CS 542G: Preliminaries, Floating-Point, Errors

Robert Bridson

September 9, 2009

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". Sign up for the google group if you want an online forum.

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. There are also some interesting gotchas that can occur with floating-point arithmetic, since it doesn't behave exactly like exact arithmetic.

2.1 Representation

Scientific computing revolves around working with real numbers (occasionally complex!), 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 decimal 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, severely limiting performance, and for many problems it turns out this flexibility isn't needed. In some important special cases, the computations can be done with floating-point hardware, interestingly enough, but the performance is still a big issue.
- 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, or programming very low-power devices like older mobile-phones.

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 and mainframes 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 **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.
- 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.

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 and can be tested, but positive and negative zero are deemed to be equal when tested. Negative zero doesn't usually occur, it should be noted, unless it would seem to be the natural limit of a sequence of negative numbers.

There are some other special values in IEEE floating-point. Infinity (or "inf") comes in both positive and negative forms, providing the result for operations like $1/0$. It interacts with the rest of the arithmetic in predictable ways—for example, $1/\mathrm{inf} = 0$. Infinity also arises in an "overflow" situation, trying to calculate a finite value that is too large to be represented by the floating-point system—there is also a rarer "underflow" situation for a nonzero value too small in magnitude to be represented, but this is more gracefully handled by denormalization.

Finally, there is the value of Not-a-Number, or "nan" for short. Nans are the result of ill-defined operations, such as $0/0$ or $\mathrm{inf} - \mathrm{inf}$. Any numeric operation with a nan returns another nan, allowing one to trace back to where the problem occured in a program. More peculiarly, any comparison operation with a nan returns false, including $nan = nan$: if using a language that doesn't provide a test for nan-ness, you can always check if $x = x$ returns false.

The IEEE floating-point standard mandates 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.[1] 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.)

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. We'll discuss this more later.

[1] Rounding by default is to the nearest value, but you can also set other rounding modes, such as up or down, which is extremely useful for efficiently implementing interval arithmetic for example.

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 usually 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 (hopefully not ones that are turned on unless specially requested by the user, but some compilers don't seem to care) will assume these to be true in reordering expressions, which can have unintended consequences. What you think is being evaluated, based on the source code you wrote, might not be what's actually going on—in the worst case, you may at times need to turn off optimization or carefully check the emitted assembly language.

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 despite both the addition and the subtraction being very accurate 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.

One approach to robustly dealing with rounding errors (and generalizable to several more useful situations—ask me for details if curious) is **interval arithmetic**. Here all computation is done with intervals guaranteed to contain the actual value rather than a single approximate value. This is of course slower, though IEEE floating-point hardware can be exploited to make some operations only twice as slow. All the usual arithmetic operations can be defined on intervals, producing output intervals which contain all possible output values based on the possible values in the input intervals. Interval arithmetic is important in cases where it's critical to have guarantees on the result of a calculation, but it's not a panacea: used naively it tends to produce intervals far too large to be useful. Special interval algorithms are required to refine the results to be as sharp as possible. Most scientific computing calculations do not use interval arithmetic as a result: with educated algorithm design, it's not generally required.

3 Well-Posed Problems

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, albeit are 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.

Often the simplest statement of a problem is ill-posed, but a small change can be made to make it well-posed without changing the actual intent. For example, the following linear system is ill-posed:

$$\begin{aligned} x + y &= 1 \\ x + y &= 0 \end{aligned}$$

First of all, there is no solution—but even if we change the right hand side to permit a solution there would be infinitely many of them. However, in this case it might be reasonable to change the strict equalities into a search for a set of x and y which come as close as possible to satisfying both equations, and of all such optimal x and y picking the smallest magnitude pair. This optimization approach to making ill-posed problems sensible is extremely common, and arises in many different situations.