will find many more examples in the associated, quite extensive yet down-to-earth documentation accompanying these downloads: the *Tutorial of Numerical Analysis for Matrix.xla*, and the *Reference Guide for Matrix.xla*. We have especially emphasized least squares applications, as well as those involving eigenfunctions and their generalization in terms of singular value decomposition. The exercises in the latter areas can also

provide practice in handling complex numbers, vectors, and matrices on the spreadsheet. Linear algebra is used increasingly in solving all kinds of practical problems, and Volpi's add-ins are bound to facilitate a greater use of Excel in science, engineering, and statistics. The advantages of the spreadsheet in its widespread distribution, direct visual display, and ease of learning, must of course be balanced against the higher computational speed and larger choice of functions of dedicated programs such as Matlab.

Matrix operations are the heart and soul of *linear* algebra. But this term should not be taken to imply that all matrix operations are linear. Matrix inversion, e.g., is nonlinear in the same sense that the value of the algebraic inverse $1/x$ is not directly proportional to the value of the number x. Moreover, as we saw in section 10.10, mutual dependencies between variables (as expressed in their covariances v_{ij} and associated linear correlation coefficients r_{ij}, or in the ill-conditioning of one or more matrices) can lead to wild fluctuations and to seeming "outliers", as the result of the rather minuscule "noise" from computer round-off errors in floating-point arithmetic. In that sense they are somewhat similar to natural phenomena, such as the movements of tectonic plates, which are usually smooth but now and then lead to violent "catastrophes" such as earthquakes, tsunamis, and volcanic eruptions, events that we know from past experience to occur occasionally, but for which the statistics are too sporadic to allow for predictions with much accuracy in either space or time.

There are of course many systems that have multiple, mutually more or less strongly dependent variables: the movement of tectonic plates is an example from nature, but the economy is clearly a man-made one. Models for such systems are invariably based on past experience, and since catastrophes occur only occasionally and follow poorly understood statistics, they are hard to model and consequently are often left out. One should therefore be highly skeptical of systems of equations representing moderately understood phenomena, and the more so the more complicated these phenomena are, regardless of the sophistication of the matrix algebra used. If you consider the disappointing retrospective power of chemical analysis of single chemical species under carefully controlled conditions, as illustrated in section 4.26, what might a similar, critical test of the predictive power of large-scale economics look like?

It is tempting to put the above observations in a more general context. Much of our economy nowadays relies on econometric models, and we have recently learned, at a great cost to society, what can happen when those models are inadequate. As Gerd Gigerenzer, author of "*Calculated Risk*" (Simon & Schuster 2003), likes to say, such models typically do well at representing the past (with which they have been calibrated), but have problems predicting the future, because they do not incorporate unforeseen events.

Some of the financial catastrophes have been defended as unforeseeable, "ten sigma" outliers, but that is the typical language of combining *mutually independent* Gaussian distributions. Econometric models will often be based on many simultaneous, empirical equations, and will use these to generate a prediction. But even assuming that the input data are accurate (which empirical data seldom are), that the models are also correct (how can we know?), and that they were indeed valid "predictors" of the past at the time they were made, over time laws will be amended, rule enforcement may become lax, circumstances change, and so will the coefficients in those equations. Insofar as such changes affect the coefficients of near-singular matrices, they will occasionally lead to otherwise quite unexpected results. Earthquakes do happen, as do tsunamis and volcanic eruptions. And so do stock market crashes and other man-made disasters. The problem doesn't seem to lie so much in assuming a particular distribution of unavoidable fluctuations, as suggested by N. N. Talib in his "*Black Swan*" (Random House 2007), but in the profoundly nonlinear behavior of near-singular matrices, as easily hidden in large-scale econometric models. Such near-singularities reflect strongly correlated phenomena, of which stock market evaluations are a prime example, since they are affected both by hard data and by herd ("bull" or "bear") psychology. Unless we deal with systems that are inherently stable, the possibility that they can become instable cannot be discounted. And unless we realize the uncertainty and collinearity of our input data and assumptions, the combination of oversimplified computer models, wishful thinking and/or greed and hubris of some will again get all of us into trouble.

There is yet another factor to consider. While outliers are by definition rare events, their *risk* is the *product* of their low probability times their potential consequences. Rejecting outliers just because their probability of occurrence is low, regardless of the risk involved, is irresponsible, just as it would be to abolish the building codes in San Francisco because earthquakes there don't occur very frequently.

11.8 The error function (AE3 pp. 536-537)

The error function

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt \tag{11.8.1}$$

is another example of sloppy Excel programming. The error function is basic to both engineering and statistics, and appears in many problems of heat and mass transport that describe 'random walk' processes, such as heat conduction and molecular diffusion. As its name indicates, the error function is also related to the cumulative Gaussian ('normal') error distribution curve

$$\frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-(t-\mu)^2/2\sigma^2}\, dt = \frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right] \tag{11.8.2}$$

with mean μ and standard deviation σ.

Excel's error function takes two forms, one following the common definition (11.8.1), the other the rather unusual, two-parameter form

$$\mathrm{erf}(a,b) = \frac{2}{\sqrt{\pi}} \int_a^b e^{-t^2}\, dt = \mathrm{erf}(b) - \mathrm{erf}(a) \tag{11.8.3}$$

which we will not consider here, but which exhibits the same problem.

For negative values of x, we have the simple symmetry relation

$$\mathrm{erf}(-x) = -\,\mathrm{erf}(x) \tag{11.8.4}$$

while for positive values of x there is a straightforward, non-alternating power series

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} e^{-x^2} \sum_{n=0}^{\infty} \frac{2^n x^{2n+1}}{1 \cdot 3 \cdot \ldots \cdot (2n+1)} \tag{11.8.5}$$

For $x > 6$, we have $\mathrm{erf}(x) = 1$ to at least 15 figures, so that the function need not be computed. A VBA code reflecting these properties is shown below. Figure 11.8.1 shows the errors of the Excel function Erf(x) and of our cErf(x), and speaks for itself. Again, the results were checked against the same function computed with much longer numberlength. As with ASINH(), Excel 2010 has a corrected version for the error function. Even prodding a giant can sometimes work.

```
Function cErf(X)

' Based on Abramowitz & Stegun eq.(7.1.6)

Dim n As Integer
Dim Factor, Term, Y

If X < -6 Then
  Y = -1
  GoTo A
ElseIf X > 6 Then
  Y = 1
  GoTo A
ElseIf Abs(X) < 1E-8 Then
  Y = 2 * X / Sqr([Pi()])
  GoTo A
Else
  Term = X * Exp(-X * X)
  Y = Term
  n = 1
  Do
    Factor = 2 * X * X / (2 * n + 1)
    Term = Term * Factor
    Y = Y + Term
    n = n + 1
  Loop Until Abs(Term / Y) < 1E-50
  Y = 2 * Y / Sqr([Pi()])
End If

A:
cErf = Y

End Function
```

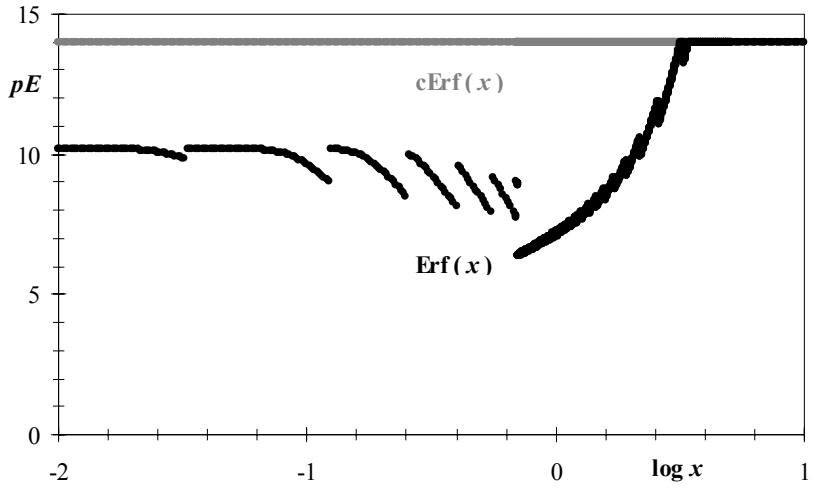


Fig. 11.8.1: Error plots of Erf(x) (black) and its corrected version cErf(x) (gray). For $x < 0$, Erf(x) gives no result, for $0 < x \delta 10^{-2}$ it is good to only 10 decimal figures (apparently because of an error in the value of $2/\sqrt{\pi}$ used), around $x = 0.707$ its accuracy dips to p$E = 6.3$, and for $x > 1.57\times10^{162}$ it again fails to give a result. Moreover, as shown here, the curve of Erf(x) vs. x shows a number of spurious discontinuities. By contrast, the function cErf is smooth and accurate to at least 14 significant decimal digits over the entire range of possible input values, 2. $2\times10^{-308} \le |x| \le 1.8\times10^{308}$.

11.9 Double precision add-in functions and macros (AE3 pp. 537-542)

Almost as a casual byproduct of his extended-precision XNumbers software, Leonardo Volpi generated a set of double-precision functions that were more accurate than their Excel equivalents, and also created many new functions not provided by Microsoft. John Beyers further added to these, making XN a quite large collection of useful functions that greatly extends the Excel toolkit. Of course, the extended precision functions listed in Table 11.11.1 and in appendix D can also yield double-precision answers when DgtMax is specified as 16, but it is better to use a larger value for DgtMax (such as 28 or 35) and to convert the results back to double precision with xCDbl.

Table 11.9.1 is a categorized listing of a number of double-precision *functions* that are accessible in this way, with a brief description, and Table 11.9.2 lists the double-precision *macros* available by clicking on the Macro button of the XN toolbar. For details on their operation, often with worked-out examples, see the XN manual, which you can access from Excel by pressing the Help button on the XN toolbar. If your monitor screen is wide enough to display the essential part of your spreadsheet and the help file side-by-side, without overlap, you can work in the spreadsheet while the help file shows. Many XN double-precision functions for operating with complex numbers were already listed in table 9.7.1, and are therefore not included here. Moreover, whole categories of functions in that table are not even discussed here, such as those for polynomial operations with names starting with "Poly", or the "ODE" solvers for *o*rdinary *d*ifferential *e*quations. If these interest you, read up on them in the XN Help menu.

The abbreviations A, H, and N in the right-most column of Table 11.9.1 refer to chapters or sections in A: the ***A***tlas of Functions by K. Oldham, J. Myland & J. Spanier, 2nd ed., Springer 2009, H: the ***H***andbook of Mathematical Functions edited by M. Abramowitz & I. Stegun, NBS 1964, reprinted many times since by Dover, and N: the ***N***IST Handbook of Mathematical Functions, edited by F. W. J. Olver, D. W. Lozier, D. F. Boisvert and C. W. Clark, Cambridge Univ. Press 2010.

function	brief description	symbol	refs.
Special functions			
AiryA	Airy function Ai(x)	$Ai(x)$	A56, H10.4, N9
AiryAD	First derivative of the Airy function Ai(x)	$Ai'(x)$	A56, H10.4, N9
AiryB	Airy function Bi(x)	$Bi(x)$	A56, H10.4, N9
AiryBD	First derivative of the Airy function Bi(x)	$Bi'(x)$	A56, H10.4, N9
BesselIx	Modified Bessel function of the 1st kind $I_n(x)$	$I_n(x)$	A49, H9, N10
BesseldI	First derivative of the modified Bessel function $I_n(x)$	$I_n'(x)$	A49, H9, *N10*
BesselJx	Bessel function of the 1st kind $J_n(x)$	$J_n(x)$	A52, H9, N10
BesseldJ	First derivative of the Bessel function $J_n(x)$	$J_n'(x)$	A52, H9, *N10*

BesselKx	Modified Bessel function of the 2nd kind $K_n(x)$	$K_n(x)$	A51, H9, N10
BesseldK	First derivative of the modified Bessel function $K_n(x)$	$K_n'(x)$	A51, H9, N10
BesselYx	Bessel function of the 2nd kind $Y_n(x)$	$Y_n(x)$	A54, H9, N10
BesseldY	First derivative of the Bessel function $Y_n(x)$	$Y_n'(x)$	A32.13, H10.1, N10
BesselSphJ	Spherical Bessel function of the 1st kind $J_n(x)$	$j_n(x)$	A32.13, H10.1, N10
BesselSphY	Spherical Bessel function of the 2nd kind $Y_n(x)$	$y_n(x)$	A32.13, H10.1, N10
DiGamma	Digamma function	$\psi(x)$	A44, H6.3, N5
ErrFun	Error function	erf(x)	A40, H7, N7
ErrFunC	Error function complement	erfc(x)	A40, H7,N7
HypGeom	Hypergeometric function	$F(a,b;c;z)$	A18.14, H15, H15
Zeta	Riemann zeta function	$\zeta(x)$	A3, H23.2, H25

Trigonometric operations

Serie_Trig	Generates a trigonometric series from its harmonics
Serie2D_Trig	Generates a 2-D trigonometric series from its harmonics

Polynomial operations

Poly	Evaluates polynomial at x
PolyAdd	Adds two polynomials
PolyBuild	Builds a polynomial for given roots
PolyCenter	Center of polynomial roots
PolyDiv	Quotient of two polynomials
PolyInt	Transformation to polynomial with same roots but integer coefficients
PolyInterp	Polynomial interpolation
PolyInterpCoef	Coefficients of polynomial interpolation
PolyMult	Product of two polynomials
Polyn	Value of polynomial $P(z)$ for specified real or complex argument z
Polyn2	Value of bivariant polynomial $P(w, z)$ for specified for real or complex arguments w, z
PolyRadius	Radius of polynomial roots
PolyRem	Remainder of polynomial division
PolyShift	Shifts a polynomial from x to $x + x_0$
PolySolve	Finds all roots of a polynomial
PolySub	Subtracts two polynomials
PolyTerms	Extracts vector of polynomial coefficients
PolyWrite	Constructs polynomial from its coefficients

Orthogonal polynomials

Poly_ChebychevT	Evaluates Chebychev polynomial of 1st kind	$T_n(x)$	A22, H22, N18
Poly_Weight_ChebychevT	Weight of Chebychev polynomial of 1st kind		A21, H22, N18
Poly_ChebychevU	Evaluates Chebychev polynomial of 2nd kind	$U_n(x)$	A22, H22, N18
Poly_Weight_ChebychevU	Weight of Chebychev polynomial of 2nd kind		A21, H22, N18
Poly_Gegenbauer	Evaluates Gegenbauer polynomial	$C_n^{(\alpha)}(x)$	A22, H22, N18
Poly_Weight_Gegenbauer	Weight of Gegenbauer polynomial		A21, H22, N18
Poly_Hermite	Evaluates Hermite polynomial	$H_n(x)$	A24, H22, N18
Poly_Weight_Hermite	Weight of Hermite polynomial		A21, H22, N18
Poly_Jacobi	Evaluates Jacobi polynomial	$P_n^{(\alpha,\beta)}(x)$	A22, H22, N18
Poly_Weight_Jacobi	Weight of Jacobi polynomial		A21, H22, N18
Poly_Laguerre	Evaluates Laguerre polynomial	$L_n(x)$	A23, H22, N18
Poly_Weight_Laguerre	Weight of Laguerre polynomial		A21, H22, N18
Poly_Legendre	Evaluates Legendre polynomial	$P_n(x)$	A21, H22, N18
Poly_Weight_Legendre	Weight of Legendre polynomial		A21, H22, N18

Special integrals

Convol	Convolution integral of two vectors	$\boldsymbol{v}_1 \otimes \boldsymbol{v}2$	
SinIntegral	Sine integral	$Si(x)$	A38, H5.2, N6
CosIntegral	Cosine integral	$Ci(x)$	A38, H5.2, N6
Exp_Integr	Exponential integral $Ei(x)$	$Ei(x)$	A37, H5, N6
Expn_Integr	Entire exponential integral $Ein(x)$	$Ein(x)$	A37, H5, N6
Fresnel_Sin	Fresnel sine integral	$S(x)$	A39, H7, N7
Fresnel_Cos	Fresnel cosine integral	$C(x)$	A39, H7, N7
IElliptic1	Elliptic integral of the 1st kind		A61, H17, N19
IElliptic2	Elliptic integral of the 2nd kind		A61, H17, N19
Kummer1	Confluent hypergeometric function of 1st kind	$M(a,b,x)$	A47, H13, N13
Kummer2	Confluent hypergeometric function of 2nd kind	$U(a,b,x)$	A48, H13, N13

Distributions	*default (dtype = 0) = probability function; dtype = 1 yields cumulative function*	
DSBeta	Beta distribution	A26.5
xBetaI	Incomplete beta function	
DSBinomial	Binomial distribution	
DSCauchy	Cauchy (Lorentz) distribution	
DSChi	Chi distribution	A26.4
DSErlang	Erlang distribution	
DSGamma	Gamma distribution	
xGammaI	incomplete gamma function	
DSLevy	Levy distribution	
DSLogistic	Logistic distribution	
DSLogNormal	Lognormal distribution	
DSMaxwell	Maxwell distribution	
DSMises	Von Mises distribution	
DSNormal	Normal (Gaussian) distribution	A26.2
DSPoisson	Poisson distribution	
DSRayleigh	Rayleigh distribution	
DSRice	Rice distribution	
DSStudent	Student distribution	A26.7
DSWeibull	Weibull distribution	
Least squares alternatives		
RegLinMM	Straight line fit minimizing the sum of absolute values	
RegLinRM	Straight line fit minimizing the deviations from the median	
Differentiation	*see section 9.2.11*	
Diff1	First derivative of $f(x)$	$f'(x)$
Diff2	Second derivative of $f(x)$	$f''(x)$
DPoly	Value of nth order derivative of polynomial $P(x)$ at specified x	
Dpolyn	Value of nth order derivative of polynomial $P(z)$ at specified real or complex argument z	
Grad	Vector of first derivatives of multivariate function	
Hessian	Hessian matrix of a multivariate function	
Jacobian	Jacobian matrix of a vector function	
Integration		
Integr	Integrates $f(x)$ between a and b	
Integr_DE	Integrates $f(x)$ using double exponential transformation	
Integr_2D	2-D integration of $f(x,y)$	
Integr_NC	Newton-Cotes integration of $f(x)$ between a and b	
Integr_Ro	Romberg integration of $f(x)$ between a and b	
IntegrData	trapezoidal integration of 1st, 3rd (default), or 5th degree	
IntegrData2D	bidimensional trapezoidal integration of a piecewise-rectangular grid	
IntegrDataC	Newton-Cotes integration of a complex data vector $\boldsymbol{v}$	
IntPowSin	Integrates $\sin^n(x)$	
IntPowCos	Integrates $\cos^n(x)$	
Fourier transformations		
FFT	Fast Fourier transform of $N = 2^n$ real input data	
FFT_INV	Fast inverse Fourier transform of 2^n real input data	
FFT2D	Fast 2-D Fourier transform	
FFT2D_INV	Fast 2-D inverse Fourier transform	
DFT	Discrete Fourier transform of any number N of real input data	
DFT_INV	Discrete inverse Fourier transform of any number N of real input data	
DFSP	Amplitude and phase angle determined by discrete Fourier transform	
DFSP_INV	The inverse of DFSP, viz. the reconstitution of $f(t)$ from the DFT amplitude and phase angle	
Fourier_Sin	Integrates $f(x)\sin(kx)$ from a to ∞	
Integr_fSin	Integrates $f(x)\sin(kx)$ with Filon's formula	
Fourier_Cos	Integrates $f(x)\cos(kx)$ from a to ∞	
Integr_fCos	Integrates $f(x)\cos(kx)$ with Filon's formula	
Equation solvers		
DiophEqu	Solves the diophantine equation $ax + by = c$	
PellEqu	Solves Brouncker-Pell equation $x^2 - dy^2 = 1$	
SysPoly2	Solves system of two 2nd-degree polynomials	
Zero_Bisec	Finds root of $f(x)$ by bisection	
Zero_Sec	Finds root of $f(x)$ with the secant method	

ODE solvers

ODE_COR	Solves ODE with multi-step corrector
ODE_PC2I	Solves ODE with implicit 2nd-order predictor-corrector
ODE_PC4	Solves ODE with 2nd-order Adams-Bashforth-Moulton method
ODE_PRE	Solves ODE with multi-step predictor
ODE_RK4	Solves ODE with 4th-order Runge-Kutta method
ODE_SYSL	Solves linear system of ODEs with constant coefficients

Interpolating tools

CSpline_Coeff	Coefficients of cubic spline
CSpline_Eval	Interpolates using fast cubic spline interpolation
CSpline_Interp	Interpolates using cubic spline interpolation
CSpline_Pre	Second derivative of cubic spline
FracInterp	Interpolation with continued fraction
FracInterpCoef	Coefficients of interpolation with continued fraction
Interp_Mesh	Linear interpolation in a rectangular mesh of data points

Extrapolating tools

ExtDelta2	Aitken's delta-square extrapolation

Primes

Factor	Decomposes an integer into its prime factors
NextPrime	First prime beyond n for primes < 2^{53}
PrevPrime	First prime before n for primes < 2^{53}
Totient	Euler's Totient function
Prime	Yields "P" if prime, smallest factor if not prime

Measures of data agreement

DgMat	Number of matching most-significant digits between x_1 and x_2
FDgMat	= pE using second argument as reference
LRE	= pE using second argument as reference
mjkLRE	= pE using second argument as reference

Data conversions

cvBinDec	Convert from binary to decimal
cvDecBin	Convert from decimal to binary
cvBaseDec	Converts from any base-n system ($1<n<27$) to decimal
cvDecBase	Converts from decimal to any base-n system ($1<n<27$)
s2Dbl	Converts string to double precision

Miscellany

dBel	Yields the decibel level, db(x) = 20 log(x)
Flip	Inverts order of vector elements
Fract	Approximates decimal fraction as a quotient of two integers
FractCont	Continued fraction of x in double precision
FractContSqr	Continued fraction of $\sqrt{n}$
MCD	Maximum (largest, greatest) common divisor
MCM	Minimum (smallest, least) common multiplier
SumDigits	Sums the values of the digits of an integer

Table 11.9.1: The double-precision functions in XN.xla(m). Advise: before first use of one of these functions, consult the relevant page in the Xnumbers v.6.0 Help file ⇨ Contents ⇨ Index of Functions, which you find on your computer by clicking on the Help button of the short XN toolbar.

macro	**brief description**	**symbol**
Matrix operations		
Transpose		$\mathbf{A}^T$
Add		$\mathbf{A} + \mathbf{B}$
Subtract		$\mathbf{A} - \mathbf{B}$
Scalar multiply		$k\,\mathbf{A}$
Matrix multiply		$\mathbf{A}\,\mathbf{B}$
Scalar product		$\mathbf{A}^T\,\mathbf{B}$
Invert		$\mathbf{A}^{-1}$
Determinant		
Similarity transformation		$\mathbf{B}^{-1}\,\mathbf{A}\,\mathbf{B}$

Solve a linear system		$\mathbf{A}\,\mathbf{X} = \mathbf{B}$
Solve an overdetermined linear system ($n > m$)		$\mathbf{A}\,\mathbf{X} = \mathbf{B}$
Norm		$\|\|\,\mathbf{A}\,\|\|$
Crout decomposition		LU
Cholesky decomposition		LL^T
Singular value decomposition		$\mathbf{U}\,\mathbf{D}\,\mathbf{V}^T$
Integration		
Double integration		
Triple integration		
Integration from $-\infty$ to $+\infty$		
Fourier transformation		
Discrete Fourier transformation		
2-D discrete Fourier transformation		
Sampler (of a specified function)		
Number theory		
Factors		
Prime number generation		
Prime test		
Integer relation finder	of polynomial or polyvariate relations	
Polynomials		
Rootfinder		
Builder		
Factors		
Orthogonal zero		
Orthogonal coeff.		
Optimizers		
Downhill simplex	Multivariate Nelder-Mead algorithm, robust & derivative-free	
Downhill simplex / Resets		
1-D divide & conquer	Monovariate, bisection-based algorithm, robust and derivative-free	
Least squares		
Linear		
Mesh fill		
ODE Solvers		
IVP Solver		
Slope grid		
Polynomial		

Table 11.9.2: The double-precision macros in XN.xla(m), accessible from the XN toolbar under Macros. For information, consult the relevant page in the Xnumbers v.6.0 Help file ⇨ Contents ⇨ Index of Macros.

11.11 The XN functions for extended precision (AE3 pp. 547-554)

Leonardo Volpi wrote two general programs using extended numberlength, implemented in VB and VBA as XNumbers.xla for incorporation in functions and directly in the spreadsheet, and as XNumbers.dll as a ***d***igital ***l***inking ***l***ibrary (dll) for use in macros. Both can still be downloaded from Volpi's web site, http://digilander.libero.it/foxes/SoftwareDownload.htm. However, they are no longer updated, and XNumbers.dll cannot be used in Excel 2007 or Excel 2010. Fortunately, John Beyers has greatly extended XNumbers.xla, made it readily usable in macros, relabeled this updated version XN.xla, and also made it available for Excel 2007 and 2010 as XN.xlam. These are freely downloadable from the website http.www.thetropicalevents.com/XNumbers60 and are relayed on my excellaneous website. It is this recent version of XNumbers.xla that will be used below, because it can be used everywhere in Excel: on the spreadsheet, and (after its installation in VBA) in functions and in subroutines, including macros. Installing XN.xla or XN.xlam is described in section 1.2.6. Moreover, you have already encountered its double-precision contributions in earlier chapters. Here we will emphasize its uses for high numerical precision.

The best explanation of all changes John Beyers has made in Xnumbers, plus a list of the many new functions he has added, can be found once XN and its XN toolbar are installed. You do the latter by clicking once on the XN book icon. (It is a toggle switch: clicking on the book icon once more will close the toolbar.) Then click on the Help button of that XN toolbar, select Help on-line, and in the resulting

Xnumbers version 6.0 file click at the end of its first paragraph on "changes to version 6.0". One special aspect of XN is that it can be configured. Fortunately, there is no need to change the pre-configured settings, which will do fine for most functions.

In what follows we will use XN.xla6051-7A, as of this writing the latest version in its simplest (and fastest) incarnation, which can operate with numbers of up to 630 decimals. Slightly slower versions with higher maximum precisions (of up to 4030 decimals) can also be downloaded, as can the corresponding Excel 2007/2010 versions of XN.xlam. Apart from their speed and maximum precision, all versions operate in identical fashion. Near the upper limit of available precision, the last few decimals are not always reliable, and it is therefore better to stay away from that very edge by two packets (i.e., by 14 decimals for 7-decimal packets) assigned in the XNToolbar under X-Edit ⇨ Configuration ⇨ Digit Max Adjustment as internal guard decimals. Also keep in mind that some XN functions can be rather slow at or near maximal precision, prime examples being xErf and xErfc, and the functions that use these: xNormal, xNormalS, and xMaxwell. By staying away from the limit by two packets as indicated above, the speed problem largely disappears: with DgtMax = 500, xErf computes its result in less than one second, and it is virtually instantaneous for DgtMax = 100. Operations on large matrices can also be slow, because the number of operations typically increases with some power larger than one of the number of elements involved. And especially slow can be iterative matrix computations, such as xSVD. But if you really need the result, what's a few seconds?

Personally I have never encountered a problem that required even 500 decimals of precision. I use 35 as my default value for all but the most difficult problems, and sometimes 28 to keep the numbers short.

Here are the advantages of XN:

(1) XN provides for a variable numerical precision. The desired precision can be specified for the entire spreadsheet, function, or subroutine, while still remaining adjustable for each individual instruction.

(2) XN can present its results either in full length or in regular, double precision format. In the latter case, i.e., when the output is converted back to double precision with an the instruction such as `=xCDbl`, it does not require any modifications of the input and output stages of already existing functions or macros, while the data processing is done in background at the selected higher precision.

(3) XN can be used directly on the spreadsheet. Many Excel instructions now have XN equivalents, including *all* of its engineering, statistical, and trigonometric functions. Moreover, there are many new functions. The required code changes in existing VBA functions and subroutines are relatively minor.

(4) If you want to make sure that data processing errors are kept to a minimum, and that at least the first 14 of the 15 decimals displaying your spreadsheet results are not distorted by limited computer precision, XN is your best bet.

Here are its disadvantages:

(1) While you can use XN functions on your spreadsheet, upgrading existing functions and macros to extended precision is only possible when you have access to their source code. Fortunately, that automatically includes your own, custom-made creations, and all open-access functions and macros (including those in the MacroBundle) that you can find in the literature.

(2) XN can be noticeably slow in execution, especially when we combine high precision with repeated use, as in multiply nested loops. This reflects the trade-off between speed and accuracy.

(3) If you want to share your active spreadsheets with colleagues (rather than just its results as, say, a pdf), they will need to install XN.xla(m) on their machines. Since XN is free, that is not much of a disadvantage, unless your institution blocks it. In that case, consult your IT manager..

While these disadvantages are rather minor compared with the possible gains, I do not recommend that you use XN for routine use, where it may often be overkill. Keep it for critical applications, and for double-checking that your intermediary and final results do not lose needed accuracy to numerically insufficient precision. Therefore, its principal applications will likely be to problems where double precision is known to produce substandard results, typically because of algorithmic cancellation and/or accumulation errors. XN can also be very useful as a final check on numerical results of consequence, and as a high-precision reference for testing algorithms and their spreadsheet implementations. Even such occasional uses fully warrant its inclusion in a book on *advanced* Excel.

Below you will find a short alphabetical list of the XN functions currently available in the XN.xla(m) downloads. In an effort to streamline the nomenclatures used in Matrix.xla(m) and XN.xla(m), some function names have recently been modified, such as xMatMult for extended-

precision matrix inversion, which has become xMMult, to be more compatible with MMULT in Excel and with xMMultC and xMMultSC in XN. Check in your Paste Function (or Insert Function) dialog box which commands your version expects; the 13 function names which used Mat instead of M to specify that they were matrix functions are identified below with asterisks. Earlier versions are grandfathered, and are still listed in the Paste Function dialog box. They will be translated automatically into their most recent names, so that spreadsheets using older function names will still work properly as spreadsheet *functions*. But VBA has become more demanding, and *macros* using older names may have to be updated to use their current names. Not featured here (but listed in the PasteFunction box) are a number of functions with the last letter R, which stands for "raw" and are meant for use inside VBA programs, where they are some 10% faster.

As for all Excel functions, capitalization is ignored by Excel (except in quoted text), and is used here merely to enhance readability by emphasizing the different parts of compound terms. A categorized listing with somewhat greater detail can be found in appendix D, and specific examples are often available by consulting the on-screen Help file accessible after you have loaded the small XN toolbar.

extended precision function	***brief function description***	***Excel double precision equivalent***
vRoundR	relative banker's rouding, default: to 15 decimal relative precision	
x_And	Boolean logic AND	AND
x_If	Boolean logic IF, including condition test on numeric strings	IF
x_Not	Boolean logic NOT	NOT
x_Or	Boolean logic OR	OR
x2Dbl	converts to double precision, slower than xCDbl, more accurate, but seldom needed	
x2Pi	$2\pi \approx 6.283\ldots$	
xAbs	absolute value	ABS
xACos	inverse cosine of an angle, =arcos(α)	ACOS
xACosH	inverse hyperbolic cosine of a number, =arcosh(x)	ACOSH
xAdd	add	+
xAddMod	modular addition	
xAdj2Pi	adjusted angle, in rad between 0 and $+2\pi$	
xAdjPi	adjusted angle, in rad between $-\pi$ and $+\pi$	
xALog	10-based antilogarithm, 10^x	
xAngleC	complement of angle α, $=\pi/2 - \alpha$	
xASin	inverse sine of an angle, =arcsin(α)	ASIN
xASinH	inverse hyperbolic sine of a number, =arsinh(x)	ASINH
xATan	inverse tangent of an angle, =arctan(α)	ATAN
xATan2	arctan in rad between $-\pi$ and $+\pi$	
xATanH	inverse hyperbolic tangent of a number, =artanh(x)	ATANH
xAveDev	average of the absolute deviation from the mean	AVEDEV
xBaseChange	base converter	BASECHANGE
xBeta	beta function	
xBinomial	binomial distribution	BINOMDIST
xCalc	formula evaluator or parser	
xCat	concatenate	&
xCDbl	convert to double precision	
xCeil	ceiling	CEILING
xClip	yields Ceil for $T \geq$ Ceil, T for Floor $< T <$ Ceiling, and Floor for $T \leq$ Floor	
xComb	binomial coefficient	
xComb_Big	binomial coefficient for large numbers	
xComp	comparison, yields 1 for $a > b$, 0 for $a = b$, and –1 for $a < b$; yields sign of a if b absent	IF
xComp1	absolute comparison, yields 1 for xAbs > 1, 0 for xAbs = 1, and –1 for xAbs < 1	
xCorrel	correlation coefficient	CORREL
xCos	cosine of an angle, =cos(α)	
xCosH	hyperbolic cosine of a number, =cosh(x)	
xCoVar	covariance	COVAR
xCplx	builds complex number z from real and imaginary component, selects i or j as $\sqrt{-1}$.	COMPLEX
xCplxAbs	absolute value of complex number, $\lvert z \rvert$	IMABS
xCplxACos	complex inverse cosine, arcos(z)	

xCplxACosH	complex inverse hyperbolic cosine, arsinh(z)	
xCplxAdd	complex addition	
xCplxArg	argument of complex number, arctan(Im/Re)	IMARGUMENT
xCplxASin	complex inverse sine, arcsin(z)	
xCplxASinH	complex inverse hyperbolic sine, arsinh(z)	
xCplxATan	complex inverse tangent, arctan(z)	
xCplxATanH	complex inverse hyperbolic tangent, artanh(z)	
xCplxConj	complex conjugate	IMCONJUGATE
xCplxCos	complex cosine, cos(z)	IMCOS
xCplxCosH	complex hyperbolic cosine, cosh(z)	
xCplxDiv	complex division	IMDIV
xCplxExp	complex exponentiation, e^z	IMEXP
xCplxInv	complex inversion, $1/z$	
xCplxLn	complex natural logarithm, ln(z)	IMLN
xCplxLog	complex logarithm to any base n, $\log_n(z)$	
xCplxLog10	complex 10-based logarithm, log(z)	IMLOG10
xCplxLog2	complex 2-based logarithm, $\log_2(z)$	IMLOG2
xCplxMult	complex multiplication	
xCplxNeg	complex negation, $-z$	
xCplxPolar	converts complex number from rectangular to polar	
xCplxPow	raises complex number to integer power n, z^n, default $n = 2$	IMPOWER
xCplxRect	converts complex number from polar to rectangular	
xCplxRoot	n^{th} root of complex number, $z^{1/n}$, default $n = 2$	
xCplxSin	complex sine, sin(z)	IMSIN
xCplxSinH	complex hyperbolic sine, sinh(z)	
xCplxSqr	complex square root, $\sqrt{z}$	IMSQRT
xCplxSub	complex subtraction	IMSUB
xCplxTan	complex tangent, tan(z)	
xCplxTanH	complex hyperbolic tangent, tanh(z)	
xCStr	converts a double-precision number to a string, default Digit_Max = 767	
xCvExp	converts into scientific notation, x = mantissa $\times 10^{\text{exponent}}$, default exponent = 0	
xDec	decimal part of number	
xDecr	decrements a number by 1, xSub(x,1)	
xDegrees	converts radians into degrees	DEGREES
xDelta	tests whether two numbers are equal	DELTA
xDevSq	sum of squares of deviations from sample average	DEVSQ
xDgMat	number of matching digits	
xDgt	the number of digits	
xDgtS	the number of significant digits	
xDiff	n^{th}-order derivative ($1 \le n \le 20$) of a function $f(x)$, default $n = 1$	
xDiff1	first-order derivative with choice of forward, central, or backward differencing	
xDiff2	second-order derivative with choice of forward, central, or backward differencing	
xDiffI	n^{th}-order derivative with choice of forward, central, or backward differencing	
xDiffOpt	displays n^{th}-order derivative plus values of h and d used	
xDiv	division	/
xDivInt	integer division	\
xDivMod	modular division	
xDivTrunc	truncated quotient Q = xTrunc(N/D)	
xDLGI	1^{st} derivative of Lagrange interpolation polynomial at specified x-value	
xDPoly	1^{st} derivative of polynomial at specified x-value	
xE	e ≈ 2.718…, the base of natural logarithms	
xErF	error function	ERF
xErFC	error function complement	ERFC
xEu	Euler's gamma	
xEval	formula parser, evaluates quasi-algebraic expressions	
xEvall	much slower version of xEval, can associate labels with input data	
xEven	rounds to nearest even number	EVEN
xExp	exponentiation, e^x	EXP
xExpa	raises a to the power x, a^x	
xExpBase	raises a to the power x, a^x	
xExponent	the exponent of a number x = mantissa $\times 10^{\text{exponent}}$	

xExtDelta2		Aitken's delta-square extrapolation	
xFact		factorial, n!	FACT
xFact2		double-step (odd/even) factorial	FACTDOUBLE
xFib		Fibonacci number	
xFisher		Fisher transformation	FISHER
xFishInv		inverse Fisher transformation	FISHERINV
xFloor		rounds towards zero to nearest significant multiple	FLOOR
xFmt		formats string with selectable lead zeroes, trailing zeroes, and rounding	
xFormat		formats number in groups of *n* digits	FORMAT
xFract		numerator and denominator of a fraction	
xFractCont		continued fraction	
xGamma		gamma function, $\Gamma(x)$	GAMMADIST
xGammaF		Fisher's gamma distribution	
xGammaLn		natural logarithm of the gamma function, $\ln \Gamma(x)$	GAMMALN
xGammaLog		10-based logarithm of the gamma function, $\ln \Gamma(x)$	
xGammaQ		ratoio of two gamma functions, $\Gamma(x_1) / \Gamma(x_2)$	
xGEstep		threshold comparison, = 1 if $x \geq$ step, otherwise = 0	GESTEP
xGm		Euler's gamma, =xEu	
xGMmean		geometric mean	GEOMEAN
xGrad		gradient vector, i.e., first derivatives of a function $f(x,y,\ldots)$	
xHMean		harmonic mean	HARMEAN
xImag		imaginary part of complex number	IMAGINARY
xIncr		increments by 1, =xAdd(*x*,1)	
xInt		integer part of number	
xIntercept		intercept of least squares straight line with *y*-axis	INTERCEPT
xIntLog10		=Int[log(|*x*|)/log(10)]	
xIntLog2		=Int[log(|*x*|)/log(2)]	
xIntMod		remainder R = sign(D) * { N – D * xInt(N/D) }	
xIntQR		quotient Q = xInt(N/D) and remainder R = N – D * Q	
xInv		inverse, 1/*x*	
xIsErr		0 if no errors in range, otherwise 1	
xIsErrNA		0 if no errors in range, otherwise 1, ignoring N/A errors	
xIsEven		TRUE if trunc(x) = even, FALSE if odd	ISEVEN
xIsInteger		TRUE if integer, FALSE if not	
xIsNumeric		= 0 for non-numeric, 1 for double precision, 2 for extended precision	
xIsOdd		TRUE if trunc(x) = odd, FALSE if even	ISODD
xIsSquare		TRUE if perfect square	
xIsXNumber		TRUE if extended precision number that cannot be converted into double precision	
xJacobian		Jacobian matrix of a vector function	
xIElliptic1		elliptic integral of 1st kind	
xIElliptic2		elliptic integral of 2nd kind	
xLGI		interpolates Lagrange polynomial	
xLn		natural logarithm	LN
xLn10		= ln(10) ≈ 2.302...	
xLn2		= ln(2) ≈ 0.693…	
xLog		logarithm of any integer base *n*, $\log_n$, default *n* = 10	LOG
xLogistic		logistic distribution	
xLogNorm		lognormal distribution	
xLRE		negative logarithm of relative or absolute error	
xMAbs		modulus of matrix or vector, $\|\| \mathbf{M} \|\|$	
xMAbsC		Euclidian norm of complex matrix **C**	
xMAdd		matrix addition, = $\mathbf{M}_1 + \mathbf{M}_2$	
xMAddC		complex matrix addition, = $\mathbf{C}_1 + \mathbf{C}_2$	
xMantissa		mantissa of a number x = mantissa $\times 10^{\text{exponent}}$	
xMatAbs	*	modulus of matrix or vector, $\|\| \mathbf{M} \|\|$, now xMAbs	
xMatAdd	*	matrix addition, = $\mathbf{M}_1 + \mathbf{M}_2$, now xMAdd	
xMatBAB	*	similarity transformation, = $\mathbf{B}^{-1}\,\mathbf{A}\,\mathbf{B}$, now xMBAB	
xMatDet	*	determinant of a matrix, now xMDet	MDETERM
xMatInv	*	inversion of a square matrix, = $\mathbf{M}^{-1}$, now xMInv	MINVERSE
xMatLL	*	Cholesky decomposition, = $\mathbf{L}\,\mathbf{L}^{H}$, now xMCholesky	
xMatLU	*	lower/upper decomposition = **L U** with Crout's algorithm, now xMLU	MLU

xMatMult	*	matrix multiplication, $\mathbf{M}_1\,\mathbf{M}_2$, now xMMult	MMULT
xMatPow	*	raising a square matrix to an integer power n, = $\mathbf{M}^n$, now xMPow	
xMatSub	*	subtraction of two real matrices, = $\mathbf{M}_1 - \mathbf{M}_2$, now xMSub	
xMax		maximum value in a range of values	
xMaxwell		Maxwell-Boltzmann distribution	
xMBAB		similarity transformation, = $\mathbf{B}^{-1}\,\mathbf{A}\,\mathbf{B}$	
xMCD		maximum common divisor = greatest common denominator	GCD
xMCharC		characteristic polynomial of complex matrix	
xMCholesky		Cholesky decomposition, = $\mathbf{L}\,\mathbf{L}^{\mathrm{H}}$	
xMCM		minimum common multiple = least common multiple	LCM
xMCond		condition number κ of a real matrix	
xMCondC		condition number κ of a complex matrix	
xMCplx		converts two real matrices into one complex matrix	
xMDet		determinant of a matrix	MDETERM
xMDetC		determinant of a complex matrix	
xMean		arithmetic mean	AVERAGE
xMedian		median	MEDIAN
xMExp		matrix series expansion	
xMExpC		complex series expansion for exp(**C**)	
xMExpErr		truncation error of xMatExp	
xMExpErrC		truncated error of complex series expansion for exp(**C**)	
xMin		minimum value in a range of values	
xMInv		inversion of a square matrix, = $\mathbf{M}^{-1}$	MINVERSE
xMInvC		regular inverse of complex square matrix, =$\mathbf{C}^{-1}$	
xMLU		lower/upper decomposition = **L U** with Crout's algorithm	MLU
xMMopUp		removes matrix elements a_{ij} when $\lvert a_{ij}\rvert <$ ErrMin	
xMMult		matrix multiplication, $\mathbf{M}_1\,\mathbf{M}_2$	MMULT
xMMultC		multiplies two complex matrices, $\mathbf{C}_1$+$\mathbf{C}_2$	
xMMultCS		multiplies a complex matrix and a complex scalar, = **C S**	
xMNormalize		normalizes the column vectors of a real matrix	
xMNormalizeC		normalizes the column vectors of a complex matrix	
xMode		most frequently occurring number in range of numbers	MODE
xMpCond		negative 10-based logarithm of condition number pκ of a real matrix κ	
xMpCondC		negative 10-based logarithm of condition number pκ of a complex matrix	
xMPow		raising a square matrix to an integer power n, = $\mathbf{M}^n$	
xMPowC		raise complex square matrix **C** to an integer power n, =$\mathbf{C}^n$	
xMPseudoInv		pseudo-inverse of matrix based on real SVD	
xMPseudoInvC		pseudo-inverse of matrix based on complex SVD	
xMRound		rounds a number to the nearest multiple	MROUND
xMSub		subtraction of two real matrices, = $\mathbf{M}_1 - \mathbf{M}_2$	
xMSubC		subtraction of two complex matrices, = $\mathbf{C}_1 - \mathbf{C}_2$	
xMTC		transpose of a complex matrix **C**	
xMTH		Hermitian (conjugate, adjoint) transpose of a complex matrix **C**	
xMult		multiplication	*
xMultinom		generates a multinomial	MULTINOMIAL
xMultMod		modular multiplication	
xNeg		negation, = –x	–
xNormal		normal distribution	NORMDIST
xNormalS		cumulative normal distribution	10 : NORM.S.DIST
xNormS		cumulative standard normal distribution	NORMSDIST
xOdd		rounds up to the nearest odd integer	ODD
xOddDen		checks if denominator is odd (TRUE) or even (FALSE)	
xPearson		Pearson product moment correlation coefficient	PEARSON
xPerm		the number of possible permutations	PERMUT
xPi		π ≈ 3.141…	PI()
xPi2		π / 2 ≈ 1.570…	
xPi4		π / 4 ≈ 0.785…	
xPoly		evaluates polynomial $P(x)$ at specified x	
xPolyAdd		adds two polynomials in x	
xPolyDiv		divides two polynomials in x	
xPolyMult		multiplies two polynomials in x	

xPolyRem	the remainder of polynomial division	
xPolySub	subtracts two polynomials in x	
xPolyTerms	extracts polynomial coefficients as a vector	
xPow	raises number to power n, XN, n integer for $x < 0$	
xPow2	integer powers of 2, 2^n	
xPowMod	modular power, $a^p \bmod m$	
xProd	the product of two or more numbers	PRODUCT
xProdScal	scalar product of real vectors	
xProdScalC	scalar product of complex vectors	
xProdVect	vector product of two 3D vectors	
xQMean	quadratic mean	
xRad12	$=\sqrt{12}$	
xRad5	$=\sqrt{5}$	
xRadians	converts degrees to radians	RADIANS
xRand	generates random numbers with uniform distribution $U(0,1)$ between 0 and 1	RAND
xRandD	generates random numbers with uniform distribution $U(a,b)$ between a and b	RANDBETWEEN
xRandI	generates random numbers with uniform distribution $U(m,n)$ between integers m and n	
xRank	the rank of a number in a list of numbers	RANK
xRayleigh	Rayleigh distribution	
xRDown	rounds towards zero, default rounds to an integer	ROUNDDOWN
xReal	real component of complex number	IMREAL
xRegLinCoef	coefficients of a multivariate least squares fit	
xRegLinCov	covariance matrix of the coefficients of a multivariate least squares fit	
xRegLinErr	standard deviations of the coefficients of a multivariate least squares fit	
xRegLinEval	evaluates the least squares results at a specified value of x	
xRegLinStat	R^2 and standard deviation s_f of a multivariate least squares fit	
xRegPolyCoef	coefficients of the least squares fit to an internally computed polynomial in x	
xRegPolyErr	standard deviations of the coefficients of the least squares fit to that polynomial in x	
xRegPolyStat	R^2 and standard deviation of the least squares fit to that polynomial in x	
xRegrL	coefficients of a SVD-based multivariate linear least squares	
xRegrLC	coefficients of a SVD-based complex multivariate linear least squares	
xRoot	n^{th} root, $=x^{1/n}$	
xRound	rounding, with ending-5 rounded away from zero, default rounds to an integer	ROUND
xRoundR	relative rounding, default to 15 decimals	
xRSq	square of the Pearson product moment correlation coefficient	RSQ
xRUp	rounds away from zero, default rounds up to an integer	ROUNDUP
xSerie	generates a series for $f(x)$ over a specified range in x	
xSerie2D	generates a doubleseries for $f(x,y)$ over specified ranges in x and y	
xSerSum	sums a power series	SERIESSUM
xSin	sine of an angle, $=\sin\alpha$)	SIN
xSinH	hyperbolic sine of a number, $=\sinh(x)$	SINH
xSlope	slope of least squares fit to a straight line	SLOPE
xSplit	splits a number x = mantissa $\times 10^{\text{exponent}}$ into its mantissa and exponent	
xSqr	square root, $=\sqrt{x}$	SQRT
xSqrPi	square root of pi, $=\sqrt{\pi}$	
xStatis	univariate statistical summary	
xStDev	sample standard deviation of repeat measurements	STDEV
xStDevP	population standard deviation of repeat measurements	STDEVP
xSub	subtract	–
xSubMod	modular subtraction	
xSum	sum of a range of cells	SUM
xSumProd	sum product of two cell ranges	SUMPRODUCT
xSumSq	sum of squares of terms in an array	SUMSQ
xSumX2mY2	sum of the differences of squares of corresponding terms in two arrays	SUMY2MY2
xSumX2pY2	sum of the sum of squares of corresponding terms in two arrays	SUMY2PY2
xSumXmY2	sum of the squares of the differences between corresponding terms in two arrays	SUMYMY2
xSVDD	$\mathbf{\Sigma}$ matrix of singular value decomposition $\mathbf{M} = \mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^{\mathrm{T}}$ of a rectangular matrix $\mathbf{M}$ with only real components	
xSVDDC	$\mathbf{\Sigma}$ matrix of singular value decomposition $\mathbf{M} = \mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^{\mathrm{H}}$ of a rectangular matrix with complex components	
xSVDU	$\mathbf{U}$ matrix of singular value decomposition $\mathbf{M} = \mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^{\mathrm{T}}$ of a rectangular matrix with only real components	
xSVDUC	$\mathbf{U}$ matrix of singular value decomposition $\mathbf{M} = \mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^{\mathrm{H}}$ of a rectangular matrix with complex components	
xSVDV	$\mathbf{V}$ matrix of singular value decomposition $\mathbf{M} = \mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^{\mathrm{T}}$ of a rectangular matrix with only real components	

xSVDVC	**V** matrix of singular value decomposition $\mathbf{M} = \mathbf{U}\,\mathbf{\Sigma}\,\mathbf{V}^H$ of a rectangular matrix with complex components	
xSysLin	Gauss-Jordan solution of linear system **A x** = **B**	
xSysLinC	Gauss-Jordan solution of complex linear system **A x** = **B**	
xTan	tangent of an angle, =tan(α)	TAN
xTanH	hyperbolic tangent of a number, =tanh(x)	TANH
xTrunc	truncation to fixed number n of placed past the decimal point, default n = 0	TRUNC
xTruncMod	remainder R = N – D * xTrunc(N/D), with sign of numerator N	
xTruncQR	truncate (number, divisor) to yield (quotient Q, remainder R)	
xTruncR	truncate to fixed number n of significant digits	
xUnformat	removes all formatting characters	
xVar	sample variance of a set of repeat measurements	VAR
xVarP	population variance of a set of repeat measurements	VARP
xWeibull	Weibull distribution	
xZeta	zeta distribution	

Table 11.11.1: Alphabetical listing of the XN functions with extended precision (beyond quadruple), readily identifiable by their prefix x. Some "raw" duplicate functions (with last letter R) have been omitted. For the sake of uniformity with the function names in Matrix.xla(m), those indicated with * were updated to function names that started with xM instead of xMat, but were grandfathered so that you can still use the older spreadsheets.

11.15 Filip, once more (AE3 pp. 578-581)

We will use Filip.dat to test xnLS, because it has been modified specifically to accommodate high powers of high-precision x-values by computing x^n inside the macro. Table 11.15.1 shows our results, for various D-values, in terms of their pE-values based on the NIST values as reference. As you can see in that table, we find complete agreement with the NIST-published values for all coefficients and all standard deviations, i.e., to pE = 15, when we use $D \geq 34$. At D = 18 we obtain pE = 0, in agreement with the total failure of LS to solve Filip.dat in double precision with the traditional least squares algorithm. The contrast could not be starker!

Both xRegPolyCoeff and xnLS clearly illustrate the power and convenience of XN. With the relatively small additions of negating the distortion involved in binary data storage, where necessary computing the higher powers of x internally, and replacing a limited number of instructions by the corresponding XN functions, we have transformed a primitive custom algorithm LS that could not get a single coefficient of Filip.dat right to one that passes this most difficult of all NIST linear least squares tests. No specialized knowledge about the subtleties of matrix inversion (such as embedded in singular value decomposition) are needed. Our "brute force" approach to use higher computer precision with an inferior algorithm certainly beats most commercial statistical software packages in fitting Filip.dat. Not bad for free, open-access spreadsheet software. The various results obtained in this book with Filip.dat are summarized in Table 11.15.2.

D	p$E(a_i)$	p$E(s_i)$	D	p$E(a_i)$	p$E(s_i)$	D	p$E(a_i)$	p$E(s_i)$
18	0.00	0.00	24	5.11	5.45	30	11.53	11.85
19	0.00	0.00	25	5.99	6.32	31	12.99	13.34
20	2.02	0.17	26	6.92	7.25	32	13.39	13.68
21	2.58	0.69	27	8.33	8.64	33	14.03	14.46
22	2.95	2.88	28	9.07	9.41	**34**	**15.00**	**15.00**
23	5.55	3.97	29	10.52	10.85	**35**	**15.00**	**15.00**

Table 11.15.1: The dependence of the results obtained by xnLS with Filip.dat to the NIST StRD reference values for the coefficients a_i and the standard deviations s_i, as measured by its lowest pE-value in each category as a function of the specified decimal precision D used. For $D > 35$ the pE is always 15.

software used	least squares coefficients a_i	standard deviations s_i	covariances v_{ij}
classical least suares (LS)	0	0	0
pre-2003 LinEst and Regression	0	0	–
BigMatrix linear regression	6.4	–	–
singular value decomposition (SVD)	7.7	–	–
post-2003 LinEst and Regression	7.7	7.5	–
LS + polynomial centering	10	–	✔
SVD + polynomial centering	13.7	–	–
post-2003 LinEst + polynomial centering	13.7	7.5	–
BigMatrix polynomial regression[#]	14	–	–
xRegPolyCoef[#] $D \geq 34$	**15**	–	–
xnLS1[#] with $D \geq 34$	**15**	**15**	✔

Table 11.15.2: A score card of linear least squares analyses of Filip.dat in terms of the error criterion pE_{min}. Only LS and xnLS provide the covariance matrix, but these are merely check-marked here, because NIST provides no reference values for comparison. $D = 35$ is our recommended default (quintuple) precision. Routines that compute x^n internally are indicated with [#].

test data set	D	test data set	D	test data set	D
Norris	19	Filip	34	Wampler3	25
Pontius	20	Longley	23	Wampler4	25
NoInt1	16*	Wampler1	28	Wampler5	25
NoInt2	16*	Wampler2	23		

Table 11.15.3: The minimum number of decimals D needed to get all coefficients a_i, their standard deviations s_i, and the standard deviation of the fit s_f, to all specified 15 decimals with xnLS. *: p$E = 15$ is already obtained with the standard double-precision macro LS.

Table 11.15.3 shows how many decimals xnLS needs for each of the eleven NIST StRD linear least squares test data to get all its least squares coefficients and standard deviations correct to all 15 decimals specified by NIST. Note that you don't always need extended precision to get p$E = 15$: with NoInt1 and NoInt2 you can already obtain this with the standard double-precision macro LS. As our test we used the function xpE based on (9.1.1), but modified here by making that criterion readily user-selectable (as the value of C in the function code), and using the more stringent precision criterion of 15 rather than the 14 used in (9.1.2). The actual function xpE used for Table 11.15.3 relies on relative rounding, and is shown here:

```
Function xpE(number, reference, Optional vRR)
' vRR is the value for Relarive Rounding used, here with
' a default of 15 (decimals) for "double precision"
Dim N, R, pE
If IsMissing(vRR) Then vRR = 15
N = vRoundR(number, vRR)
R = vRoundR(reference, vRRC)
If xComp(N, R) = 0 Then pE = C
If (xComp(N, R) <> 0 And xComp(R, 0) = 0) Then _
  pE = xNeg(xLog(xAbs(N)))
If (xComp(N, R) <> 0 And xComp(R, 0) <> 0) Then _
  pE = xNeg(xLog(xAbs(xDiv(xSub(N, R), R))))
If xComp(pE, 0) <= 0 Then pE = 0
If xComp(pE, C) = 1 Then pE = vRR
xpE = vRoundR(pE, vRR)
End Function
```

Given the central role of least squares analysis in this book, it was perhaps fitting to return to Filip.dat as our last example. We encountered the traditional least squares formalism in section 10.5, but its implementation in our custom macro LS could not make any sense of Filip.dat, and neither could the pre-2003 Excel routines LinEst and Regression, which were likewise based on the classical pseudo-inverse $(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T$. Singular value decomposition described in section 10.11 made it possible to find the coefficients a_i to p$E \geq 7.5$, and the same applied to Excel's LinEst and Regression since their 2003 update, because they use the related QR factorization. Further progress could be made by polynomial centering, see

section 10.7, and their combined results came close to the NIST reference data for the coefficients a_i, with a minimum pE of 13.7, i.e., with relative errors no larger than 2×10^{-14}. BigMatrix has a macro that, with quadruple precision and internally computing the powers of x, can even achieve slightly better results, with p$E \geq 14$ for both the coefficients a_i and their standard deviations s_i. However, all of these would still leave us without the covariances v_{ij}. In section 11.14 we therefore modified LS with XN, in order to suit all of our requirements.

Why so much emphasis on the covariances? First, look at the data obtained from Filip.dat for not only a_i but also s_i. The ratios a_i/s_i are less than five for all eleven coefficients, indicating that the answers are not very precise. But let's assume that Filip.dat reflects actually observed data, as claimed by NIST. In that case we can either fit them to a simple empirical model, using as few parameters as possible, to describe the overall character of the results of our measurement, and be able to interpolate them and, perhaps, even extrapolate them to adjacent x-values, or we can use a theoretical framework. In the first case, a simple empirical fit to the general shape of the curve would certainly not use a power series in x, see *Am. J. Phys*. 75 (2007) 617, especially since a tenth-order polynomial will start to oscillate violently just outside the measurement range, making it unsuitable for extrapolation, and even questionable for interpolation. The polynomial model therefore implies some theoretical basis, in which the parameters obtained should be interpretable. And therein lies the rub: if any conclusions drawn from the fitting coefficients a_i involve more than one coefficient, we definitely need the covariances, because the standard deviations by themselves are then powerless to provide reliable uncertainty estimates. Why? Just do the fit with xnLS, observe its output on your monitor screen or send the output to a color printer, and look at the linear correlation coefficients r_{ij}, as defined in (2.10.2), between the various coefficients a_i. It is a veritable sea of red ink, indicating that these coefficients show very high mutual dependencies. Every r_{ij} of adjacent powers of x (i.e., every r_{ij} for which $j = i \pm 1$) is at least 0.9996, every r_{ij} where $j = i \pm 2$) is at least 0.9985, etc., and even the least strongly coupled coefficient, that between a_0 and a_{10}, has an r_{ij} of 0.965. The mutual dependence of these coefficients is so strong that, without their covariances, no valid uncertainties can be assigned to any functions depending on two or more a_i-values.

Incidentally, you might well ask how realistic Filip.dat is as a test set. NIST labels it as from an "observed" rather than a "generated" source, but that does not necessarily exclude a large amount of doctoring. One seldom encounters raw experimental data that are significant to ten decimal places, with such uniformly high covariances, that would need to be fitted to a tenth-order polynomial. But even if Filip.dat was artfully crafted, perhaps starting from a set of real experimental data, it is a vivid illustration of the potential liabilities of numerical least squares analysis in a double-precision environment. This is why it has been a recurring theme in this book, and why table 11.15.2 summarized the results of our encounters with it. It does inspire confidence when software can pass even unrealistically severe tests in real problem areas, especially when it is within such easy reach as through the combination of Excel and XN.

The power of XN is that it directly addresses the root cause of most problems specific to numerical analysis, viz. the limited precision provided by software based on the IEEE 754 protocol. In fact, XN has far more numerical precision than you may ever need, and this spare precision makes it ready for much tougher problems. Using 600 decimals for Filip.dat is certainly overkill for the NIST data: with xnLS you can get identical results with just 34 decimals. But when you reduce that number to 28, the fit is poor, mainly because of errors in computing the higher powers of x, and with D < 20 you will get no valid answers whatsoever, see Table 11.15.1. Because the double precision of IEE-754 is really the limiting factor, and XN completely bypasses it, there is no need to introduce SVD into xnLS, which would merely slow it down.