Software Design and Development

Numerical Issues

N.M. Maclaren

nmm1@cam.ac.uk

September 2019

3.1 Numerical Analysis

This analyses the effects of approximate calculations, both those caused by floating-point arithmetic and the way that variations build up in the underlying mathematics, and is **not** covered in this course. The Department of Applied Mathematics and Theoretical Physics has three courses on it. In the days when it was taught more widely, it was the subject of year-long diplomas, and even the scientists' introductions were several afternoons long; unfortunately, it is almost never taught nowadays. You are recommended to use a reasonably trustworthy package or library, because it is more reliable than doing your own thing.

This is not a numerical analysis course.

For a very brief introduction to numerical analysis, see:

`https://www-internal.lsc.phy.cam.ac.uk/nmm1/Arithmetic/na1.pdf`

and, if you are really interested:

`https://www-internal.lsc.phy.cam.ac.uk/nmm1/Arithmetic/na2.pdf`

Other useful references include:

`https://www.cl.cam.ac.uk/teaching/1819/Numerical_Analysis_2019.pdf`
`http://www.damtp.cam.ac.uk/user/hf323//L19-IB-NA/index.html`

There are many good numerical analysis books, but a large proportion are hard going, old (though not outdated), expensive, out-of-print, or all of the above! The NAG library documentation is also very good, and is a recommended source:

`https://www.nag.com/content//software-documentation`

The modern authoritative reference is:

The Accuracy and Stability of Numerical Algorithms by *Nicholas J. Higham*

The classic reference is:

The Algebraic Eigenvalue Problem by *J.H. Wilkinson*

Another very highly regarded book is:

Matrix Computations (now 4th ed.) by *Gene H. Golub* and *Charles F. Van Loan*

The NAG library is most general reliable library, by a very large margin; the Numerical Algorithms Group is the leading numerical software house in the world. There are some very good open-source libraries (e.g. LAPACK), but many others are seriously unreliable or worse. For reasons that I am not going into now, do **not** trust Numerical Recipes, most especially not the first edition. The same applies to the Web, which has not been called

The Web of a Million Lies for no reason. In both cases, some parts are good, and some are ghastly.

Your field may well have a preferred library, which may be almost mandatory; as usual, those vary immensely in quality. As always, check with your supervisor or experts in your field for recommendations.

3.2 Low-Level Accuracy Issues

These are commonly taught, but a surprising number of people either don't realise their importance, or have forgotten them. They are worth repeating.

A program's results are almost never better than its input; the old term is GIGO, which stands for Garbage In, Garbage Out, and there is a great deal of truth in it. They can be much less accurate, too, and a common error is to assume machine precision in the result when the input is supposedly exact. Some forms of input error can be reduced statistically, but not all, and there are some tricky issues which are described later. N digits of input means at most $10^{-(N-1)}$ accuracy, and it is common for the last few digits on data loggers not to be correct, so it can be a lot lower. Also, many physical constants are imprecise, even the fundamental ones.

It is important not to expect the impossible in calculations. Given a function F, and absolute error in its input δ, the absolute error in its result will be at least $F'(X) \times \delta$. A simple example is arc tangent near $\pi/2$. Any function with singularities has this problem very seriously in their vicinity, but it can occur almost as badly in functions with none. You should always do a quick check for such issues, which may require rethinking the approach if there is a problem.

4.2.1 Cancellation

This is the low-level cause of most loss of accuracy during calculations, and is caused by subtracting two nearly-equal values – this obviously includes adding two with different signs, but also includes dividing two nearly-equal numbers (and multiplying by almost a number's inverse). If we assume numbers have P digits of precision, and X and Y have Q leading digits in common, then $X - Y$ and $X/Y - 1.0$ have $P - Q$ digits of precision.

Let us start with a semi-realistic example. The K'th differences of a sequence give an approximation to the K'th derivative, and the K'th derivative of x^K should be K factorial. As numerical analysts know, that is a classically ill-conditioned problem - but did you? In double precision, here are the true values, the range of the calculated values and the relative errors of the K'th differences of x^K calculated from 0.5 to 2.0 in steps of 0.01. They give 1 significant figure for K=7, and nonsense for K=8 and above.

K:	$K!$	Values	Relative Error
1:	1	1.0 to 1.0	1.1×10^{-14}
2:	2	2.0 to 2.0	6.7×10^{-12}
3:	6	6.0 to 6.0	7.4×10^{-10}
4:	24	24.0 to 24.0	6.3×10^{-08}
5:	120	120.0 to 120.0	3.6×10^{-06}
6:	720	719.6 to 720.4	5.1×10^{-04}
7:	5040	4914.1 to 5192.6	2.8×10^{-02}
8:	40320	–21032.1 to 102033.9	$1.5 \times 10^{+00}$
9:	362880	–18299033.4 to 16980266.1	$4.9 \times 10^{+01}$

This example is only semi-realistic, though I have seen a user fall into this very trap. More realistic examples include the solutions of linear equations, determinants, eigensystems, polynomials, regression and parameter fitting (linear and non-linear), analysis of variance, ordinary and partial differential equations, finite elements, graph theoretic work (e.g. Dirichlet tesselation a.k.a. Voronoi diagrams) and so on. In fact, it is quite rare to have a non-trivial problem which does not show this effect to some extent.

4.2.2 Reducing Cancellation

Restructuring expressions can help a lot. Where it matters, consider changes like the following:

$(X + D) * *2 - X * *2 \quad \Rightarrow \quad (2 * X + D) * D$

$x^5 - y^5 \quad \Rightarrow \quad (x^4 + x^3 * y + x^2 * y^2 + x * y^3 + y^4) * (x - y)$

$sin(x + d) - sin(x) \quad \Rightarrow \quad sin(x) * (cos(d) - 1.0) + cos(x) * sin(d)$

I haven't used the following software, but it claims to do this automatically, and might be worth trying:

`http://herbie.uwplse.org/`

When approximating functions, a continued fraction is usually the best method, because it has a great many useful mathematical properties and rarely much cancellation; don't rush in and code, because the Lentz algorithm is worth considering. Taylor series are more often used, because they are easier to derive - even with those, you don't have to use derivatives, and can just fit polynomials. It is also worth considering Padé approximants (ratios of polynomials), which you can also get by expanding continued fractions. These often converge much faster than polynomials, especially for functions without a useful Taylor series, but can have serious cancellation problems.

For polynomials, Taylor Series, multinomials generally, Padé approximants etc., cancellation is very often associated with slow convergence. One classic example is $log(1+X) = X - X^2/2 + X^3/3 - ... - (-x)^k/k + ...$ for X close to unity. But that is not always true, unfortunately, because even $exp(X) = 1 + X + X^2/2 + X^3/6 - ... + x^k/k! + ...$ is quite hopeless for $X = \pm 100$. Both have evil cancellation problems, and the classic reference on this is available online:

The Perfidious Polynomial by *J.H. Wilkinson*

https://en.wikipedia.org/wiki/Wilkinson's_polynomial

Asymptotic expansions, such as Stirling's formula are a bit odd, because there is a best number of terms for any argument; both more and fewer terms give a less accurate result. The exact number depends on the function and argument value, so care is needed.

For all such approximations, if there is an issue, you should do the following:

1. Reduce the range if feasible to simplify the problem
2. Reorganise the expansion to reduce cancellation (logarithm is a classic here)
3. Reorganise the expansion to accelerate convergence, which is not always quite the same
4. Consider whether to solve a related function and convert. For example, $Y = log(X)$ can be done by solving $X = exp(Y)$
5. Consider other approaches, such as integrating another function; this sounds strange, but normal scores really are best done that way

Cancellation also occurs in large reductions. The most common ones are summations and inner products, and products can be converted to summations by using logarithms, but there are many others. With products, watch out for $1 + \epsilon \approx 1$ and accuracy losses. Kahan summation is usually more accurate than simple summation, and I use a similar method, which has some advantages over it. The most reliable method is to emulate extra precision, but none of the methods will guarantee good accuracy.

That is too tricky to teach here, because it needs advanced floating-point hackery (plus a deep knowledge of the language standards), and compiler optimisation often breaks the code (precisely because of the language standard issues). But, for example code, see:

https://www-internal.lsc.phy.cam.ac.uk/nmm1/Development/Programs/fancy_accumulate.cpp

https://www-internal.lsc.phy.cam.ac.uk/nmm1/Development/Programs/fancy_inner.cpp

https://www-internal.lsc.phy.cam.ac.uk/nmm1/Development/Programs/double_emulation.cpp

Be warned that those files contain extreme geekery, and it is critical to read the comments in the files for details and warnings. Even Kahan summation is non-trivial in that respect, though many compilers have been hacked to recognise its code pattern and not optimise it out!

Unfortunately, cancellation may be implicit in the algorithm, and there is no specific expression in which it occurs, so you cannot fix it by the above methods. This is common with ones that use numerical derivatives. In such cases, you hope that you have algebraic form for the derivative, which allows you to provide them, reorganising the expression to avoid cancellation. Many computer algebra packages have a derivative function which can help with this.

But sometimes the cancellation is deep within the logic (i.e. it is an *emergent property*), the only real solution is a more stable algorithm, or even to change the problem being considered. This is not teachable, in general, because it is too problem-specific, but don't assume that one doesn't exist.

The above is **not** an exhaustive list of the sources of cancellation problems, just some of the most common causes. Always keep a watch out for such low-level inaccuracies, and

you can usually reduce them considerably if needed. If the simple approaches fail, think laterally, and ask how else could you do it? A good, clear book on this topic, with a rather over-grandiose title, is:

Real Computing Made Real:Preventing Errors in Scientific and Engineering Calculations

by *Foreman S. Acton*

Remember that almost any method will give good results in simple, small cases — you can even solve well-conditioned 3×3 sets of equations using Cramer's rule — but the poor methods will fail or give wildly inaccurate answers in complex, larger cases, such as the ones you really want the answer for and cannot check by hand! So do not rely on small test cases.

3.3 Higher Level Issues

This is the domain of classical numerical analysis; remember that it is a branch of applied mathematics, not computing, and fiddling with low-level (programming) solutions will rarely help. Furthermore, numerical analysis applies to **all** arithmetics that model mathematical real numbers, and to all algebras derived from them, such as complex numbers, quaternions, matrices, and so on. Do not be taken in by the widespread claims on the Web that there is such a thing as exact real arithmetic.

I will mention a lot of techniques but not describe them, so look them up. It is always worth starting with Wikipedia, but it is not an authoritative source. If you actually need to use them, rather than remind yourself of what they are, find a proper reference to the technique.

To remind you, transformations are immensely useful, and even simple variable transformations can help immensely. There are many reasons, but I will mention only a couple. They are often taught to improve convergence, and faster convergence often reduces cancellation, but you can often turn an asymptotic expansion into a normal one. And they are also useful to avoid singularities and discontinuities; for example, 3-D rotations using roll, pitch and yaw is numerically evil but, convert to direction cosines, and it becomes simple.

Root finding and minimisation are very common requirements, and can be applied to polynomials or other functions, but extracting eigenvalues solving many non-linear systems and parameter estimation are all closely related. Newton-Raphson is the classic method, and is simple and moderately good. Fancier methods can be much faster if the function is well-behaved, which means essentially quadratic in the regions around well-separated roots. Close roots makes all methods **much** slower, and can make them fail to converge. Very similar remarks apply to complex numbers and minimisation, but there isn't a semi-canonical method, though there are equivalent to Newton's method (or derivatives of it, if you prefer).

Sometimes functions are not well-behaved, and then it is generally better to use the most robust method available, because it is usually faster and more accurate. Binary chop needs only monotonicity, and Fibonacci minimisation is similar for a minimum. Steepest descents, perhaps fiddled a bit, is nearly as robust for multi-dimensional problems. These

still have problems with very close roots, and truly evil multidimensional problems can run like drains; one example is an exponential helix trough with the minimum at the centre. But nothing else will do any better in such cases.

4.3.1 Linear Systems

These include most uses of matrices, and a great deal more. The area is very well-understood and researched, and most algorithms have known, reliable error bounds. Numerical analysis books go into huge detail on this, but we shall not be doing so! We shall start with some general rules, which are often applicable much more widely. The following assumes a knowledge of basic matrix theory, but does not use anything very advanced.

It is worth saying that C and C++ use the Algol order for matrices, which is the other way round to the order used by Fortran. When converting code between the two, it is often important to reverse the loop order (or, alternatively, the subscript order). This is not an accuracy requirement, but a performance one.

Errors in the results are usually relative to the largest result, though sometimes to the largest input element, which means that very small results can have huge relative errors. These matrix errors are usually $N \times eps \times condition\,number$, where N is the dimension of the matrix, eps is the accuracy of the data (necessarily larger than the machine precision), and the condition number is how 'evil' the matrix is. You multiply that by the largest result or element to get the absolute errors, of course.

The appropriate condition number varies with the analysis, and 'nice' problems are near one, but 'nasty' ones are much higher. An infinite condition number means that the problem is ill-posed, and that there are either no answers or there is more than one possible answer (which is usually any value in a subspace, not one of a discrete set). If you want to calculate a condition number, look up how for your particular analysis, and there are usually routines in most libraries to do it.

Note that that the error bounds are not due to rounding error in the arithmetic, and fixed-point and floating-point arithmetic have very similar ones. The bounds are are inherent in the mathematics, and derive from the less accurate of the errors in the input values and the precision of their storage representation. For this reason, it applies even to 'accurate' measurements from data loggers and physical constants, even those whose value is known perfectly.

Also, it means that there is usually no point in using extra arithmetic precision, but it is critical to use an appropriate algorithm. Sometimes using more accurate accumulation, or other such tweaks, helps for particularly important and sensitive sections of code. And, as always, start by using a good library or reference book, such as the NAG library, LAPACK etc., or the authoritative references given above. As an aside, FFTs have an error of $N \times eps$ at worst, and usually much less, relative to the largest input element; they are about as well-posed as it is possible for a problem to be.

4.3.2 Matrices

Real symmetric and Hermitian matrices are fairly simple, mathematically, and have faster and more robust algorithms, as well as fewer difficult cases. Pivoting and scaling are less likely to be needed, and they never are for for positive definite or semi-definite matrices. These techniques are not mentioned further – you can see them described in any good reference, as well as when they are necessary.

Sparse matrices are a lot more difficult, both in performance and in the robustness of algorithms, but a great deal of work has been done on them. The following book is by some well-respected authors, and is written to be usable by computational scientists:

Direct Methods for Sparse Matrices by *I.S. Duff, A.M. Erisman and J.K. Reid*

The performance of matrix methods obviously depends critically on the algorithm, and it is important to note that the fastest is often not the most accurate or robust, especially for sparse matrices and borderline problems (e.g. problematic matrices). Also, large matrix codes can be very cache-unfriendly, and the best solution is to use blocked algorithms, where they exist. It is not always possible, but is often tens of times faster, and sometimes more. Unfortunately, blocked algorithms lead to messy code, so this is a reason to use a good library.

Note that this applies even for simple algorithms like transposition, where the optimal code will depend on your system. Like sorting, transposition on modern systems is **not** a computational problem, but a data management one. The following is an example of a method that is much better than the obvious code; the large blocks are tackled in their numeric order, and the elements are transposed in the order given in the first block.

ONE Blocked Transposition

1	5	11	15
8	2	6	12
13	9	3	7
16	14	10	4

Figure 3.1

4.3.3 Classic Matrix Algorithms

Solving equations and matrix inversion is normally done by $L.L^T$ (Cholesky) and $L.U$ decompositions, where L is lower triangular, L^T is its transpose (conjugate transpose for complex numbers) and U is upper triangular. It is usually not a problem, except for

matrices that are almost singular; this causes inaccuracy and overflow, if not detected. Generally, avoid inversion, because it adds errors, but it is needed for some multivariate statistical calculations.

Both decompositions are sometimes needed for singular matrices, which can be done in many ways, such as using a $L.D.L^T$ decomposition, where D is a diagonal matrix. A well-posed set of linear equations has the near-zeroes cancelling in its coefficients and its right hand side; in this case, there are solutions, which are discussed later. The easiest way is to use SVD (described later) and to limit inverse values to avoid overflow, butit is important to check that your near-zeroes really do cancel!

Eigensystems are nearly as simple, and the standard algorithms are the QR and QL ones. Consider a $N \times N$ matrix; there are always exactly N eigenvalues, but some of them may be equal. It's really just an esoteric form of root finding, as described above. However, equal eigenvalues mean that some eigenvectors are ill-defined; they can be any vector within their subspace, though good algorithms will return an orthonormal basis. All eigenvectors exist for all real symmetric and Hermitian matrices, but may not for really nasty unsymmetric ones. You are unlikely to encounter these, except as the result of a miscalculation, but an example is:

1 1
0 1

There are a huge number of other decompositions, most of which are used heavily in some problem domains and not at all in others. They are particularly useful in problematic cases; $L.D.L^T$ instead of $L.L^T$ was mentioned as useful for the positive semi-definite ones so common in statistics. The most general is singular value decomposition (SVD), which doesn't have any of the problems mentioned above, but is often inconvenient to use. But you can sometimes take short-cuts, and may even need to, especially for positive semi-definite matrices, such as adding $delta \times N \times maxval$ to the diagonal, where $maxval$ is the maximum diagonal element and $delta$ is a suitable small constant.

The characteristic polynomial of a matrix is the polynomial that has the eigenvalues as roots, and is defined as the determinant of $x.I - A$, where A is the matrix, x is the polynomial variable and I is the unit diagonal matrix. This is generally best avoided, because it has a lot of unobvious numerical issues, and there can be foul cancellation problems both in calculating the polynomial coefficients and when evaluating its value. Polynomials look simple, numerically, but aren't, and they should always be viewed with suspicion. The Perfidious Polynomial was mentioned above, and it is well worth looking at.

Determinants are another common calculation and are a useful example, because they can be done in many ways; for example:

Using Cholesky or $L.U$ decomposition and taking the product of the diagonal elements

Evaluating the eigenvalues using QL, taking the product of the eigenvalues

From the characteristic polynomial in two ways

And more ...

Use one of the first two methods, or call a library routine to do it. An example is worthwhile. With an 8×8 Hilbert matrix, rotated to randomise it, which is a nasty but realistic matrix, and one where the determinant can be calculated exactly:

Method	Error
Using Cholesky diagonals	1.5×10^{-7}
Eigenvalue product	1.9×10^{-7}
Polynomial constant	2.4×10^{2}
Polynomial root product	6.3×10^{2}

The eigenvalues vary from 1.11×10^{-10} to 1.70, and the absolute errors in the roots of the polynomial vary from 2.7×10^{-8} to 4.4×10^{-16}; yes, the smallest ones have the largest absolute errors, and their relative errors are ridiculous! This is largely due to rounding error and cancellation and, in this case, it might be 'fixable' using extra precision, but that is not so in general.

An important general rule is to use the right algorithm for the task, which is often not the fastest, when such problems arise. Always keep a watch for problematic matrices and similar difficulties.

4.3.4 Non-Linear Systems

Partial and ordinary differential equations are the most common examples, but there are others. This is much trickier than for linear systems, as the problems are as dependent on the actual function as on the algorithm, and they often vary considerably with where in its domain the function is being evaluated. A classic example is fluid flow, with Reynolds number used as an indication of the domain; low-speed (laminar) flow is easy, high-speed (turbulent) flow is much harder, and transonic flow brings in extra problems. Similar problems may need different approaches, so always watch out for unexpected results.

If you need help, you need to find a specialist expert, though approaches used for apparently related problems may work, and are always worth at least looking at. If not, you need someone who knows about your particular problem, or a reliable reference that addresses it. In the last resort, you may have to sit down and analyse it yourself. Some useful rules for these problems include:

Never just bull ahead and ignore problems that arise, because that stores up trouble for later

Always watch out for weird results, because they may indicate an error in your code or the failure of an algorithm

Lastly, remember that experts aren't omniscient, and can make mistakes

Errors can often build-up exponentially or worse. In such cases, increasing the precision of your calculations definitely (e.g. doubling it) definitely will **not** help. You have no option, and must improve the algorithm you use to solve the problem, or tackle it another way. A trivial but not very realistic example was given above: the K'th differences of x^K, for $x = 0.5, 0.51, ..., 1.99, 2.0$ in double precision gives 1 significant figure at $K = 7$, and complete nonsense thereafter. The old term was numerically unstable systems, but the worst examples are now called chaotic, and there may be **no** useful algorithm; in such

a case, you must take another approach. Often you can calculate some properties, but only representative details; classic examples include:

Long-term orbital mechanics – the orbit is easy, but to predict exactly where the planets will be is infeasible

Turbulent fluid flow – this is not easy, though it can be done, but to predict individual vortices exactly is infeasible

Weather forecasting – it's now pretty reliable, but not whether it will rain at noon at Great St Mary's nor even whether next summer will be a wet one

3.4 Other Issues

These are generic issues, but not every programmer will encounter them, though the first is a very common cause of error, and parameter estimation is important in most branches of science. The others are included mainly because they are not widely known, and much of the Web (and many references) is quite simply wrong; they are more likely to be relevant to people who work in biomedical areas than in traditional physics.

4.4.1 Parameter Estimation

It is very easy to fit data as closely as you like but, unfortunately, estimating parameters reliably is extremely hard; this problem is well-known to statisticians, but often not by others. The difficulty is often factorial in number of parameters, because many different functions will also fit the same data; an old saying was "*With seven parameters, you can fit an elephant.*" Another consequences is that even a good fit does **not** mean the estimates are reliable; parameters can be too closely related to be estimated well.

An extreme but realistic example is the sum of negative exponentials, where it is virtually impossible to estimate the parameters above two exponentials – seriously!

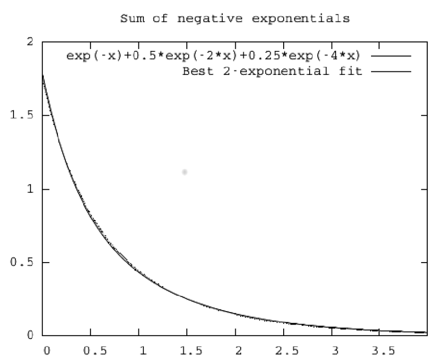

Figure 4.1

The only solution is to estimate the errors in estimates, and this is a multivariate problem, though far too often each variable is treated independently, which gives incorrect and vastly too low estimates of the errors. Each estimate may be precise, if others are taken as fixed, but the estimates are hopeless when taken as a whole, as in that example. The correct approach is to use the inverse of the second derivative matrix; if that is almost singular, it means that there are too many parameters to estimate, as in the horrible

example above. The errors are then diagonal elements, scaled appropriately according to the confidence you need.

4.4.2 Factorials and Related Functions

These are extremely common in statistics and combinatorics, and include the gamma and beta functions, as well as many others; they are widespread elsewhere, and are often counter-intuitive. Something that should be more widely appreciated is that $N!$ in 32-bit integers overflows at $N = 14$ and, even in 64-bit reals, overflows at $N = 171$. Using logarithms avoids overflow, but is slow for large N, and the solution is to use Stirling's formula in its logarithmic form:

$$log(N!) \;=\; (Nlog(N) - N + 0.5log(2\pi N)) + \frac{1}{12N} - \frac{1}{360N^3} + ...$$

The error in expansion that far is $-1/(1260N^5)$, and it gives better than double precision accuracy for $N \geq 170$. There can be accuracy problems, though, especially in combinatorics for very large values. Consider $Binomial(10^9, 0.4, 4 \times 10^8)$. The usual formula is:

$$Binomial(N, P, K) \;=\; N! \times P^K(1 - P)^{(N-K)}/(K!(N - K)!)$$

The relative error in that is 4×10^{-4}. You can improve that by using Stirling's formula algebraically to fix the problem.

$$Binomial(N, P, K) \;=\; (NP/K)^K \times (N(1 - P)/(N - K))^{(N-K)} \times$$
$$\sqrt{N/(K(N - K)2\pi} \times (1 + (1/N - 1/K - 1/(N - K))/12 - ...)$$

Note that it is now using the non-logarithmic form of Stirling's formula. Algebra packages, such as Mathematica, Maple or Axiom, can help, but they are better for checking your work than doing it.

4.4.3 Monte-Carlo Simulations

Any program that uses randomised data is likely to encounter these issues, even if it is not a Monte-Carlo simulation as such. Far too many people assume this is simple; statisticians know better, but it's a specialised knowledge, and most Web references and non-statistical books are unreliable. Unfortunately and much worse, a great many are actually erroneous. Naturally, there are also good ones but your problem is telling which is which.

The error always $O(1/\sqrt{N}$, and there no way round it except in rare cases, but you can reduce the constant considerably. Look for books called "Monte Carlo Methods" if this is an issue.

However, not all distributions have a mean, and the Law of Large Numbers does **not** always hold, despite the common myth that it does; this is not just a theoretical problem, either, as such distributions are not rare. In fact, you need a second moment if you want a decent rate of convergence to the mean. Note that this is mathematically equivalent to calculating an infinite integral. One extreme but realistic example is the ratio of two independent Gaussian variances, giving the error as plus or minus two standard errors of the sample:

Sample size	Estimate	Error
100	1.31	6.39
1000	2,460	1,919
10000	1.34×10^{10}	6.67×10^{10}

If you encounter this, you can sometimes use a transformation, to convert the distribution to one that does have a mean, and you are also recommended to look up truncated sampling, which can also help. But there is a sledgehammer – you can always use the median and its estimated range (based on quantiles, not the sample variance). Two standard errors is roughly element $N/2 \pm \sqrt{N}$ in the sorted sample.

Sample size	Estimate	Range
100	0.87	0.59–1.37
1000	0.998	0.866–1.172
10000	0.997	0.993–1.001

Both transforming your data and using this approach estimates slightly different forms of the average from using the plain mean, of course, so you need to allow for that.

4.4.4 Pseudo-Random Numbers

You should choose a basic generator with at least the following properties:

It returns a double precision floating-point number uniformly distributed between 0 and 1, referred to as U(0,1), with at least 50 independent bits in the number. Converting a 32-bit integer to double is asking for trouble.

It needs a long period (at least 10^{18}), and ideally, at least the square of the **total** count of numbers you will use in your simulation

It needs good pseudo-independence between all numbers, and not just adjacent ones.

It needs to have only a few rare or subtle failure modes.

The last two are not easy to judge, even if you are an expert, but failures to deliver them has caused a lot of trouble in the past. There are several test suites on the net. Diehard is dated and limited, and TestU01 is better, but it still lets quite a few flawed generators through.

`http://simul.iro.umontreal.ca/testu01/tu01.html`

Adjacency properties start failing surprisingly early, and numbers become either too uniform or not uniform enough. This will always happen by the 2/3 power of the period or 10^{15} numbers (in double precision), whichever is the lower, and many common ones fail by 10^7. This is partly due to precision problems, but not entirely. Some popular generators fail by 2×10^4, and using them is a common cause of erroneous results. I have much better tests for adjacency problems than TestU01. I can't claim credit for, as they have been known to statisticians since the year after I was born! But nobody else seems to use them.

A summary of some common generators may be useful.

- The following are ghastly or worse, and should **never** be used: some Numerical Recipes ones, gfortran `rand`, ISO C `rand`, C++ `minstd_rand`, C++ `minstd_rand0`,

g++ `default_random_engine`, and any 32-bit generators or ones that return `float`.

- The following are usable but flawed (watch out!): anything modulo 2^{64} (the best I know of is $x = x \times 13^{13} + 1\ |2^{64}|$), C++ `ranlux48_base` and C++ `knuth_b`.
- The following pass almost all the above tests, but have a few known weaknesses: almost all Mersenne Twisters, and my own `dprand`.
- The following pass all the above tests: gfortran `RANDOM_NUMBER`, C++ `ranlux24_base`, C++ `ranlux24`, C++ `ranlux48`, the better cryptographic ones, and my tweaked version of dprand.

I do not advise using the cryptographic ones, as they are not designed for simulation, and the requirements are subtly different. They may well have failure mores that can be ignored by cryptographers, but are serious for simulations. I mention my own ones mainly because people are welcome to them; they are extremely portable and much simpler than the other good ones. Nor do I advise using 'true random' ones, such as those based on `/dev/random`, for similar reasons.

C++ `ranlux24` is slow and `ranlux48` much slower, though. There are much faster, and equally good, generators.

But **all** generators have weaknesses, as Knuth pointed out half a century ago. Some are known to cause trouble, and the generators should be avoided, others are known but not known to cause trouble, and others are not known.

It is well worthwhile writing your program so that you can use multiple generators. Different initialisations of the same generator give an indication of variability, but only due to the actual number sequence. You should always re-run *critical* results using another generator, which **must** be based on different principles, to check your results are not an artifact of some subtle problem in the generator. And, if you are really cautious, try a third. Spurious results due to interactions between the generator's properties and a program's analysis are common, even in generators that have passed all known tests.

Generating parallel sequences is a seriously difficult problem, and most Web pages and books are wrong about it. It is too nasty to describe here, but it is possible to give some guidelines.

- Firstly, two sequences being disjoint does **not** imply any kind of independence, despite what many people say. It may be the best you can do, but is no more than that.

You can generate genuinely pseudo-independence parallel sequences, but it is not easy to do in a scalable fashion, and even that provides only in a limited form of pseudo-independence. That is not what most people do.

One approach is to use a common instance of a generator for all threads; this is never done for parallel processes (i.e. MPI). All threads then call the same instance of same generator. **Don't** call the generator unsynchronised, because it introduces serious race conditions. At best, it will be slow, and it may fail horribly; worse, it may seem to work in testing, but fail in actual use. A simple way round this is to create a large buffer of them for each thread, as for I/O, synchronised, and extract them from that, unsynchronised and efficiently; when the buffer runs out, you can refill it, synchronised.

Another approach is to use separate instances for each thread or process. You need a very long period, and try to use different sections for each instance; this is reliable only if it is a very high-quality generator, and can fail badly if not. Ideally, each thread's or process's sequences should disjoint from all of the others, but this may be possible to arrange only probabilistically. The details of how to do it depend on the details of generator but, generally, avoid using similar seeds (such as the thread or process index) to start the instances, unless the generator randomises them before use.

You usually will want numbers from other (non-uniform) distributions, of which the normal (Gaussian) is the most common, but but there are too many others to describe. Almost all work by converting one or more U(0,1) generators. These are exactly like special functions, but there are more techniques available, and with correspondingly more failure modes. Unfortunately, there doesn't seem to be a good test suite, even for the common ones; writing one needs statistical skills and is distribution-dependent.

Be cautious with these, as they vary a lot in quality, and sensitive simulations go wrong very easily; watch out for overall poor accuracy, inaccuracies in the tails, breaking invariants in subtle ways, and rare failure for some input values. But they can also introduce poor independence between numbers, though that is rare. Using more than one implementation can help to detect such issues. All of this is exactly the same as for special function implementations.