An Introduction to the Quality of Computed Solutions[1]

Sven Hammarling

The Numerical Algorithms Group Ltd
Wilkinson House
Jordan Hill Road
Oxford, OX2 8DR, UK

sven@nag.co.uk

October 12, 2005

[1]Based on [Hammarling, 2005]

Contents

1 Introduction

This report is concerned with the quality of the computed numerical solutions of mathematical problems. For example, suppose we wish to solve the system of linear equations $Ax = b$ using a numerical software package. The package will return a computed solution, say $\tilde{x}$, and we wish to judge whether or not $\tilde{x}$ is a reasonable solution to the equations. Sadly, all too often software packages return poor, or even incorrect, numerical results and give the user no means by which to judge the quality of the numerical results. In 1971, Leslie Fox commented [Fox, 1971, p. 296]

> "I have little doubt that about 80 per cent. of all the results printed from the computer are in error to a much greater extent than the user would believe, ..."

More than thirty years on that paper is still very relevant and worth reading. Another very readable article is Forsythe [1970].

The quality of computed solutions is concerned with assessing how good a computed solution is in some appropriate measure. Quality software should implement reliable algorithms and should, if possible, provide measures of solution quality.

In this report we give an introduction to ideas that are important in understanding and measuring the quality of computed solutions. In particular we review the ideas of condition, stability and error analysis, and their realisation in numerical software. We take as the principal example LAPACK [Anderson et al., 1999], a package for the solution of dense and banded linear algebra problems, but also draw on examples from the NAG Library [NAG] and elsewhere. The aim is not to show how to perform an error analysis, but to indicate why an understanding of the ideas is important in judging the quality of numerical solutions, and to encourage the use of software that returns indicators of the quality of the results. We give simple examples to illustrate some of the ideas that are important when designing reliable numerical software.

Computing machines use floating point arithmetic for their computation, and so we start with an introduction to floating point numbers.

2 Floating Point Numbers and IEEE Arithmetic

Floating point numbers are a subset of the real numbers that can be conveniently represented in the finite word length of a computer, without unduly restricting the range of numbers represented. For example, the ANSI/IEEE standard for binary floating point arithmetic [IEEE, 1985] uses 64 bits to represent double precision numbers in the approximate range $10^{\pm 308}$.

A *floating point number*, x, can be represented in terms of four integers as

$$x = \pm m \times b^{e-t},$$

where b is the *base* or *radix*, t is the *precision*, e is the *exponent* with an *exponent range* of $[e_{\min}, e_{\max}]$ and m is the *mantissa* or *significand*, satisfying $0 \leq m \leq b^t - 1$. If $x \neq 0$ and

$m \geq b^{t-1}$ then x is said to be *normalized*. An alternative, equivalent representation of x is

$$\begin{aligned} x &= \pm 0.d_1 d_2 \dots d_t \times b^e \\ &= \pm \left(\frac{d_1}{b} + \frac{d_2}{b^2} + \cdots + \frac{d_t}{b^t} \right) \times b^e, \end{aligned}$$

where each digit satisfies $0 \leq d_i \leq b-1$. If $d_1 \neq 0$ then we see that x is normalized. If $d_1 = 0$ and $x \neq 0$ then x is said to be *denormalized*. Denormalized numbers between 0 and the smallest normalized number are called *subnormal*. Note that denormalized numbers do not have the full t digits of precision.

The following example, which is not intended to be realistic, illustrates the model.

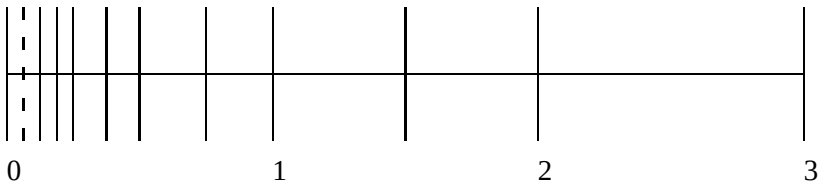
Example 2.1 (Floating point numbers)

$$b = 2,\ t = 2,\ e_{\min} = -2,\ e_{\max} = 2.$$

All the normalized numbers have $d_1 = 1$ and either $d_2 = 0$ or $d_2 = 1$, that is m is one of the two binary integers $m = 10\ (= 2)$ or $m = 11\ (= 3)$. Denormalized numbers have $m = 01\ (= 1)$. Thus the smallest positive normalized number is $2 \times 2^{e_{\min}-t} = \frac{1}{8}$ and the largest is $3 \times 2^{e_{\max}-t} = 3$. The value $1 \times 2^{e_{\min}-t} = \frac{1}{16}$ is the only positive subnormal number in this system. The complete set of non-negative normalized numbers is:

$$0, \frac{1}{8}, \frac{3}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{2}, \frac{3}{4}, 1, \frac{3}{2}, 2, 3$$

The set of non-negative floating point numbers is illustrated in Figure 1, where the subnormal number is indicated by a dashed line.



$$b = 2,\ t = 2,\ e_{\min} = -2,\ e_{\max} = 2$$

Figure 1: Floating Point Number Example

Note that floating point numbers are not equally spaced absolutely, but the relative spacing between numbers is approximately equal. The value

$$u = \frac{1}{2} \times b^{1-t} \tag{1}$$

is called the *unit roundoff*, or the *relative machine precision* and is the furthest distance relative to unity between a real number and the nearest floating point number. In Example 2.1, $u = \frac{1}{4} = 0.25$

and we can see, for example, that the furthest real number from 1.0 is 1.25 and the furthest real number from 2.0 is 2.5. u is fundamental to floating point error analysis.

The value $\epsilon_M = 2u$ is called *machine epsilon*.

The ANSI/IEEE standard mentioned above (usually referred to as IEEE arithmetic), which of course has $b = 2$, specifies:

- floating point number formats,
- results of the basic floating point operations,
- rounding modes,
- signed zero, infinity $(\pm\infty)$ and not-a-number (NaN),
- floating point exceptions and their handling and
- conversion between formats.

Thankfully, nowadays almost all machines use IEEE arithmetic. There is also a generic ANSI/IEEE, base independent, standard [IEEE, 1987]. The formats supported by the ANSI/IEEE binary standard are indicated in Table 1.

Format	Precision	Exponent	Approx Range	Approx Precision
Single	24 bits	8 bits	$10^{\pm 38}$	10^{-8}
Double	53 bits	11 bits	$10^{\pm 308}$	10^{-16}
Extended	≥ 64 bits	≥ 15 bits	$10^{\pm 4932}$	10^{-20}

Table 1: IEEE Arithmetic Formats

The default rounding mode for IEEE arithmetic is *round to nearest*, in which a real number is represented by the nearest floating point number, with rules as to how to handle a tie [Overton, 2001, Chapter 5].

Whilst the ANSI/IEEE standard has been an enormous help in standardizing floating point computation, it should be noted that moving a computation between machines that implement IEEE arithmetic does not guarantee that the computed results will be the same. Variations can occur due to such things as compiler optimization, the use of extended precision registers, and fused multiply-add.

Further discussion of floating point numbers and IEEE arithmetic can be found in Higham [2002] and Overton [2001].

The value u can be obtained from the LAPACK function SLAMCH, for single precision arithmetic, or DLAMCH for double precision arithmetic by calling the function with the argument CMACH as `'e'`, and is also returned by the NAG Fortran Library routine X02AJF[1]. It should be noted that on machines that truncate, rather than round, ϵ_M is returned in place of u, but such machines are now rare. It should also be noted that 'e' in S/DLAMCH represents `eps`, but this should not

[1] In some ports it actually returns $u + b^{1-2t}$. See the X02 Chapter introduction [NAG, 2003].

be confused with ϵ_M. The Matlab built in variable `eps` returns ϵ_M [MathWorks], as does the Fortran 95/Fortran 2003 numeric enquiry function `epsilon` [Metcalf and Reid, 1996; Metcalf et al., 2004].

3 Why Worry about Computed Solutions?

In this section we consider some simple examples of numerical computation that need care in order to obtain reasonable solutions. For clarity of exposition, most of the examples in this and the following sections are illustrated using decimal floating point (significant figure) arithmetic, with round to nearest.

The first example illustrates the problem of damaging subtraction, usually referred to as *cancellation*.

Example 3.1 (Cancellation)
Using four figure decimal arithmetic, suppose we wish to compute $s = 1.000 + 1.000 \times 10^4 - 1.000 \times 10^4$. *If we compute in the standard way from left to right we obtain*

$$s = 1.000 + 1.000 \times 10^4 - 1.000 \times 10^4 \Rightarrow (1.000 + 1.000 \times 10^4) - 1.000 \times 10^4$$
$$\Rightarrow 1.000 \times 10^4 - 1.000 \times 10^4 \Rightarrow 0,$$

instead of the correct result of 1.0. *Although the cancellation (subtraction) was performed exactly, it lost all the information for the solution.*

As Example 3.1 illustrates, the cause of the poor result often happens before the cancellation, and the cancellation is just the final nail in the coffin. In Example 3.1, the damage was done in computing $s = 1.000 + 1.000 \times 10^4$, where we lost important information (1.000). It should be said that the subtraction of nearly equal numbers is not always damaging.

Most problems have alternative formulations which are theoretically equivalent, but may computationally yield quite different results. The following example illustrates this in the case of computing sample variances.

Example 3.2 (Sample variance [Higham, 2002], Section 1.9)
The sample variance of a set of n *values* $x_1, x_2, \ldots, x_n$ *is defined as*

$$s_n^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2, \tag{2}$$

where $\bar{x}$ *is the sample mean of the* n *values*

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i.$$

An alternative, theoretically equivalent, formula to compute the sample variance which requires only one pass through the data is given by

$$s_n^2 = \frac{1}{n-1} \left(\sum_{i=1}^{n} x_i^2 - \frac{1}{n} \left(\sum_{i=1}^{n} x_i \right)^2 \right). \tag{3}$$

If $x^{\mathrm{T}} = \begin{pmatrix} 10000 & 10001 & 10002 \end{pmatrix}$ *then using 8 figure arithmetic (2) gives* $s^2 = 1.0$*, the correct answer, but (3) gives* $s^2 = 0.0$*, with a relative error of 1.0.*

(3) can clearly suffer from cancellation, as illustrated in the example. On the other hand, (2) always gives good results unless n is very large [Higham, 2002, Problem 1.10]. See also Chan et al. [1983] for further discussion of the problem of computing sample variances.

Sadly, it is not unknown for software packages and calculators to implement the algorithm of (3). For example in Excel 2002 from Microsoft Office XP (and in previous versions of Excel also), the function STDEV computes the standard deviation, s, of the data

$$x^{\mathrm{T}} = \begin{pmatrix} 100000000 & 100000001 & 100000002 \end{pmatrix}$$

as $s = 0$. Considering the pervasive use of Excel and the importance of standard deviation and its use in applications, it is disappointing to realise that (3) has been used by these versions of Excel[2]. See Cox et al. [2000] for further information, as well as Knüsel [1998], McCullough and Wilson [1999] and McCullough and Wilson [2002]. The spreadsheet from OpenOffice.org version 1.0.2 produces the same result, but gives no information on the method used in its help system; on the other hand the Gnumeric spreadsheet (version 1.0.12) gives the correct result, although again the function description does not describe the method used.

A result that is larger than the largest representable floating point number is said to *overflow*. For example, in double precision IEEE arithmetic for which the approximate range is $10^{\pm 308}$, if $x = 10^{200}$, then x^2 would overflow. Similarly, x^{-2} is said to *underflow* because it is smaller than the smallest non-zero representable floating point number.

As with the unit rounding error or machine epsilon discussed in Section 2, the overflow and underflow thresholds can be obtained from the LAPACK function S/DLAMCH by calling the function with the argument CMACH as `'o'` and `'u'` respectively; from the NAG Fortran Library routines X02ALF and X02AKF respectively; the Matlab built in variables `realmax` and `realmin`; and from the Fortran 95 numeric enquiry functions `huge` and `tiny`.

Care needs to be taken to avoid unnecessary overflow and damaging underflow. The following example illustrates this care in the case of computing the hypotenuse of the right angled triangle shown in Figure 2.

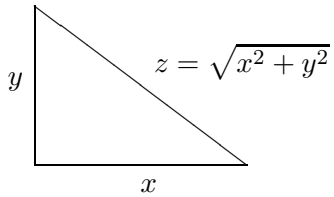


Figure 2: Hypotenuse of a right angled triangle

[2]The algorithm has at last been replaced in Excel from Office 2003, which now gives the correct answer.

Example 3.3 (Hypotenuse)
In Figure 2, if x or y is very large there is a danger of overflow, even if z is representable. Assuming that x and y are non-negative, a safe method of computing z is

$$a = \max(x, y),\ b = \min(x, y)$$

$$z = \begin{cases} a\sqrt{1 + \left(\frac{b}{a}\right)^2}, & a > 0 \\ 0, & a = 0. \end{cases}$$

This also avoids z being computed as zero if x^2 and y^2 both underflow. We note that Stewart [1998, p.139 and p.144] actually recommends computing z as

$$z = \begin{cases} s\sqrt{\left(\frac{x}{s}\right)^2 + \left(\frac{y}{s}\right)^2},\ \textit{where } s = x + y, & s > 0 \\ 0, & s = 0 \end{cases}$$

because this is slightly more accurate on a hexadecimal machine. An interesting alternative for computing Pythagorean sums is given in Moler and Morrison [1983]; see also Dubrulle [1983] and Higham and Higham [2005, section 22.9] .

We can see that (3) of Example 3.2 also has the potential for overflow and underflow and, as well as implementing this formula rather than a more stable version, Excel does not take the necessary care to avoid underflow and overflow. For example, for the values (1.0E200, 1.0E200), STDEV in Excel 2003 from Microsoft Office 2003 returns the mysterious symbol `#NUM!`, which signifies a numeric exception, in this case overflow, due to the fact that $(10.0^{200})^2$ overflows in IEEE double precision arithmetic. The correct standard deviation is of course 0. Similarily, for the values (0, 1.0E-200, 2.0E-200), STDEV returns the value 0 rather than the correct value of 1.0E-200. OpenOffice.org version 1.0.2 also returns zero for this data, and overflows on the previous data. Mimicking Excel is not necessarily a good thing!

The computation of the modulus of a complex number $x = x_r + ix_i$ requires almost the same computation as that in Example 3.3.

Example 3.4 (Modulus of a complex number)

$$|x| = \sqrt{x_r^2 + x_i^2}.$$

$$a = \max(|x_r|, |x_i|),\ b = \min(|x_r|, |x_i|)$$

$$|x| = \begin{cases} a\sqrt{1 + \left(\frac{b}{a}\right)^2}, & a > 0 \\ 0, & a = 0. \end{cases}$$

Again this also avoids $|x|$ being computed as zero if x_r^2 and x_i^2 both underflow.

Another example where care is needed in complex arithmetic is complex division

$$\frac{x}{y} = \frac{x_r + ix_i}{y_r + iy_i} = \frac{(x_r + ix_i)\,(y_r - iy_i)}{y_r^2 + y_i^2}.$$

Again, some scaling is required to avoid overflow and underflow. See for example Smith [1962], Stewart [1985] and Priest [2004]. Algol 60 procedures for the complex operations of modulus, division and square root are given in Martin and Wilkinson [1968] and the NAG Library Chapter, A02, for complex arithmetic has routines based upon those Algol procedures, see for example NAG [2003]. A careful C function is given in the Priest reference above. Occasionally, some aspect of complex floating point arithmetic is incorrectly implemented, see for example Blackford et al. [1997, Section 7].

Another example, similar to the previous examples, requiring care to avoid overflow and damaging underflow is that of real plane rotations where we need to compute $c = \cos\theta$ and $s = \sin\theta$ such that

$$c = \frac{x}{z},\ s = \frac{y}{z},\ \text{ where } z = \sqrt{x^2 + y^2}$$

or alternatively

$$c = \frac{-x}{z},\ s = \frac{-y}{z}.$$

Another convenient way to express the two choices is as

$$c = \frac{\pm 1}{\sqrt{1+t^2}},\ s = c\,t,\ \text{ where } t \equiv \tan\theta = \frac{x}{y}\,. \tag{4}$$

If G is the *plane rotation matrix*

$$G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix},$$

then, with the choices of (4),

$$G \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \pm z \\ 0 \end{pmatrix}.$$

When used in this way for the introduction of zeros the rotation is generally termed a *Givens plane rotation* [Givens, 1954; Golub and Van Loan, 1996]. Givens himself certainly took care in the computation of c and s. To see an extreme case of the detailed consideration necessary to implement a seemingly simple algorithm, but to be efficient, to preserve as much accuracy as possible throughout the range of floating point numbers, and to avoid overflow and damaging underflow see Bindel et al. [2002], where the computation of the Givens plane rotation is fully discussed.

Sometimes computed solutions are in some sense reasonable, but may not be what the user was expecting. In the next example, the computed solution is close to the exact solution, but does not meet a constraint that the user might have expected the solution to meet.

Example 3.5 (Sample mean [Higham, 1998])
Using three figure floating point decimal arithmetic:

$$(5.01 + 5.03)/2 \Rightarrow 10.0/2 \Rightarrow 5.00$$

and we see that the computed value is outside the range of the data, although it is not inaccurate.

Whether or not such a result matters depends upon the application, but is an issue that needs to be considered when implementing numerical algorithms. For instance if

$$y = \cos x$$

then we probably expect the property $|y| \leq 1$ to be preserved computationally, so that a value $|y| > 1$ is never returned. For a monotonic function we may expect monotonicity to be preserved computationally.

In the next section we look at ideas that help our understanding of what constitutes a quality solution.

4 Condition, Stability and Error Analysis

4.1 Condition

Firstly we look at the condition of a problem. The *condition* of a problem is concerned with the sensitivity of the problem to perturbations in the data. A problem is ill-conditioned if small changes in the data cause relatively large changes in the solution. Otherwise a problem is well-conditioned. Note that condition is concerned with the sensitivity of the problem, and is independent of the method we use to solve the problem. We now give some examples to illustrate somewhat ill-conditioned problems.

Example 4.1 (Cubic equation)
Consider the cubic equation

$$x^3 - 21x^2 + 120x - 100 = 0,$$

whose exact roots are $x_1 = 1, x_2 = x_3 = 10$*. If we perturb the coefficient of* x^3 *to give*

$$0.99x^3 - 21x^2 + 120x - 100 = 0,$$

the roots become $x_1 \approx 1.000, x_2 \approx 11.17, x_3 \approx 9.041$*, so that the changes in the two roots* x_2 *and* x_3 *are significantly greater than the change in the coefficient. On the other hand, the roots of the perturbed cubic equation*

$$1.01x^3 - 21x^2 + 120x - 100 = 0,$$

are $x_1 \approx 1.000,\ x_2, x_3 \approx 9.896 \pm 1.044i$*, and this time the double root has become a complex conjugate pair with a significant imaginary part.*

We can see that the roots x_2 *and* x_3 *are ill-conditioned. Note that we cannot deduce anything about the condition of* x_1 *just from this data. The three cubic polynomials are plotted in Figure 3.*

Example 4.2 (Eigenvalue problem)
The matrix

$$A = \begin{pmatrix} 10 & 100 & 0 & 0 \\ 0 & 10 & 100 & 0 \\ 0 & 0 & 10 & 100 \\ 0 & 0 & 0 & 10 \end{pmatrix}$$

has eigenvalues $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = 10$*, whereas the slightly perturbed matrix*

$$B = \begin{pmatrix} 10 & 100 & 0 & 0 \\ 0 & 10 & 100 & 0 \\ 0 & 0 & 10 & 100 \\ 10^{-6} & 0 & 0 & 10 \end{pmatrix}$$

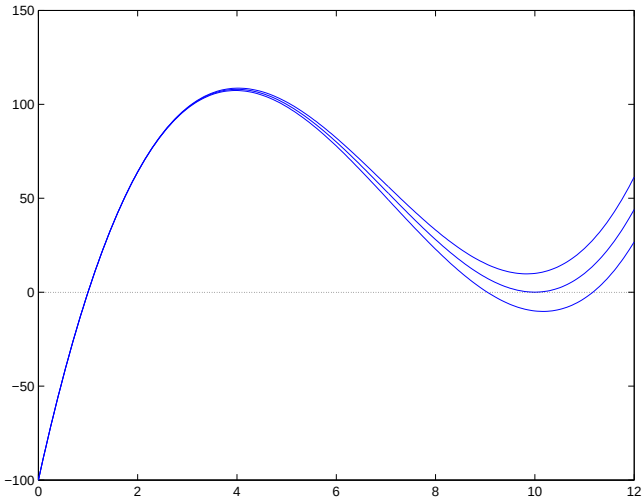


Figure 3: Cubic Equation Example

has eigenvalues $\lambda_1 = 11,\ \lambda_2, \lambda_3 = 10 \pm i,\ \lambda_4 = 9$.

Example 4.3 (Integral)

$$I = \int_{-10}^{10} \left(ae^{x} - be^{-x} \right)\, dx$$

When $a = b = 1,\ I = 0$, *but when* $a = 1, b = 1.01,\ I \approx -220$. *The function* $f(x) = ae^{x} - be^{-x}$, *when* $a = b = 1$ *is plotted in Figure 4. Notice that the vertical scale has a scale factor* 10^4, *so that a small change in function can make a large change in the area under the curve.*

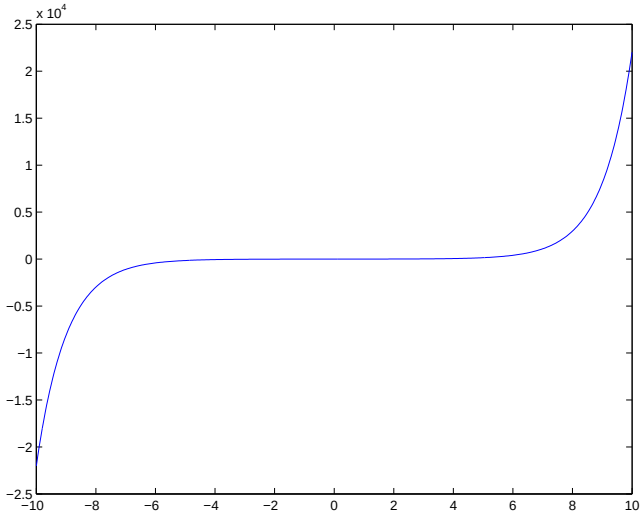


Figure 4: Integral Example

Example 4.4 (Linear equations)
The equations $Ax = b$ given by

$$\begin{pmatrix} 99 & 98 \\ 100 & 99 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 197 \\ 199 \end{pmatrix} \tag{5}$$

have the solution $x_1 = x_2 = 1$, but the equations

$$\begin{pmatrix} 98.99 & 98 \\ 100 & 99 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 197 \\ 199 \end{pmatrix}$$

have the solution $x_1 = 100, x_2 = -99$. The two straight lines represented by (5) are plotted in Figure 5, but to the granularity of the graph we cannot tell the two lines apart.

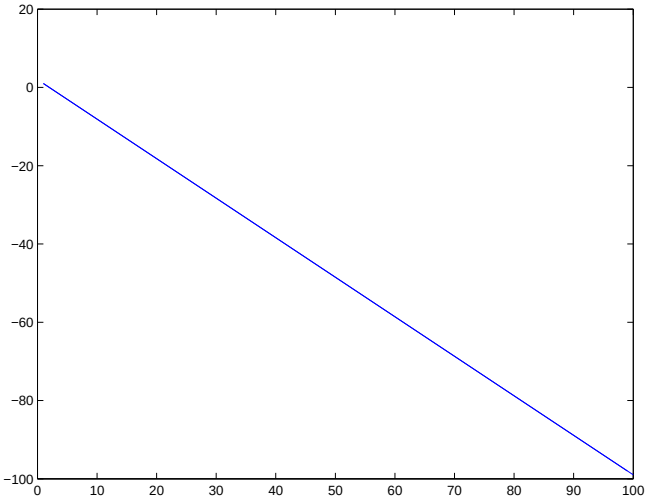


Figure 5: Linear Equations Example

To be able to decide whether or not a problem is ill-conditioned it is clearly desirable to have some measure of the condition of a problem. We show two simple cases where we can obtain such a measure, and quote the result for a third example. For the first case we derive the condition number for the evaluation of a function of one variable [Higham, 2002, Section 1.6].

Let $y = f(x)$ with f twice differentiable and $f(x) \neq 0$. Also let $\hat{y} = f(x + \epsilon)$. Then, using the mean value theorem

$$\begin{aligned} \hat{y} - y &= f(x + \epsilon) - f(x) \\ &= f'(x)\epsilon + \frac{f''(x + \theta\epsilon)}{2!}\epsilon^2, \ \theta \in (0, 1) \end{aligned}$$

giving

$$\frac{\hat{y} - y}{y} = \left(\frac{xf'(x)}{f(x)}\right)\frac{\epsilon}{x} + O\left(\epsilon^2\right).$$

The quantity

$$\kappa(x) = \left|\frac{xf'(x)}{f(x)}\right|$$

is called the *condition number* of f since

$$\left|\frac{\hat{y}-y}{y}\right| \approx \kappa(x)\left|\frac{\epsilon}{x}\right|.$$

Thus if $\kappa(x)$ is large the problem is ill-conditioned, that is small perturbations in x can induce large perturbations in the solution y.

Example 4.5
Let $y = f(x) = \cos x$. Then we see that

$$\kappa(x) = |x \tan x|$$

and, as we might expect, $\cos x$ is most sensitive close to asymptotes of $\tan x$, such as x close to $\pi/2$.[3] *If we take $x = 1.57$ and $\epsilon = -0.01$ then we find that*

$$\kappa(x)\left|\frac{\epsilon}{x}\right| \approx 12.5577,$$

which is a very good estimate of $|(\hat{y}-y)/y| = 12.55739\ldots$.

For the second example we consider the condition number of a system of linear equations $Ax = b$. If we let $\hat{x}$ be the solution of the perturbed equations

$$(A+E)\hat{x} = b,$$

then

$$A(\hat{x}-x) = -E\hat{x}, \text{ so that } \hat{x}-x = -A^{-1}E\hat{x},$$

giving

$$\frac{\|\hat{x}-x\|}{\|\hat{x}\|} \leq \|A^{-1}\|\cdot\|E\| = \left(\|A\|\cdot\|A^{-1}\|\right)\frac{\|E\|}{\|A\|}. \tag{6}$$

The quantity

$$\kappa(A) = \|A\|\cdot\|A^{-1}\|$$

is called the condition number of A with respect to the solution of the equations $Ax = b$, or the condition number of A with respect to matrix inversion. Since $I = AA^{-1}$, for any norm such that $\|I\| = 1$, we have that $1 \leq \kappa(A)$, with equality possible for the $1, 2$ and ∞ norms. If A is singular then $\kappa(A) = \infty$.

Example 4.6 (Condition of matrix)
For the matrix of Example 4.4 we have that

$$A = \begin{pmatrix} 99 & 98 \\ 100 & 99 \end{pmatrix}, \ \|A\|_1 = 199$$

and

$$A^{-1} = \begin{pmatrix} 99 & -98 \\ -100 & 99 \end{pmatrix}, \ \left\|A^{-1}\right\|_1 = 199,$$

[3] The given condition number is not valid at $x = \pi/2$, since $\cos \pi/2 = 0$.

so that

$$\kappa_1(A) = 199^2 \approx 4 \times 10^4.$$

Thus we can see that if A is only accurate to about 4 figures, we cannot guarantee any accuracy in the solution.

The term condition number was first introduced by Turing in the context of systems of linear equations [Turing, 1948]. Note that for an orthogonal or unitary matrix $Q, \kappa_2(Q) = 1$.

As a third illustration we quote results for the sensitivity of the root of a polynomial. Consider

$$f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0,$$

and let α be a single root of $f(x)$ so that $f(\alpha) = 0$, but $f'(\alpha) \neq 0$. Let $p(x)$ be the perturbed polynomial

$$p(x) = f(x) + \epsilon g(x),\ g(x) = b_n x^n + b_{n-1}x^{n-1} + \ldots + b_1 x + b_0,$$

with root $\hat{\alpha} = \alpha + \delta$, so that $p(\hat{\alpha}) = 0$. Then [Wilkinson, 1963, Section 7, Chapter 2] shows that

$$|\delta| \approx \left| \frac{\epsilon g(\alpha)}{f'(\alpha)} \right|.$$

Wilkinson also shows that if α is a double root then

$$|\delta| \approx \left| \left(-\frac{2\epsilon g(\alpha)}{f''(\alpha)} \right)^{\frac{1}{2}} \right|.$$

Example 4.7 (Condition of roots of cubic equation)
For the root $\alpha = x_1 = 1$ of the cubic equation of Example 4.1, with $g(x) = x^3$ and $\epsilon = -0.01$, we have

$$f'(x) = 3x^2 - 42x + 120$$

so that

$$|\delta| \approx \left| \frac{-0.01 \times 1^3}{81} \right| \approx 0.0001$$

and hence this root is very well-conditioned with respect to perturbations in the coefficient of x^3. On the other hand, for the double root $\alpha = 10$, we have

$$f''(x) = 6x - 42,$$

so that

$$|\delta| \approx \left| \left(\frac{-2 \times -0.01 \times 10^3}{18} \right)^{\frac{1}{2}} \right| \approx 1.054$$

and this time the perturbation of ϵ produces a rather larger perturbation in the root. Because ϵ is not particularly small the estimate of δ is not particularly accurate, but we do get a good warning of the ill-conditioning.

Higham [2002, Section 25.4] gives a result for the sensitivity of a root of a general nonlinear equation.

Problems can be ill-conditioned simply because they are poorly scaled, often as the result of a poor choice of measurement units. Some algorithms, or implementations of algorithms, are insensitive to scaling or attempt automatic scaling, but in other cases a good choice of scaling can be important to the success of an algorithm. It is also all too easy to turn a badly scaled problem into a genuinely ill-conditioned problem.

Example 4.8 (Badly scaled matrix)
If we let A be the matrix

$$A = \begin{pmatrix} 2 \times 10^9 & 10^9 \\ 10^{-9} & 2 \times 10^{-9} \end{pmatrix},$$

then $\kappa_2(A) \approx 1.67 \times 10^{18}$ and so A is ill-conditioned. However we can row scale A as

$$B = DA = \begin{pmatrix} 10^{-9} & 0 \\ 0 & 10^9 \end{pmatrix} \begin{pmatrix} 2 \times 10^9 & 10^9 \\ 10^{-9} & 2 \times 10^{-9} \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix},$$

for which $\kappa_2(B) = 3$, so that B is well-conditioned. On the other hand if we perform a plane rotation on A with $c = 0.8, s = 0.6$ we get

$$\begin{aligned} C = GA &= \begin{pmatrix} 0.8 & 0.6 \\ -0.6 & 0.8 \end{pmatrix} \begin{pmatrix} 2 \times 10^9 & 10^9 \\ 10^{-9} & 2 \times 10^{-9} \end{pmatrix} \\ &= 2 \begin{pmatrix} 8 \times 10^8 + 3 \times 10^{-10} & 4 \times 10^8 + 6 \times 10^{-10} \\ -6 \times 10^8 + 4 \times 10^{-10} & -3 \times 10^8 + 8 \times 10^{-10} \end{pmatrix}. \end{aligned}$$

Since G is orthogonal, $\kappa_2(C) = \kappa_2(A) \approx 1.67 \times 10^{18}$, and so C is of course as ill-conditioned as A, but now scaling cannot recover the situation. To see that C is genuinely ill-conditioned, we note that

$$C \approx 2 \times 10^8 \begin{pmatrix} 8 & 4 \\ -6 & -3 \end{pmatrix}$$

which is singular. In double precision IEEE arithmetic, this would be the floating point representation of C.

Many of the LAPACK routines perform scaling, or have options to equilibrate the matrix in the case of linear equations [Anderson et al., 1999, Sections 2.4.1 and 4.4.1], [Higham, 2002, Sections 7.3 and 9.8], or to balance in the case of eigenvalue problems [Anderson et al., 1999, Sections 4.8.1.2 and 4.11.1.2].

4.2 Stability

The *stability* of a method for solving a problem is concerned with the sensitivity of the method to (rounding) errors in the solution process. A method that guarantees as accurate a solution as the data warrants is said to be stable, otherwise the method is unstable. To emphasise the point we note that, whereas condition is concerned with the sensitivity of the problem, stability is concerned with the sensitivity of the method of solution.

An example of an unstable method is that of (3) for computing sample variance. We now give two more simple illustrative examples.

Example 4.9 (Quadratic equation)
Consider the quadratic equation

$$1.6x^2 - 100.1x + 1.251 = 0.$$

Four significant figure arithmetic when using the standard formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

gives

$$x_1 = 62.53,\ x_2 = 0.03125.$$

If we use the relationship $x_1 x_2 = c/a$ to compute x_2 from x_1 we instead find that

$$x_2 = 0.01251.$$

The correct solution is $x_1 = 62.55, x_2 = 0.0125$. We can see that in using the standard formula to compute the smaller root we have suffered from cancellation, since $\sqrt{b^2 - 4ac}$ is close to $(-b)$.

Even a simple problem such as computing the roots of a quadratic equation needs great care. A very nice discussion is to be found in Forsythe [1969].

Example 4.10 (Recurrence relation)
Consider the computation of y_n defined by

$$y_n = (1/e) \int_0^1 x^n e^x dx, \tag{7}$$

where n is a non-negative integer. We note that, since in the interval $[0, 1], (1/e)e^x$ is bounded by unity, it is easy to show that

$$0 \leq y_n \leq 1/(n+1). \tag{8}$$

Integrating (7) by parts gives

$$y_n = 1 - n y_{n-1},\ y_0 = 1 - 1/e = 0.63212055882856\ldots \tag{9}$$

and we have a seemingly attractive method for computing y_n for a given value of n. The result of using this forward recurrence relation, with IEEE double precision arithmetic, to compute the values of y_i up to y_{21} is shown in Table 2. Bearing in mind the bounds of (8), we see that later values are diverging seriously from the correct solution.

A simple analysis shows the reason for the instability. Since y_0 cannot be represented exactly, we cannot avoid starting with a slightly perturbed value, $\hat{y_0}$. So let

$$\hat{y_0} = y_0 + \epsilon.$$

y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
0.6321	0.3679	0.2642	0.2073	0.1709	0.1455	0.1268	0.1124
y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}
0.1009	0.0916	0.0839	0.0774	0.0718	0.0669	0.0627	0.0590
y_{16}	y_{17}	y_{18}	y_{19}	y_{20}	y_{21}		
0.0555	0.0572	-0.0295	1.5596	-30.1924	635.0403		

Table 2: Forward Recurrence for y_n

Then, even if the remaining computations are performed exactly we see that

$$\begin{aligned} \hat{y_1} &= 1 - \hat{y_0} &&= y_1 - \epsilon \\ \hat{y_2} &= 1 - 2\hat{y_1} &&= y_2 + 2\epsilon \\ \hat{y_3} &= 1 - 3\hat{y_2} &&= y_3 - 6\epsilon \\ \hat{y_4} &= 1 - 4\hat{y_3} &&= y_4 + 24\epsilon \end{aligned}$$

and a straightforward inductive proof shows that

$$\hat{y_n} = y_n + (-1)^n n!\epsilon.$$

When $n = 21, n! \approx 5.1091 \times 10^{19}$. We see clearly that this forward recurrence is an unstable method of computing y_n, since the error grows rapidly as we move forward.

The next example illustrates a stable method of computing y_n.

Example 4.11 (Stable recurrence)
Rearranging (9) we obtain the backward recurrence

$$y_{n-1} = (1 - y_n)/n,$$

Suppose that we have an approximation, $\hat{y}_{n+m}$, to y_{n+m} and we let

$$\hat{y}_{n+m} = y_{n+m} + \epsilon.$$

Then, similarly to the result of Example 4.10, we find that

$$\hat{y}_n = y_n + \frac{(-1)^{m-n}\epsilon}{(n+m)(n+m-1)\dots(n+1)}$$

and this time the initial error decays rapidly, rather than grows rapidly as in Example 4.10. If we take an initial guess of $y_{21} = 0$, we see from (8) that

$$|\epsilon| \leq 1/21 < 0.05.$$

Using this backward recurrence relation, with IEEE double precision arithmetic, gives the value

$$y_0 = 0.63212055882856,$$

which is correct to all the figures shown. We see that this backward recurrence is stable.

It should also be said that the integral of (7) can be evaluated stably without difficulty using a good numerical integration (quadrature) formula, since the function $f(x) = (1/e)x^n e^x$ is non-negative and monotonic throughout the interval $[0, 1]$.

In the solution of ordinary and partial differential equations one form of instability can arise by replacing a differential equation by a difference equation. We first illustrate the problem by the solution of a simple nonlinear equation.

Example 4.12 (Parasitic solution)
The equation

$$e^{-x} = 99x \tag{10}$$

has a solution close to $x = 0.01$. By expanding e^{-x} as a power series we have that

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \ldots \approx 1 - x + \frac{x^2}{2!}$$

and hence an approximate solution of (10) is a root of the quadratic equation

$$x^2 - 200x + 2 = 0,$$

*which has the two roots $x_1 \approx 0.0100005, x_2 \approx 199.99$. The second root clearly has nothing to do with the original equation and is called a **parasitic** solution.*

In the above example we are unlikely to be fooled by the parasitic solution, since it so clearly does not come close to satisfying (10). But in the solution of ordinary or partial differential equations such bifurcations, due to truncation error, may not be so obvious.

Example 4.13 (Instability for ODE)
For the initial value problem

$$y' = f(x, y),\ y = y_0 \text{ when } x = x_0, \tag{11}$$

the mid-point rule, or leap-frog method, for solving the differential equation is given by

$$y_{r+1} = y_{r-1} + 2hf_r, \tag{12}$$

where $h = x_i - x_{i-1}$ for all i and $f_r = f(x_r, y_r)$. This method has a truncation error of $O(h^3)$ [Isaacson and Keller, 1966, Section 1.3, Chapter 8][4] This method requires two starting values, so one starting value must be estimated by some other method. Consider the case where

$$f(x, y) = \alpha y,\ y_0 = 1,\ x_0 = 0,$$

so that the solution of (11) is $y = e^{\alpha x}$. Figures 6 and 7 show the solution obtained by using (12) when $h = 0.1$ for the cases $\alpha = 2.5$ and $\alpha = -2.5$ respectively. In each case the value of y_1 is taken as the correct four figure value, $y_1 = 1.284$ when $\alpha = 2.5$ and $y_1 = 0.7788$ when $\alpha = -2.5$. We see that in the first case the numerical solution does a good job in following the exact solution, but in the second case oscillation sets in and the numerical solution diverges from the exact solution.

[4] Here it is called the centered method. It is an example of a Nyström method.

The reason for the behaviour in the above example is that (12) has the solution

$$y_r = A\left(\alpha h + \left(1+\alpha^2 h^2\right)^{\frac{1}{2}}\right)^r + B\left(\alpha h - \left(1+\alpha^2 h^2\right)^{\frac{1}{2}}\right)^r, \tag{13}$$

where A and B are constants that depend on the intial conditions. With the initial conditions $y_0 = 1, x_0 = 0$ and $y_1 = e^{\alpha h}, x_1 = h$ we find that $A = 1 + O(h^3), B = O(h^3)$. We can see that the first term in (13) approximates the exact solution, but the second term is a parasitic solution. When $\alpha > 0$ the exact solution increases and the parasitic solution decays, and so is harmless, but when $\alpha < 0$ the exact solution decays and the parasitic solution grows as illustrated in Figure 7. An

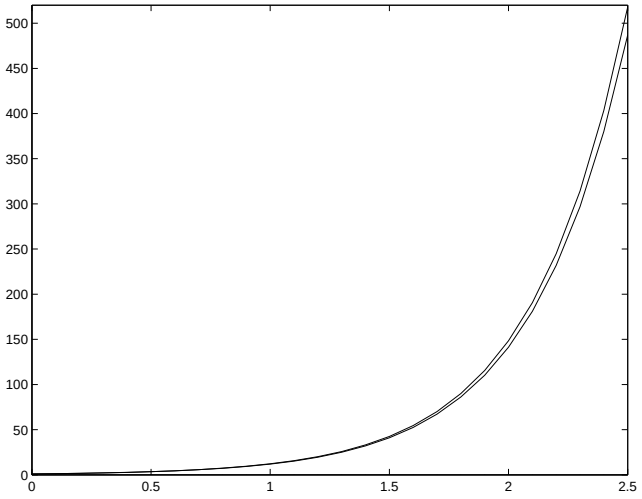


Figure 6: Stable ODE Example

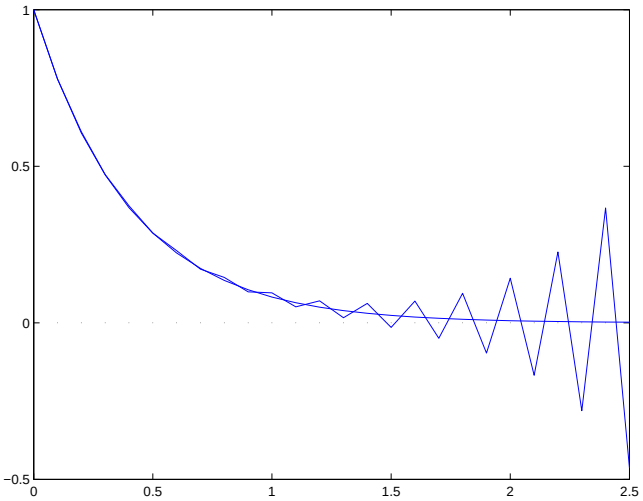


Figure 7: Unstable ODE Example

entertaining discussion, in the context of the Milne-Simpson method, of the above phenomenon is

given in Acton [1970, Chapter 5], a book full of good advice and insight. A more recent book by Acton in the same vein is [Acton, 1996].

4.3 Error Analysis

Error analysis is concerned with analysing the cumulative effects of errors. Usually these errors will be rounding or truncation errors. For example, if the polynomial

$$p(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_n x^n$$

is evaluated at some point $x = \alpha$ using Horner's scheme (nested multiplication) as

$$p(\alpha) = p_0 + \alpha\left(p_1 + \cdots + \alpha\left(p_{n-2} + \alpha\left(p_{n-1} + \alpha p_n\right)\right)\ldots\right),$$

we might ask under what conditions, if any, on the coefficients $p_0, p_1, \ldots, p_n$ and α, the solution will, in some sense, be reasonable? To answer the question we need to perform an error analysis.

Error analysis is concerned with establishing whether or not an algorithm is stable for the problem in hand. A *forward error analysis* is concerned with how close the computed solution is to the exact solution. A *backward error analysis* is concerned with how well the computed solution satisfies the problem to be solved. On first acquaintance, that a backward error analysis, as opposed to a forward error analysis, should be of interest often comes as a surprise. The next example illustrates the distinction between backward and forward errors.

Example 4.14 (Linear equations)
Let

$$A = \begin{pmatrix} 99 & 98 \\ 100 & 99 \end{pmatrix} \text{ and } b = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Then the exact solution of the equations $Ax = b$ is given by

$$x = \begin{pmatrix} 1 \\ -1 \end{pmatrix}.$$

Also let $\hat{x}$ be an approximate solution to the equations and define the ***residual*** *vector r as*

$$r = b - A\hat{x}. \tag{14}$$

Of course, for the exact solution $r = 0$ and we might hope that for a solution close to x, r should be small. Consider the approximate solution

$$\hat{x} = \begin{pmatrix} 2.97 \\ -2.99 \end{pmatrix}, \text{ for which } \hat{x} - x = \begin{pmatrix} 1.97 \\ -1.99 \end{pmatrix},$$

and so $\hat{x}$ looks to be a rather poor solution. But for this solution we have that

$$r = \begin{pmatrix} -0.01 \\ 0.01 \end{pmatrix}$$

and we have almost solved the original problem. On the other hand the approximate solution

$$\hat{x} = \begin{pmatrix} 1.01 \\ -0.99 \end{pmatrix}, \text{ for which } \hat{x} - x = \begin{pmatrix} 0.01 \\ 0.01 \end{pmatrix},$$

gives

$$r = \begin{pmatrix} -1.97 \\ -1.97 \end{pmatrix}$$

and, although $\hat{x}$ is close to x, it does not solve a problem close to the original problem.

Once we have computed the solution to a system of linear equations $Ax = b$ we can, of course, readily compute the residual of (14). If we can find a matrix E such that

$$E\hat{x} = r, \tag{15}$$

then

$$(A + E)\hat{x} = b$$

and we thus have a measure of the perturbation in A required to make $\hat{x}$ an exact solution. A particular E that satisfies (15) is given by

$$E = \frac{r\hat{x}^{\mathrm{T}}}{\hat{x}^{\mathrm{T}}\hat{x}}.$$

From this equation we have that

$$\|E\|_2 \leq \frac{\|r\|_2\|\hat{x}\|_2}{\|\hat{x}\|_2^2} = \frac{\|r\|_2}{\|\hat{x}\|_2}$$

and from (15) we have that

$$\|r\|_2 \leq \|E\|_2\|\hat{x}\|_2, \text{ so that } \|E\|_2 \geq \frac{\|r\|_2}{\|\hat{x}\|_2}$$

and hence

$$\|E\|_2 = \frac{\|r\|_2}{\|\hat{x}\|_2}.$$

Thus, this particular E minimizes $\|E\|_2$. Since $x^{\mathrm{T}}x = \|x\|_F^2$, it also minimizes E in the Frobenius norm. This gives us an *a posteriori* bound on the backward error.

Example 4.15 (Perturbation in linear equations)
Consider the equations $Ax = b$ of Example 4.14 and the 'computed' solution

$$\hat{x} = \begin{pmatrix} 2.97 \\ -2.99 \end{pmatrix} \text{ for which } r = \begin{pmatrix} -0.01 \\ 0.01 \end{pmatrix}.$$

Then

$$r\hat{x}^{\mathrm{T}} = \begin{pmatrix} -0.0297 & 0.0299 \\ 0.0297 & -0.0299 \end{pmatrix}, \ \hat{x}^{\mathrm{T}}\hat{x} = 17.761$$

and

$$E \approx \begin{pmatrix} -0.00167 & 0.00168 \\ 0.00167 & -0.00168 \end{pmatrix}.$$

Note that $\|E\|_F/\|A\|_F \approx 1.695\times10^{-5}$ *and so the computed solution corresponds to a small relative perturbation in* A*.*

From (6) we have that

$$\frac{\|\hat{x}-x\|}{\|\hat{x}\|} \leq \kappa(A)\frac{\|E\|}{\|A\|}$$

and so, if we know $\kappa(A)$, then an estimate of the backward error allows us to estimate the forward error.

As a general rule, we can say that approximately:

$$\boxed{\text{forward error} \leq \text{condition number} \times \text{backward error.}}$$

Although the idea of backward error analysis had been introduced by others, it was James Hardy Wilkinson who really developed the theory and application, and gave us our understanding of error analysis and stability, particularly in the context of numerical linear algebra. See the classic books Wilkinson [1963] and Wilkinson [1965]. A typical picture of Wilkinson, cheerfully expounding his ideas, is shown in Figure 8. A wonderful modern book that continues the Wilkinson tradition is Higham [2002]. The solid foundation for the numerical linear algebra of today relies heavily on the pioneering work of Wilkinson; see also Wilkinson and Reinsch [1971][5]

Wilkinson recognised that error analysis could be tedious and often required great care, but was nevertheless essential to our understanding of the stability of algorithms.

> "The clear identification of the factors determining the stability of an algorithm soon led to the development of better algorithms. The proper understanding of inverse iteration for eigenvectors and the development of the QR algorithm by Francis are the crowning achievements of this line of research.
>
> "For me, then, the primary purpose of the rounding error analysis was insight." [Wilkinson, 1986, p. 197.]

As a second example to illustrate forward and backward errors, consider we consider the quadratic equation of Example 4.9.

Example 4.16 (Errors in quadratic equation)
For the quadratic equation of Example 4.9 we saw that the standard formula gave the roots $x_1 = 62.53, x_2 = 0.03125$*. Since the correct solution is* $x_1 = 62.55, x_2 = 0.0125$ *the second root has a*

[5] In Givens 1954 technical report quoted earlier [Givens, 1954], which was never published in full and must be one of the most oft quoted technical reports in numerical analysis, as well as the introduction of Givens plane rotations, it describes the use of Sturm sequences for computing eigenvalues of tridiagonal matrices, and contains probably the first explicit backward error analysis. Wilkinson, who so successfully developed and expounded the theory and analysis of rounding errors, regarded the *a priori* error analysis of Givens as "one of the landmarks in the history of the subject" [Wilkinson, 1965, Additional notes to Chapter 5].

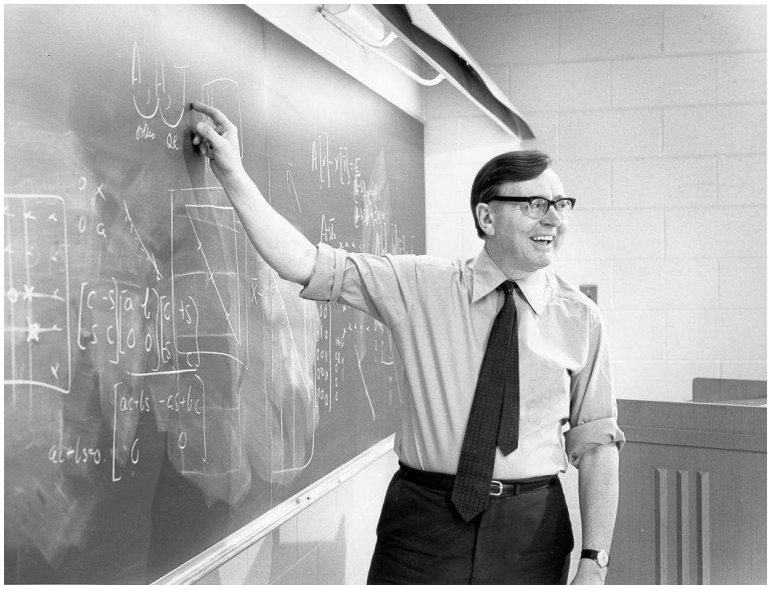


Figure 8: James Hardy Wilkinson (1919 – 1986)

large forward error. If we form the quadratic $q(x) = 1.6(x - x_1)(x - x_2)$*, rounding the answer to four significant figures, we get*

$$q(x) = 1.6x^2 - 100.1x + 3.127$$

and the constant term differs significantly from the original value of 1.251, so that there is also a large backward error. The standard method is neither forward nor backward stable. On the other hand, for the computed roots $x_1 = 62.53, x_2 = 0.01251$ *we get*

$$q(x) = 1.6x^2 - 100.1x + 1.252,$$

so this time we have both forward and backward stability.

An example of a computation that is forward stable, but not backward stable is that of computing the coefficients of the polynomial

$$p(x) = (x - x_1)(x - x_2)\dots(x - x_n),\ x_i > 0$$

In this case, since the x_i are all of the same sign, no cancellation occurs in computing the coefficients of p(x) and the computed coefficients will be close to the exact coefficients; thus we have small forward errors. On the other hand, as Example 4.1 illustrates, the roots of polynomials can be sensitive to perturbations in the coefficients and so the roots of the computed polynomial could differ significantly from $x_1, x_2, \dots, x_n$.

Example 4.17 (Ill-conditioned polynomial)
The polynomial whose roots are $x_i = i, i = 1, 2, \dots, 20$*, is*

$$p(x) = x^{20} - 210x^{19} + \dots + 20!$$

Suppose that the coefficient of x^{19} is computed as $-(210 + 2^{-23})$; then we find that $x_{16}, x_{17} \approx 16.73 \pm 2.813i$. Thus a small error in computing a coefficient produced a polynomial with significantly different roots from those of the exact polynomial. This polynomial is discussed in Wilkinson [1963, Chapter 2, Section 9] and Wilkinson [1984]. See also Wilkinson [1985, Section 2].

5 Floating Point Error Analysis

Floating point error analysis is concerned with the analysis of errors in the presence of floating point arithmetic. It is based on the relative errors that result from each basic operation. We give just a brief introduction to floating point error analysis in order to illustrate the ideas.

Let x be a real number; then we use the notation $\text{fl}(x)$ to represent the floating point value of x. The fundamental assumption is that

$$\boxed{\text{fl}\,(x) = x(1+\epsilon),\ |\epsilon| \le u} \tag{16}$$

where u is the unit roundoff of (1). Of course,

$$\frac{\text{fl}\,(x) - x}{x} = \epsilon.$$

A useful alternative is

$$\text{fl}\,(x) = \frac{x}{1+\delta},\ |\delta| \le u, \text{ so that } \frac{\text{fl}\,(x) - x}{\text{fl}\,(x)} = \delta. \tag{17}$$

Example 5.1 (Floating point numbers)
Consider four figure decimal arithmetic with

$$u = \frac{1}{2} \times 10^{-3} = 5 \times 10^{-4}.$$

If $x = \sqrt{2} = 1.414213\ldots$ then fl$(x) = 1.414$ and

$$|\epsilon| = \left| \frac{\text{fl}\,(x) - x}{x} \right| \approx 1.5 \times 10^{-4}.$$

If $x = 1.000499\ldots$ then fl$(x) = 1.000$ and

$$|\epsilon| = \left| \frac{\text{fl}\,(x) - x}{x} \right| \approx 5 \times 10^{-4} = u.$$

If $x = 1000.499\ldots$ then fl$(x) = 1000$ and again

$$|\epsilon| = \left| \frac{\text{fl}\,(x) - x}{x} \right| \approx 5 \times 10^{-4} = u.$$

Bearing in mind (16), if x and y are floating point numbers, then the standard model of floating point arithmetic, introduced by Wilkinson [1960], is given by

$$\boxed{\begin{array}{c}\text{fl}\,(x \otimes y) = (x \otimes y)(1+\epsilon),\ |\epsilon| \le u, \\ \\ \text{where } \otimes \equiv +, -, \times, \div.\end{array}} \tag{18}$$

It is assumed, of course, that $x \otimes y$ produces a value that is in the range of representable floating point numbers. Comparable to (17), a useful alternative is

$$\text{fl}\,(x \otimes y) = \frac{x \otimes y}{1+\delta},\ |\delta| \le u.$$

When we consider a sequence of floating point operations we frequently obtain products of error terms of the form

$$(1+\epsilon) = (1+\epsilon_1)(1+\epsilon_2)\dots(1+\epsilon_r)$$

so that

$$(1-u)^r \le 1+\epsilon \le (1+u)^r.$$

If we ignore second order terms then we have the reasonable assumption that[6]

$$|\epsilon| \le ru. \tag{19}$$

We now give three illustrative examples. In all three examples the x_i are assumed to be floating point numbers, that is, they are values that are already represented in the computer. This is, of course, a natural assumption to make when we are analysing the errors in a computation.

Example 5.2 (Product of values)
Let $x = x_0x_1\dots x_n$ and $\tilde{x} = fl(x)$. Thus we have n products to form, each one introducing an error bounded by u. Hence from (18) we get

$$\tilde{x} = x_0x_1(1+\epsilon_1)x_2(1+\epsilon_2)\dots x_n(1+\epsilon_n),\ |\epsilon_i| \le u \tag{20}$$

and from (19) we see that

$$\tilde{x} = x(1+\epsilon),\ |\epsilon| \le nu, \tag{21}$$

where

$$1+\epsilon = (1+\epsilon_1)(1+\epsilon_2)\dots(1+\epsilon_n).$$

We can see from (21) that this computation is forward stable, because the result is close to the exact result, and from (20) the computation is also backward stable, because the result is exact for a slightly perturbed problem; that is the result is exact for the data $x_0, x_1(1+\epsilon_1), x_2(1+\epsilon_2), \dots, x_n(1+\epsilon_n)$.

[6] Those who are uncomfortable with the approximation may prefer to replace the bound $|\epsilon| \le ru$ with one of the form $|\epsilon| \le \gamma_r$, where $\gamma_r = (ru)/(1-ru)$ and $ru < 1$ is assumed. See Higham [2002], Lemma 3.1.

Example 5.3 (Sum of values)
Let $s = x_1 + x_2 + \ldots + x_n$ and $\tilde{s} = fl(s)$. By considering

$$s_r = fl\left(s_{r-1} + x_r\right),\ s_1 = x_1$$

it is straightforward to show that

$$\begin{aligned} \tilde{s} &= x_1(1+\epsilon_1) + x_2(1+\epsilon_1) + x_3(1+\epsilon_2) + \ldots + x_n(1+\epsilon_{n-1}) \\ &= s + (x_1\epsilon_1 + x_2\epsilon_1 + x_2\epsilon_2 + \ldots + x_n\epsilon_{n-1}),\ |\epsilon_r| \leq (n-r+1)u. \end{aligned}$$

Here we see that summation is backward stable, but is not necessarily forward stable. Example 3.1 gives a case where summation is not forward stable.

Note that if the x_i all have the same sign, then summation is forward stable because then

$$|\tilde{s} - s| \leq (|x_1| + |x_2| + \ldots + |x_n|)nu = |s|nu$$

so that

$$\frac{|\tilde{s} - s|}{|s|} \leq nu.$$

Example 5.4 (Difference of two squares)
Consider the computation

$$z = x^2 - y^2 \tag{22}$$

We can, of course, also express z as

$$z = (x+y)(x-y). \tag{23}$$

If we compute z from (22) we find that

$$\begin{aligned} \tilde{z} &= fl\left(x^2 - y^2\right) = x^2(1+\epsilon_1) - y^2(1+\epsilon_2) \\ &= z + (x^2\epsilon_1 - y^2\epsilon_2),\ \epsilon_1, \epsilon_2 \leq 2u \end{aligned}$$

and so this is backward stable, but not forward stable. On the other hand, if we compute z from (23) we find that

$$\begin{aligned} \hat{z} &= fl\left((x+y)(x-y)\right) = (x+y)(x-y)(1+\epsilon) \\ &= z(1+\epsilon),\ \epsilon \leq 3u \end{aligned}$$

and so this is both backward and forward stable. As an example, if we take

$$x = 543.2,\ y = 543.1,\ \textit{so that}\ \ z = 108.63$$

and use four significant figure arithmetic we find that

$$\tilde{z} = 100,\ \textit{but}\ \ \hat{z} = 108.6.$$

Clearly $\tilde{z}$ has suffered from cancellation, but $\hat{z}$ has not.

We now quote some results, without proof, of solving higher level linear algebra problems to illustrate the sort of results that are possible. Principally we consider the solution of the n linear equations

$$Ax = b \tag{24}$$

by Gaussian elimination and we assume that the reader is familiar with Gaussian elimination. The kth step of Gaussian elimination can be expressed as

$$A_k = M_k P_k A_{k-1} Q_k,\ A_0 = A, \tag{25}$$

where P_k and Q_k are permutation matrices, one or both of which may be the unit matrix, chosen to implement whatever pivoting strategy is used and M_k is the multiplier matrix chosen to eliminate the elements below the diagonal of the kth column of A_{k-1}. This results in the factorization

$$A = PLUQ,$$

where P and Q are permutation matrices, L is a unit lower triangular matrix and U is upper triangular. To simplify analysis it is usual to assume that, with hindsight, A has already been permuted so that we can work with $A \Leftarrow P^{\mathrm{T}} A Q^{\mathrm{T}}$. In this case (25) becomes

$$A_k = M_k A_{k-1},\ A_0 = A$$

and M_k and A_{k-1} have the form

$$M_k = \begin{pmatrix} I & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -m_k & I \end{pmatrix},\ A_{k-1} = \begin{pmatrix} U_{k-1} & u_{k-1} & X_{k-1} \\ 0 & \alpha_{k-1} & b_{k-1}^{\mathrm{T}} \\ 0 & a_{k-1} & \hat{A}_{k-1} \end{pmatrix}.$$

m_k is chosen to eliminate a_{k-1}, so that

$$a_{k-1} - \alpha_{k-1} m_k = 0, \text{ giving } m_k = a_{k-1}/\alpha_{k-1},$$

$\hat{A}_{k-1}$ is updated as

$$\tilde{A}_k = \hat{A}_{k-1} - m_k b_{k-1}^{\mathrm{T}} \equiv \begin{pmatrix} \alpha_k & b_k^{\mathrm{T}} \\ a_k & \hat{A}_k \end{pmatrix}$$

and

$$A = LU, \text{ where } L = M_1^{-1} M_2^{-1} \ldots M_{n-1}^{-1}, \text{ and } U = A_{n-1}.$$

Since

$$M_k^{-1} = \begin{pmatrix} I & 0 & 0 \\ 0 & 1 & 0 \\ 0 & m_k & I \end{pmatrix}$$

we have that

$$L = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ m_{21} & 1 & \cdots & 0 & 0 \\ m_{31} & m_{32} & \cdots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ m_{n-1,1} & m_{n-1,2} & \cdots & 1 & 0 \\ m_{n1} & m_{n2} & \cdots & m_{n,n-1} & 1 \end{pmatrix}.$$

It can be shown that the computed factors $\tilde{L}$ and $\tilde{U}$ satisfy

$$\tilde{L}\tilde{U} = A + F,$$

where various bounds on F are possible; for example, for the $1, \infty$ or F norms

$$\|F\| \leq 3ngu\|A\|, \; g = \frac{\max \|\tilde{A}_k\|}{\|A\|}.$$

g is called the *growth factor*. Similarly it can be shown that the computed solution of (24), $\tilde{x}$, satisfies

$$(A + E)\tilde{x} = b,$$

where a typical bound is

$$\|E\| \leq 3n^2 gu\|A\|.$$

We can see that this bound is satisfactory unless g is large, so it is important to choose P or Q, or both, in order to control the size of g. This is essentially the classic result of Wilkinson [1961] and Wilkinson [1963, Section 25], where the ∞-norm is used and the use of partial pivoting is assumed; see also Higham [2002, Theorem 9.5].

The next example gives a simple demonstration of the need for pivoting.

Example 5.5 (The need for pivoting)
Consider the matrix

$$A = \begin{pmatrix} 0.001 & 12 \\ 10 & -10 \end{pmatrix}.$$

and the use of four significant figure arithmetic. Since this is just a two by two matrix we have that $M_1^{-1} = L$ and $M_1 A = U$. Denoting the computed matrix X by $\tilde{X}$, we find that

$$L = \tilde{L} = \begin{pmatrix} 1 & 0 \\ 10000 & 1 \end{pmatrix}, \; U = \begin{pmatrix} 0.001 & 12 \\ 0 & -120010 \end{pmatrix} \; \text{and} \; \tilde{U} = \begin{pmatrix} 0.001 & 12 \\ 0 & -120000 \end{pmatrix},$$

which gives

$$U - \tilde{U} = \begin{pmatrix} 0 & 0 \\ 0 & 10 \end{pmatrix}$$

and

$$F = \tilde{L}\tilde{U} - A = \begin{pmatrix} 0 & 0 \\ 0 & 10 \end{pmatrix} = U - \tilde{U}.$$

Thus whilst $\|F\|$ is small relative to $\|U\|$, it corresponds to a large relative perturbation in $\|A\|$. On the other hand if we permute the two rows of A to give

$$\bar{A} = \begin{pmatrix} 10 & -10 \\ 0.001 & 12 \end{pmatrix},$$

we have that

$$L = \tilde{L} = \begin{pmatrix} 1 & 0 \\ 0.0001 & 1 \end{pmatrix}, \; U = \begin{pmatrix} 10 & -10 \\ 0 & 12.001 \end{pmatrix} \; \text{and} \; \tilde{U} = \begin{pmatrix} 10 & -10 \\ 0 & 12.00 \end{pmatrix},$$

which gives

$$U - \tilde{U} = \begin{pmatrix} 0 & 0 \\ 0 & -0.001 \end{pmatrix}$$

and

$$F = \tilde{L}\tilde{U} - A = \begin{pmatrix} 0 & 0 \\ 0 & -0.001 \end{pmatrix} = U - \tilde{U}.$$

This time $\|F\|$ is small relative to both $\|U\|$ and $\|A\|$.

If we put $m = \max |\tilde{m}_{ij}|$ then we can show that

$$g \leq (1+m)^{n-1}.$$

Partial pivoting ensures that

$$m \leq 1 \text{ and hence } g \leq 2^{n-1}.$$

Only very special examples get anywhere near this bound, one example due to Wilkinson being matrices of the form

$$A = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 1 \\ -1 & 1 & 0 & \cdots & 0 & 1 \\ -1 & -1 & 1 & \cdots & 0 & 1 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ -1 & -1 & -1 & \cdots & 1 & 1 \\ -1 & -1 & -1 & \cdots & -1 & 1 \end{pmatrix}, \text{ for which } U = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 1 \\ 0 & 1 & 0 & \cdots & 0 & 2 \\ 0 & 0 & 1 & \cdots & 0 & 4 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 2^{n-2} \\ 0 & 0 & 0 & \cdots & 0 & 2^{n-1} \end{pmatrix}.$$

Despite such examples, in practice partial pivoting is the method of choice, but careful software should at least include an option to monitor the growth factor.

There are classes of matrices for which pivoting is not needed to control the growth of g [Higham, 2002, Table 9.1]. Perhaps the most important case is that of symmetric positive definite matrices for which it is known a priori that growth cannot occur, and so Gaussian elimination is stable when applied to a system of equations for which the matrix of coefficients is symmetric positive definite[7].

The choice of pivots is affected by scaling and equilibration, and a poor choice of scaling can lead to a poor choice of pivots. A full discussion on pivoting strategies, equilibration and scaling, as well as sage advice, can be found in Higham [2002].

For methods that use orthogonal transformations we can usually obtain similar error bounds, but without the growth factor, since orthogonal transformations preserve the $2-$norm and $F-$norm. For example, if we use Householder transformations to perform a QR factorization of A for the solution of the least squares problem $\min_x \|b - Ax\|_2$, where A is an m by $n, m \geq n$ matrix of rank n [Golub, 1965], the computed solution $\tilde{x}$ satisfies

$$\min_x \|(b+f) - (A+E)\tilde{x}\|_2,$$

where f and E satisfy bounds of the form

$$\|f\|_F \leq c_1 mnu \|b\|_F, \ \|E\|_F \leq c_2 mnu \|A\|_F.$$

[7] The variant of Gaussian elimination that is usually used in this case is *Cholesky's method.*

and c_1 and c_2 are small integer constants [Lawson and Hanson, 1995, page 90].

Similarly, for the solution of the eigenvalue problem $Ax = \lambda x$, where A is an n by n matrix, using Housholder transformations to reduce A to upper Hessenberg form, followed by the QR algorithm to further reduce the Hessenberg form to upper triangular Schur form, the computed solution satisifies

$$(A + E)\tilde{x} = \tilde{\lambda}\tilde{x}$$

where

$$\|E\|_F \leq p(n)u\|A\|_F$$

and $p(n)$ is a modestly growing function of n [Wilkinson, 1965; Anderson et al., 1999].

We note that the bounds discussed so far are called *normwise* bounds, but in many cases they can be replaced by *componentwise* bounds which bound the absolute values of the individual elements, and so are rather more satisfactory. For instance, if A is a sparse matrix, we would probably prefer not to have to perturb the elements that are structurally zero. As a simple example, consider the triangular equations

$$Tx = b,\ T - n \text{ by } n \text{ triangular},$$

and let $\tilde{x}$ be the solution computed by forward or backward substitution, depending on whether T is lower or upper triangular respectively. Then it can readily be shown that $\tilde{x}$ satisifes

$$(T + E)\tilde{x} = b, \text{ with } |e_{ij}| \leq nu|t_{ij}|,$$

which is a strong componentwise result showing backward stability [Higham, 2002, Theorem 8.5].

Associated with componentwise error bounds are componentwise condition numbers. Once again see Higham [2002] for further details and references.

6 Posing the Mathematical Problem

In this short section we merely wish to raise awareness of the need to model a problem correctly, without offering any profound solution.

It can be all too easy to transform a well-conditioned problem into an ill-conditioned problem. For instance, in Example 4.10 we transformed the well-conditioned quadrature problem of finding

$$y_n = (1/e)\int_0^1 x^n e^x dx,\ n \geq 0,$$

into the ill-conditioned problem of finding y_n from the forward recurrence relation

$$y_n = 1 - ny_{n-1},\ y_0 = 1 - 1/e.$$

As another example, we noted in Section 4.3 that polynomials can be very ill-conditioned. It follows that the eigenvalues of a matrix A should most certainly not be computed via the characteristic equation of A. For example, if A is a symmetric matrix with eigenvalues $\lambda_i = i, i = 1, 2, \ldots, 20$, then the characteristic equation of A, $\det(A - \lambda A)$, is very ill-conditioned (see Example 4.17).

On the other hand, the eigenvalues of a symmetric matrix are always well-conditioned [Wilkinson, 1965, Section 31, Chapter 2].

The above two examples illustrate the dangers in transforming the mathematical problem. Sometimes it can be poor modelling of the physical problem that gives rise to an ill-conditioned mathematical problem, and so we need to think carefully about the whole modelling process.

We cannot blame software for giving us poor solutions if we provide the wrong problem. We can, of course, hope that the software might provide a measure for the condition of the problem, or some measure of the accuracy of the solution to give us warning of a poorly posed problem.

At the end of Section 4.1 we also mentioned the desirability of careful choice of measurement units, in order to help avoid the effects of poor scaling.

7 Error Bounds and Software

In this section we give examples of reliable software that return information about the quality of the solution. Firstly we look at the freely available software package LAPACK [Anderson et al., 1999], and then at an example of a commercial software library, the NAG Library [NAG]. The author of this report has to declare an interest in both of these software products; he is one of the authors of LAPACK and is a software developer employed by NAG Ltd. Naturally, the examples are chosen because of familiarity with the products and belief in them as quality products, but I have nevertheless tried not to introduce bias.

LAPACK stands for **L**inear **A**lgebra **PACK**age and is a numerical software package for the solution of dense and banded linear algebra problems aimed at PCs, workstations and high-performance shared memory machines. One of the aims of LAPACK was to make the software efficient on modern machines, whilst retaining portability, and to this end it makes extensive use of the Basic Linear Algebra Subprograms (BLAS), using block-partitioned algorithms based upon the Level 3 BLAS wherever possible. The BLAS specify the interface for a set of subprograms for common scalar and vector (Level 1), matrix-vector (Level 2) and matrix-matrix operations (Level 3). Their motivation and specification are given in Lawson et al. [1979], Dongarra et al. [1988a] and Dongarra et al. [1990] respectively. Information on block-partitioned algorithms and performance of LAPACK can be found in Anderson et al. [1999, Chapter 3]. See also Golub and Van Loan [1996, particularly Section 1.3], and [Stewart, 1998, Chapter 2], which also has some nice discussion on computation.

LAPACK has routines for the solution of systems of linear equations, linear least squares problems, eigenvalue and singular value problems, including generalized problems, as well as routines for the underlying computational components such as matrix factorizations. In addition, a lot of effort was expended in providing condition and error estimates. Quoting from the first paragraph of Chapter 4 – Accuracy and Stability – of the LAPACK Users' Guide:

> "In addition to providing faster routines than previously available, LAPACK provides more comprehensive and better error bounds. Our goal is to provide error bounds for most quantities computed by LAPACK."

In many cases the routines return the bounds directly; in other cases the Users' Guide gives details of error bounds and provides code fragments to compute those bounds.

As an example, routine DGESVX[8] solves a system of linear equations $AX = B$, where B is a matrix of one or more right-hand sides, using Gaussian elimination with partial pivoting. Part of the interface is

```
SUBROUTINE DGESVX( ..., RCOND, FERR, BERR, WORK, ..., INFO)
```

where the displayed arguments return the following information:

RCOND	-	Estimate of reciprocal of condition number, $1/\kappa(A)$
FERR(j)	-	Estimated forward error for X_j
BERR(j)	-	Componentwise relative backward error for X_j (smallest relative change in any element of A and B_j that makes X_j an exact solution)
WORK(1)	-	Reciprocal of pivot growth factor, $1/g$
INFO	-	Returns a positive value if the computed triangular factor U is singular or nearly singular

Thus DGESVX is returning all the information necessary to judge the quality of the computed solution.

The routine returns an estimate of $1/\kappa(A)$, rather than $\kappa(A)$ to avoid overflow when A is singular, or very ill-conditioned. The argument INFO is the LAPACK warning or error flag, and is present in all the LAPACK user callable routines. It returns zero on successful exit, a negative value if an input argument is incorrectly supplied, for example $n < 0$, and a positive value in the case of failure, or near failure as above. In the above example, INFO returns the value i if $u_{ii} = 0$, in which case no solution is computed since U is exactly singular, but returns the value $n+1$ if $1/\kappa(A) < u$, in which case A is non-singular to working precision. In the latter case a solution is returned, and so INFO = $n+1$ acts as a warning that the solution may have no correct digits. The routine also has the option to equilibrate the matrix A. See the documentation of the routine for further information, either in the Users' Guide, or in the source code available from netlib (http://www.netlib.org/lapack/index.html).

As a second example from LAPACK, routine DGEEVX solves the eigenproblem $Ax = \lambda x$ for the eigenvalues and eigenvectors, $\lambda_i, x_i, i = 1, 2, \ldots, n$ of the n by n matrix A. Optionally, the matrix can be balanced and the left eigenvectors of A can also be computed. Part of the interface is

```
SUBROUTINE DGEEVX( ..., ABNRM, RCONDE, RCONDV, ...  )
```

where the displayed arguments return the following information:

ABNRM	-	Norm of the balanced matrix
RCONDE(i)	-	Reciprocal of the condition number for the ith eigenvalue, s_i
RCONDV(i)	-	Reciprocal of the condition number for the ith eigenvector, sep$_i$

[8] In the LAPACK naming scheme the D stands for double precision, GE for general matrix, SV for solver and X for expert driver

Following a call to DGEEVX, approximate error bounds for the computed eigenvalues and eigenvectors, say EERRBD(i) and VERRBD(i), such that

$$\begin{aligned} |\tilde{\lambda}_i - \lambda_i| &\leq \text{EERRBD}(i) \\ \theta(\tilde{\nu}_i, \nu_i) &\leq \text{VERRBD}(i), \end{aligned}$$

where $\theta(\tilde{\nu}_i, \nu_i)$ is the angle between the computed and true eigenvector, may be returned by the following code fragment, taken from the Users' Guide:

```
      EPSMCH = DLAMCH('E')
      DO 10 I = 1, N
         EERRBD(I) = EPSMCH*ABNRM/RCONDE(I)
         VERRBD(I) = EPSMCH*ABNRM/RCONDV(I)
   10 CONTINUE
```

These bounds are based upon Table 3, extracted from Table 4.5 of the LAPACK Users' Guide, which gives approximate asymptotic error bounds for the nonsymmetric eigenproblem. These bounds

Simple eigenvalue	$\lvert\tilde{\lambda}_i - \lambda_i\rvert \lesssim \lVert E\rVert_2 / s_i$
Eigenvector	$\theta(\tilde{\nu}_i, \nu_i) \lesssim \lVert E\rVert_F / \text{sep}_i$

Table 3: Asymptotic Error Bounds for $Ax = \lambda x$

assume that the eigenvalues are simple eigenvalues. In addition if the problem is ill-conditioned, these bounds may only hold for extremely small $\|E\|_2$ and so the Users' Guide also provides a table of global error bounds which are not so restrictive on $\|E\|_2$. The tables in the Users' Guide include bounds for clusters of eigenvalues and for invariant subspaces, and these bounds can be estimated using DGEESX in place of DGEEVX. For further details see The LAPACK Users' Guide [Anderson et al., 1999, Chapter 4] and for further information see Golub and Van Loan [1996, Chapter 7] and Stewart and Sun [1990].

LAPACK is freely available via netlib[9], is included in the NAG Fortran 77 Library and is the basis of the dense linear algebra in the NAG Fortran 90 and C Libraries. Tuned versions of a number of LAPACK routines are included in the NAG Fortran SMP Library. The matrix computations of MATLAB have been based upon LAPACK since Version 6 [MathWorks; Higham and Higham, 2000].

We now take an example from the NAG Fortran Library. Routine D01AJF is a general purpose integrator using an adaptive procedure, based on the QUADPACK routine QAGS [Piessens et al., 1983], which performs the integration

$$I = \int_a^b f(x)dx,$$

where $[a, b]$ is a finite interval. Part of the interface to D01AJF is

```
SUBROUTINE D01AJF( ..., EPSABS, EPSREL, RESULT, ABSERR, ...  )
```

[9] http://www.netlib.org/lapack/index.html

where the displayed arguments return the following information:

EPSABS	-	The absolute accuracy required
EPSREL	-	The relative accuracy required
RESULT	-	The computed approximation to I
ABSERR	-	An estimate of the absolute error

In normal circumstances ABSERR satisfies

$$|I - \mathsf{RESULT}| \leq \mathsf{ABSERR} \leq \max(\mathsf{EPSABS}, \mathsf{EPSREL} \times |I|).$$

See the NAG Library documentation [NAG, 2003] and Piessens et al. [1983] for further details. QUADPACK is freely available from netlib[10], and a Fortran 90 version of QAGS is available from the more recent quadature package, CUBPACK [Cools and Haegemans, 2003], which is also available from netlib. Typically the error estimate for a quadrature routine is obtained at the expense of additional computation with a finer interval, or mesh, or the use of a higher order quadrature formula.

As a second example from the NAG Library we consider the solution of an ODE. Routine D02PCF integrates

$$y' = f(t, y), \text{ given } y(t_0) = y_0,$$

where y is the n element solution vector and t is the independent variable, using a Runge-Kutta method. Following the use of D02PCF, routine D02PZF may be used to compute global error estimates. Part of the interface to D02PZF is

```
SUBROUTINE D02PZF( RMSERR, ERRMAX, TERRMX, ...  )
```

where the displayed arguments return the following information:

RMSERR(i)	-	Approximate root mean square error for y_i
ERRMAX	-	Maximum approximate true error
TERRMX	-	First point at which maximum approximate true error occurred

The assessment of the error is determined at the expense of computing a more accurate solution using a higher order method to that used for the original solution.

The NAG D02P routines are based upon the RKSUITE software by Brankin et al. [1992], which is also available from netlib[11]. See also Shampine and Gladwell [1992] and Brankin et al. [1993]. A Fortran 90 version of RKSUITE is also available[12], see Brankin and Gladwell [1997].

Many routines in the NAG Library attempt to return information about accuracy. The documentation of the routines includes a section labelled "Accuracy" which, when appropriate, gives further advice or information. For instance, the optimization routines generally quote the optimality conditions that need to be met for the routine to be successful. These routines are cautious, and sometimes return a warning, or error, when it is likely that an optimum point has been found, but not all the optimality conditions have been met. NAG and the authors of the routines feel that this is much the best approach for reliability – even if users would sometimes prefer that we were more optimistic!

[10] http://www.netlib.org/quadpack/

[11] http://www.netlib.org/ode/rksuite/

[12] http://www.netlib.org/ode/rksuite/ or http://www.netlib.org/toms/771

8 Other Approaches

What does one do if the software does not provide suitable estimates for the accuracy of the solution, or the sensitivity of the problem? One approach is to run the problem with perturbed data and compare solutions. Of course, the difficulty with this approach is to know how best to choose perturbations. If a small perturbation does significantly change the solution, then we can be sure that the problem is sensitive, but of course we cannot rely on the converse. If we can have trust that the software implements a stable method, then any sensitivity in the solution is due to the problem, but otherwise we cannot be sure whether it is the method or problem that is sensitive.

To help estimate such sensitivity there exists software that uses stochastic methods to give statistical estimates of backward error, or of sensitivity. One such example, PRECISE, is described in Chaitin-Chatelin and Frayssé [1996, Chapter 8] and provides a module for statistical backward error analysis as well as a module for sensitivity analysis. Another example is CADNA[13]; see for example Vignes [1993].

Another approach to obtaining bounds on the solution is the use of interval arithmetic, in conjunction with interval analysis [Moore, 1979; Kreinovich; Alefeld and Mayer, 2000]. Some problems can be successfully solved using interval arithmetic throughout, but for some problems the bounds obtained would be far too pessimistic; however interval arithmetic can often be applied as an posteriori tool to obtain realistic bounds. We note that there is a nice interval arithmetic toolbox for Matlab, INTLAB, by Rump [1999] that is freely available[14]; see also Hargreaves [2002]. It should be noted that in general, the aim of interval arithmetic is to return forward error bounds on the solution.

Example 8.1 (Cancellation and interval arithmetic)
As a very simple example consider the computation of s in Example 3.1 using four figure interval arithmetic. Bearing in mind that interval arithmetic works with intervals that are guaranteed to contain the exact solution, we find that

$$\begin{aligned} s = \begin{bmatrix} s1 & s2 \end{bmatrix} &= \begin{bmatrix} 1.000 & 1.000 \end{bmatrix} + \begin{bmatrix} 1.000 \times 10^4 & 1.000 \times 10^4 \end{bmatrix} - \begin{bmatrix} 1.000 \times 10^4 & 1.000 \times 10^4 \end{bmatrix} \\ &= \begin{bmatrix} 1.000 \times 10^4 & 1.001 \times 10^4 \end{bmatrix} - \begin{bmatrix} 1.000 \times 10^4 & 1.000 \times 10^4 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 10 \end{bmatrix}, \end{aligned}$$

so whilst the result is somewhat pessimistic, it does give due warning of the cancellation.

Finally we comment that one should not be afraid to exert pressure on software developers to provide features that allow one to estimate the sensitivity of the problem and the accuracy of the solution.

9 Summary

We have tried to illustrate the niceties of numerical computation and the detail that needs to be considered when turning a numerical algorithm into reliable, robust numerical software. We have also tried to describe and illustrate the ideas that need to be understood to judge the quality of

[13] At the time of writing, a free academic version is available from http://www-anp.lip6.fr/cadna/Accueil.php
[14] http://www.ti3.tu-harburg.de/english/index.html

a numerical solution, especially condition, stability and error analysis, including the distinction between backward and forward errors.

We emphasise that one should most certainly be concerned about the quality of computed solutions, and use trustworthy quality software. We cannot just blithely assume that results returned by software packages are correct.

This is not always easy since scientists wish to concentrate on their science and should not really need to be able to analyse an algorithm to understand whether or not it is a stable method for solving their problem. Hence the emphasis in this report on the desirability of software providing proper measures of the quality of the solution.

We conclude with a quotation:

> "You have been solving these damn problems better than I can pose them."
> Sir Edward Bullard, Director NPL, in a remark to Wilkinson in the mid 1950s. See Wilkinson [1985, p. 11].

Software developers should strive to provide solutions that are at least as good as the data deserves.

References

F. S. Acton. *Numerical Methods that Usually Work*. Harper and Row, New York, USA, 1970.

F. S. Acton. *Real Computing Made Real: Preventing Errors in Scientific and Engineering Calculations*. Princeton University Press, Princeton, NJ, USA, 1996. ISBN 0-691-03663-2.

G. Alefeld and G. Mayer. Interval analysis: Theory and applications. *J. Comput. Appl. Math.*, 121: 421–464, 2000.

E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, 3rd edition, 1999. ISBN 0-89871-447-8. (www.netlib.org/lapack/lug/).

D. Bindel, J. Demmel, W. Kahan, and O. Marques. On computing Givens rotations reliably and efficiently. *ACM Trans. Math. Software*, 28:206–238, 2002.

L. S. Blackford, A. Cleary, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, A. Petitet, H. Ren, K. Stanley, and R. C. Whaley. Practical experience in the numerical dangers of heterogeneous computing. *ACM Trans. Math. Software*, 23:133–147, 1997.

R. W. Brankin and I. Gladwell. Algorithm 771: `rksuite_90`: Fortran 90 software for ordinary differential equation initial-value problems. *ACM Trans. Math. Software*, 23:402–415, 1997.

R. W. Brankin, I. Gladwell, and L. F. Shampine. RKSUITE: A suite of runge-kutta codes for the initial value problem for ODEs. Softreport 92-S1, Mathematics Department, Southern Methodist University, Dallas, TX 75275, USA, 1992.

R. W. Brankin, I. Gladwell, and L. F. Shampine. RKSUITE: A suite of explicit runge-kutta codes. In R. P. Agarwal, editor, *Contributions to Numerical Mathematics*, pages 41–53. World Scientific, River Edge, NJ, USA, 1993. (WSSIAA, vol. 2).

J. L. Britton, editor. *Collected Works of A. M. Turing: Pure Mathematics*. North-Holland, Amsterdam, The Netherlands, 1992. ISBN 0-444-88059-3.

F. Chaitin-Chatelin and V. Frayssé. *Lectures on Finite Precision Computations*. SIAM, Philadelphia, PA, USA, 1996. ISBN 0-89871-358-7.

T. F. Chan, G. H. Golub, and R. J. LeVeque. Algorithms for computing the sample variance: Analysis and recommendations. *The American Statistician*, 37:242–247, 1983.

R. Cools and A. Haegemans. Algorithm 824: CUBPACK: A package for automatic cubature; framework description. *ACM Trans. Math. Software*, 29:287–296, 2003.

M. G. Cox, M. P. Dainton, and P. M. Harris. Testing spreadsheets and other packages used in metrology: Testing functions for the calculation of standard deviation. NPL Report CMSC 07/00, Centre for Mathematics and Scientific Computing, National Physical Laboratory, Teddington, Middlesex TW11 0LW, UK, 2000.

D. S. Dodson. Corrigendum: Remark on "Algorithm 539: Basic Linear Algebra Subroutines for FORTRAN usage". *ACM Trans. Math. Software*, 9:140, 1983.

D. S. Dodson and R. G. Grimes. Remark on algorithm 539: Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Software*, 8:403–404, 1982.

J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 14:1–32, 399, 1988a. (Algorithm 656. See also Dongarra et al. [1988b]).

J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. Corrigenda: "An extended set of FORTRAN Basic Linear Algebra Subprograms". *ACM Trans. Math. Software*, 14:399, 1988b. (See also Dongarra et al. [1988a]).

J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–28, 1990. (Algorithm 679).

A. A. Dubrulle. A class of numerical methods for the computation of Pythagorean sums. *IBM J. Res. Develop.*, 27(6):582–589, November 1983.

G. E. Forsythe. Pitfalls in computation, or why a math book isn't enough. *Amer. Math. Monthly*, 9: 931–995, 1970.

G. E. Forsythe. What is a satisfactory quadratic equation solver. In B. Dejon and P. Henrici, editors, *Constructive Aspects of the Fundamental Theorem of Algebra*, pages 53–61. Wiley, New York, NY, USA, 1969.

L. Fox. How to get meaningless answers in scientific computation (and what to do about it). *IMA Bulletin*, 7:296–302, 1971.

W. Givens. Numerical computation of the characteristic values of a real symmetric matrix. Technical Report ORNL-1574, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, USA, 1954.

G. H. Golub. Numerical methods for solving linear least squares problems. *Numer. Math.*, 7: 206–216, 1965.

G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, USA, 3rd edition, 1996. ISBN 0-8018-5414-8.

S. Hammarling. An introduction to the quality of computed solutions. In B. Einarsson, editor, *Accuracy and Reliability in Scientific Computing*, pages 43–76. SIAM, Philadelphia, PA, USA, 2005. (Accompanying web site for book: www.nsc.liu.se/wg25/book/).

G. Hargreaves. Interval analysis in MATLAB. Master's thesis, Department of Mathematics, University of Manchester, Manchester M13 9PL, UK, 2002.

D. J. Higham and N. J. Higham. *MATLAB Guide*. SIAM, Philadelphia, PA, USA, 2000. ISBN 0-89871-469-9.

D. J. Higham and N. J. Higham. *MATLAB Guide*. SIAM, Philadelphia, PA, USA, 2nd edition, 2005. ISBN 0-89871-578-4.

N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, USA, second edition, 2002. ISBN 0-89871-521-0.

N. J. Higham. Can you "count" on your computer? www.maths.man.ac.uk/~higham/talks/, 1998. (Public lecture for Science Week 1998).

IEEE. *ANSI/IEEE Standard for Binary Floating Point Arithmetic: Std 754-1985*. IEEE Press, New York, NY, USA, 1985.

IEEE. *ANSI/IEEE Standard for Radix Independent Floating Point Arithmetic: Std 854-1987*. IEEE Press, New York, NY, USA, 1987.

E. Isaacson and H. B. Keller. *Analysis of Numerical Methods*. Wiley, New York, NY, USA, 1966. (Reprinted with corrections and new Preface by Dover Publications, New York, 1994, ISBN 0-486 68029-0).

L. Knüsel. On the accuracy of statistical distributions in Microsoft Excel 97. *Comput. Statist. Data Anal.*, 26:375–377, 1998.

V. Kreinovich. Interval computations. www.cs.utep.edu/interval-comp/.

C. L. Lawson and R. J. Hanson. *Solving Least Squares Problems*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1974. (Republished as Lawson and Hanson [1995]).

C. L. Lawson and R. J. Hanson. *Solving Least Squares Problems*. Classics in Applied Mathematics, 15. SIAM, Philadelphia, PA, USA, 1995. ISBN 0-89871-356-0. (Revised version of Lawson and Hanson [1974]).

C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Trans. Math. Software*, 5:308–323, 1979. (Algorithm 539. See also Dodson and Grimes [1982] and Dodson [1983]).

R. S. Martin and J. H. Wilkinson. Similarity reduction of a general matrix to Hessenberg form. *Numer. Math.*, 12:349–368, 1968. (See also [Wilkinson and Reinsch, 1971, pp 339–358]).

MathWorks. MATLAB. The Mathworks, Inc, www.mathworks.com.

B. D. McCullough and B. Wilson. On the accuracy of statistical procedures in Microsoft Excel 2000 and Excel XP. *Comput. Statist. Data Anal.*, 40:713–721, 2002.

B. D. McCullough and B. Wilson. On the accuracy of statistical procedures in Microsoft Excel 97. *Comput. Statist. Data Anal.*, 31:27–37, 1999.

M. Metcalf and J. K. Reid. *Fortran 90/95 Explained*. Oxford University Press, Oxford, UK, 1996.

M. Metcalf, J. K. Reid, and M. Cohen. *Fortran 95/2003 Explained*. Oxford University Press, Oxford, UK, 2004. ISBN 0 19 852693 8.

C. Moler and D. Morrison. Replacing square roots by Pythagorean sums. *IBM J. Res. Develop.*, 27 (6):577–581, November 1983.

R. E. Moore. *Methods and Applications of Interval Analysis*. SIAM, Philadelphia, PA, USA, 1979.

NAG. *The NAG Fortran Library Manual, Mark 20*. The Numerical Algorithms Group Ltd, Wilkinson House, Jordan Hill Road, Oxford OX2 8DR, UK., 2003.
(www.nag.com/numeric/fl/manual/html/FLlibrarymanual.asp, or
www.nag.co.uk/numeric/fl/manual/html/FLlibrarymanual.asp).

NAG. The NAG Library. NAG Ltd, www.nag.com/numeric/numerical_libraries.asp, or www.nag.co.uk/numeric/numerical_libraries.asp.

M. L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM, Philadelphia, PA, USA, 2001. ISBN 0-89871-482-6.

R. Piessens, E. de Doncker-Kapenga, C. W. Überhuber, and D. K. Kahaner. *QUADPACK – A Subroutine Package for Automatic Integration*. Springer-Verlag, Berlin, Germany, 1983.

D. M. Priest. Efficient scaling for complex division. *ACM Trans. Math. Software*, 30:389–401, 2004.

S. M. Rump. INTLAB – INTerval LABoratory. In T. Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic, Dordrecht, The Netherlands, 1999.

L. F. Shampine and I. Gladwell. The next generation of rünge-kutta codes. In Cash J. R. and I. Gladwell, editors, *Computational Ordinary Differential Equations*, pages 145–164. Oxford University Press, Oxford, UK, 1992. (IMA Conference Series, New Series 39).

R. L. Smith. Algorithm 116: Complex division. *Communs Ass. comput. Mach.*, 5:435, 1962.

G. W. Stewart. *Matrix Algorithms: Basic Decompositions*, volume I. SIAM, Philadelphia, PA, USA, 1998. ISBN 0-89871-414-1.

G. W. Stewart. A note on complex division. *ACM Trans. Math. Software*, 11:238–241, 1985.

G. W. Stewart and J. Sun. *Matrix Perturbation Theory*. Academic Press, London, UK, 1990.

A. M. Turing. Rounding-off errors in matrix processes. *Q. J. Mech. appl. Math.*, 1:287–308, 1948. (Reprinted in Britton [1992] with summary, notes and corrections).

J. Vignes. A stochastic arithmetic for reliable scientific computation. *Math. and Comp. in Sim.*, 35: 233–261, 1993.

J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Notes on Applied Science, No.32. HMSO, London, UK, 1963. (Also published by Prentice-Hall, Englewood Cliffs, NJ, USA, 1964, translated into Polish as Bledy Zaokragleń w Procesach Algebraicznych by PWW, Warsaw, Poland, 1967 and translated into German as Rundungsfehler by Springer-Verlag, Berlin, Germany, 1969. Reprinted by Dover Publications, New York, 1994).

J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, Oxford, UK, 1965. (Also translated into Russian by Nauka, Russian Academy of Sciences, 1970).

J. H. Wilkinson. The perfidious polynomial. In G. H. Golub, editor, *Studies in Numerical Analysis, Volume 24*, chapter 1, pages 1–28. The Mathematical Association of America, 1984. (Awarded the Chauvenet Prize of the Mathematical Association of America).

J. H. Wilkinson. Error analysis revisited. *IMA Bulletin*, 22:192–200, 1986. (Invited lecture at Lancaster University in honour of C. W. Clenshaw, 1985).

J. H. Wilkinson. Error analysis of direct methods of matrix inversion. *J. ACM*, 8:281–330, 1961.

J. H. Wilkinson. The state of the art in error analysis. *NAG Newsletter*, 2/85:5–28, 1985. (Invited lecture for the NAG 1984 Annual General Meeting).

J. H. Wilkinson. Error analysis of floating-point computation. *Numer. Math.*, 2:319–340, 1960.

J. H. Wilkinson and C. Reinsch, editors. *Handbook for Automatic Computation, Vol.2, Linear Algebra*. Springer-Verlag, Berlin, Germany, 1971.