18.330 Lecture Notes: Machine Arithmetic: Fixed-Point and Floating-Point Numbers

Homer Reid

March 1, 2016

Contents

1	Overview	2
2	Fixed Point Representation of Numbers	3
3	Floating-Point Representation of Numbers	8
4	The Big Floating-Point Kahuna: Catastrophic Loss of Numer- ical Precision	11
5	Other Floating-Point Kahunae	16
6	Fixed-Point and Floating-Point Numbers in Modern Computers	
7	Roots can be found more accurately than extrema	22
Α	How much information is required to specify a number?	25

1 Overview

Consider an irrational real number like $\pi = 3.1415926535...$, represented by an infinite non-repeating sequence of decimal digits. Clearly an exact specification of this number requires an infinite amount of information.¹ In contrast, computers must represent numbers using only a finite quantity of information, which clearly means we won't be able to represent numbers like π without some error. In principle there are many different ways in which numbers could be represented on machines, each of which entails different tradeoffs in convenience and precision. In practice, there are two types of representations that have proven most useful: *fixed-point* and *floating-point* numbers. Modern computers use both types of representation. Each method has advantages and drawbacks, and a key skill in numerical analysis is to understand where and how the computer's representation of your calculation can go catastrophically wrong.

The easiest way to think about computer representation of numbers is to imagine that the computer represents numbers as finite collections of decimal digits. Of course, in real life computers store numbers as finite collections of *binary* digits. However, for our purposes this fact will be an unimportant implementation detail; all the concepts and phenomena we need to understand can be pictured most easily by thinking of numbers inside computers as finite strings of decimal digits. At the end of our discussion we will discuss the minor points that need to be amended to reflect the base-2 reality of actual computer numbers.

¹See the Appendix for an expanded discussion of this point.

2 Fixed Point Representation of Numbers

The simplest way to represent numbers in a computer is to allocate, for each number, enough space to hold N decimal digits, of which some lie before the decimal point and some lie after. For example, we might allocate 7 digits to each number, with 3 digits before the decimal point and 4 digits after. (We will also allow the number to have a sign, \pm .) Then each number would look something like this, where each box stores a digit from 0 to 9:



Figure 1: In a 7-digit fixed-point system, each number consists of a string of 7 digits, each of which may run from 0 to 9.

For example, the number 12.34 would be represented in the form



Figure 2: The number 12.34 as represented in a 7-digit fixed-point system.

The representable set

The numbers that may be exactly represented form a finite subset of the real line, which we might call $S^{\text{representable}}$ or maybe just S^{rep} for short. In the fixed-point scheme illustrated by Figure 1, the exactly representable numbers

```
\mathcal{S}^{\rm rep} = \left\{ \begin{array}{c} -999.9999 \\ -999.9998 \\ -999.9997 \\ \vdots \\ -000.0001 \\ +000.0001 \\ +000.0001 \\ +000.0002 \\ \vdots \\ +999.9998 \\ +999.9999 \end{array} \right.
```

Notice something about this list of numbers: They are all separated by the same absolute distance, in this case 0.0001. Another way to say this is that the *density* of the representable set is uniform over the real line (at least between the endpoints, $R_{\min}^{\max} = \pm 999.9999$): Between any two real numbers r_1 and r_2 lie the same number of exactly representable fixed-point numbers. For example, between 1 and 2 there are 10^4 exactly-representable fixed-point numbers, and between 101 and 102 there are *also* 10^4 exactly-representable fixed-point numbers.

Rounding error

Another way to characterize the uniform density of the set of exactly representable fixed-point numbers is to ask this question: Given an arbitrary real number r in the interval $[R^{\max}, R^{\min}]$, how near is the nearest exactly-representable fixed-point number? If we denote this number by fi(r), then the statement that holds for fixed-point arithmetic is:

for all
$$r \in \mathbb{R}$$
, $R^{\min} < r < R^{\max}, \exists \epsilon \text{ with } |\epsilon| \le \text{EPSABS such that}$
 $\texttt{fi}(r) = r + \epsilon.$ (1)

In equation (1), EPSABS is a fundamental quantity associated with a given fixedpoint representation scheme; it is the maximum absolute error incurred in the approximate fixed-point representation of real numbers. For the particular fixedpoint scheme depicted in (1), we have EPSABS = 0.00005.

The fact that the *absolute* rounding error is uniformly bounded is characteristic of fixed-point representation schemes; in floating-point schemes it is the *relative* rounding error that is uniformly bounded, as we will see below.

Error-free calculations

There are many calculations that can be performed in a fixed-point system with *no error*. For example, suppose we want to add the two numbers 12.34 and 742.55. Both of these numbers are exactly representable in our fixed-point

are

18.330 Lecture Notes

system, as is their sum (754.89), so the calculation in fixed-point arithmetic yields the exact result:



Figure 3: Arithmetic operations in which both the inputs and the outputs are exactly representable incur *no error*.

We repeat again that the computer representation of this calculation introduces $no \ error$. In general, arithmetic operations in which both the inputs and outputs are elements of the representable set incur no error; this is true for both fixed-point and floating-point

Non-error-free calculations

On the other hand, here's a calculation that is not error-free.



Figure 4: A calculation that is not error-free. The exact answer here is 24/7=3.42857142857143..., but with finite precision we must round the answer to nearest representable number.

Overflow

The error in (4) is not particularly serious. However, there is one type of calculation that can go seriously wrong in a fixed-point system. Suppose, in the calculation of Figure 3, that the first summand were 412.34 instead of 12.34. The correct sum is

```
412.24 + 742.55 = 1154.89.
```

However, in fixed-point arithmetic, our calculation looks like this:



Figure 5: Overflow in fixed-point arithmetic.

The leftmost digit of the result has fallen off the end of our computer! This is the problem of *overflow:* the number we are trying to represent does not fit in our fixed-point system, and our fixed-point representation of this number is not even close to being correct (154.89 instead of (1154.89). If you are lucky, your computer will detect when overflow occurs and give you some notification, but in some unhappy situations the (completely, totally wrong) result of this calculation may propagate all the way through to the end of your calculation, yielding highly befuddling results.

The problem of overflow is greatly mitigated by the introduction of *floatingpoint* arithmetic, as we will discuss next.

3 Floating-Point Representation of Numbers

The idea of floating-point representations is to allow the decimal point in Figure 1 to move around – that is, to *float* – in order to accommodate changes in the scale of the numbers we are trying to represent.

More specifically, if we have a total of 7 digits available to represent numbers, we might set aside 2 of them (plus a sign bit) to represent the *exponent* of the calculation – that is, the order of magnitude. That leaves behind 5 boxes for the actual significant digits in our number; this portion of a floating-point number is called the *mantissa*. A general element of our floating-point representation scheme will then look like this:



Figure 6: A floating-point scheme with a 5-decimal-digit mantissa and a twodecimal-digit exponent.

For example, some of the numbers we represented above in fixed-point form look like this when expressed in floating-point form:



Vastly expanded dynamic range

The choice to take digits from the mantissa to store the exponent does not come without cost: now we can only store the first 5 significant digits of a number, instead of the first 7 digits.

However, the choice buys us enormously greater dynamic range: in the number scheme above, we can represent numbers ranging from something like $\pm 10^{-103}$ to $\pm 10^{+99}$, a dynamic range of of more than 200 orders of magnitude. In contrast, in the fixed-point scheme of Figure 1, the representable numbers span a piddling 7 orders of magnitude! This is a huge win for the floating-point scheme.

Of course, the dynamic range of floating-point scheme is not *infinite*, and there do exist numbers that are too large to be represented. In the scheme considered above, these would be numbers greater than something like $R^{\max} \approx$

 10^{100} ; in 64-bit IEEE double-precision binary floating-point (the usual floatingpoint scheme you will use in numerical computing) the maximum representable number is something closer to $R^{\max} \approx 10^{308}$. We are not being particularly precise in pinning down these maximum representable numbers, because in practice you should never get anywhere near them: if you are doing a calculation in which numbers on the order of 10^{300} appear, you are doing something wrong.

The representable set

Next notice something curious: The number of empty boxes in Figure 6 is the same as the number of empty boxes in Figure 1. In both cases, we have 7 empty boxes, each of which can be filled by any of the 10 digits from 0 to 9; thus in both cases the total number of representable numbers is something like 10^7 . (This calculation omits the complications arising from the presence of sign bits, which give additional factors of 2 but don't change the thrust of the argument). Thus the sets of exactly representable fixed-point and exactly representable floating-point numbers have roughly the same cardinality. And yet, as we just saw, the floating-point set is distributed over a fantastically wider stretch of the real axis. The only way this can be true is if the two representable sets have very different densities.

In particular, in contrast to fixed-point numbers, the density of the set of exactly representable floating-point numbers is *non-uniform*. There are more exactly representable floating-point numbers in the interval [1, 2] then there are in the interval [101, 102]. (In fact, there are roughly the same number of exactly-representable floating-point numbers in the intervals [1, 2] and [100, 200].)

Some classes of exactly representable numbers

- 1. Integers. All integers in the range $[-I^{\max}, I^{\max}]$ are exactly representable, where I^{\max} depends on the size of the mantissa. For our 5-decimal-digit floating-point scheme, we would have $I^{\max} = 99,999$. For 64-bit (double precision) IEEE floating-point arithmetic we have $I^{\max} \approx 10^{16}$.
- 2. Integers divided by 10 (in decimal floating-point)
- 3. Integers divided by 2 (in binary floating-point)
- 4. Zero is always exactly representable.

Rounding error

For a real number r, let fl(r) be the real number closest to r that is exactly representable in a floating-point scheme. Then the statement analogous to (1) is

for all
$$r \in \mathbb{R}$$
, $|r| < R^{\max}$, $\exists \epsilon$ with $|\epsilon| \leq \text{EPSREL}$ such that
 $\mathtt{fl}(r) = r(1+\epsilon)$
(2)

where EPSREL is a fundamental quantity associated with a given floating-point representation; it is the maximum *relative* error incurred in the approximate floating-point representation of real numbers. EPSREL is typically known as "machine precision" (and often denoted $\epsilon^{\text{machine}}$ or simply EPS). In the decimal floating-point scheme illustrated in Figure 6, we would have EPSREL $\approx 10^{-5}$.

For actual real-world numerical computations using 64-bit (double-precision) IEEE floating-point arithmetic, the number you should keep in mind is EPSREL $\approx 10^{-15}$. Another way to think of this is: *double-precision floating-point can represent real numbers to about 15 digits of precision*. High-level languages like MATLAB and JULIA have built-in commands to inform you of the value of EPSREL on whatever machine you are running on:

julia> eps() 2.220446049250313e-16

4 The Big Floating-Point Kahuna: Catastrophic Loss of Numerical Precision

In the entire subject of machine arithmetic there is one notion which is so important that it may be singled out as the most crucial concept in the whole discussion. If you take away only one idea from our coverage of floating-point arithmetic, it should be this one:

> Never compute a small number as the difference between two nearly equal large numbers.

The phenomenon that arises when you subtract two nearly equal floating-point numbers is called *catastrophic loss of numerical precision*; to emphasize that it is the main pitfall you need to worry about we will refer to it as the *big floating-point kahuna*.

A population dynamics example

As an immediate illustration of what happens when you ignore the admonition above, suppose we attempt to compute the net change in the U.S. population during the month of February 2011 by comparing the nation's total population on February 1,2011 and March 1, 2011. We find the following data:²

Date	US population (thousands)
2011-02-01	$311,\!189$
2011-03-01	311,356

Table 1: Monthly U.S. population data for February and March 2011.

These data have enough precision to allow us to compute the actual change in population (in thousands) to three-digit precision:

$$311,356 - 311,189 = 167.$$
 (3)

But now suppose we try to do this calculation using the floating-point system discussed in the previous section, in which the mantissa has 5-digit precision. The floating representations of the numbers in Table 1 are

$$fl(311,356) = 3.1136 \times 10^5$$

fl(311,189) = 3.1119 × 10⁵

²http://research.stlouisfed.org/fred2/series/POPTHM/downloaddata?cid=104

Subtracting, we find

$$3.1136 \times 10^{5}$$

-3.1119 × 10⁵
=1.7000 × 10² (4)

Comparing (3) and (4), we see that the floating-point version of our answer is 170, to be compared with the exact answer of 167. Thus our floating-point calculation has incurred a relative error of about $2 \cdot 10^{-2}$. But, as noted above, the value of EPSREL for our 5-significant-digit floating-point scheme is approximately 10^{-5} ! Why is the error in our calculation 2000 times larger than machine precision?

What has happened here is that *almost all of our precious digits of precision are wasted* because the numbers we are subtracting are much bigger than their difference. When we use floating-point registers to store the numbers 311,356 and 311,189, almost all of our precision is used to represent the digits 311, which are the ones that give zero information for our calculation because they cancel in the subtraction.

More generally, if we have N digits of precision and the first M digits of x and y agree, then we can only compute their difference to around N - M digits of precision. We have thrown away M digits of precision! When M is large (close to N), we say we have experienced *catastrophic loss of numerical precision*. Much of your work in practice as a numerical analyst will be in developing schemes to avoid catastrophic loss of numerical precision.

In 18.330 we will refer to catastrophic loss of precision as the big floatingpoint kahuna. It is the one potential pitfall of floating-point arithmetic that you must always have in the back of your mind.

The big floating-point kahuna in finite-difference differentiation

In our unit on finite-difference derivatives we noted that the forward-finitedifference approximation to the first derivative of f(x) at a point x is

$$f'_{\rm FD}(h,x) = \frac{f(x+h) - f(x)}{h}$$
(5)

where h is the stepsize. In exact arithmetic, the smaller we make h the more closely this quantity approximates the exact derivative. But in your problem set you found that this is only true down to a certain critical stepsize h^{crit} ; taking h smaller than this critical stepsize actually makes things *worse*, i.e. increases the error between f'_{FD} . Let's now investigate this phenomenon using floating-point arithmetic. We will differentiate the simplest possible function imaginable, f(x) = x, at the point x = 1; that is, we will compute the quantity

$$\frac{f(x+h)-f(x)}{h}$$

for various floating-point stepsizes h.

Stepsize $h = \frac{2}{3}$

First suppose we start with a stepsize of $h = \frac{2}{3}$. This number is not exactly representable; in our 5-decimal-digit floating-point scheme, it is rounded to

$$fl(h) = 0.66667$$
 (6a)

The sequence of floating-point numbers that our computation generates is now

$$f(x+h) = 1.6667$$
(6b)

$$f(x) = 1.0000$$

$$f(x+h) - f(x) = 0.6667$$

and thus

$$\frac{f(x+h) - f(x)}{h} = \frac{0.66670}{0.66667}$$
(6c)

The numerator and denominator here begin to differ in their 4th digits, so their ratio deviates from 1 by around 10^{-4} . Thus we find

$$f'_{\rm FD}\left(h=\frac{2}{3},x\right) = 1 + O(10^{-4})$$
 (6d)

and thus, since $f'_{\text{exact}} = 1$,

for
$$h = \frac{2}{3}$$
 the error in $f'_{\text{FD}}(h, x)$ is about 10^{-4} . (6e)

Stepsize $h = \frac{1}{10} \cdot \frac{2}{30}$

Now let's shrink the stepsize by 10 and try again. Like the old stepsize h = 2/3, the new stepsize $h = \frac{2}{30}$ is not exactly representable. In our 5-decimal-digit floating-point scheme, it is rounded to

$$fl(h) = 0.066667$$
 (7a)

Note that our floating-point scheme allows us to specify this h with just as much precision as we were able to specify the previous value of h [equation (6a)] – namely, 5-digit precision. So we certainly don't suffer any loss of precision at *this* step.

The sequence of floating-point numbers that our computation generates is now

$$\begin{array}{ll} f(x+h) = 1.0667 & (7b) \\ f(x) = 1.0000 \\ f(x+h) - f(x) = 0.0667 \end{array}$$

and thus

$$\frac{f(x+h) - f(x)}{h} = \frac{0.066700}{0.066667}$$
(7c)

Now the numerator and denominator begin to disagree in the *third* decimal place, so the ratio deviates from 1 by around 10^{-3} , i.e. we have

$$f'_{\rm FD}\left(h = \frac{1}{30}, x\right) = 1 + O(10^{-3})$$
 (7d)

and thus, since $f'_{\text{exact}} = 1$,

for
$$h = \frac{2}{30}$$
 the error in $f'_{\rm FD}(h, x)$ is about 10^{-3} . (7e)

Comparing equation (7e) to equation (6e) we see that shrinking h by a factor of 10 has *increased* the error by a factor of ten! What went wrong?

Analysis

The key equations to look at are (6b) and (7b). As we noted above, our floatingpoint scheme represents $\frac{2}{3}$ and $\frac{2}{30}$ with the same precision – namely, 5 digits. Although the second number is 10 times smaller, the floating-point uses the same mantissa for both numbers and just adjusts the exponent appropriately.

The problem arises when we attempt to cram these numbers inside a floatingpoint register that must also store the quantity 1, as in (6b) and (7b). Because the overall scale of the number is set by the 1, we can't simply adjust the exponent to accommodate all the digits of $\frac{2}{30}$. Instead, we lose digits off the right end – more specifically, we lose one more digit off the right end in (7b) then we did in (7b). However, when we go to perform the division in (6c) and (7c), the numerator is the same 5-digit-accurate h value we started with [eqs. (6a) and (7a)]. This means that each digit we lost by cramming our number together with 1 now amounts to an extra lost digit of precision in our final answer.

Avoiding the big floating-point kahuna

Once you know that the big floating-point kahuna is lurking out there waiting to bite, it's easier to devise ways to avoid him. To give just one example, suppose we need to compute values of the quantity

$$f(x,\Delta) = \sqrt{x+\Delta} - \sqrt{x}.$$

When $\Delta \ll x$, the two terms on the RHS are nearly equal, and subtracting them gives rise to catastrophic loss of precision. For example, if x = 900, $\Delta = 4e-3$, the calculation on the RHS becomes

30.00006667 - 30.0000000

and we waste the first 6 decimal digits of our floating-point precision; in the 5-decimal-digit scheme discussed above, this calculation would yield precisely *zero* useful information about the number we are seeking.

However, there is a simple workaround. Consider the identity

$$\left(\sqrt{x+\Delta} - \sqrt{x}\right)\left(\sqrt{x+\Delta} + \sqrt{x}\right) = (x+\Delta) - x = \Delta$$

which we might rewrite in the form

$$\left(\sqrt{x+\Delta} - \sqrt{x}\right) = \frac{\Delta}{\left(\sqrt{x+\Delta} + \sqrt{x}\right)}$$

The RHS of this equation is a safe way to compute a value for the LHS; for example, with the numbers considered above, we have

$$\frac{4e-3}{30.0000667+30.0000000} \approx 6.667e-5.$$

Even if we can't store all the digits of the numbers in the denominator, it doesn't matter; in this way of doing the calculation those digits aren't particularly relevant anyway.

5 Other Floating-Point Kahunae

Random-walk error accumulation

Consider the following code snippet, which adds a number to itself N times:

```
function DirectSum(X, N)
Sum=0.0;
for n=1:N
Sum += X;
end
Sum
end
```

Suppose we divide some number Y into N equal parts and add them all up. How accurately to we recover the original value of Y? The following figure plots the quantity

$$\frac{|\texttt{DirectSum}\left(\frac{Y}{N},N\right)-Y|}{V}$$

for the case $Y = \pi$ and various values of N. Evidently we incur significant errors for large N.



Figure 7: Relative error in the quantity $\operatorname{DirectSum}(\frac{Y}{N}, N)$.

The cure for random-walk error accumulation

Unlike many problems in life and mathematics, the problem posed in previous subsection turns out to have a beautiful and comprehensive solution that, in practice, utterly eradicates the difficulty. All we have to do is replace DirectSum with the following function:³

```
function RecursiveSum(X, N)
if N < BaseCaseThreshold
Sum = DirectSum(X,N)
else
Sum = RecursiveSum(X,N/2) + RecursiveSum(X,N/2);
end
Sum
end</pre>
```

What this function does is the following: If N is less than some threshold value **BaseCaseThreshold** (which may be 100 or 1000 or so), we perform the sum directly. However, for larger values of N we perform the sum *recursively*: We evaluate the sum by adding together two return values of **RecursiveSum**. The following figure shows that this slight modification completely eliminates the error incurred in the direct-summation process:



Figure 8: Relative error in the quantity $\operatorname{RecursiveSum}\left(\frac{Y}{N},N\right)$.

³**Caution**: The function **RecursiveSum** as implemented here actually only works for even values of N. Can you see why? For the full, correctly-implemented version of the function, see the code **RecursiveSum.jl** available from the "Lecture Notes" section of the website.

Analysis

Why does such a simple prescription so thoroughly cure the disease? The basic intuition is that, in the case of DirectSum with large values of N, by the time we are on the 10,000th loop iteration we are adding X to a number that is 10^4 times bigger than X. That means we instantly lose 4 digits of precision of the right end of X, giving rise to a random rounding error. As we go to higher and higher loop iterations, we are adding the small number X to larger and larger numbers, thus losing more and more digits off the right end of our floating-point register.

In contrast, in the RecursiveSum approach we never add X to any number that is more than BaseCaseThreshold times greater than X. This limits the number of digits we can ever lose off the right end of X. Higher-level additions are computing the sum of numbers that are roughly equal to each other, in which case the rounding error is on the order of machine precision (i.e. tiny).

For a more rigorous analysis of the error in direct and pairwise summation, see the Wikipedia page on the topic⁴, which was written by MIT's own Professor Steven Johnson.

⁴http://en.wikipedia.org/wiki/Pairwise_summation

6 Fixed-Point and Floating-Point Numbers in Modern Computers

As noted above, modern computers use both fixed-point and floating-point numbers.

Fixed-point numbers: int or integer

Modern computers implement fixed-point numbers in the form of *integers*, typically denoted **int** or **integer**. Integers correspond to the fixed-point diagram of Figure 1 with zero digits after the decimal place; the quantity EPSABS in equation (1) is 0.5. Rounding is always performed toward zero; for example, 9/2=4, -9/2=-4. You can get the remainder of an integer division by using the % symbol to perform modular arithmetic. For example, 19/7 = 2 with remainder 5:

julia> 19%7 5

Floating-point numbers: float or double

The floating-point standard that has been in use since the 1980s is known as *IEEE 754 floating point* (where "754" is the number of the technical document that introduced it). There are two primary sizes of floating-point numbers, 32-bit (known as "single precision" and denoted float or float32) and 64-bit (known as "double precision" and denoted double or float64).

Single-precision floating-point numbers have a mantissa of approximately 7 digits (EPSREL $\approx 10^{-8}$) while double-precision floating-point numbers have a mantissa of approximately 15 digits (EPSREL $\approx 10^{-16}$.)

You will do most of your numerical calculations in double-precision arithmetic, but single precision is still useful for, among other things, storing numbers in data files, since you typically won't need to store all 15 digits of the the numbers generated by your calculations.

Inf and NaN

The floating-point standard defines special numbers to represent the result of ill-defined calculations.

If you attempt to divide a non-zero number by zero, the result will be a special number called Inf. (There is also -Inf.) This special number satisfies x+Inf=Inf and x*Inf=Inf if x > 0. You will also get Inf in the event of *overflow*, i.e. when the result of a floating-point calculation is larger than the largest representable floating-point number:

julia> exp(1000) Inf

On the other hand, if you attempt to perform an ill-defined calculation like 0.0/0.0 then the result will be a special number called NaN ("not a number.") This special number has the property that all arithmetic operations involving NaN result in NaN. (For example, 4.0+NaN=NaN, -1000.0*NaN.)

What this means is that, if you are running a big calculation in which any one piece evaluates to NaN (for example, a single entry in a matrix), that NaN will propagate all the way through the rest of your calculation and contaminate the final answer. If your calculation takes hours to complete, you will be an unhappy camper upon arriving the following morning to check your data and discovering that a NaN somewhere in the middle of the night has corrupted everything. (I speak from experience.) Be careful!

NaN also satisfies the curious property that it is not equal to itself:

```
julia> x=0.0 / 0.0
NaN
julia> y=0.0 / 0.0
NaN
julia> x==y
false
julia>
```

This fact can actually be used to test whether a given number is NaN.

Distinguishing floating-point integers from integer integers

If, in writing a computer program, you wish to define a integer-valued constant that you want the computer to store as a floating-point number, write 4.0 instead of 4.

Arbitrary-precision arithmetic

In the examples above we discussed the kinds of errors that can arise when you do floating-point arithmetic with a finite-length mantissa. Of course it is possible to chain together multiple floating-point registers to create a longer mantissa and achieve any desired level of floating-point precision. (For example, by combining two 64-bit registers we obtain a 128-bit register, of which we might set aside 104 bits for the mantissa, roughly doubling the number of significant digits we can store.) Software packages that do this are called *arbitrary-precision* arithmetic packages; an example is the GNU MP library⁵.

Be forewarned, however, that arbitrary-precision arithmetic packages are not a panacea for numerical woes. The basic issue is that, whereas single-precision

⁵http://gmplib.org

and double-precision floating-point arithmetic operations are performed in *hard-ware*, arbitrary-precision operations are performed in *software*, incurring massive overhead costs that may well run to $100 \times$ or greater. So you should think of arbitrary-precision packages as somewhat extravagant luxuries, to be resorted to only in rare cases when there is absolutely no other way to do what you need.

7 Roots can be found more accurately than extrema

(Note: This is a section of the lecture notes on numerical root-finding that I am reproducing here because it kinda belongs in both places.)

An important distinction between numerical root-finding (numerically determining zeroes of a function) and derivative-free numerical optimization (numerically determining minima or maxima) is that the former can generally be done much more accurately. Indeed, if a function f(x) has a root at a point x_0 , then in many cases we will be able to approximate x_0 to roughly machine precision—that is, to 15-decimal-digit accuracy on a typical modern computer. In contrast, if f(x) has an extremum at x_0 , then in general we will only be able to pin down the value of x_0 to something like the square root of machine precision—that is, to just 8-digit accuracy! This is a huge loss of precision compared to the root-finding case.

To understand the reason for this, suppose f has a minimum at x_0 , and let the value of this minimum be $f_0 \equiv f(x_0)$. Then, in the vicinity of x_0 , f has a Taylor-series expansion of the form

$$f(x) = f_0 + \frac{1}{2}(x - x_0)^2 f''(x_0) + O\left((x - x_0)^3\right)$$
(8)

where the important point is that the linear term is absent because the derivative of f vanishes at x_0 .

Now suppose we try to evaluate f at floating-point numbers lying very close to, but not exactly equal to, the nearest floating-point representation of x_0 . (Actually, for the purposes of this discussion, let's assume that x_0 is exactly floating-point representable, and moreover that the magnitudes of x_0 , f_0 , and $f''(x_0)$ are all on the order of 1. The discussion could easily be extended to relax these assumptions at the expense of some cluttering of the ideas.) In 64-bit floating-point arithmetic, where we have approximately 15-decimal-digit registers, the floating-point numbers that lie closest to x_0 without being equal to x_0 are something like⁶ $x_{\text{nearest}} \approx x_0 \pm 10^{-15}$. We then find

$$f(x_{\text{nearest}}) = \underbrace{f_0}_{\sim 1.0} + \frac{1}{2} \underbrace{(x - x_0)^2}_{\sim 1.0\text{e}-30} f''(x_0) + O\left(\underbrace{(x - x_0)^3}_{\sim 1.0\text{e}-45}\right).$$

Since x_{nearest} deviates from x_0 by something like 10^{-15} , we find that $f(x_{\text{nearest}})$ deviates from $f(x_0)$ by something like 10^{-30} , i.e. the digits begin to disagree in the 30th decimal place. But our floating-point registers can only store 15 decimal digits, so the difference between $f(x_0)$ and $f(x_{\text{nearest}})$ is completely lost; the two function values are utterly indistinguishable to our computer.

Moreover, as we consider points x lying further and further away from x_0 , we find that f(x) remains floating-point indistinguishable from $f(x_0)$ over a

⁶This is where the assumption that $|x_0| \sim 1$ comes in; the more general statement would be that the nearest floating-point numbers not equal to x_0 would be something like $x_0 \pm 10^{-15} |x_0|$.

wide interval near x_0 . Indeed, the condition that f(x) be floating-point distinct from $f(x_0)$ requires that $(x - x_0)^2$ fit into a floating-point register that is also storing $f_0 \approx 1$. This means that we need⁷

$$(x - x_0)^2 \gtrsim \epsilon_{\text{machine}} \tag{9}$$

or

$$(x - x_0) \gtrsim \sqrt{\epsilon_{\text{machine}}}$$
 (10)

This explains why, in general, we can only pin down minima to within the square root of machine precision, i.e. to roughly 8 decimal digits on a modern computer.

On other hand, suppose the function g(x) has a root at x_0 . In the vicinity of x_0 we have the Taylor expansion

$$g(x) = (x - x_0)g'(x_0) + \frac{1}{2}(x - x_0)^2 g''(x_0) + \cdots$$
(11)

which differs from (8) by the presence of a linear term. Now there is generally no problem distinguishing $g(x_0)$ from $g(x_{\text{nearest}})$ or g at other floating-point numbers lying within a few machine epsilons of x_0 , and hence in general we will be able to pin down the value of x_0 to close to machine precision. (Note that this assumes that g has only a *single* root at x_0 ; if g has a *double* root there, i.e. $g'(x_0) = 0$, then this analysis falls apart. Compare this to the observation we made earlier that the convergence of Newton's method is worse for double roots than for single roots.)

Figures 7 illustrates these points. The upper panel in this figure plots, for the function $f(x) = f_0 + (x - x_0)^2$ [corresponding to equation (8) with $x_0 = f_0 = \frac{1}{2}f''(x_0) = 1$], the deviation of f(x) from its value at $f(x_0)$ versus the deviation of x from x_0 as computed in standard 64-bit floating-point arithmetic. Notice that f(x) remains indistinguishable from $f(x_0)$ until x deviates from x_0 by at least 10^{-8} ; thus a computer minimization algorithm cannot hope to pin down the location of x_0 to better than this accuracy.

In contrast, the lower panel of Figure 7 plots, for the function $g(x) = (x-x_0)$ [corresponding to equation (11) with $x_0 = g'(x_0) = 1$], the deviation of g(x) from $g(x_0)$ versus the deviation of x from x_0 , again as computed in standard 64-bit floating-point arithmetic. In this case our computer is easily able to distinguish points x that deviate from x_0 by as little as $2 \cdot 10^{-16}$. This is why numerical root-finding can, in general, be performed with many orders of magnitude better precision than minimization.

⁷This is where the assumptions that $|f_0| \sim 1$ and $|f''(x_0)| \sim 1$ come in; the more general statement would be that we need $(x - x_0)^2 |f''(x_0)| \gtrsim \epsilon_{\text{machine}} \cdot |f_0|$.



Figure 9: In standard 64-bit floating-point arithmetic, function *extrema* can generally be pinned down only to roughly 8-digit accuracy (upper), while *roots* can typically be identified with close to 15-digit accuracy (lower).

A How much information is required to specify a number?

At the beginning of these notes we noted that specifying the number π requires an infinite quantity of information. This might seem strange to you if you have ever used a computer algebra system⁸like MATHEMATICA, in which one only need type Pi—two characters, a highly finite amount of information—to specify the quantity in question to infinite precision! Of course, what is happening here is that with MATHEMATICA we agree *in advance* to pre-select some finite set of special real numbers like π and *e*, then agree on a labeling or indexing convention for the elements of this finite set, in which case we can clearly describe any element of the set using only finitely much information.

The more interesting question to ask is this: Given an arbitrary real number $x \in \mathbb{R}$, how much information is required to communicate this number to another person (our counterpart) in the *absence* of any predefined finite set of special values? (And by "communicate" we mean *really communicate*, as in, our communication must suffice to enable our counterpart to pinpoint the *exact* point $x \in \mathbb{R}$, not some point in its vicinity such as a rational approximation.) The answer to this question properly belongs in the domain of computational complexity theory and specifically the field of *computability* theory, but here's a ranking of some categories of numbers in order of increasing information content.

- Integers. Integers are the lowest-information-content of all numbers: any integer may be represented by finite string of decimal digits (or binary digits, or digits in whatever base we like) and thus we may communicate any integer x to our counterpart by simply transmitting its finitely many digits.
- Non-integer rational numbers. Non-integer rational numbers may have infinitely many decimal digits in their decimal expansion (such as $\frac{2}{7}$) or in their binary expansion (such as $\frac{1}{10}$), but any rational number can be represented as a ratio of two integers. Thus, to communicate a rational number $x = \frac{p}{q}$ to our counterpart, we can always just communicate the integers p and q, each of which again requires only finite information.

⁸In a numerical math system like MATLAB or JULIA one has similarly the built-in constant **pi**; however, the situation here is quite different from that of MATHEMATICA. In MATLAB/JULIA, the symbol **pi** refers to the best floating-point approximation to π that is available on the hardware you are using; this is a rational number that *approximates* π to (typically) 15 or so digits, but is not equal to π . In contrast, in MATHEMATICA the symbol **Pi** specifies abstractly the exact number π , in the sense of the labeling/indexing scheme described in the text. If you ask MATHEMATICA to print out a certain number of digits of **Pi**, you will get a rational approximation similar to that in JULIA; however, in MATHEMATICA you can ask for any number of these digits, and moreover a calculation such as Exp[I*Pi/4] will yield the exact number $\sqrt{\frac{2}{2}}(1+i)$, represented abstractly by a symbol. In contrast, in JULIA the calculation exp(im*pi/4.0) will yield a rational number number that approximates $\frac{\sqrt{2}}{2}(1+i)$ to roughly 15 digits). Be aware of this distinction between symbolic math software and numerical math software!

- Irrational algebraic numbers. Because irrational numbers like $\sqrt{2}$ cannot be represented as ratios of integers, one might think their specification requires an infinite quantity of information. However, within the irrationals lives a subset of numbers that maybe specified with finite information: the algebraic numbers, defined as the roots of polynomials with rational coefficients. Thus, the number $x = \sqrt{2}$, though irrational, satisfies the equation $x^2 + 0x - 2 = 0$ and is thus an algebraic number; similarly, the roots of the polynomial $7x^3 + \frac{4}{3}x^2 + 9$ are algebraic numbers. If we tried to specify these numbers to our counterpart by communicating their digits (in any base), we would have to send an infinite amount of information; but we just send the coefficients of the polynomials that define them we can specify the exact numbers using only finitely much information.
- Transcendental numbers. Finally, we come to the transcendental numbers. These are real numbers like π or e which are not the roots of any polynomial with rational coefficients⁹ and which thus cannot be communicated to our counterpart using only a finitely amount of information.

$$\sin x = x - \frac{x^3}{6} + \frac{x^5}{120} + \dots = 0$$

and e-1 is a root of

$$-1 + \ln\left[1 + x\right] = -1 + x - \frac{x^2}{2} + \frac{x^3}{3} + \dots = 0.$$

However, this is also cheating, because polynomials by definition must have only finitely many terms.

⁹If we allow non-rational coefficients, then the numbers π and e are, of course, the roots of polynomials (for example, π is a root of $x - \pi = 0$), but that's cheating. Also, π and e are the the roots of non-polynomial equations which may be written in the form of infinite power series: for example, π is a root of