1 Best approximation

We first spend a few moments finishing our treatment of best approximation. Texts which treat this material in more depth include a book by Cheney, as well as books by Rice and by Rivlin. Cheney’s book was once used here as the primary text for a class on approximation theory.

We are given a target function to be approximated over some subdomain, a family of functions from which approximations may be drawn, and a measure of the error at each point. The goal is to minimize the maximum error.

In general, there is no theory which says there even exists a best approximation unless we are optimizing over a space like the polynomials of degree \( n \) or less. Except under special circumstances, spaces of rational functions with specified degree numerator and denominator also contain best approximations. But in general the theory is much less well-developed.

Suppose a rational function with quadratic numerator and denominator approximates the above function \( f \). We have

\[
|f(x) = \frac{\alpha x^2 + \beta x + \gamma}{x^2 + \delta x + \eta}| < \epsilon
\]

This expression has six coefficients: the five parameters for the rational function and \( \epsilon \). But suppose, by fluke, you got more than you paid for: 7 equi-alternations rather than 6. If you then tried to get a better approximation by ratios of cubic functions, you would find that any cubic of the form

\[
\frac{Ax^3 + Bx^2 + Cx + D}{x^3 + Ex^2 + Fx + G} = \frac{(\alpha x^2 + \beta x + \gamma)(1 - \lambda)}{(x^2 + \delta x + \eta)(1 - \lambda)}
\]

would be a best approximant. So the parameters are not determined uniquely, and we have degeneracy.

Degeneracy appears in the form of singular (or nearly singular) linear systems to solve. You might get a system of the form \( Mz = r \) where \( M \) is singular, but \( r \) is in the column space of \( M \), or nearly so. If we do Gaussian elimination on \( M \), we will typically get a small or zero element in the lower right corner of the upper triangular factor \( U \). But as we prepare for the solve involving \( U \) after doing the solve involving \( L \), the vector on the right hand side will also have a very tiny last element. Dividing those two tiny elements is what causes trouble; we essentially have roundoff divided by roundoff, which may be huge. But if we perturb the nearly zero entry in \( U \) by a little bit, then we don’t have division by zero or nearly-zero. We get a solution that looks like \( z + z_0 \), where \( z_0 \in \ker(M) \); and as long as \( z_0 \) is small and \( z \) is not lost, we’re fine.

It is also possible to do a rank-revealing decomposition (perhaps an SVD) and solve a truncated system, but that’s really overkill for what tend to be relatively small and simple problems.

So degeneracy is addressable and, while it is a nuisance, it is not to be feared. Degeneracy has actually arisen in approximating \( \Gamma(x) \) by rationals.
2 Of tetrahedra and troubles

We turn now once more to the matter of language support for floating point computation. This lecture is, in part, preparation for a talk which was given at Stanford on April 17, 2001.

Why is floating point different from practically everything else? Kahan argued that the failure of transitivity of accuracy, and the consequent difficulty in composing correctly functioning floating point modules, results in a set of challenges fundamentally different from those occurring in other areas of computer science. The assertion that such failures of compositability are unique to floating point was not uniformly accepted by the class, and some discussion ensued. A more complete version of that argument took much of the next lecture period, and is reported upon in those notes.

Here is one example of the intransitivity of accuracy, drawn from “Matlab’s Loss is Nobody’s Gain.” Suppose we compute, in correctly rounded floating point arithmetic,

\[
G(Y) := (-\log(Y))^{-1/4} \quad \text{for } 0 < Y < 1
\]

\[
H(X) := \exp(-X^{-4}) \quad \text{for } X > 1
\]

\[
F(X) := G(H(X)) \quad \text{for } X > 1
\]

What relationship will the computed function \(F\) bear to the identity, which is what we would have in exact arithmetic? In fact, the graph of \(F\) will not look much like the graph of the identity. Rather, it looks like a series of steps:

2.1 Determinants and tetrahedra volumes

Suppose we wanted to compute the volume of a tetrahedron from the coordinates of its corner points. If \(x, y, z,\) and \(t\) are the coordinates expressed as vectors in 3-space, then

\[
\text{vol} = \frac{1}{6} \left| \det \begin{pmatrix} 1 & 1 & 1 & 1 \\ x & y & z & t \end{pmatrix} \right|
\]

\[
= \frac{1}{6} \left| \det \begin{pmatrix} y-x & z-x & t-x \\ y & z & t \end{pmatrix} \right|
\]

Usually the latter formula is used to compute the volume. But often the volume of the tetrahedron is unimportant; rather, you want to know whether a point is on one side or the other of the plane formed by the other points. This information is given by the sign of the determinant.

As numerical analysts, we should know how to undertake an accurate determinant computation. If the dimension is small, then we could expand by minors to evaluate the determinant as a polynomial. As we evaluate the polynomial, we can keep a running error bound:

\[
S := X + Y \quad \Delta S := (\Delta X + \Delta Y + \epsilon|S|)(1 + 2\epsilon)
\]

\[
P := X \ast Y \quad \Delta P := (\epsilon|P| + \Delta X \cdot |Y| + \Delta Y \cdot |X| + \Delta X \cdot \Delta Y)(1 + 6\epsilon)
\]

The derivation of a similar error bound for quotients is left as an exercise to the student.

Ultimately, you could compute with as many words as are needed to get a determinant exactly; or, using the error bound, you could decide to compute exactly only when your answer was too uncertain to be useful. But this adds time cost and complexity for what is generally a rare event.

So what can you do to get the best result with the precision you have?
2.1.1 Equilibration

If we are trying to compute \( \det(B) \), we may find it easier to compute

\[
\det(B) = \frac{\det(ABV)}{\det(A)\det(V)}
\]

where \( A \) and \( V \) are diagonal matrices whose elements are powers of the radix, so that the rescaling can be done exactly.

Recall that in terms of edge lengths, we have another determinantal formula for the volume. Without equilibration, that determinant can look very much like

\[
\begin{vmatrix}
0 & 1 & 1 \\
1 & 1 & 0 \\
1 & 1 & 1
\end{vmatrix}
\]

If the determinant is to be evaluated via Gaussian elimination, this type of determinant can lead to unreasonable pivot selection. In particular, try using this formula and scaling all the edge lengths by the same factor. Since the volume is a homogeneous function of the edge lengths, it should change as the cube of the scale factor; but that is not what happens when this formula is evaluated on a computer!

2.1.2 Iterative refinement

Suppose that in computing \( \det(B) \), we factor

\[
P B = LU
\]

and then determine that \( L \) and \( U \) are substantially contaminated. One possible remedy is to iteratively improve \( L \) and \( U \):

\[
P B = (L + \Delta L)(U + \Delta U)
\]

\[
P \Delta B := PB - LU \text{ is calculated carefully with extra precision}
\]

\[
= (L + \Delta L) \Delta U + \Delta L \cdot U
\]

where \( \Delta L \) is strictly lower triangular and \( \Delta U \) is upper triangular with zeros along the top row. Solving for \( \Delta L \) and \( \Delta U \) is left as an exercise for the student; is is just a matter of eliminating in the right order.

2.1.3 Preconditioning

Suppose \( \det(B) \approx 0 \) due to cancellation. Then \( B \) is nearly singular, which means \( B^{-1} \) is huge. Suppose \( r^T \) is the row of \( B^{-1} \) with largest entries. We know that

\[
r^T B = e_k
\]

where \( e_k \) is the \( k \)th row of the identity matrix. But compared to the size of \( r^T \), \( e_k \) is tiny. That is, \( r^T \) is approximately a null vector of \( B \). Suppose \( i^T \) is a scaled version of \( r^T \), rounded to only a few bits. Then we can replace a row of \( B \) by \( i^T \) \( B \) to get

\[
\det \left( \begin{array}{c}
i^T B \\
B
\end{array} \right) = \lambda \det(B)
\]
where $\lambda$ is a factor coming from $i^T$. Since we can think of the change from $B$ to $B$ with the first row replaced as multiplication on the left by the identity with the first row replaced by $i^T$, it is clear what $\lambda$ will be.

By doing this transformation exactly, we have cancelled exactly computed quantities, thereby dodging a certain amount of cancellation that might otherwise reveal damage due to roundoff. If the original data fits into your word with space to spare, this sort of transformation can often be done without extra precision.

2.2 A different formula

All these tricks, though, are really for numerical analysts. They are a bit arcane, and they don’t always work! What casual programmer is going to use them? It’s possible for numerical analysts to justify their existence this way, but that’s not all that satisfying.

What if we write the formula a different way? For nice, fat tetrahedra, there are few problems with any of the obvious formulae. For a tetrahedron which is shaped like an axe-head, nearly flat, the volume may be a very ill-conditioned function of the edge lengths, so we could argue that a poor answer is deserved. But for a javelin-shaped tetrahedron, the problem is well-conditioned: the condition number is three if each point projects perpendicularly onto the opposite face. And for just about any way you would evaluate

$$\sqrt{(\text{cubic in } u^2, v^2, w^2, U^2, V^2, \text{ and } W^2)/6}$$

there is some tetrahedron for which you would get a much worse result than the condition number indicates.

There is an algorithm, though. The details are on the web, but the idea is this. Compute a number of facial differences and sums, taking care with the ordering so that everything is evaluated accurately. Use nine of those numbers (which must include the three smallest) out of the twelve. Then compute certain products.

It seems fair to declare this formula unobvious. It also seems fair to say that it’s not obvious why this formula should be backward stable. Figuring out what this formula that the others lack is a challenge to the student.

3 What to do?

Computing the volume of a tetrahedron seems like a simple problem, but look how much effort it is! Is it reasonable to expect anyone to make such an effort, even if they have the expertise to do so?

Few programmers will get more than an hour or two of exposure to these issues. Even those who take a numerics course may not get a version taught competently. However, even numerically inexpert programmers may well find occasion to use floating point in their codes. How will even an expert, using a module written by a programmer adept in many areas but unware of floating point issues, manage to correctly diagnose the problem when the module fails? Most floating point problems are misdiagnosed; how much more difficult will diagnoses be when the problem occurs in a downloaded module for which the source code may not even be available?

But there is a way to improve the situation. We need new rules of thumb, both for compiler writers implementing floating point support and for programmers using floating point. These rules of thumb should be simple enough that they can be explained in a few pages in an elementary
programming text. With luck, they may supplant older rules of thumb inherited from well before the advent of the digital computer and increasingly inappropriate to modern computation.

The rules of thumb, in essence, would be

Rule 0: The rules of thumb are just rules of thumb. They are not a substitution for numerical expertise, but are rather meant to increase the likely success of the casual floating point user.

Rule 1: Store large volumes of data as floats. In many cases, what you compute represents some thing, perhaps some geometric object. At certain points in the calculation, you will have data representing that object which is as good as you can trust. Store those “fiducial values” no more accurately than you need.

Rule 2: Store constants, intermediates, etc. in the widest precision that does not run too slow.

Codacil: Sometimes we compute objects in motion and, from time to time, take a “snapshot” of the object and perform some additional computations based on that snapshot. When doing this, there should be two copies of the data: one, stored in double, for further time evolution; and a second, stored in float, to be used as the basis for computations based just on the snapshot.

In general, calculations only lose substantial accuracy near singularities. Those singularities may be in the problem, or in the intermediate formulas used to solve the problem. By increasing the precision of intermediate values, we decrease the volume of the regions about such singularities where bad things happen. If the precision of the data remains the same, then increases in the intermediate precision usually mean decreases in the incidence of numerical embarrassment.

(Why say “incidence” instead of “probability”? Because we generally don’t know the distribution of our input data; and even if we did, users generally don’t ask problems at random.)

The default for language designers, then, should be to evaluate everything not explicitly desired as a float in the widest precision available with reasonable speed. That’s the motivation for the 8087 design; and these rules were honored by the Standard Apple Numerics Environment compilers. Additional formats like quad, which may be too slow for heavy use, make things more complicated.

We also would like locutions to allow compilers to selectively apply certain optimizations. For instance, floating-point arithmetic obeys neither the distributive nor the associative laws; but many (not all) codes are insensitive to reorganizations of expressions based on assumptions of distributivity and associativity. Ideally, the programmer could grant the compiler license to make such optimizations in some areas of the code, and not have to fight the compiler to prevent such optimizations in other subroutines.