Appendix K

Roundoff Noise in Scientific Computations

Roundoff errors in calculations are often neglected by scientists. The success of the IEEE double precision standard makes most of us think that the precision of a simple personal computer is virtually infinite. Common sense cannot really grasp the meaning of 16 precise decimal digits.

However, roundoff errors can easily destroy the result of a calculation, even if it looks reasonable. Therefore, it is worth investigating them even for IEEE double-precision representation.

K.1 COMPARISON TO REFERENCE VALUES

Investigation of roundoff errors is most straightforwardly based on comparison of the results to *reference values*, that is, to the outcome of an ideally precise calculation.

First of all, we need to note that the way of *comparison* is not well defined. It is usually done by looking at the difference of the finite precision result and of the reference value. This is very reasonable, but cannot be applied in all cases. If, for example, we investigate a stand-alone resonant system, the reference and the finite precision results could be two similar sine waves, with slightly different frequency. In such a case, the difference of the imprecise result and precise result grows significantly with time, although the outputs are still very similar. Therefore, the basis of comparison needs to be chosen very carefully.

Secondly, having infinitely precise reference values seems to be a dream only. However, this is sometimes quite reasonable.

K.1.1 Comparison to Manually Calculable Results

Relying on calculable results seems to be applicable in trivial cases only, but this is not true. The general condition of usability is to have

- the possibility to calculate the exact output or at least a part of the exact output,
- complex enough input which can represent the set of typical data.

A good example is the evaluation of the FFT or of any digital filter by using random phase multisine excitation (Paduart, Schoukens and Rolain, 2004). We use integer periods of a periodic signal containing certain harmonic components with phases randomly set, and certain harmonic components set to exactly zero. Random phases make the amplitude distribution of the time domain signal more or less Gaussian (Pintelon and Schoukens, 2001). Zero amplitudes make the steady-state result at the corresponding frequencies contain roundoff noise only. Evaluation of these gives a measure of the roundoff noise variances at the output.

K.1.2 Increased Precision

When theoretically correct results are not available, the most precise determination of the reference value is available by using increased precision. This is however a paradoxical requirement, since when extended precision is available for calculation of the reference values, why would somebody be interested in the lower-precision result at all?

We may want to characterize the error of usual-precision calculations, accepting slower determination of the reference value. When an *overall roundoff analysis* of an algorithm with given precision is to be performed, having a reference value available we can compare the results of several runs to this value.

There are two major ways to obtain increased-precision results on an IEEE double-precision machine.

(a) Use the so-called *long double* numbers available in certain compilers (e.g. Borland C, or Microsoft C and so on). These define operations on numbers with extra long bit numbers, so the result has a much lower roundoff error than usual ones. Their use however requires special care, since the precision is only increased significantly if all the operations (like trigonometric functions etc.) are done in long double precision, and it is also difficult to extract the results in extended precision.

(b) Use the so-called *multiple-precision* arithmetic packages. These implement increased precision calculations on double-precision machines.

Both solutions are rather slow, since they are not directly supported by the fast arithmetic co-processors.

K.1.3 Ambiguities of IEEE Double-Precision Calculations

In many cases, IEEE double precision is sufficient for calculating a reference value. One of the aims of IEEE double precision (page 343) as a standard was that people

needed reproducible results: the same computer code executed on two different, but IEEE-compatible arithmetic processors was assumed to yield the same result. Therefore, the results in IEEE double precision are accurately defined in the standard.

This aim was almost reached with the appearance of IEEE-compatible machines, like the personal computer. However, there is one significant difference between these machines, and sensitive calculations must consider this. Certain operations, like "fused multiply and add" or "multiply and accumulate", MAC ($A = A + B * C$), can be directly evaluated in the accumulator. The IEEE standard allows the use of *extended-precision* arithmetic, with longer than usual significand (mantissa) and exponent (IEEE, 1985), which is usually implemented as increased precision of the accumulator. Certain processors, like the Pentium processors in PCs make use of this, so that operations executed in the accumulator are calculated with 64 mantissa bits. 53 mantissa bits are used in memory storage.

However, it is determined not by the processor itself if this extended precision is utilized or not. The compiler has the ultimate decision about the implementation of certain operations. For example, in Matlab, earlier versions of the program calculated intermediate results of fused multiply and add operations (e.g. the scalar product of vectors, or matrix multiplications) in the accumulator, and therefore on a PC the result was more precise than e.g. on a Sun computer. However, in Matlab this is not done any more (Kahan, 1998). Every operation is executed with the same, that is, non-extended precision. This has the advantage that Matlab produces exactly the same results on every machine, let it be a PC, a Sun, or a Macintosh, that is, even the roundoff errors are the same, so the results are directly interchangeable. On the other hand, it is a pity that while the processor of a PC could work with less roundoff in certain operations, we do not make use of this possibility.

The standard clearly defines the result of multiplications, additions, subtractions and divisions, even of the square root function. They must be as close as possible to the theoretical results. This is expressed as "the error may not be larger than 0.5 ULPs" (units in the last place[1]). However, this is not the case for more general functions, such as $\sin(\cdot)$, or $\log(\cdot)$. Because of difficulties of hardware realization, it is accepted in the evaluation of transcendental functions to have an error of 1 ULP, which allows a small hardware-dependent ambiguity in these evaluations.

Furthermore, for results theoretically at (integer$+0.5$) LSB, the result of rounding is not uniquely defined. Although the standard contains a description, and most modern arithmetic processors implement round-to-even, this is not ubiquitous. Therefore, arithmetic units may differ even in this. Even Matlab, which has become a de facto standard program in scientific calculations, implements a $\text{round}(\cdot)$ function

[1] Unit in the Last Place ($\text{ulp}(x)$) "is the gap between the two finite floating-point numbers nearest x, even if x is one of them." For definitions of the ulp function, see (Muller, 2005). In other words, ulp is the smallest possible increment or decrement that can be made using the machine's floating point arithmetic. For representable numbers, this is almost always equal to 1 LSB (Least Significant Bit) of the number, except for non-underflow powers of 2, where it equals 0.5 LSB.

which rounds all halves towards $\pm\infty$ (round([$-3.5 : 1 : 3.5$])=[$-4, -3, -2, -1, 1, 2, 3, 4$]), while the processor itself executes round-to-even. This is probably inherited from earlier Matlab implementations, and stays so. Therefore, simulations in Matlab need extra care if these 0.5 LSB values often occur.

K.1.4 Decreased-Precision Calculations

When increased precision is not directly available, we may also evaluate the algorithm with *decreased precision*. This means that the same operation is performed with quantizers of lower bit numbers inserted into the signal path. We can recognize that for fixed-point number representation the magnitude of the roundoff error is approximately proportional to $q/2$, and for floating-point the magnitude of the relative error is proportional to 2^{-p}. When a certain algorithm is performed with different bit length, the tendency of the roundoff error can be estimated. Then, double-precision results can be handled as reference values, and the lower-precision results can yield the roundoff errors which allow extrapolation to double precision or further.

Since the roundoff errors can usually be considered as random, a desirable method would calculate a set of independent results, and make a statistical analysis. This would usually give a precise indication of the error levels.

K.1.5 Different Ways of Computation

An easy and useful way can be followed when the inverse of an algorithm can also be evaluated. When the algorithm and its inverse are repeatedly calculated, like the FFT and the IFFT, the differences of the results of the sequences FFT, FFT–IFFT–FFT, FFT–IFFT–FFT–IFFT–FFT and so on is a characteristic of the roundoff errors. In *open-loop* calculations like the FFT, the roundoff errors will accumulate, but the error level can be evaluated directly in successive differences of the results (Fig. K.1). The only difficulty is that we cannot distinguish between the error of the FFT and of the IFFT.

Figure K.2 illustrates the errors obtained by the execution of Matlab's `fft/ifft` pair (these are IEEE double-precision floating-point algorithms).

In general, a good test of the calculation result is to calculate the same quantity in two or more ways, possibly with different algorithms, and compare the results. Even when one algorithm is available, application to different signals and analysis of the results can yield a reasonable estimate of the roundoff error. For example, for linear operations the input signal can be decomposed into two or more parts, while for nonlinear operations the input signal may be slightly perturbed. An example for the linear case is when the roundoff error of an FFT algorithm is to be checked. The decomposition of the input sequence into two sequences: $\{x_i\} = \{y_i\} + \{z_i\}$, and comparison of $\{X_k\}$ with $\{Y_k\} + \{Z_k\}$ will give a picture of the order of magnitude of

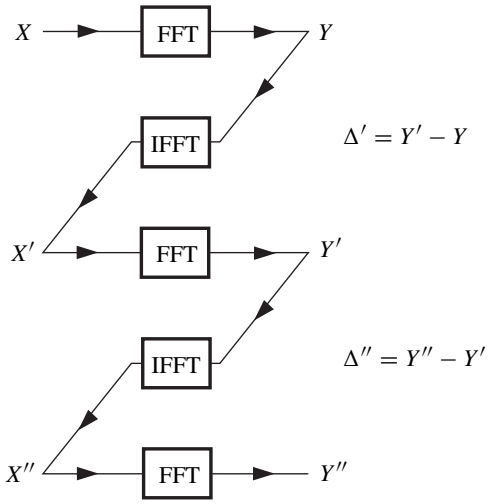

Figure K.1 IFFT–FFT pairs yielding roundoff errors.

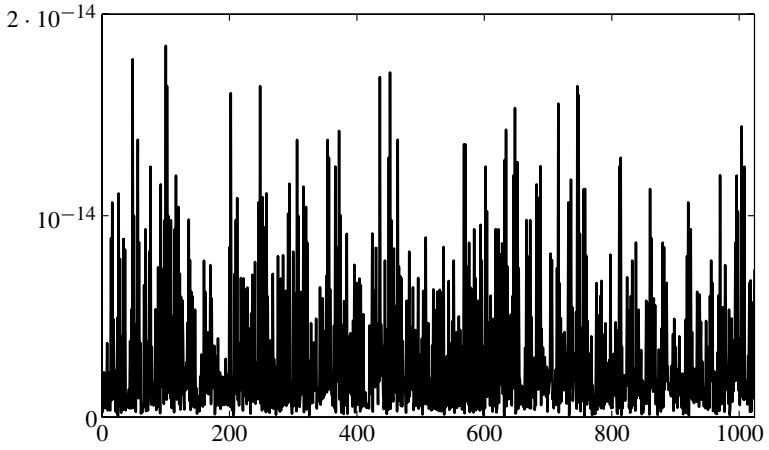

Figure K.2 The absolute value of the error after execution of Matlab's `fft/ifft` command pair on an $N = 1024$-point white Gaussian sequence, with zero mean and $\sigma = 1$.

the roundoff errors, at least when $\{y_i\}$ and $\{z_i\}$ have significantly different nature. The basic requirement is that the roundoff errors of the two calculation methods be at least approximately independent. Therefore, the choice $\{y_i\} = \{z_i\} = \{x_i\}/2$ is obviously wrong, $\{y_i\} = (1/\pi)\{x_i\}$ and $\{y_i\} = (1 - 1/\pi)\{x_i\}$ is better. We might recommend

e.g. the selection of a sequence $\{r_i\}$ whose members are all randomly distributed between (0,1), and use the decomposition $\{y_i\} = \{r_i\}.*\{x_i\}$ and $\{z_i\} = \{1-r_i\}.*\{x_i\}$, where $.*$ denotes pointwise multiplication. Fortunately, there are infinitely many such decompositions, depending on the sequence, which provide varying roundoff errors.

K.1.6 The Use of the Inverse of the Algorithm

Sometimes the *inverse algorithm* can be evaluated more reliably than the forward algorithm. In such cases *error feedback* calculations can be applied, where the error in the restoration of the input is used to generate a correction of the result (Fig. K.3). The simplest examples are the correction of the result of division by calculating the product of the divisor and the ratio and subtraction of this from the dividend:

$$Y(k+1) = Y(k) + (X - Y(k) * D)/D\,, \tag{K.1}$$

or Newton's algorithm to calculate the square root of X:

$$Y(k+1) = \frac{1}{2}\left(Y(k) + \frac{X}{Y(k)}\right)\,. \tag{K.2}$$

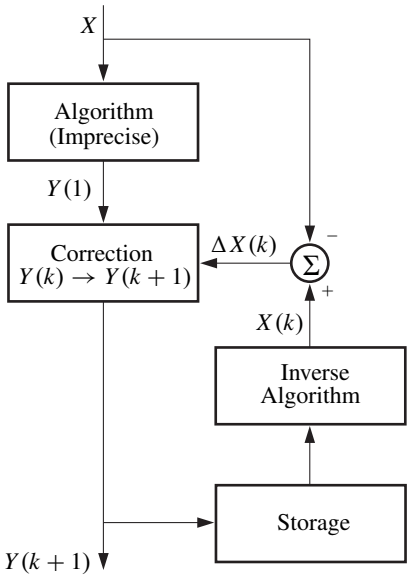

Figure K.3 Evaluation based on the inverse algorithm (e.g. division based on multiplication). $\Delta X(1) = 0$.

K.2 THE CONDITION NUMBER

Modern engineering is often based on solutions of linear equations. In such cases, the usual measure of reliability of the results is the *condition number* defined as $||\mathbf{A}|| \cdot ||\mathbf{A}^{-1}||$. For $||\cdot||$ being the l_2 norm, the condition number equals the ratio of the largest and smallest singular values of the system matrix $\mathbf{A}$. Since the work of Wilkinson (1994) we know that the solution of a linear equation is only reliable if the condition number is reasonably smaller than the reciprocal of the machine precision.[2]

While the condition number seems to be a universal measure for the calculability of the solution of linear matrix equations, sometimes it may be misleading, even in simple cases. The following example which illustrates that the condition number alone is not a sufficient indicator of roundoff problems, is due to De Moor (1991).

Example K.1 Non-associative Matrix Product
The machine precision is equal in IEEE double precision to $eps = 2.2204 \cdot 10^{-16}$. Let us define now ϵ_m as the largest number for which $1 + \epsilon_m$ is rounded to 1. This is thus smaller than *eps*. On machines that implement 'round to even' or 'convergent' rounding for the numbers whose fraction (mod LSB) is exactly 0.5 LSB, ϵ_m is equal to the half of *eps*. For IEEE double precision, $\epsilon_m = eps/2 = 1.1102 \cdot 10^{-16}$.

Let us choose now a value α for which the following two inequalities hold:

$$\alpha^2 < \epsilon_m < \alpha\,, \tag{K.3}$$

and that can be represented on the given computer.

The largest such value is $\alpha = \mathrm{fl}(\sqrt{\epsilon_m} * (1 - eps))$, where $\mathrm{fl}(\cdot)$ denotes the floating-point rounding operation. For IEEE double precision, e.g. $\alpha = 10^{-8}$ meets these conditions.

Let us define now the following matrices:

$$\mathbf{A}_1 = \begin{pmatrix} 1 & \alpha & 0 \\ 1 & 0 & \alpha \end{pmatrix} \qquad \mathbf{A}_2 = \begin{pmatrix} 1 & 1 \\ \alpha & 0 \\ 0 & \alpha \end{pmatrix}, \tag{K.4}$$

$$\mathbf{A}_3 = \begin{pmatrix} -1 & \alpha & 0 \\ 1 & 0 & \alpha \end{pmatrix} \qquad \mathbf{A}_4 = \begin{pmatrix} -1 & 1 \\ \alpha & 0 \\ 0 & \alpha \end{pmatrix}. \tag{K.5}$$

Let us evaluate the product $\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4$. The theoretical value is

[2]Machine precision is the resolution in the given number representation: the size of the step from 1 to the next larger representable number.

$$\Pi = \mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4 = \begin{pmatrix} 2\alpha^2+\alpha^4 & 0 \\ 0 & 2\alpha^2+\alpha^4 \end{pmatrix} = (2\alpha^2+\alpha^4)\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \quad \text{(K.6)}$$

From linear algebra we know that the product can be evaluated executing the matrix multiplications in arbitrary order (associativity), as long as the order of the matrices themselves is maintained. However, for the above case, the order of the execution of the multiplications significantly changes the result, due to the arithmetic rounding performed after each scalar multiplication and addition. Let us notice that

$$\mathrm{fl}(\mathbf{A}_1\mathbf{A}_2) = \mathrm{fl}\left(\begin{pmatrix} 1+\alpha^2 & 1 \\ 1 & 1+\alpha^2 \end{pmatrix}\right) = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad \text{(K.7)}$$

$$\mathrm{fl}(\mathbf{A}_3\mathbf{A}_4) = \mathrm{fl}\left(\begin{pmatrix} 1+\alpha^2 & -1 \\ -1 & 1+\alpha^2 \end{pmatrix}\right) = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}, \quad \text{(K.8)}$$

$$\mathrm{fl}(\mathbf{A}_2\mathbf{A}_3) = \begin{pmatrix} 0 & \alpha & \alpha \\ -\alpha & \alpha^2 & 0 \\ \alpha & 0 & \alpha^2 \end{pmatrix}. \quad \text{(K.9)}$$

Evaluating the product as $\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4 = (\mathbf{A}_1\mathbf{A}_2)(\mathbf{A}_3\mathbf{A}_4)$:

$$\Pi_{\mathrm{fl}1} = \mathrm{fl}\big(\mathrm{fl}(\mathbf{A}_1\mathbf{A}_2)\mathit{fl}(\mathbf{A}_3\mathbf{A}_4)\big) = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}. \quad \text{(K.10)}$$

The "natural order" of evaluation, $\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4 = \big((\mathbf{A}_1\mathbf{A}_2)\,\mathbf{A}_3\big)\mathbf{A}_4$ gives

$$\Pi_{\mathrm{fl}2} = \mathrm{fl}\Big(\mathrm{fl}\big(\mathrm{fl}(\mathbf{A}_1\mathbf{A}_2)\mathbf{A}_3\big)\mathbf{A}_4\Big) = \begin{pmatrix} \alpha^2 & \alpha^2 \\ \alpha^2 & \alpha^2 \end{pmatrix}. \quad \text{(K.11)}$$

$\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4 = \big(\mathbf{A}_1(\mathbf{A}_2\mathbf{A}_3)\big)\mathbf{A}_4$ yields

$$\Pi_{\mathrm{fl}3} = \mathrm{fl}\Big(\mathrm{fl}\big(\mathbf{A}_1\mathrm{fl}(\mathbf{A}_2\mathbf{A}_3)\big)\mathbf{A}_4\Big) = \begin{pmatrix} 2\alpha^2 & 0 \\ 0 & 2\alpha^2 \end{pmatrix}. \quad \text{(K.12)}$$

The ranks of the calculated products are equal to 0, 1, 2, respectively!

Because of the roundoff errors, associativity of multiplications does not precisely hold in practical realizations. The example illustrates this for small matrices, but numerical imprecision in the associativity can be shown even for scalar numbers.

The difference is in most cases so small that we do not care. However, in the example, the results qualitatively differ (observe the rank of the matrices). The important difference between the three calculated products is surprising because the condition number of each that used matrix $\mathbf{A}_i$ is $1/\alpha$, far below the reciprocal of the machine epsilon ($10^8 \ll 1/eps = 4.5 \cdot 10^{15}$ for IEEE double precision), and the nonzero elements in the first two products are all ± 1, so no one would suspect any serious problem. Therefore, a check of the condition numbers of the multiplicands and an examination of the results do not reveal the danger, unless we require the very conservative condition that the *product of the condition numbers* of the multiplicands must be lower than the reciprocal of the machine epsilon, a rule which is almost never followed.

The cause of the trouble is that at certain stages we calculate the difference of numbers which are very close to each other, and rounding errors can bring the difference to either positive, negative or zero value.

A simple example illustrates that the condition numbers of the multiplicands is not really relevant to the problem.

Example K.2 Good and Bad Conditioning of Product
Define two products:

$$\Pi_a = \begin{pmatrix} 1 & 0 \\ 0 & 10^{-8} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 10^{-8} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 10^{-16} \end{pmatrix}, \tag{K.13}$$

$$\Pi_b = \begin{pmatrix} 1 & 0 \\ 0 & 10^{-8} \end{pmatrix} \begin{pmatrix} 10^{-8} & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 10^{-8} & 0 \\ 0 & 10^{-8} \end{pmatrix}. \tag{K.14}$$

The first product is extremely badly conditioned, while the second one is extremely well conditioned, while the condition numbers of the multiplicands are exactly the same.

Does this mean that we should stop looking at the condition numbers? Not at all. The condition number is a very useful indication of possible problems in the solution of linear equations, calculating the eigenvalues, and so on. But a good condition number alone does not guarantee the correctness of the result of a linear or nonlinear operation.

Unfortunately, there is no general recipe. Each calculation needs to be considered individually, and reasonable approximations and estimations must be made. Some general principles can however be formulated.

K.3 UPPER LIMITS OF ERRORS

In most cases, we are interested not in the actual value of the error, but in the *upper limit* of it. We are then content to know that the error is certain or likely to be smaller

than a given quantity. In some cases a good theoretical upper bound of the roundoff errors is available, for example: (Wilkinson, 1994; Forsythe and Moler, 1967).

Another possibility is performing the calculations with *interval arithmetic*. This means that instead of obtaining the error, its upper and lower bounds are calculated, assuming worst-case conditions. Since roundoff errors are very unlikely to act all in the same direction, the obtained upper and lower bounds of the results are usually very pessimistic, so we obtain a very wide interval which contains the true result with 100% certainty. Interval arithmetic assures that the true value will be in any case between these bounds, but cannot account for the fact that the result is likely to be close to the middle of the interval rather than close to the bounds. This makes the results practically unusable for most iterative algorithms (e.g. for the output of IIR filters or of control loops).

A more realistic, although theoretically not perfectly correct bound can be obtained using the PQN model in the calculations. The basic idea is to perform increased-precision calculations, and instead of quantization, to add independent PQN noise at every quantizing step. Evaluation can be done by using Monte Carlo experiments, or by evaluation of the accumulation of variances.

The "QDSP Toolbox for Matlab", available from the website of the book provides basic tools to perform PQN-based Monte Carlo evaluation of the error.

Since we usually want to evaluate the roundoff error of a given precision on the same machine, the theoretically correct result is not available. We cannot eliminate quantization noise during evaluation. However, we can make it independent (uncorrelated) by adding dither at every step.[3] The results will be obtained for the roundoff noise *and* dither.

Sometimes the effect of roundoff errors may be significantly magnified by the algorithm itself, and then linear error models may not apply. Such a case is the minimization of cost functions approximately quadratic around the minimum $x_{\min}$ (Press, Flannery, Teukolsky and Vetterling, 1988, Section 10.2), as in least squares or maximum likelihood estimation. We look for argument values whose small change does not cause significant changes in the cost function. If, for example, the cost function can be expressed as $C(x_{\min} + \mathrm{d}x) = C_{\min} + c \cdot \mathrm{d}x^2$, a change of $\mathrm{d}x$ causes a change of $c \cdot \mathrm{d}x^2$ in it. Since only a relative change larger than *eps/2* can be detected, with *eps* being the machine precision, the location of the minimum (the argument value) can be determined only to the following precision:

$$\begin{aligned} \frac{c \cdot \mathrm{d}x^2}{C_{\min}} &> \frac{eps}{2} \\ \frac{\mathrm{d}x}{x_{\min}} &> \frac{1}{x_{\min}} \sqrt{\frac{C_{\min}}{2c}} \sqrt{eps}\,. \end{aligned} \tag{K.15}$$

[3] In order to have uncorrelated error, triangular dither in $(\pm q)$ can be recommended.

This relative error is usually much larger than the machine precision *eps*. The location of the minimum can be less precisely calculated than the value of the cost function at the minimum, although we are interested in the former value.

K.4 THE EFFECT OF NONLINEARITIES

Special care should be taken when applying nonlinear operations.

With statistics at hand, one might consider the mean value of the set as a reference value, and calculate the deviation from it as the error. This is often wrong. The mean value is consistent (the arithmetic mean converges to the true value when increasing the number of different calculations) only if the mean value of the errors is zero. This may not be true if nonlinear operations are involved.

Depending on the form of nonlinearity, quantization theorems may not hold any more, and quantization noise becomes highly non-uniformly distributed. A simple example is the use of the logarithm function. For the number $x = 8.94 \cdot 10^{306}$, the expression $10^{\log_{10} x}$-x which is theoretically zero, gives $-1.04 \cdot 10^{294}$, that is, the error relative to x is $-1.16 \cdot 10^{-13} \approx -523 \cdot eps$. The cause of this deviation is certainly not an error in evaluation of the functions. The problem is that the logarithm contracts the numbers to a grid finer than representable in IEEE double precision. Quantization which is necessary to store the result in this format rounds to the nearest integer, and this error is magnified back when applying the function $10^{(\cdot)}$.

It is very difficult to give general rules which apply to many cases. There are problems which are numerically difficult to handle, and there are cases which require special care. Instead of further discussion, we present here two more examples.

Example K.3 Roots of Wilkinson's Polynomial

A well-known problem with bad conditioning is due to Wilkinson, the great numerical mathematician, involving polynomials whose roots are successive integer numbers. We will discuss a related case. For sake of simplicity, we give the Matlab code of the calculations.

Define a polynomial whose roots are 1,2,...,10, and solve it, then calculate the errors.

```
r=[1:10]'; err=flipud(roots(poly(r)))-r;
%Roots and their errors:
 1  -5.8842e-015
 2   1.2226e-012
 3  -2.9842e-011
 4   2.7292e-010
 5  -1.2226e-009
 6   3.0459e-009
 7  -4.4523e-009
 8   3.8089e-009
```

```
 9  -1.7700e-009
10   3.4584e-010
```

These errors are well above $eps = 2.2204 \cdot 10^{-16}$.

Let us look at a similar problem with one more root at 10.

```
r=[1:10,10]'; err=flipud(roots(poly(r)))-r;
%Roots and their errors:
 1  -5.2514e-014
 2   1.7337e-012
 3  -2.5234e-011
 4   2.2806e-010
 5  -1.2284e-009
 6   3.8950e-009
 7  -7.4407e-009
 8   8.7565e-009
 9  -6.7227e-009
10   1.2679e-009 -2.9784e-005i
10   1.2679e-009 +2.9784e-005i
```

The double root at 10 seemingly causes increased error. It is well known that numerical polynomial solvers have difficulties in solving for multiple roots. Here the double root became a complex pair, with an error much higher than above with single roots. But is this error large or it is not large?

The answer depends on our point of view. If the question is whether the polynomial has complex poles or only real ones, this error is obviously not tolerable. If the question is whether there are double poles or all poles are different, the criterion needs to be carefully chosen. We can e.g. look for pole pairs not further than 10^{-4} from each other, and recognize the first two poles as a pole pair. But how can we set the error limit properly? This is a very difficult question, the answer depending on the actual requirements.

This example emphasizes a general problem of numerical calculations. In practice, virtually every numerical result is prone to roundoff errors. In most cases, this is not a big problem, but when the results qualitatively change because of the roundoff, like the nature of the roots above, puzzling situations may occur. It is even more problematic when the nature of the results serves as a basis of further decisions.

Example K.4 LS Solution of an Overdetermined Linear Equation
In system identification, an often occurring task is to find the solution $\mathbf{p}$ of

$$\mathbf{z} = \mathbf{X}\mathbf{p} \tag{K.16}$$

when the system matrix $\mathbf{X}$ and the observation vector $\mathbf{z}$ are known. The size of the system matrix $\mathbf{X}$ is $n_z \times n_p$, where $n_z > n_p$. The solution is usually sought in the LS sense: $|\mathbf{z} - \mathbf{X}\mathbf{p}|^2$ should be minimal. Theoretically, the solution is

$$\hat{\mathbf{p}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{z}\,. \tag{K.17}$$

This expression is usually not evaluated as it is, but a numerical approximation is given using a matrix factorization of $\mathbf{X}$. In the following, we will use the singular value decomposition of $\mathbf{X}$ (Golub and van Loan, 1989):

$$\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^{\mathrm{T}}, \tag{K.18}$$

where U and V are unitary matrices: $\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{E}$ and $\mathbf{V}^{\mathrm{T}}\mathbf{V} = \mathbf{E}$, and $\mathbf{S}$ is a diagonal matrix of nonnegative singular values. With these, the LS solution (K.17) becomes:

$$\hat{\mathbf{p}} = \mathbf{V}\mathbf{S}^{-\mathbf{1}}\mathbf{U}^{\mathrm{T}}\mathbf{z}. \tag{K.19}$$

If the rank of $\mathbf{X}$ is less than n_p, the inverse does not exist. In such cases the so-called Moore–Penrose pseudoinverse is used. In terms of the SVD, (K.19) is modified: degenerate rank means that at least one singular value equals zero, and the pseudoinverse is defined by replacing the reciprocals of the zero singular values by zero in the inverse of the diagonal matrix $\mathbf{S}$.

The numerical problem arises in practice since none of the theoretically zero singular values really equals zero, because of roundoff errors. We need to decide however, which are the singular values which are equal to zero, that is, their numerically evaluated value contains roundoff noise only. Since the *algorithm* needs to be changed by this decision, careful consideration is needed. Bad decisions cause significant changes in the estimated parameter vector $\hat{\mathbf{p}}$.[4]

On the basis of PQN theory, the power of the roundoff noise on the singular values can be estimated. From this, a reasonable upper bound can be obtained. If a singular value is below this level, then there is no reason to think that this singular value differs from zero. If it is above the level, we consider it as nonzero.

[4] Although the solution in $\hat{\mathbf{p}}$ may significantly change, the value of the least squares cost function is practically insensitive to this change. This is a phenomenon similar to that discussed considering Eq. (K.15).