Numerical Analysis Lecture Notes

Peter J. Olver

1. Computer Arithmetic

The purpose of computing is insight, not numbers.
— R.W. Hamming, [**24**]

The main goal of *numerical analysis* is to develop efficient algorithms for computing precise numerical values of mathematical quantities, including functions, integrals, solutions of algebraic equations, solutions of differential equations (both ordinary and partial), solutions of minimization problems, and so on. The objects of interest typically (but not exclusively) arise in applications, which seek not only their qualitative properties, but also quantitative numerical data. The goal of this course of lectures is to introduce some of the most important and basic numerical algorithms that are used in practical computations. Beyond merely learning the basic techniques, it is crucial that an informed practitioner develop a thorough understanding of how the algorithms are constructed, why they work, and what their limitations are.

In any applied numerical computation, there are four key sources of error:

(i) Inexactness of the mathematical model for the underlying physical phenomenon.

(ii) Errors in measurements of parameters entering the model.

(iii) Round-off errors in computer arithmetic.

(iv) Approximations used to solve the full mathematical system.

Of these, (i) is the domain of mathematical modeling, and will not concern us here. Neither will (ii), which is the domain of the experimentalists. (iii) arises due to the finite numerical precision imposed by the computer. (iv) is the true domain of *numerical analysis*, and refers to the fact that most systems of equations are too complicated to solve explicitly, or, even in cases when an analytic solution formula is known, directly obtaining the precise numerical values may be difficult.

There are two principal ways of quantifying computational errors.

Definition 1.1. Let x be a real number and let $x^\star$ be an approximation. The *absolute error* in the approximation $x^\star \approx x$ is defined as $|\, x^\star - x \,|$. The *relative error* is defined as the ratio of the absolute error to the size of x, i.e., $\dfrac{|\, x^\star - x \,|}{|\, x \,|}$, which assumes $x \neq 0$; otherwise relative error is not defined.

For example, 1000001 is an approximation to 1000000 with an absolute error of 1 and a relative error of 10^{-6}, while 2 is an approximation to 1 with an absolute error of 1 and a relative error of 1. Typically, relative error is more intuitive and the preferred determiner of the size of the error. The present convention is that errors are always ≥ 0, and are $= 0$ if and only if the approximation is exact. We will say that an approximation $x^\star$ has k *significant decimal digits* if its relative error is $< 5 \times 10^{-k-1}$. This means that the first k digits of $x^\star$ following its first nonzero digit are the same as those of x.

Ultimately, numerical algorithms must be performed on a computer. In the old days, "computer" referred to a person or an analog machine, but nowadays inevitably means a digital, electronic machine. A most unfortunate fact of life is that all digital computers, no matter how "super", can only store finitely many quantities. Thus, there is no way that a computer can represent the (discrete) infinity of all integers or all rational numbers, let alone the (continuous) infinity of all real or all complex numbers. So the decision of how to approximate more general numbers using only the finitely many that the computer can handle becomes of critical importance.

Each number in a computer is assigned a location or *word*, consisting of a specified number of binary digits or *bits*. A k bit word can store a total of $N = 2^k$ different numbers. For example, the standard single precision computer uses 32 bit arithmetic, for a total of $N = 2^{32} \approx 4.3 \times 10^9$ different numbers, while double precision uses 64 bits, with $N = 2^{64} \approx 1.84 \times 10^{19}$ different numbers. The question is how to distribute the N exactly representable numbers over the real line for maximum efficiency and accuracy in practical computations.

One evident choice is to distribute them evenly, leading to *fixed point arithmetic*. For example, one can use the first bit in a word to represent a sign, and treat the remaining bits as integers, thereby representing the integers from $1 - \frac{1}{2}N = 1 - 2^{k-1}$ to $\frac{1}{2}N = 2^{k-1}$ exactly. Of course, this does a poor job approximating most non-integral numbers. Another option is to space the numbers closely together, say with uniform gap of 2^{-n}, and so distribute our N numbers uniformly over the interval $-2^{-n-1}N < x \leq 2^{-n-1}N$. Real numbers lying between the gaps are represented by either *rounding*, meaning the closest exact representative, or *chopping*, meaning the exact representative immediately below (or above if negative) the number. Numbers lying beyond the range must be represented by the largest (or largest negative) representable number, which thus becomes a symbol for overflow. When processing speed is a significant bottleneck, the use of such fixed point representations is an attractive and faster alternative to the more cumbersome floating point arithmetic most commonly used in practice.

The most common non-uniform distribution of our N numbers is the *floating point* system, which is based on scientific notation. If x is any real number it can be written in the form

$$x = \pm .d_1 d_2 d_3 d_4 \ \ldots \ \times 2^n,$$

where $d_\nu = 0$ or 1, and $n \in \mathbb{Z}$ is an integer. We call $d_1 d_2 d_3 d_4 \ \ldots$ the *mantissa* and n the exponent. If $x \neq 0$, then we can uniquely choose n so that $d_1 = 1$. In our computer, we approximate x by a finite number of mantissa digits

$$x^\star = \pm .d_1 d_2 d_3 d_4 \ \ldots \ d_{k-1} \widehat{d}_k \times 2^n,$$

by either chopping or rounding the final digit. The exponent is also restricted to a finite range of integers $n_\star \leq N \leq N^\star$. Very small numbers, lying in the gap between $0 < |x| < 2^{n_\star}$, are said to cause *underflow*.

- In *single precision* floating point arithmetic, the sign is 1 bit, the exponent is 7 bits, and the mantissa is 24 bits. The resulting nonzero numbers lie in the range

$$2^{-127} \approx 10^{-38} \leq |x| \leq 2^{127} \approx 10^{38},$$

  and allow one to accurately represent numbers with approximately 7 significant decimal digits of real numbers lying in this range.

- In *double precision* floating point arithmetic, the sign is 1 bit, the exponent is 10 bits, and the mantissa is 53 bits, leading to floating point representations for a total of 1.84×10^{19} different numbers which, apart from 0. The resulting nonzero numbers lie in the range

$$2^{-1023} \approx 10^{-308} \leq |x| \leq 2^{1023} \approx 10^{308}.$$

In double precision, one can accurately represent approximately 15 decimal digits. Keep in mind floating point numbers are *not uniformly spaced*! Moreover, when passing from $.111111\ldots \times 2^n$ to $.100000\ldots \times 2^{n+1}$, the inter-number spacing suddenly jumps by a factor of 2. The non-smoothly varying spacing inherent in the floating point system can cause subtle, undesirable numerical artifacts in high precision computations.

Remark: Although they are overwhelmingly the most prevalent, fixed and floating point are not the only number systems that have been proposed. See [**9**] for another intriguing possibility.

In the course of a calculation, intermediate errors interact in a complex fashion, and result in a final total error that is not just the sum of the individual errors. If $x^\star$ is an approximation to x, and $y^\star$ is an approximation to y, then, instead of arithmetic operations $+, -, \times, /$ and so on, the computer uses a "pseudo-operations" to combine them. For instance, to approximate the difference $x - y$ of two real numbers, the computer begins by replacing each by its floating point approximation $x^\star$ and $y^\star$. It the subtracts these, and replaces the exact result $x^\star - y^\star$ by its nearest floating point representative, namely $(x^\star - y^\star)^\star$. As the following example makes clear, this is *not* necessarily the same as the floating point approximation to $x - y$, i.e., in general $(x^\star - y^\star)^\star \neq (x - y)^\star$.

Example 1.2. . Lets see what happens if we subtract the rational numbers

$$x = \frac{301}{2000} \approx .15050000\ldots, \qquad y = \frac{301}{2001} \approx .150424787\ldots.$$

The exact answer is

$$x - y = \frac{301}{4002000} \approx .00007521239\ldots.$$

However, if we use rounding to approximate with 4 significant digits[†], then

$$x = \frac{301}{2000} \approx .1505, \qquad y = \frac{301}{2001} \approx .1504 \qquad \text{and so} \qquad x - y \approx .0001,$$

[†] To aid comprehension, we are using base 10 instead of base 2 arithmetic in this example.
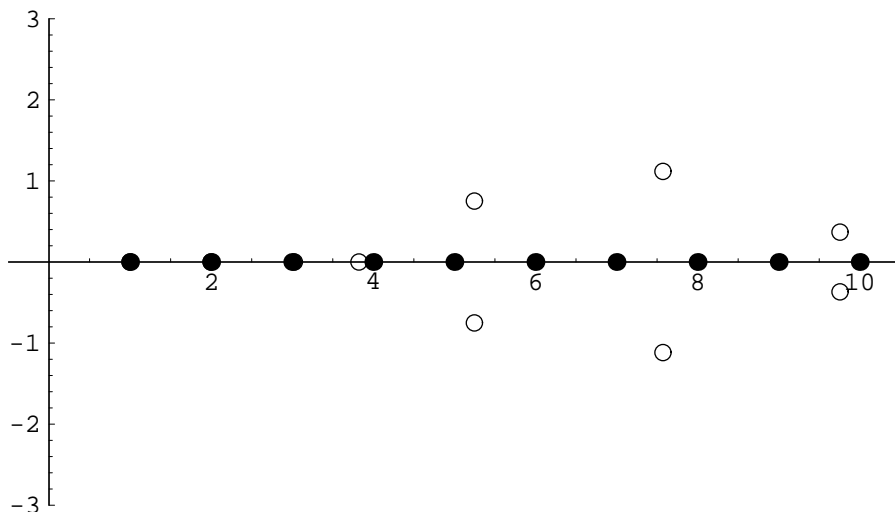
Figure 1.1.   Roots of Polynomials.

which has *no* significant digits in common with the actual answer. On the other hand, if we evaluate the difference using the alternative formula

$$x - y = \frac{301 \times 2001 - 301 \times 2000}{4002000} = \frac{602301 - 602000}{4002000}$$
$$\approx \frac{6.023 \times 10^5 - 6.020 \times 10^5}{4.002 \times 10^6} = \frac{.003 \times 10^5}{4.002 \times 10^6} \approx .00007496,$$

we at least reproduce the first two significant digits. Thus, the result of a floating point computation *depends on the order of the arithmetic operations*! In particular, the associative and distributive laws *are not valid in floating point arithmetic*! In the development of numerical analysis, one tends to not pay attention to this "minor detail', although its effects must always be kept in the back of one's mind when evaluating the results of any serious numerical computation.

Just in case you are getting a little complacent, thinking "gee, a few tiny round off errors can't really make that big a difference", let us close with two cautionary examples.

Example 1.3. Consider the pair of degree 10 polynomials

$$p(x) = (x - 1)(x - 2)(x - 3)(x - 4)(x - 5)(x - 6)(x - 7)(x - 8)(x - 9)(x - 10)$$

and

$$q(x) = p(x) + x^5.$$

They only differ in the value of the coefficient of their middle term, which, by a direct expansion, is $-902055\,x^5$ in $p(x)$, and $-902054\,x^5$ in $q(x)$; all other terms are the same. The relative error between the two coefficients is roughly one-thousandth of one percent.

Our task is to compute the roots of these two polynomials, i.e., the solutions to $p(x) = 0$ and $q(x) = 0$. Those of $p(x)$ are obvious. One might expect the roots of $q(x)$ to
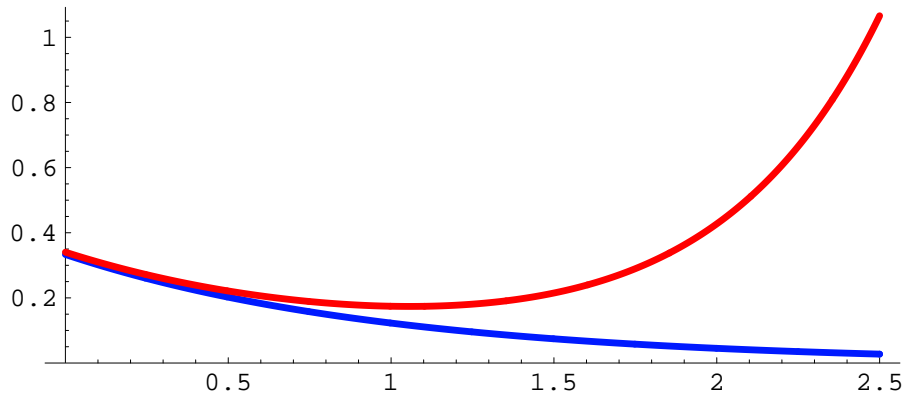
Figure 1.2.   Sensitive Dependence on Initial Data.

be rather close to the integers $1, 2, \ldots, 10$. However, their approximate values are

$$1.0000027558, \qquad 1.99921, \qquad 3.02591, \qquad 3.82275,$$
$$5.24676 \pm 0.751485\,\mathrm{i}\,, \qquad 7.57271 \pm 1.11728\,\mathrm{i}\,, \qquad 9.75659 \pm 0.368389\,\mathrm{i}\,.$$

Surprisingly, only the smallest two roots are relatively unchanged; the third and fourth roots differ by roughly 1% and 27%, while the final six roots have mysteriously metamorphosed into three complex conjugate pairs of roots of $q(x)$. The two sets of roots are plotted in Figure 1.1; those of $p(x)$ are indicated by solid disks, while those of $q(x)$ are indicated by open circles.

We have thus learned that an almost negligible change in a single coefficient of a real polynomial can have dramatic and unanticipated effects on its roots. Needless to say, this indicates that finding accurate numerical values for the roots of high degree polynomials is a very challenging problem.

Example 1.4. Consider the initial value problem

$$\frac{du}{dt} - 2\,u = -e^{-t}, \qquad u(0) = \tfrac{1}{3}.$$

The solution is easily found:

$$u(t) = \tfrac{1}{3}\,e^{-t},$$

and is exponentially decaying as $t \to \infty$.

However, in a floating point computer, we are not able to represent the initial condition $\tfrac{1}{3}$ exactly, and make some small round-off error (depending on the precision of the computer). Let $\varepsilon \neq 0$ represent the error in the initial condition. The solution to the perturbed initial value problem

$$\frac{dv}{dt} - 2\,v = -e^{-t}, \qquad v(0) = \tfrac{1}{3} + \varepsilon,$$

is

$$v(t) = \tfrac{1}{3}\,e^{-t} + \varepsilon\,e^{2t},$$

which is exponentially growing as t increases. As sketched in Figure 1.2, the two solutions remain close only for a very short time interval, the duration of which depends on the

size in the initial error, but then they eventually diverge far away from each other. As a consequence, a tiny error in the initial data can have a dramatic effect on the solution. This phenomenon is referred to as *sensitive dependence on initial conditions*.

The numerical computation of the exponentially decaying exact solution in the face of round-off errors is an extreme challenge. Even the tiniest of error will immediately introduce an exponentially growing mode which will eventually swamp the true solution. Furthermore, in many important applications, the physically relevant solution is the one that decays to zero at large distances, and is usually distinguished from the vast majority of solutions that grow at large distances. So the computational problem is both important and very difficult.

Examples 1.3 and 1.4 are known as *ill-conditioned problems* meaning that tiny changes in the data have dramatic effects on the solutions. Simple numerical methods work as advertised on well-conditioned problems, but all have their limitations and a sufficiently ill-conditioned problem will test the limits of the algorithm and/or computer, and, in many instances, require revamping a standard numerical solution scheme to adapt to the ill-conditioning. Some problems are so ill-conditioned as to defy even the most powerful computers and sophisticated algorithms. For this reason, numerical analysis will forever remain a vital and vibrant area of mathematical research.

So, numerical analysis cannot be viewed in isolation as a black box, and left in the hands of the mathematical experts. Every practitioner will, sooner or later, confront a problem that tests the limitations of standard algorithms and software. Without a proper understanding of the mathematical principles involved in constructing basic numerical algorithms, one is ill-equipped, if not hopelessly handicapped, when dealing with such problems. The purpose of this series of lectures is to give you the proper mathematical grounding in modern numerical analysis.

© 2008   Peter J. Olver