CS 542G: Preliminaries, Floating-Point, Errors

Robert Bridson

September 8, 2008

1 Preliminaries

The course web-page is at: `http://www.cs.ubc.ca/~rbridson/courses/542g`. Details on the instructor, lectures, assignments and more will be available there. There is no assigned text book, as there isn't one available that approaches the material with this breadth and at this level, but for some of the more basic topics you might try M. Heath's "Scientific Computing: An Introductory Survey".

This course aims to provide a graduate-level overview of scientific computing: it counts for the "breadth" requirement in our degrees, and should be excellent preparation for research and further studies involving scientific computing, whether for another subject (such as computer vision, graphics, statistics, or data mining to name a few within the department, or in another department altogether) or in scientific computing itself. While we will see many algorithms for many different problems, some historical and others of critical importance today, the focus will try to remain on the bigger picture: powerful ways of analyzing and solving numerical problems, with an emphasis on common approaches that work in many situations. Alongside a practical introduction to doing scientific computing, I hope to develop your insight and intuition for the field.

2 Floating Point

Unfortunately, the natural starting place for virtual any scientific computing course is in some ways the most tedious and frustrating part of the subject: floating point arithmetic. Some familiarity with floating point is necessary for everything that follows, and the main message—that all our computations will make (somewhat predictable) errors along the way—must be taken to heart, but we'll try to get through it to more fun stuff fairly quickly.

2.1 Representation

Scientific computing revolves around working with real numbers, which must be appropriately represented (and frequently approximated) to be used on the computer. Several possibilities other than floating point for this representation have been explored:

- Exact symbolic representation: symbolic computing packages such as Maple or Mathematica can take a lazy approach to numbers, storing them as mathematical formulas until evaluation to a required precision is necessary. Low performance limits the applicability of this approach, though it has clear benefits in terms of accuracy; how exactly evaluation happens usually relies on the next representation...
- Arbitrary-precision arithmetic: here numbers are truncated at some point, e.g. $1/3$ represented with five digits as 0.33333, but that point can be extended as far as might be needed to guarantee an accurate answer. Dealing with such numbers requires non-fixed-size data structures as well as branching logic that severely limit performance, and for many problems it turns out this flexibility isn't needed.
- Fixed-point arithmetic: at the other end of the spectrum, numbers can be represented as fixed-size (say 32-bit) integers divided by some power of two. In a decimal version, this corresponds to allowing numbers to have some pre-determined fixed number of digits before and after the decimal point. This is wonderful from the point of view of making high performance hardware, but only works for an application if it's known ahead of time that all relevant numbers are going to be in the desired range; this has relegated fixed-point to only a few small niches in areas such as Digital Signal Processing.

None of these hit the same sweet spot as floating-point in terms of performance and flexibility, and are not typically used in scientific computing.

Floating-point is similar in spirit to **scientific notation** of numbers, which you've almost certainly seen in a science class: writing 1534.2 as 1.5342×10^3 for example, or in C/C++ short-hand `1.5342e3`. This represents a number as a **mantissa**, the sequence of **significant digits** which in this example is 1.5342, multiplied by a **base** (10 here) raised to an **exponent** (3 here). This makes it easy to deal with both extremely large and extremely small numbers, by changing the exponent (i.e. moving the decimal point—so it "floats" instead of being fixed in place), and makes it clear what the **relative precision** is: how many significant digits there are. More on this in a moment.

Floating-point uses a fixed size representation, usually with a binary format (so the mantissa's significant digits are instead significant bits, and the base for the exponent is 2). Most commonly either 32-bit ("single precision") or 64-bit ("double precision") is used, although 80-bit is sometimes found (perhaps even just internally in a processor), in the context of graphics smaller versions such as 16-bit ("half precision") or 24-bit can appear, and some calculators use a decimal form. Most floating point hardware these days conforms to the IEEE standard (i.e. is "IEEE floating point"), which makes precise requirements on the layout of the 32-bit and 64-bit versions along with the behaviour of standard operations such as addition and a few additional options.

The fixed number of bits can be broken into three parts:

- one **sign bit**, indicating if the number is positive (0) or negative (1).
- some number of **mantissa** bits: 24 bits for standard 32-bit, 53 for 64-bit. These form a binary representation of a number usually between 1 and 2, with some exceptions.
- some number of **exponent** bits: these encode the power to which the base of 2 is raised. Standard 32-bit floats use 8 bits, 64-bit uses 11.

If you add that up, you'll see there appears to be one too many bits: this is because if the mantissa is restricted to lie in the interval $[1, 2)$, its first bit is always a 1, and thus that first bit needn't be explicitly stored—so the mantissa is actually stored with one bit less (23 for single precision, 52 for 64-bit). Such floating point numbers, with this restriction on the mantissa, are called **normalized**.

Right now we can represent a huge range of numbers: for single precision, the smallest magnitude numbers are on the order of $\pm 2^{-127} \approx \pm 10^{-38}$ and the biggest are on the order of $\pm 2^{127} \approx \pm 10^{38}$, and with a great deal of precision in all ranges (the gap from one number to the next is about $2^{-23} \approx 10^{-7}$ times the size of the number). However, you may have noticed that we are missing one very important number: zero.

The requirement that the mantissa be between 1 and 2 precludes storing zero, which is obviously ridiculous. The first special case added to floating point is thus **denormalized** numbers: ones with the minimum possible exponent but a mantissa strictly less than 1, which includes zero. Zero aside, these represent dangerous territory in terms of precision, since the number of significant bits (i.e. bits after the first 1) is less than expected, but are so small that in usual practice they don't show up.

Zero is an extra special case in fact, since there are two zeros: $+0$ and -0. The sign bit is still there, though the floating-point standard mandated to have good *relative* error: the answer you get should be equivalent to taking the exact value (assuming the operands are exactly represented in floating-point) and rounding it to the number of bits stored in the mantissa. Bounds on the relative error can be made precise with a quantity called **machine epsilon**, which is equal to the first bit not stored in the mantissa—or more precisely, the largest number which when added to 1 still evaluates to 1 in floating-point arithmetic. For single-precision floats machine epsilon is $\epsilon = 2^{-24} \approx 6 \times 10^{-8}$ and for double-precision numbers machine epsilon is $\epsilon = 2^{-53} \approx 1 \times 10^{-16}$. The relative error for any finite floating point operation is bounded by machine epsilon. (And in particular, if the operation should exactly evaluate to zero, the floating-point calculation will be zero.) This is sometimes written symbolically as:

$$x + y = \widehat{(x+y)}(1 + \epsilon_1)$$

where $|\epsilon_1| < \epsilon$. Keep in mind that the Greek letter ϵ is frequently used for many other things just within scientific computing; I'll try to make clear in the context when machine epsilon is being used, not some other epsilon.

So far so good. However, this rounding of every operation has some serious consequences. The most fundamental is that it's not good enough to design algorithms which give the right answer assuming arithmetic is exact: numerical algorithms have to be tolerant of errors to be useful. This gets into an important notion called **stability**, which we will visit several times: if a small error early on in a computation can unstably grow into a large error, the final answer can't be trusted, since we know for sure floating-point arithmetic makes errors all the time.

While we do usually pretend that floating-point arithmetic behaves just like exact arithmetic, only with some "fuzz", the effect of rounding is a bit more subtle. For example, floating-point arithmetic doesn't obey some of the standard laws of regular arithmetic like associativity and distributivity: in floating-point, the following often are **not** true:

$$\begin{aligned} (x+y)+z &= x+(y+z) \\ (x\cdot y)\cdot z &= x\cdot(y\cdot z) \\ x\cdot(y+z) &= x\cdot y + x\cdot z \\ x/y = (1/y)\cdot x & \end{aligned}$$

Sometimes aggressive compiler optimizations (generally not ones that are turned on unless specially requested by the user) will assume these to be true in reordering expressions, which can have unintended consequences. Another tricky source of floating-point bugs is when the processor carries around greater precision internally for operations; if a calculation is done fully in registers the result could be quite different than if an intermediate result needs to be stored and then fetched from memory. It's possible for the value in a variable to change from one part of a function to another despite apparently (at the source code level) being held constant, if at one point it's still in a high-precision register and at another point it has been rounded by being stored to memory. This means that from time to time the level of compiler optimization and/or the presence of debugging "printf" statements can change how a program works, making debugging a challenge. Thankfully these sorts of problems are very rare in most applications, but it's worth remembering that they do happen.

Another important point to make about errors is that the bound on relative error for a floating point operation only applies to the operation itself, assuming the operands (the inputs) were **exact**. If several operations are applied after another, so the input to one is the inexact output of another, the final error can be much worse if care is not taken.

As a decimal example, consider a calculator that only stores four significant digits: $(1+0.00001)-1$ has the exact value of 0.00001 but evaluates to 0, giving a gigantic final relative error of 1 despite both the addition and the subtraction having perfectly fine relative errors when considered on their own. If the calculation had been reordered as $(1 - 1) + 0.00001$, the exact answer with relative error of 0 would have been obtained. This is an example of something called **cancellation error**: when the operands of a subtraction are nearly the same in magnitude but at least one is *inexact*, the answer may have a high relative error because the input error is large relative to the final result. Significant digits in the operands got cancelled out in the subtraction, leaving few accurate digits in the result.

Sometimes cancellation errors can be avoided neatly by rethinking the calculation. A classic example is the solution of the quadratic equation:

$$\begin{aligned} ax^2 + bx + c &= 0 \\ x &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \end{aligned}$$

If $4ac$ is small relative to b^2, the root which is calculated from $-b + \sqrt{b^2 - 4ac}$ (if $b > 0$; for negative b the other one) may be very inaccurate thanks to cancellation. In this case, however, the other root is accurate, and may be used to safely recover the bad one by the formula $x_2 = \frac{c}{ax_1}$.

In some cases cancellation errors may be harder to avoid, or you might not even realize there is a serious problem happening because, quite frankly, keeping track of rigourous bounds on errors in a big complicated program is incredibly tedious.[1] It is **strongly** recommended, therefore, that by default you always use double precision 64-bit floating-point numbers: single precision 32-bit should only be used if you are confident enough that errors are being properly controlled, and performance/memory requirements make double precision impractical.

[1]There is an automatic approach to this, by the way, called **interval arithmetic**, where along with every number you store an upper bound on the error associated with it, which gets conservatively updated with every calculation (or alternatively put, replaces numbers with intervals in which the true number must be found). Unfortunately, apart from the performance hit, this frequently is too pessimistic and after many operations may provide uselessly loose error bounds.

3 Well-Posedness and Stability

Since errors are a fact of life with floating-point arithmetic—and for some problems arise in a variety of other ways from measurement error to approximation errors—they need to be taken into account when discussing problems and solutions.

The first concept we need is that of a **well-posed problem**: before we even think about algorithms for solving a problem, or even touching a computer, we should check that the problem itself makes sense. A problem is well-posed when:

- there is a solution
- the solution is unique
- the solution only changes a little when the data is changed a little

The importance of the first two points should be obvious, but the third one bears some discussion. Most problems involve data of some sort: for example, if you're trying to predict how fast an object will cool, the data will include the shape and materials of the object along with the initial temperature distribution. There is invariably going to be error associated with that data, say from imprecision in measurement. If perturbing the data by amounts smaller than the expected measurement error causes the solution to drastically change, there's no way we can reliably solve the problem and it's pointless to talk about algorithms.

It should be immediately noted that scientists and engineers routinely tackle problems which are not proven to be well-posed though strongly suspected to be. We also deal with problems which on the face of it are definitely not well-posed—this is what so-called "chaos" is all about. This is handled by rephrasing the problem in some averaged sense, but we won't delve any further into that in this course.

A well-posed problem is one that deals well with error. We can also talk about algorithms which deal well with error. A **stable** algorithm is one where introducing small errors in the calculation only has a small effect on the final answer. As we said before, an **unstable** algorithm—which might work fine in exact arithmetic but can give wildly wrong final answers when small errors are made along the way—is probably useless.

4 Libraries

To add to all the complications of error we have just discussed, there are many performance issues to be considered when doing scientific computing. The world's most powerful supercomputers churn for weeks on end to solve some problems, and there is a ready supply of problems that are still too big to be solved on any existing computer: performance matters!

Modern architectures have many features to allow high performance in numerical software: vectorized instructions (like the SSE instructions on x86, or AltiVec on PPC), instruction-level parallelism and out-of-order execution, instruction pipelines (often separated between integer and floating-point operations) to overlap processing, sophisticated branch prediction to make best use of pipelines, multicore chips, cache hierarchies to avoid being slowed down by main memory, cache prefetching to start memory transfers early enough that the data is in registers when it's needed, virtual memory systems with efficient translation lookaside buffers to make the address space simple for programs, parallel machines with all sorts of interconnects, etc. The complexity of these features, and their interactions, is staggering. Mastering this complexity (since compilers can't do it all for you automatically) can mean orders of magnitude performance improvement, but at considerable development cost.

The moral of the story is that reliably accurate and high performance numerical software is usually not easy to write. This makes it imperative to exploit high quality libraries when they are available, when appropriate; luckily for many core operations, there are very good libraries out there. We'll discuss a few of the most important as we come to them.