Running Review Doc for Math 128A

Ethan Ebbighausen

April 7, 2026

Contents

13 Week 1: Calculus Review (1) **23**

This document gives a running review of our Math 128A course, taught in Spring 2026 by Professor Ryan A. Hass using *Numerical Analysis : Mathematics of Scientific Computing , Third Edition* by David Kincaid and Ward Cheney. This review is not comprehensive, and is just to give an overview of the topics.

1 Week 11: (Current)

2 Week 10.5: Midterm 2 Prep

Midterm 2 will cover sections 4.1-4.4 (Matrix Algebra, LU, Doolittle, and Crout Factorizations, Gaussian Elimination and Pivoting, Matrix and Subordinate Norms) and 6.1 - 6.4 (Polynomial Interpolation, Divided Differences, Hermite Interpolation, Cubic Splines). The Professor has already provided a review document, and your homework is also a great resource for problems to understand, but here I will also collect a few extra questions that will be used in discussion.

2.1 Matrix Theory

Factorizations

The foundation of this area is your knowledge of solving linear systems via row reduction from Math 54 or an equiaveltn linear algebra class. You should remember what row operations are, and how we can write row operations as left or right multiplication by an elementary matrix. One easy way to practice this is to compute the inverse by using elementary matrices. For example, consider

$$\begin{bmatrix} 4 & 0 & 2 \\ 2 & 6 & 2 \\ 2 & 3 & 10 \end{bmatrix}$$

It should be easy to see that this matrix is diagonally dominant, so pivoting is straightforward. Use row reduction to convert this matrix into row echelon form (upper triangular). Keep track of the elementary matrices that do this, i.e. $E_3E_2E_1A = U$. Computing the LU factorization them amounts to just having $L = (E_3E_2E_1)^{-1}$ (What property do we need from L? Is this verified in this case and why?). Continue until you reach the identity for extra practice.

What makes Doolittle's factorization useful? Is Crout's factorization more or less difficult to compute (or the same difficulty)? In the above example, write it in Crout's factorization and use that factorization to quickly solve

$$Ax = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$$

What do we know about when LU factorizations Exist? What about when Cholesky factorizations exist?

Practice Problems:

4.1.17) Show that any skew-symmetric matrix has $x \cdot Ax = 0$ for all x. Further, show that any matrix A may be written $A = A_0 + A_1$ where A_0 is symmetric and A_1 is skew symmetric.

4.2.1) Show that the inverse of an upper triangular matrix is also upper triangular, and notice the same is true of lower triangular matrices. Show the product of upper triangular matrices is upper triangular as well. Use this to show that... 4.2.2) Doolittle's factorization, if it exists for an invertible matrix A, is unique. Furthemore, 4.2.12) Show that every matrix of the form $A = \begin{bmatrix} 0 & a \\ 0 & b \end{bmatrix}$ has an LU-factorization, but that it is not unique (even if L is unit).

4.2.36) Determine the LU factorization of the matrix

$$A = \begin{bmatrix} 6 & 10 & 0 \\ 12 & 26 & 4 \\ 0 & 9 & 12 \end{bmatrix}$$

Gaussian Elimination

We need to compute row operations to get the factorizations above, and the process of solving linear systems via row reduction is called Gaussian elimination. When the diagonal of the matrix is quite large compared to the rest of the matrix, this is easy because we don't need to swap rows or columns around. Show with the case

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} x = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

for small epsilon why we might expect issues in our computer systems if we choose a pivot to be a small value. Hence, we need to find a way to pivot, and we try to do so without swapping rows or columns because it is memory expensive. This is why we developed the algorithm on scaled row pivoting. Recall that you must also keep track of the permutation matrix, as well as the elementary matrices involved.

Carry out this algorithm to find the LU factorization of

$$A = \begin{bmatrix} 1 & 2 & -1 \\ 2 & 1 & -2 \\ -3 & 1 & 4 \end{bmatrix}$$

What is complete pivoting compared to this? Also, recall why diagonal dominance actually allows us to ignore pivoting and look at the tridiagonal case.

[These problems mostly end up being applying the algorithm]

Norms

We need a way to measure error in our systems and matrices. Since we may view $M(n \times k) \cong \mathbb{R}^{nk}$ as a vector space, it is natural to measure these by norms (recall the definition/properties of norms). What are some basic examples of such norms? However, we may also view the matrices as linear operators on other vector spaces, which leads to the idea of the subordinate norm. Let $||\cdot||_O$ be some norm on $\mathbb{R}^n$. We define the subordinate norm of a $n \times n$ matrix M to be

$$||M|| = \sup_{||x||_O=1} ||Mx||_O$$

Show that this is indeed a norm by showing it satisfies the properties you recalled earlier. Also show $||Ax||_O \leq ||A||||x||_O$.

For example, show that if the vector norm is the standard Euclidean norm, all orthogonal matrices have subordinate norm 1.

Recall that $\kappa(A) = ||A||||A^{-1}||$ is called the condition number. Why? What does it condition, and what does it tell us? You will probably need the residual and error vectors to explain this. What should we expect when we have a large condition number?

Practice Problems:

Compute $||\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}||$ where the norm is subordinate to the infinity norm.

What is the connection of the 2-norm to the singular values?

4.4.7) Let $n(x) = \sum_{i=1}^{n} |x_i|^3$. Is this a norm?

4.4.8) Show that $||A|| = \sum_{i,j=1}^{n} |a_{i,j}|$ is a norm on matrices but not a subordinate norm.

4.4.11) Show that $||A|| = \max_j \sum_{i=1}^{n} |a_{ij}|$ is subordinate to the 1-norm. Compute the condition number under this norm of $\begin{bmatrix} 1 & 1+\epsilon \\ 1-\epsilon & 1 \end{bmatrix}$

4.4.26) Give an example of a well-conditioned matrix with very small determinant.

4.4.39) Show $\kappa(AB) \leq \kappa(A)\kappa(B)$ and 4.4.48) $\kappa(\lambda A) = \kappa(A)$ for $\lambda \neq 0$.

2.2 Functional Approximation

The main point of this subsection is to regain a function from a finite list of values $(x_i, f(x_i))$, where the x_i are called nodes.

Interpolation Basics

Two points determine a line, three determine a polynomial, and, in general, $n+1$ points determine a degree-n polynomial (when the x values are distinct, this polynomial is a function in x). This idea gives the easiest approximation, where we make a polynomial passing through our points. If we have $n+1$ points, this degree-n polynomial is unique. There are multiple ways to compute this polynomial, but an easy one is to create

$$l_j(x) = \prod_{\substack{j=0 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j}$$

which disappear at the nodes other than x_j and are of degree n. Then, $p(x) = \sum y_i l_i(x)$ passes through data points y_i. If we have some function f producing y_i, so $p = \sum f(x_i) l_i(x)$, we have the interpolating polynomial in its Lagrange form. Since this is taxing to compute, we often instead use Newton's form, which computes in a recursive way. Given data (x_i, y_i), we take $p_0(x) = c_0$ and $p_i(x) = p_{i-1}(x) + c_i \prod_{j=0}^{i-1}(x - x_j)$ where we must compute the c_i via

$$c_k = \frac{y_k - p_{k-1}(x_k)}{(x_k - x_0)...(x_k - x_{k-1})}$$

This is where the recursion comes into computation, so we can easily compute this for higher degrees or when adding data as compared to the Lagrange form. You should understand the proof of the error bound of this interpolant, too,

$$f(x) - p(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi_x) \prod_{i=0}^{n}(x - x_i)$$

If we have a choice, can we pick the nodes in an optimal way to make the approximation very fast? Is it true that the interpolating polynomial will always converge to f?

Practice Problems

6.1.7) How many long operations (multiplication and division) does it take to compute c_k in the Newton interpolation method?

6.1.10) Write a formula for the coefficient of x^n in the lagrange interpolant $p(x) = \sum_{k=0}^{n} y_k l_k(x)$.

6.1.11) Prove that for any polynomial q of degree $\leq n-1$, $\sum_{i=0}^{n} q(x_i) \prod_{j \neq i}(x_i - x_j)^{-1} = 0$.

Other) Can we find a degree-3 polynomial p so $p''(x_0) = f''(x_0)$, $p'(x_0 = f'(x_0)$ and $p(x_0) = f(x_0)$? Is this unique?

Hermite Interpolation and Divided Differences

Divided differences rewrite the recursive computation above in a eay easier to compute by hand and to generalize. Let $f[x_0, ..., x_n]$ be the leading coefficient of the polynomial interpolating f at $x_0, ..., x_n$ (one of the problems above gives a strict formula for this). Then,

$$f[x_0, ..., x_n] = \frac{f[x_1, x_2, ..., x_n] - f[x_0, x_1, ..., x_{n-1}}{x_n - x_0}$$

hence the name "divided difference". You should try to prove this by comparing polynomials associated with each coefficient. This gives quite a slick way to compute coefficients by making an appropriate chart (which you will practice), and the symbolic usage gives a good way to show the dependence of coefficients and write formulas. For example, if t is not a node,

$$f(t) - p(t) = f[x_0, ..., x_n, t] \prod_{j=0}^{n}(t - x_j)$$

is another form of the error formula that relates to the above by noting $f[x_0, ..., x_n] = \frac{1}{n!} f^{(n)}(\xi)$ for some ξ.

Practice Problems

6.2.3) Let $f \in C^n[a,b]$ and let $x_0 \in (a,b)$. Prove that if $x_1, ..., x_n$ all converge to x_0, then

$$f[x_0, ..., x_n] \to \frac{f^{(n)}(x_0)}{n!}$$

(Hint: show the divided difference coefficient is continuous in its inputs in the region where they are distinct).

6.2.16) Compute the interpolant coefficients of $f(x) = 1/x$ for general $x_0, ..., x_n$.

One generalization possible by these divided differences is to note that we may use them to involve derivative information in interpolation. Let us define

$$f[x_0, ..., x_0] = \frac{1}{n!} f^{(n)}(x_0)$$

i.e. to be the Taylor coefficient (where we repeated x_0 n times). Then, when $x_0 \leq x_1 \leq ... \leq x_n$,

$$f[x_0, ..., x_n] = \frac{f[x_1, x_2, ..., x_n] - f[x_0, x_1, ..., x_{n-1}}{x_n - x_0}$$

when $x_n > x_0$. Thus, we may easily include curvature information in interpolation. When we have derivative information, we think about this data as repeating a node, which looks like a new piece of data when it comes to our formulas even though it has the same node. Almost all the formulas look the same in the case of repetition with this in mind.

Practice Problems

6.3.1 Altered) Give a quintic polynomial so $p(0) = 2$, $p(1) = -4$, $p(2) = 44$, $p'(0) = -9$, $p'(1) = 4$, and $p''(0) = 5$.

Splines, Natural Cubic Splines

For splines, we create piecewise functions that fit the function between the nodes. The easiest is to draw lines connecting the data points, which gives linear splines. Cubic splines are usually a strong contender to fit data, hence why we default to those. Further, odd-degree splines are well posed in the structure we presented in class. Recall that we assume that the function created by the piecewise parts is C^2, with $S(x_i) = y_i$ (how many equations does this generate? you should have $4n - 2$– write them out), where we then get natural cubic splines by the assumption $S''(x_0) = 0 = S''(x_n)$. You should understand how to derive the splines formulas from these conditions (remember the homework problem where you had to change the conditions and solve for new splines!). To actually solve for the splines, we use these conditions to develop formulas:

$$S_i(x) = \frac{z_i}{6h_i}(x_{i+1} - x)^3 + \frac{z_{i+1}}{6h_i}(x - x_i)^3 + (\frac{y_{i+1}}{h_i} - \frac{z_{i+1}h_i}{6})(x - x_i) + (\frac{y_i}{h_i} - \frac{z_i h_i}{6})(x_{i+1} - x)$$

where we have to solve for some of these coefficients. First, $h_i = x_i - x_{i-1}$. Next, form

$$\begin{aligned} u_i &= 2(h_i + h_{i-1}) \\ b_i &= \frac{6}{h_i}(y_{i+1} - y_i) \\ v_i &= b_i - b_{i-1} \end{aligned}$$

and z_i solve $z_0 = 0 = z_n$ and

$$\begin{bmatrix} u_1 & h_1 & 0 & \dots & \dots & 0 \\ h_1 & u_2 & h_2 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \dots & 0 \\ 0 & \dots & 0 & h_{n-3} & u_{n-2} & h_{n-2} \\ 0 & \dots & 0 & 0 & h_{n-2} & u_{n-1} \end{bmatrix} \begin{bmatrix} z_1 \\ \vdots \\ z_{n-1} \end{bmatrix} = \begin{bmatrix} v_1 \\ \vdots \\ v_{n-1} \end{bmatrix}$$

I won't add more problems here, but you should be able to do and understand the solutions for problems 6 and 15C on the review document.

3 Week 10: Numerical Integration Based on Interpolation (7.2) and Gaussian Quadrature (7.3)

Numerical integration techniques all take after what we talked about for differentiation: we approximate by taking some alrge or small value for a limit in the definition, and try to find approximations that converge more quickly.

Our starting point for integration would then be to take Riemann sums. Since (integral lower sum)

$$\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=0}^{n-1} \frac{b-a}{n} f(x_i^*)$$

for any $x_i^* \in (a + i\frac{b-a}{n}, a + (i+1)\frac{b-a}{n})$ when f is integrable, we could just take sufficiently large n and use this approximation. For simplicity, let $x_i = a + i\frac{b-a}{n}$ and $\delta x_i = x_{i+1} - x_i$.

However, the trapezoid rule

$$\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=0}^{n-1} \delta x_i [\frac{f(x_i) + f(x_{i+1})}{2}]$$

performs better in most cases. To analyze the error, recall that for continuous functions u, v where v is nonnegative and $a < b$,

$$\int_a^b u(x)v(x)dx = u(\xi)\int_a^b v(x)dx$$

for some $\xi \in (a, b)$. This is called the integral MVT, and it's proof is basically just IVT.

Using this, let $f \in C^2$ and $I = \int_a^b f dx$. Since the error computation is the same on each sub-interval we consider, we begin with just one sub-interval. Let $p(x)$ be a linear interpolant of f with nodes $1, b$, and by using divided differences at the data point x (theorem 3 of the lecture notes for 6.2),

$$\begin{aligned} I - \int_a^b p(x)dx &= \int_a^b f(x) - (f(x_0)l_0(x) + f(x_1)l_1(x)dx \\ &= \int_0^1 f[0, 1, x]x(x-1)dx \end{aligned}$$

The integral MVT then yields $\int_0^1 f[0, 1, x]x(x-1)dx = f[0, 1\xi_0]\int_0^1 x(x-1)dx = \frac{f''(\xi)}{2}\int_0^1 x(x-1)dx$ for some $\xi_0, \xi \in [a, b]$ (using another theorem from 6.2). Evaluating the integral gives

$$I - \frac{(b-a)}{2}(f(a) + f(b)) = -\frac{1}{12}(b-a)^3 f''(\xi)$$

From the interpolant proof, we also know that the Trapezoid rule is exact when f is a constant or linear polynomial, but not when curvature is involved (quadratic or higher). If we now consider the larger case when we subdivide (a, b) n times, and let $h = \frac{b-a}{n} = \delta x_i$,

$$I - T_n = -\frac{1}{12}(b-a)h^2 f''(\xi)$$

for some $\xi \in (a, b)$ and where T_n is the trapezoidal approximation for this n. This is justified by the computation

$$I - T_n = -\frac{1}{12}h^3 \sum_{i=0}^{n-1} f''(\xi_i)$$

and noticing that $\sum_{i=0}^{n-1} f''(\xi_i) = nf''(\xi)$ for some $\xi \in (a,b)$ by the intermediate value theorem.

We also have an alternative called Simpson's rule:

$$\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=0}^{n-1} \delta x_i [\frac{f(x_i) + 4f(\frac{x_i+x_{i+1}}{2}) + f(x_{i+1})}{6}]$$

Using a very similar analysis to the Trapezoid rule,

$$I - \frac{b-a}{6}(f(a) + 4f((a+b)/2) + f(b)) = -\frac{1}{90}\left(\frac{b-a}{2}\right)^5 f^{(4)}(\xi)$$

and Simpson's rule is exact for polynomials of degree at-most 3. Similarly, in the composite case with n even and $f \in C^4$,

$$I - S_n = -\frac{1}{180}(b-a)h^4 f^{(4)}(\xi)$$

Both rules have built-in functions in matlab for approximation.

What other options are there for integration? Both of the above are quite standard, but depending on the purpose, it may be useful to employ randomness. Monte Carlo integration uses random variables, and so it may give different values on different computations. At first, this seems like it wouldn't be great, but it is very useful for high-dimensional integration. The idea is that if we have $\Omega \subset \mathbb{R}^n$, a subset of volume V, then if we sample N points from Ω uniformly,

$$I \approx \frac{V}{N} \sum_{i=1}^{N} f(x_i)$$

where the law of large numbers guarantees convergence. In this case, the estimation error depends on the sample variance of the function and the volume of the space, but notably not the dimension of the space (whereas the rules above would depend on dimension). Often, the sampling distribution must be geared to the problem in mind (such as importance sampling) instead of being random to produce useful results, however. This topic isn't in the lecture notes or on the exam, but is nice to have in your back pocket.

Gaussian Quadrature is yet another way to compute these integrals in the flavor of the functional approximation. For every method so far, we pick some nodes $x_0, ..., x_n$ and will compute coefficients A_i so $\int_{-1}^{1} f dx \approx \sum A_i f(x_i)$. If we were to approximate f by the interpolant (in Lagrange form), we would have

$$f(x) = \sum f(x_i) l_i(x)$$

where

$$l_j(x) = \prod_{j \neq k} \frac{x - x_k}{x_j - x_k}$$

are the Lagrange cardinal functions. If we set $A_j = \int_{-1}^{1} l_j(x)dx$, then we have a perfect approximation for a polynomial of degree at most the number of nodes minus one. If we choose the location of the nodes quite wisely, we can reach further.

Let p be a polynomial of degree at most $2n+1$ and q a polynomial of degree $n+1$ with roots $x_0, ..., x_n$. If we have that $\langle x^k, q(x) \rangle = 0$ (the integral inner product), i.e. that our polynomial is orthogonal to powers of x for $k \leq n$, then we have equality again in quadrature: $\int_{-1}^{1} p(x)dx = \sum_{i=0}^{n} A_i p(x_i)$. This occurs because we can divide $p = qm + r$, integrate this equality, and use that m must have order at most n to realize that the first term disappears completely from the integral. Since it also disappears at the nodes x_i, only this remainder "matters" for computation of the integral at the nodes.

This orthogonality led Legendre to focus on a basis of polynomials for L^2 (a function space important in theoretical math) which are orthonormal and start with powers of x. Indeed, we take $\tilde{q}_i = x^i$ for $i \geq 0$, then apply Gram-Schmidt to get $q_0 = 1$, $q_1 = x$, and more. It turns out that these also satisfy a recurrence relationship

$$q_n = \frac{2n-1}{n} x q_{n-1} - \frac{n-1}{n} q_{n-2}$$

that makes for easy computation. The roots of q_n then apply to the theory of the last paragraph, and are called the Gaussian nodes of degree or order n.

For a general integral, apply a change-of-variables to get back to $[-1, 1]$. Now, it involves some functional analysis to prove, but for $f \in C([-1, 1])$ and Gaussian nodes $x_0, ..., x_n$ with coefficients as above,

$$\int_{-1}^{1} f dx = \lim_{n\to\infty} \sum_{j=0}^{n} A_j f(x_j)$$

The proof of this amounts to noting that f can be uniformly approximated by some polynomial $p(x)$, so the integral of f may also be approximated by the integral of the polynomial. Since we can compute this integral exactly through quadrature for large n, we are done.

[Need to Add Gaussian Quadrature]

4 Week 9: Numerical Differentiation (7.1)

Calculus is an extremely powerful tool that reduces complex problems to some simple operations. However, it only works on *continuous* things, and this often leaves many discrete analogues that are computationally difficult. We would like to use calculus and do operations with calculus, but computers can only work with discrete things. This is why we needed to store continuous functions as a finite sequence of data previously, as well. How do we take derivatives and integrals in this discrete setting? Your first example was my MatLab quickstart guide: finite-step differentiation. Instead of fully computing the limits in the definition of derivatives and integrals, we just "get close".

Finite-step differentiation relies on $f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$, so that $f'(x) \approx \frac{1}{h}[f(x+h) - f(x)]$. To make the approximation precise, we just apply Taylor's theorem: $f'(x) = \frac{1}{h}[f(x+h) - f(x)] - \frac{h}{2} f''(\xi)$, so that a "nice" function sees the error term decrease rapidly in h compared to the quotient term.

Remember that for complicated functions (square roots, inverse trig, logs...) we often compute the function values via something like Newton's method as well, so we do need to be careful about error. Furthermore, $f(x+h)$ and $f(x)$ should be similar values, so we expect to see subtractive cancellation as displayed in chapter 2, which limits our significant digits for small h. We then have some tradeoff between making h small enough to be accurate for the function and large enough to prevent such errors. Further formulas try to get around this, such as

$$f'(x) \approx \frac{1}{2h}[f(x+h) - f(x-h)]$$

which has $O(h^2)$ error as opposed to $O(h)$ (in the case that f is thrice differentiatible, that is). Similar manipulations fo Taylor's theorem give higher derivatives.

We have a second case to worry about. What if the function we care about is only known empirically, i.e. as a discrete list $(x_i, f(x_i))$ for $i = 0, 1, 2, ..., n$ such as for our polynomial approximation? In this case, we write the interpolant $p(x)$ in the Lagrange form at the nodes x_i and, from our error theorems,

$$f(x) = p(x) + \frac{1}{(n+1)!} f^{(n+1)}(\xi_x) w(x)$$

such that, if we fix $x = x_\alpha$

$$f'(x_\alpha) = p'(x_\alpha) + \frac{1}{(n+1)!} f^{(n+2)}(\xi_{x_\alpha}) w(x_\alpha) \frac{d}{dx}|_{x=x_\alpha} \xi_x + \frac{1}{(n+1)!} f^{(n+1)}(\xi_{x_\alpha}) w'(x_\alpha)]$$

in the case that x_α is a node, and using $w(x) = \prod_{i=0}^{n}(x - x_i)$ this simplifies to

$$f'(x_\alpha) = p'(x_\alpha) + \frac{1}{(n+1)!} f^{(n+1)}(\xi_{x_\alpha}) \prod_{i \neq \alpha} (x_\alpha - x_i)$$

which allows us to approximate the derivative of f with the polynomial derivative up to specified error.

We can squeeze even more precision out of these numerical formulas using something called Richardson's Extrapolation in a similar way to the sharper derivative formula. If we write $f(x+h) = \sum_{i=0}^{\infty} f^{(i)}(x) h^i/(i!)$, then

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{2}{3!}h^3 f'''(x) + \frac{2}{5!}h^5 f^{(5)}(x) + ...$$

Let $\phi(h) = \frac{1}{2h}[f(x+h) - f(x-h)]$, so rearranging this gives

$$f'(x) = \phi(h) + a_2 h^2 + a_4 h^4 + ...$$

If we write this with $h/2$ instead of h, we get $f'(x) = \phi(h/2) + a_2 h^2/4 + ...$ Next, we assume that this series is absolutely convergent, so that we may operate on the series termwise, and take

$$\begin{aligned} f'(x) = (4f'(x) - f'(x))/3 &= \frac{1}{3}\left(4[\phi(h/2) + a_2 h^2/4 + ...] - [\phi(h) + a_2 h^2 + a_4 h^4 + ...]\right) \\ &= \frac{4}{3}\phi(h/2) - \frac{1}{3}\phi(h) - a_4 h^4/4 - 5a_6 h^6/16 - ... \end{aligned}$$

where we picked this value of 4 to make the a_2 terms cancel. We now have $f'(x)$ in $O(h^4)$ precision! In the general setting, we don't need the whole Taylor series for this and only need f to be a C^5 function.

In the case that $f \in C^{2n+1}$, we may compute a higher-order scheme to get better error at the cost of more function evaluations. Let $D(n,0) = \phi(h/2^n)$, where $n = 0, ..., M$ and

$$D(n,k) = \frac{4^k D(n,k-1) - D(n-1,k-1)}{4^k - 1}$$

for $k = 1, 2, ..., M$ and $n = k, k+1, ..., M$. Then, $D(n,k-1) = f'(x_0) + O(h^{2k})$ as $h \to 0$ for $f \in C^{2n+1}$.

5 Week 8: Divided Differences, Hermite Interpolation, Splines

We're continuing the track of approximating functions. Last week, we looked at a rigid structure of interpolation and looked at different representations of the same polynomial. We assumed we had $n+1$ distinct nodes $x_0, ..., x_n$ and data points $y_0, ..., y_n$ to interpolate from. We could construct the interpolatant by using the basis of polynomials $1, x, x^2, ..., x^n$, but this is computationally inefficient, so we used Newton's form and Lagrange's form. This week, we formalize these computations a bit more and make more general fittings of the function.

5.1 Divided Differences

Recall that in Newton's form, we pivoted to the basis $q_j(x) = \prod_{k=0}^{j-1}(x - x_k)$. We then had $f(x_i) = \sum c_j q_j(x_i)$ to solve for the coefficients, which led to a system of equations with coefficient matrix A so $a_{ij} = q_j(x_i)$, which was lower triangular by the construction of the q_j. The system looks like

$$Ac = \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix}$$

and we know how to solve it quickly with forward substitution. The relevant values are then what we call **divided differences**: $f[x_0] = f(x_0)$ and $f[x_0, x_1] = \frac{f(x_1)-f(x_0)}{x_1-x_0}$,...,$f[x_0, ..., x_k] = c_k$. This emphasizes that the coefficient c_k only depends on $x_0, ..., x_k$, and really just gives a new name to an object we already knew about. We can then say quite a bit about these, like that $f[x_0, ..., x_k]$ is the leading coefficient of the interpolating polynomial of f over $x_0, ..., x_k$ (show this using the formula). In other words,

$$p(x) = \sum_{k=0}^{n} f[x_0, ..., x_k] \prod_{i=0}^{k-1} (x - x_i)$$

is the interpolated polynomial. If we change the values inside the square brackets, this amounts to changing the system involved.

Using this, we have a general form for computing the interpolant coefficients

Higher-Order Divided Differences:

$$f[x_0, ..., x_k] = \frac{f[x_1, ..., x_k] - f[x_0, ..., x_{k-1}]}{x_k - x_0}$$

(notice that these start at different values!)

Proof: Let $p_n(x)$ be the polynomial interpolating f at $x_0, ..., x_n$, and let q_n denote that interpolating from $x_1, ..., x_n$. Then,

$$p_k(x) = q_k(x) + \frac{x - x_n}{x_n - x_0}[q_k(x) - p_{k-1}(x)]$$

which amounts to evaluating both sides at the points $x_0, ..., x_k$ to verify (since they are degree k polynomials, this is enough). Tracking the leading coefficient, we are done. *QED*

We may then use this to compute the $f[x_0, ..., x_k]$ coefficients in practice by taking simple differences of the $f[x_i, ..., x_j]$. See page 332 of the text for pseudocode.

Notice also that if π is some permutation of $n + 1$-tuples, e.g. $\pi(x, y) = (y, x)$ in the 2-tuple case, $f[x_0, ..., x_n] = f[(\pi(x_0, ..., x_n)]$ since these are the leading coefficient of the same interpolating polynomial. Thus, we have some freedom in ordering things nicely. We also have the same error bounds as before written more nicely: if p is the interpolating polynomial of f at $x_0, ..., x_n$, then for $t \neq x_j$,

$$f(t) - p(t) = f[x_0, ..., x_n, t] \prod_{j=0}^{n} (t - x_j)$$

and, combining this with our MVT-form error gives for $f \in C^n$, $x_0 < ... < x_n$, $f[x_0, ..., x_n] = \frac{1}{n!} f^{(n)}(\xi)$ for some $\xi \in (x_0, x_n)$.

There is another formula for divided differences called the Hermite-Genocchi formula that relies on integration, but we won't focus on it here.

5.2 Hermite Interpolation

The name Hermite is pronounced "Air-meet". Hermite looked at a separate, orthonormal basis for polynomials that is quite useful in Fourier series computations and also simplifies our work here in some cases.

Instead of just having (x_j, y_j), imagine we have (x_j, y_j, y_j') where the second data point is for f', and we want to fit a polynomial passing through the data whose derivative passes through the primed-data. We can fit this, but must increase our degree:

Theorem on Hermite Interpolation: Let $x_0 < x_1 < ... < x_n$. For any $y_0, ..., y_n, y_0', ..., y_n' \in \mathbb{R}$, we have a unique polynomial p of degree at-most $2n+1$ such that $p(x_j) = y_j$ and $p'(x_j) = y_j'$.

The proof is just to create a system of $2n+2$ equations and notice it is overdetermined. Hermite interpolation refers to both this setting and also fitting higher-order derivatives. We can connect this to the previous section– let $f \in C^1(\mathbb{R})$. Then, define

$$f[x_0, x_0] = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0} = f'(x_0)$$

We think about this as the leading coefficient of the polynomial interpolating f twice at x_0, in the Hermite sense. This would look like $p(x) = f[x_0] + f[x_0, x_0](x - x_0)$. We can clearly extend to coefficients like $f[x_0, x_0, x_1]$ under this interpretation. We further define $f[x_0, x_0, ..., x_0] = \frac{1}{n!} f^{(n)}(x_0)$ to extend this to hgiher repetitions. This allows us to create a recursion as before:

Let $x_0 \leq x_1 \leq ... \leq x_n$. Then,

$$f[x_0, x_1, ..., x_n] = \begin{cases} \frac{f[x_1, ..., x_n] - f[x_0, ..., x_{n-1}]}{x_n - x_0} & x_n \neq x_0 \\ \frac{1}{n!} f^{(n)}(x_0) & x_n = x_0 \end{cases}$$

[Write out what this means in the interpolation table]

Example: Let's assume we have $f(0) = 1$, $f'(0) = 1$ and $f(1) = 3$. We can interpolate at $0, 0, 1$ as follows. First, $f[x_0] = 1$, $f[x_0, x_0] = f'(x_0) = 1$, and $f[x_0, x_1] = \frac{3-1}{1-0} = 2$. Then, $f[x_0, x_0, x_1] = \frac{2-1}{1-0} = 1$. Hence, we would fit $p(x) = f[x_0] + f[x_0, x_0](x - x_0) + f[x_0, x_0, x_1](x - x_0)^2 = 1 + (x) + 1(x^2)$.

Notice that we may overlap our taylor polynomials. If $f \in C^n$ and p is a polynomial of degree at-most n so $p^k(x_0) = f^k(x_0)$ for all $k \leq n$, then p is the Taylor polynomial of f of degree n centered at x_0. Lastly, we have a similar error bound to the other cases:

Hermite Interpolation Error Estimate: Let $x_0, ..., x_n$ be distinct nodes in $[a, b]$ and let $f \in C^{2n+2}([a, b])$. If p is the polynomial of degree at-most $2n+1$ such that $p(x_i) = f(x_i)$, $p'(x_i) = f'(x_i)$ for $0 \leq i \leq n$, then to each x in $[a, b]$ there corresponds $\xi \in (a, b)$ such that

$$f(x) - p(x) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \prod_{i=0}^{n} (x - x_i)^2$$

5.3 Splines

A spline function is a piecewise polynomial function. Suppose we have $n+1$ points $t_0 < t_1 < ... < t_n$ (we call these points "knots"), and that we have some integer $k \geq 0$. A spline function of degree k with these knots is a function S so that on each $[t_{i-1}, t_i)$, S is a polynomial of degree at-most k which has a continuous $(k-1)$st derivative on $[t_0, t_n]$. This allows us to keep small degree while fitting functions that do not grow like polynomials, such as $\sin(x)$. Cubic splines are commonly used in practice, and we will construct them here.

Assume we have a table of values $t(i, y_i)$ for $i = 0, ..., n$. We will use these both as knots and as interpolation nodes. Let S_i denote the cubic polynomial representing S on $[t_i, t_{i+1}]$. We should have $S_{i-1}(t_i) = y_i = S_i(t_i)$. We must also have $S_{i-1}'(t_i) = S_i'(t_i)$ and $S_{i-1}''(t_i) = S_i''(t_i)$ for the continuity of the derivative. These three types of conditions give $4n-2$ equations. However, we have $4n$ coefficients that we are treating as variables to solve for (4 coefficients for each of n S_i). Thus, we have two remaining degrees of freedom– $S''(t_0)$ and $S''(t_n)$ are unspecified.

The **natural spline** takes $S''(t_0) = 0 = S''(t_n)$. We have a special way to calculate this that involves using Gaussian elimination for tridiagonal matrices like the coding assignment created a couple weeks ago.

Define

$$h_j = t_{j+1} - t_j$$
$$u_j = 2(h_j + h_{j-1})$$
$$b_j = \frac{6}{h_j}(y_{j+1} - y_j)$$
$$v_j = b_j - b_{j-1}$$
$$v = \begin{bmatrix} 0 \\ v_1 \\ v_2 \\ \vdots \\ v_{n-1} \\ 0 \end{bmatrix}$$
$$A = \begin{bmatrix} 1 & 0 & \dots & \dots & \dots & 0 \\ h_0 & u_1 & h_1 & \dots & \dots & 0 \\ 0 & h_1 & u_2 & h_2 & \dots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & h_{n-2} & u_{n-1} & h_{n-1} \\ 0 & \dots & \dots & 0 & 0 & 1 \end{bmatrix}$$

Let z solve $Az = v$.

Then, the interpolating cubic spline of f is given by

$$S_j(x) = \frac{z_{j+1}}{6h_j}(x - t_j)^3 + \frac{z_j}{6h_j}(t_{j+1} - x)^3$$
$$+ \left(\frac{y_{j+1}}{h_j} - \frac{h_j z_{j+1}}{6}\right)(x - t_j) \left(\frac{y_j}{h_j} - \frac{h_j z_j}{6}\right)(t_{j+1} - x)$$

The derivation of this formula and matrix is on pages 351-354 of the text (there is a good bit of extra explanation), but it roughly amounts to starting with the continuity condition $S_i''(t_i) = z_i$ and $S''(t_{i+1}) = z_{i+1}$, such that

$$S_i''(x) = \frac{z_i}{h_i}(t_{i+1} - x) + \frac{z_{i+1}}{h_i}(x - t_i)$$

is the line between these two points. Integrating, and enforcing the continuity conditions on S' and S gives the above formula, while the matrix is a way of writing the continuity conditions in a vectorized format.

Anytime you see the word "natural" in mathematics, you should question why it is or what makes it natural to do. In the case of natural splines, this is because they are "good" in some sense. Let f'' be continuous on $[a, b]$ and $a = t_0 < ... < t_n = b$ be our knots. Let S be the natural spline interpolating f. Then, notice that

$$\int_a^b (f'')^2 dx = \int_a^b (S'')^2 dx + \int_a^b (f'' - S'')^2 dx + \int_a^b 2S''(f'' - S'')dx$$

where, by integration by parts,

$$\int_a^b S''(f'' - S'')dx = \sum_{i=1}^n S''(t_i)(f' - S')(t_i) - S''(t_{i-1})(f' - S')(t_{i-1}) - \int_{t_{i-1}}^{t_i} S'''(f' - S')dx$$

$$= S''(b)(f' - S')(b) - S''(a)(f' - S')(a) + \sum_{i=1}^n - \int_{t_{i-1}}^{t_i} S'''(f' - S')dx = 0 - 0 + \sum_{i=1}^n - \int_{t_{i-1}}^{t_i} S'''(f' - S')dx$$

$$= -\sum_{i=1}^n c_i[(f(t_i) - f(t_{i-1}) + S(t_{i-1}) - S(t_i)] = 0$$

where we eliminate the sums by the defintion of the spline. Notice that the naturality is what allowed us to eliminate the first sum from integration by parts. Now, this tells us that

$$\int_a^b [S'']^2 dx \leq \int_a^b [f'']^2 dx$$

which roughly says that we have a "lower energy" or "lower curvature" on the spline S than the function f. The book calls this an optimality condition, but it doesn't exactly show that these splines are optimal, just less poorly behaved than the original function. There are certainly other methods of computing splines, such as tension splines, taut splines, and some special cases. Splines may also be taken to both higher degrees than cubic and lower (usually linear splines), but even values aren't always well behaved. For odd degrees, there is a unique natural spline, however (see page 359), and a similar integral condition appears for higher derivatives.

6 Week 7: Matrix Norms (4.4) and Polynomial Interpolation (6.1)

6.1 Matrix Norms

Recall (from 110, 104, or 54) that we may define a norm $||\cdot||$ on a vector space V, which has the propoerties that it is positive definite, homogeneous, and satisfies the triangle inequality. Our standard norm on $\mathbb{R}^n$ is the Euclidean norm $||x|| = \sqrt{\sum_{i=1}^n x_i^2}$. However, we may also define the p-norm $||x||_p = [\sum_{i=1}^n |x_i|^p]^{1/p}$, and the limiting infinity-norm
$normx_\infty = \max |x_i|$. As an exercise, check that these satisfy the norm properties. It will not be relevant for our course, but all norms on $\mathbb{R}^n$ are equivalent (via a continuity argument).

For our first definition of a norm on matrices, we view matrices as linear transformations and apply the operator norm (which you may see in many other contexts)

$$||A|| = \sup\{||Au|| \mid u \in \mathbb{R}^n,\ ||u|| = 1\}$$

where we have some defined norm on $\mathbb{R}^n$. Notice there are two different meanins og the norm bars in this statement– this usage is common, so try to use context to determine what norm is meant. Our text calls this the **subordinate matrix norm** because of this intrinstic tie. Proving that this is truly a norm also relies on the properties of the norm on $\mathbb{R}^n$ (and you should check that they hold). In the case that the underlying norm is Euclidean, this is called the spectral norm (because $||A||_2 = \max_{1 \leq ileqn} |\sigma_i|$, the maximum singular value of the matrix, which we prove later).

Example
Since $||Ix|| = ||x||$ (where I is the identity), $||I|| = 1$. In the case that our underlying norm is the Euclidean norm (which arises from an inner product), any othogonal matrix preserves distances, and also has norm 1. This includes rotation matrices.

Using the homogeneity of the norm $||Ax|| = ||x|| \cdot ||A\frac{x}{||x||}|| \leq ||A||||x||$, an important property of this norm. Expanding similarly, we see that $||ABx|| \leq ||A||||B||||x||$ implies $||AB|| \leq ||A||||B||$.

Another commonly used matrix norm is

$$||A||_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^n |a_{i,j}|$$

which takes the largest row sum. This is the matrix norm subordinate to the infinity norm defined above (prove this and check norm properties).

Suppose we have a matrix equation $Ax = b$. We now have a way to understand "moving this equation slightly": if A is perturbed slightly, A^{-1} is perturbed to some new matrix B and the resulting $x = A^{-1}b$ will be pertubred to $\tilde{x} = Bb$ as well. We can measure this error:

$$||x - \tilde{x}|| = ||x - BAx|| \leq ||I - BA||||x||$$

or

$$\frac{||x-\tilde{x}||}{||x||} \leq ||I - BA||$$

giving a realtive error bound. We may aslo measure the residual error instead:

$$||x-\tilde{x}|| \leq ||A^{-1}||||b-A\tilde{x}|| = ||A^{-1}||||Ax||||b-\tilde{b}||||b||^{-1}$$

or

$$\frac{||x-\tilde{x}||}{||x||} \leq \kappa(A)\frac{||b-\tilde{b}||}{||b||}$$

where $\kappa(A) = ||A^{-1}||||A||$ is called the condition number of the matrix A. Let $e = x - \tilde{x}$ and $r = b - \tilde{b}$. We can get simple bounds from the above examples:

$$\frac{1}{\kappa(A)}\frac{||r||}{||b||} \leq \frac{||e||}{||x||} \leq \kappa(A)\frac{||r||}{||b||}$$

this shows how large condition numbers can lead to big errors, and so we expect a loss of precision in solving $Ax = b$ numerically. The whole of this study gives us an easy number to compute to encapsulate this.

6.2 Polynomial Interpolation

Assume you have some sort of function that you can get outputs for, but you have no formula or easy way to represent. How might you work with said function? One solution would be to compute the function at a bunch of points, connect them with lines, and use those lines to represent the function itself. If the function appears like it should curve a lot, maybe we could use a qudratic of polynomial instead. This is the idea of polynomial interpolation. We have some data points (x_i, y_i) for $i = 0, 1, ..., n$, and we wish to interpolate this data into a function, so we fit some polynomials to pass through them.

For n data points, we can fit a unique degree-n polynomial to pass through all of them. One way to do this is to make polynomials of increasing degree as we fit more data. For example,

$$p_k(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + ... + c_k(x - x_0)...(x - x_{k-1})$$

for each $0 \leq k \leq n$. In each step, we add 1 term with coefficient to account for the new y value. Hence, $c_0 = y_0$, $c_1 = \frac{y_1 - c_0}{x_1 - x_0}$.

These polynomials are called interpolation polynomials in Newton's form. We may compute values of the polynomial efficiently using Horner's method, as we already discussed, so this is an easy way to represent functions and bridge the gap in data. In general, we also can easily compute

$$c_k = \frac{y_k - p_{k-1}(x_k)}{(x_k - x_0)...(x_k - x_{k-1})}$$

An alternative method is to represent the polynomial as

$$p(x) = \sum_{k=0}^{n} y_k l_k(x)$$

where the $l_k(x)$ depend on the nodes x_i but not the ordinates y_j. In this case, we have

$$l_k(x) = \prod_{\substack{j=1 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j}$$

known as the cardinal functions or Lagrange form of interpolating polynomials. This will always agree with the polynomial computed in Newton's form, and is simply a different representation that makes evaulation a bit less expensive .

How good is this approximation?

Theorem on Interpolation Error Let f be a $C^{n+1}([a,b])$ function and let p be the polynomial of degree at-most n that interpolates f at the $n+1$ distinct points $x_0, x_1, ..., x_n$ in $[a,b]$. To each $x \in [a,b]$, these is a $\xi_x \in (a,b)$ so

$$f(x) - p(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi_x) \prod_{i=0}^{n} (x - x_i)$$

Proof We fix x, then set $w(t) = \prod_{i=0}^{n}(t - x_i)$ and $\phi(t) = f(t) - p(t) - \frac{f(x)-p(x)}{w(x)} w(t)$. By Rolle's theorem, $\phi^{(n+1)}$ has at least one zero, ξ_x, in (a,b) and so $0 = \phi^{(n+1}(\xi_x) = f^{(n+1)}(\xi_x) - (n+1)! \frac{f(x)-p(x)}{w(x)}$. *QED*

This is where the professor stopped, but we can expand a tad bit more. First, we are assuming that the nodes are given to us, but what if we can choose them? If we choose the nodes in a special way, maybe we can get a better approximation to the function. This was investigated by Chebyshev, leading to the Chebyshev polynomials. They are defined recursively: $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$ with $T_0(x) = 1$ and $T_1(x) = x$. If we allow the nodes x_i to be the roots of the Chebyshev polynomial T_{n+1} (and consider only $|x| \leq 1$), then we get an approximation error $|f(x) - p(x)| \leq \frac{1}{2^n(n+1)!} \max_{|t|\leq 1} |f^{(n+1)}(t)|$, an improvement over the last bound.

Now, we have some cool error bounds, but why should we trust interpolation at all? If we focus on a closed interval $[a,b]$, or really any compact set, the Stone-Weierstrass Theorem says that polynomials are dense in $C([a,b])$ under the uniform norm. This is to say we can approximate a function arbitrarily well. There are results that allow us to get an appropriate polynomial approximating f to be an interpolant, if the nodes are selected carefully. However, for poorly selected nodes, we can always break this uniform approximation, so we do need to be careful (see Faber's theorem).

7 Week 6.5: Midterm Review

The midterm covers chapters 1-3, so the calculus review, floating point numbers and the involved error, root finding techniques, and functional iteration. This whole document is a review document, bit here are some problems pulled from the text that I will use as practice during our review session in discussion. You should understand the pratice midterm and prioritize that over these problems, however.

1.) State Taylor's theorem with Lagrange Error. How does the Taylor polynomial approximate the function in big and little O notation?

1.1) Compute $ln(1.5)$ with accuracy at least 1/10 (you can write a sum and not evaluate it).

You should use the Taylor series for natural log about $x = 1$ and take a large enough sum so the error term may be bounded above by 1/10th. The 10-term sum is plenty for this

1.2) Show $1 + x \leq e^x$ for all $x \geq 0$. *Use the Taylor series for e^x and look at the first two terms*

2.) What is the larges k so $\arctan(x) = x + O(x^k)$.

The Taylor series for arctan *shows this is $k = 3$*

2.1) Is it true that $\frac{1}{n\ln(n)} = o(1/n)$

Yes, this is a direct definition-checking problem

2.2) Let $a_n \to 0$ and $\lambda > 1$. Show $\sum_{k=0}^{n} a_k \lambda^k = o(\lambda^n)$.

Compute $\lambda^{-n} \sum_{k=0}^{n} a_k \lambda^k = a_n + a_{n-1}\lambda^{-1} + ... + a_0 \lambda^{-n}$. As $n \to \infty$, terms like a_n, a_{n-1} approach 0 because the sequence does, and terms like $a_0 \lambda^{-n}$ approach 0 because $\lambda > 1$ and n is large

3.) Let $x = 2^{12} + 2^{-12}$. Find a machine number representation x^* for x (in Marc-32). What is the relative error between x and x^*?

You should see that this is evenly between two machine numbers, so either works, and that the relative error is 2^{-24}

3.1) If we compute the product of 3 numbers x_1, x_2, x_3 under floating point arithmetic, what is the maximum relative error between $fl(x_1x_2x_3)$ and $x_1x_2x_3$?

We must view the product as $fl(fl(fl(x_1)fl(x_2))fl(x_3)$. We have $fl(x_i) = x_i^* = x_i(1+\delta_i)$ where $|\delta_i| < 2^{-24}$ in Marc-32. Then, $fl(fl(x_1)fl(x_2)) = (x_1)(1+\delta_1)(x_2)(1+\delta_2)(1+\epsilon_1)$, where the extra error comes from the float of the final product. Repeating this again gives $x_1x_2x_3(1+\delta_1)(1+\delta_2)(1+\delta_3)(1+\epsilon_1)(1+\epsilon_2)$. Multiplying these gives a relative error of order approximately $5 \cdot 2^{-24}$.

3.2) Let $n\epsilon < 0.01$. Find an upper bound for $(1+\epsilon)^n - 1$ (not floating point!). One bound is 0.01006.

8 Week 6: Gaussian Elimination (4.3)

Linear algebra is extremely important to numerical problems. In last week's material, we went into a whole section on breaking down matrices via several factorizations just to make solving them easier in special cases. How can we compute these factorizations easily? We use a similar process to row reduction, but make the choices more consistent so that we can automate the process. Our goal will be Doolittle's factorization.

Let $A \in M(n)$. To perform Gaussian elimination on A, we perform a sequence of $n-1$ steps resulting in a sequence of matrices $A = A^{(1)} \to ... \to A^{(n)}$ where the fimal matrix is upper triangular. We will have

$$A^{(k)} = \begin{bmatrix} U^{(k-1)} & B^{(k)} \\ 0 & C^{(k)} \end{bmatrix}$$

where $U^{(k-1)}$ is upper triangular. In other words, we perform row reduction sequentially so that the rows above the row in the step of interest remain unchanged. How is this possible? In general, it may not be, and this is where our previous theorems on LU factorization make more sense. Recall that, if a matrix has nonsingular principal minors, it has a LU factorization.

The algorithm proceeds as follows. Assume we have matrix $A^{(k)} = (a_{i,j}^k)$ and want to form $A^{(k+1)}$. For the first k rows of $A^{(k)}$, we leave them be. Next, if i and j are both at least $k+1$, we reset values by subtracting off multiples of the pivot rows above:

$$a_{i,j}^{k+1} = a_{i,j}^k - \left(a_{i,k}^k / a_{k,k}^k\right) a_{k,j}^k$$

the remaining entries are all zeroes. For example, on the matrix

$$\begin{bmatrix} 6 & -2 & 2 \\ 12 & -8 & 6 \\ 3 & -13 & 9 \end{bmatrix}$$

we perform the iteration

$$\begin{bmatrix} 6 & -2 & 2 \\ 12 & -8 & 6 \\ 3 & -13 & 9 \end{bmatrix} \to \begin{bmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & -12 & 8 \end{bmatrix} \to \begin{bmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & 0 & 2 \end{bmatrix}$$

If all the pivots $a_{k,k}^k$ of this process are nonzero, then this produces $U = A^{(n)}$ and we collect the row operations for L:

$$l_{i,k} = \begin{cases} a_{i,k}^k / a_{k,k}^k & i \geq k+1 \\ 1 & i = k \\ 0 & i \leq k-1 \end{cases}$$

When doing this by hand, it's easier to just keep track of the multipliers you used for each step (for the above, $L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/2 & 3 & 1 \end{bmatrix}$). To prove this statement, check that for L and U formed this way, $LU = A$. This amounts to chasing definitions.

This algorithm is great when it applies, but because it assumes the matrix has good structure, we should also try to form an algorithm that can address zeroes in the pivots as well as floating point and rounding issues that arise from having small pivots (you should be able to explain why small pivots can be an issue, perhaps using the example $\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix}$). We call this process pivoting. You already know this process from using row reduction to get row echelon form, or perhaps reduced row echelon form to account for large pivots. In that case, we actually swap the rows, but here we will not swap them but still treat them as such and keep track of what permutation needs to be done to get an upper triangular matrix.

An algorithm for this process is scaled pivoting. First, we start with a square matrix $A \in M(n)$. To find the "large pivots", compute the scales $s_i = \max_{1 \leq k \leq n}\{|a_{i,k}|\}$ for $i = 1, ..., n$ and select the first pivot row $A_{p_1}^{(1)}$ where $|a_{p_1,1}|/s_{p_1} \geq |a_{i,1}|/s_1$ (where we at least reach the maximum relative size for the first column). We use this row to reduce the others, and log the number p_1 so the first column of our permutation matrix is e_{p_1}. Next, we repeat throughout the rest of the columns, where we exclude previous pivot rows from our maximum choices. Doing this n times returns $A^{(n)}$ and P, so $PA = LU$ where $U = PA^{(n)}$, and L again tracks our progress.

Example: For a small-scale example, consider

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 1 & 1 \\ 3 & 2 & 4 \end{bmatrix}$$

Our first step will choose $p_1 = 3$ and subtract to

$$A^{(1)} = \begin{matrix} 01 & & 2 \\ 01/3 & & -1/3 \\ 32 & & 4 \end{matrix}$$

with

$$P = \begin{bmatrix} 0 & ? & ? \\ 0 & ? & ? \\ 1 & ? & ? \end{bmatrix}$$

For column 2, we exclude the third row and choose $p_2 = 1$, so

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Gaussian elimination computes $A^{(2)}$ and $A^{(3)}$ in the obvious way.

There is another algorithm in the book that we will not focus on called complete pivoting. When the previous processes move forward th column k, they each consider $n - k + 1$ elements in the column to make pivot choices (n elements minus the previous pivot rows). Complete pivoting instead considers the $(n-k+1)^2$ elements of the submatrix formed by excluding previous pivot rows and columns, then picks either a pivot row or column.

Let the resulting permutation matrix be P, and this solve $PA = LU$. When we now go to solve $Ax = b$, we perform this process and have $PAx = Pb$, giving $LUx = Pb$. Updating $y = Pb$, we solve $Lz = y$ then $Ux = z$ to find x.

There are some cases where we may safely used Gaussian Elimination as opposed to one of these pivoting algorithms. A matrix is called diagonally dominant if

$$|a_{i,i}| > \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{i,j}|$$

for all $1 \leq i \leq n$. Gaussian elimination without pivoting preserves the diagonal dominance of a matrix, and so by our previous discussion it gives and LU factorization where U is diagonally dominant. As U is upper triangular, this implies it has nonzero diagonal entries and that the original A was also nonsingular. Applying full or scaled row pivoting to a diagonally dominant matrix will also be the exact same process to Gaussian elimination, so this is really the special case which amounts to tracking that choice of pivots.

9 Week 5: Computing roots of polynomials (3.5), Matrix Algebra and Factorizations (4.1-4.2)

9.1 Polynomial Roots

Due to the work of Gauss, we know that a degree n polynomial has n complex roots (the Fundamental Theorem of algebra paired with polynomial division). Following our trend on root finding to replace various proceeses like rational roots, we would like to find an efficient method of computing these roots. We start with localization theorems, which tell us a range of where the roots may lie. Consider a polynomial $p(x) = a_n z^n + a_{n-1} z^{n-1} + ... + a_0$.

Localization Theorem All the roots of $p(x)$ written as above lie in $B(0,\rho) \subset \mathbb{C}$ where $\rho = 1 + |a_n|^{-1} \max_{0\leq k\leq n} |a_k|$.

Proof: Let $c = \max_{0\leq k\leq n} |a_k|$. The case $c = 0$ is immediate, so assume $c > 0$ and $\rho > 1$. For $|z| \geq \rho$, (use the reverse triangle inequality and geometric sums).

$$|p(z)| \geq |a_n z^n| - c\sum_{k=1}^{n-1} |z|^k > |a_n z^n| - c|z|^n(|z|-1)^{-1} \geq |a_n z^n|(1 - c|a_n|^{-1}(\rho-1)^{-1}) = 0$$

QED

This theorem implies a second:

Corollary: If all the roots of $z^n p(1/z)$ are in the disk $B(0,\rho)$, then all the non-zero roots of p are outside $B(0,\rho^{-1})$.

Proof: The polynomial $z^n p(1/z)$ is $s(z) = a_n + a_{n-1}z + ... + a_0 z^n$. Then, $p(z_0) = 0$ is equivalent to $s(1/z_0) = 0$. Since the roots of s have $|z| \leq \rho$, we then have $\frac{1}{|z|} \geq \rho^{-1}$, as desired. *QED*

These results give a starting point for algorithms to take hold. We will focus on Newton's method, but the text also covers Bairstow's method and Laguerre iteration. For Newton's method, we want an easy way to comput the polynomial, especially centered at different points or in the nested multiplication form from the first week. For this, we use Horner's method.

Starting with a polynomial $p(z)$ of the above form, we perform an artificial polynomial division to reduce the degree of the polynomial. Since $p(z) - p(z_0)$ has a root at z_0, we may form $q(z) = \frac{p(z)-p(z_0)}{z-z_0}$ a degree $n-1$-polynomial such that $p(z) = p(z_0) + q(z)(z - z_0)$. Horner's method tells us how to compute this $q(z)$ and $p(z_0)$. Let $q(z) = b_{n-1}z^{n-1} + ... + b_0$. Then, using the product relationship $p(z) = p(z_0) + q(z)(z - z_0)$, $a_n = b_{n-1}$, and $a_j + z_0 b_j = b_{j-1}$ for $0 \leq j \leq n-1$. Rewriting this relationship, we have $b_k = a_{k+1} + z_0 a_{k+2}$. We may compute this all the way down to b_0, and if we put in $k = -1$, the value b_{-1} we solve for is actually $p(z_0)$.

One way this helps our algorithms on the computer side is that it allows us to compute $p'(z_0)$ without needed to use symbolic logic in the comuter (i.e. without using the power rule). Using a taylor series on p, we may see

$$p(z) = c_n(z - z_0)^n + ... + c_1(z - z_0) + c_0$$

where $c_1 = p'(z_0)$. Applying Horner's method at z_0 gives $p(z) = (z - z_0)q(z) + c_0$. It then follows that $q(z_0) = p'(z_0)$ from the product rule. Another way to see this is to write

$$p(z) = c_0 + (z - z_0)(c_1 + (z - z_0)(c_2 + ... + (z - z_0)c_n)...)$$

where, aligning with Horner's relationship, $q(z) = (c_1 + (z - z_0)(c_2 + ... + (z - z_0)c_n)...)$. Repeating Horner's method on q then gives the higher Taylor coefficients of p: for $p(z) = (z - z_0)q_0(z) + p(z_0)$ and $q_i(z) = q_{i+1}(z)(z - z_0) + q_i(z_0)$; then $q_k(z_0) = p^{(k)}(z_0)/k!$ and these values are the last value produced in each step of Horner's method.

We may now work on finding roots. In the special case z_0 is a root, our formula is simply $q(z) = \frac{p(z)}{z-z_0}$, so this division merely decreases the degree of p without any added complication. We may apply Newton's method to $p(z)$, computing its values and its derivative's values by Horner's method, then have $p(z) = (z - z_0)q(z)$ and repeat this idea on $q(z)$. In a finite number of steps, we will have fully factored $p(z)$.

The localization methods give us a good starting area, and we may use polynomial coefficients or estimates to get an idea on a radius that implies convergence for Newton's method. We also have the following:

Theorem on Successive Newton Estimates (Bodewig 1946) Let x_k and x_{k+1} be two successive iterates when NEwton's method is applied to a degree n polynomial p. Then, there is a root of p within distance $n|x_k - x_{k+1}|$ of x_k in the complex plane.

Proof Let $r_1, ..., r_n$ be the roots of p. Then, $p(z) = c\prod_{j=1}^{n}(z - r_j)$, and

$$p'(z) = c\sum_{i=1}^{n}\prod_{j\neq i}(z - r_j) = \sum_{i=1}^{n}\frac{p(z)}{z - r_i}$$

We claim that for and z, there is some index j so $|z - r_j| \leq n|p(z)/p'(z)|$. If this were to fail, we would have

$$|z - r_j|^{-1} < \frac{1}{n}|p'(z)/p(z)| = \frac{1}{n}|\sum(z - r_i)^{-1} \leq \frac{1}{n}\sum|z - r_i|^{-1}$$

Since the number on the right is an average, this is impossible. Notice now that $n|x_k - x_{k+1}| = n|\frac{p(x_k)}{p'(x_k)}|$, so that we have shown root is within this radius of x_k.

QED

This theorem may be useful for your code or HW, and gives us an idea of how to set error bounds.

9.2 Matrix Algebra and Factorizations

In Math 54, we learned several factorizations like diagonalization, LU-factorization, QR-factorization, and Singular Value decomposition. These will be useful in making code and algorithms more efficient, so we review. Let $A \in M(n, \mathbb{R})$, which means A is a $n \times n$ matrix with entries in $\mathbb{R}$. We denote $A = (a_{i,j})$ to say that the $a_{i,j}$ are the entries of A in the i, j positions. We also sometimes denote the rows of A by A_i or the columns by A_j.

Recall that we do many processes like Gaussian elimination, row reduction, and even diagonalization by elementary row operations. These operations include adding rows, interchanging rows, and multiplying rows by a nonzero constant. For each operation, there exists a matrix E so EA is the same as doing the row operation to A. These matrices E are invertible, and their inverse signifies the opposite row operation. Recall that EA and AE may be different, as multiplication is not commutative for matrices. You should

review what being an invertible matrix means, how to compute simple inverses, and how invertibility for a square matrix is the same as being injective, surjective, being the product of elementary matrices, or having nonzero determinant. There are some guiding questions on a MAth 54 review doc from my older teaching sites.

We begin with LU-factorization. Recall that L is a lower triangular matrix if all entries strictly below the diagonal are 0, or $l_{i,j} = 0$ for $i < j$. We may similarly have an upper triangular matrix U. These types of matrices are useful for numerical methods because solving $Lx = b$ or $Ux = b$ is quite easy compared to for a standard matrix. For example, consider

$$\begin{bmatrix} a & 0 \\ b & c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \end{bmatrix}$$

Then, we have $ax_1 = d_1$, so $x_1 = d_1/a_1$. This helps us solve the second equation, $bx_1 + cx_2 = d_2$ then becomes $x_2 = c^{-1}(d_2 - bd_1a_1^{-1})$. For larger matrices, the same recursion applies to simplyifying inverting L.

A square matrix A has an LU factorization if $A = LU$ where L is lower triangular and U is upper triangular. Like many other factorizations, this is not unique: if D is an invertible diagonal matrix, and $A = LU$, then $A = (LD)(D^{-1}U)$ is another LU-factorization. When A is invertible, it is possible to show that all LU factorizations of A are related by such a relationship.

What criteria do we have to imply an LU-factorization exists? Let $B \in M(k)$ be a sub-matrix of A. The leading principal matrices of A are those B where $b_{i,j} = a_{i,j}$ for all i, j, or where B is located in the top left corner of A. If all leading principal matrices of A are invertible, then A has n LU-factorization (the proof of this is semi-long, but should be understood. I will replicate it here at a later date).

We usually compute the LU-factorization by applying row operations to A to get an upper triangular matrix U, then collecting the elementary matrices to what should be a lower triangular matrix for L (if an LU factorization even exists). For example, consider the matrix

$$\begin{bmatrix} 8 & 4 \\ 4 & 1 \end{bmatrix}$$

By subtracting the half of the first row from the second, we obtain an upper triangular matrix

$$U = \begin{bmatrix} 8 & 4 \\ 0 & -1 \end{bmatrix}$$

and this elemetary row operation is encoded by $\begin{bmatrix} 1 & 0 \\ -0.5 & 1 \end{bmatrix}$. The inverse of this operation is $L = \begin{bmatrix} 1 & 0 \\ 0.5 & 1 \end{bmatrix}$, and so we have $A = LU$ is an LU-factorization.

A more foolproof way to compute it theoeretically would be to set up systems of equations, but such a method includes exactly the type of problem we are trying to solve. Simplifying this system leads to the special case of Doolittle's factorization, an LU-factorization with $l_{i,i} = 1$ for $i = 1, ..., n$ (depending on your reference, these may just be nonzero, which is equivalent by the comments above about diagonal matrices). We may then recursively solve for the values of L and U using the equations

$$u_{k,j} = a_{k,j} - \sum_{s=1}^{k-1} l_{k,s}u_{s,j}$$

$$l_{i,k} = \left[a_{i,k} - \sum_{s=1}^{k-1} l_{i,s}u_{s,k} \right] / u_{k,k}$$

Try multiplying out abstract matrices in the 3×3 case to make sense of these equations. You should notice things like the first row of U and the first row of A being the same, and move from there.

Similarly, Crout factorization is an LU-factorization with $u_{i,i} = 1$ for all i, and also has a recursive algorithm for computing the entries of L and U. We also have a decomposition that looks to LU-factorization as orthogonal diagonalization does to diagonalization. Cholesky's factorization is an LU-factorization where $U = L^T$, or $A = LL^T$. Such a factorization exists and is unique if A is a symmetric and positive definite matrix, but may exist in much broader cases.

10 Week 4: Fixed Points and Functional Iteration (3.4)

The generalizations of Newton's method may be posed as a type of function iteration. If we define $F(x) = x - \frac{f(x)}{f'(x)}$, where f is the function we are trying to find a root of, then Newton's method becomes $F(x_n) = x_{n+1}$. This is a useful situation because there is a strong theorem in analysis that helps us find limits in such cases, called the contraction mapping theorem.

This theorem supposes that we have a non-empty complete metric space X, and defines a function $F : X \to X$ to be a contraction if $|F(x) - F(y)| \leq c|x - y|$ where $c \in (0, 1)$. Under this condition, it poses that we have a unique fixed point x^*, i.e. a point where $F(x^*) = x^*$, and the proof of the theorem shows that we may reach such x^* by taking any point y and $F^n(x) = F(F(...F(x)...))$ converges to x^* as $n \to \infty$.

In the context of the Secant method or Newton's method, we then converge to a root if we can pose appropriate F as a contraction. The MVT is often helpful in showing appropriate functions are contractions on the right spaces.

Finally, we care about the rate of convergence. Given F and $x_{n+1} = F(x_n)$, where F is a contraction with fixed point x^*, we trace $e_n = x_n - x^*$. Under suitable assumptions,

$$e_{n+1} = F(x_n) - F(s) = F(s + e_n) - F(s) = e_n F'(s) + \frac{1}{2}e_n^2 F''(s) + ... + \frac{1}{(q-1)!}e_n^{q-1}F^{(q-1)}(s) + \frac{1}{q!}e_n^q F^{(q)}(\zeta_n)$$

so that if s is an order -q root of F, then $e_{n+1} = \frac{1}{q!}e_n^q F^{(q)}(\zeta_n)$ and we have q-order convergence. Try checking the convergence for Newton's method and the Secant method.

Example: Determine the order of convergence of the sequence $x_n = n^{-1/2}$. We should expect linear convergence from normal limit arguments, but let us check using this method. Squaring the terms, we see $x_n^2 = n$, so that $x_n^{-2} + 1 = n + 1 = x_{n+1}^{-2}$. This is to say

$$F(x) = \frac{1}{\sqrt{\frac{1}{x^2} + 1}} = \frac{x}{\sqrt{1 + x^2}}$$

$$F'(x) = \frac{1 - x^2}{(x^2 + 1)^{3/2}}$$

so that $F'(0) \neq 0$. We expect linear convergence, and this may also be checked directly from the definition of the order of convergence.

10.1 Aitken's Extrapolation Formula

Let $F \in C^1(\mathbb{R})$ be a contraction with fixed point s, and $x_{n+1} = F(x_n)$ defining a sequence converging to that point. Then,

$$s - x_n \approx \frac{\lambda_n}{1 - \lambda_n}(x_n - x_{n-1})$$

$$\lambda_n = \frac{x_n - x_{n-1}}{x_{n-1} - x_{n-2}}$$

Aitken's method stipulates that to compute s, starting with x_1, we compute $x_2 = F(x_1)$ and $x_3 = F(x_2)$, then compute x_4 via the above formula. To proceed, we compute $x_n = F(x_{n-1})$ if $n \equiv 1, 2, 3 \mod 4$ and use Aitken's formula for x_{4k}.

11 Week 3: 3 Root finding Algorithms (3.1-3.3)

11.1 Bisection Method

The bisection method starts with a *continuous* function f and an interval $[a, b]$ on which we have $f(a)f(b) < 0$. This is a special assumption to guarantee that we have a root of our function f on $[a, b]$ by the Intermediate Value Theorem.

We then produce $c = \frac{a+b}{2} = a + \frac{b-a}{2}$, the midpoint of the interval. We check whether $f(a)f(c) < 0$, in which case we set $[a_1, b_1] = [a, c]$. In the case $f(a)f(c) > 0$, we know $f(a)f(b) < 0$, so we et $[a_1, b_1] = [c, b]$. Finally, if $f(c) = 0$, we terminate. We repeat this process, producing c_1 and then $[a_2, b_2]$.

This way, each step cuts the interval in half. We use c_n as out approximation of the root. If r is the root we eventually converge to, we know $|c_n - r| \leq \frac{1}{2}|b_n - a_n| \leq 2^{-n}|b - a|$.

11.2 Newton's Method

For Newton's method, we assume that we have a C^2 function f for which we want to find a root. We start with some initial value x_0. Then, we assume there exists some root r, and so we know $0 = f(r) = f(x_0 - e_0)$ where $e_0 = x_0 - r$. Expanding by Taylor's theorem, $0 = f(x_0) - e_n f'(x_0) + ...$, so we approximate $e_0 \approx f(x_0)/f'(x_0)$ and update

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

at each subsequent step.

This isn't guaranteed to converge for any starting point, but let's look at the case of convergence.

$$e_{n+1} = x_{n+1} - r = x_n - \frac{f(x_n)}{f'(x_n)} - r = e_n - \frac{f(x_n)}{f'(x_n)}$$

Thus, we use that $0 = f(x_n) - e_n f'(x_n) + e_n^2 f''(\zeta_n)/2$ from above to rewrite this as

$$e_{n+1} = \frac{e_n f'(x_n) - f(x_n)}{f'(x_n)} = \frac{1}{2}\frac{f''(\zeta_n)}{f'(x_n)} e_n^2$$

When do we see convergence? We need the $e_n \to 0$ for this, which is okay when this coefficient is bounded. This is a second place where we use the C^2 assumption. Set

$$c(\delta) = \frac{1}{2}\frac{\max_{|x-r|<\delta} |f''(x|)}{\min_{|x-r|<\delta} |f'(x)|}$$

Using continuity, take δ small enough so $\delta c(\delta) < 1$. Then, it is easy to note that if $|e_n| \leq \delta$, then $|e_{n+1}| \leq \delta$ as well and that $|e_{n+1}| \leq C|e_n|^2$, so that we have quadratic convergence when x_0 is chosen within a δ-ball of r.

11.3 Alternatives to Newton's method

In cases where we don't want to compute the derivative, several alternatives exist. The first approximation we might take is that $f'(x) \approx \frac{f(x+h)-f(x)}{h}$ for small h. Using this in Newton's method gives the Secant method,

$$x_{n+1} = x_n - \frac{f(x_n)}{(f(x_n) - f(x_{n-1}))/(x_n - x_{n-1})}$$

though several alternatives exist, including Steffenson's method (further approximation) and Halley's method (which requires the second derivative). We will explore their convergence rates through functional iteration next week.

12 Week 2: Floating Point Numbers and Machine Numbers (Ch. 2)

12.1 Floating Point Numbers

It would be amazing if we could punch numbers into computers and store exact values, but out computers only have finite memory and numbers like π that are irrational can't be stored in a finite number of base-10 digits. Furthermore, the hardware restrictions of semiconductors when computers were developed means that base-10 is not even natural to create mechanically. For these reasons, we instead store numbers in a finite string of 0s and 1s called bits, which represent a rounded version of the value. This type if storage is called floating point storage, and it can cause some issues if we aren't careful.

First, we represent numbers in scientific notation: $x = (-1)^s r \times 10^n$ where $r \in [0.1, 1)$ and n is an integer. This gives a way to store a spread of values in a consistent way. In binary, this would be of the form $x = (-1)^s q \times 2^m$. The number q is called the mantissa, and m is called the exponent. Since $q \in [1/2, 1)$, we always have a leading 1, so we actually shift the mantissa over by one digit to squeeze more precision out of the same number of bits. Since we want to store both large and very small numbers without needing an extra negative sign for the exponent (which takes extra storage), we also bias the exponent

$$x = (-1)^s 1.f \times 2^{e-127}$$

This is called a normalized floating point form, and is also the IEEE standard floating point representation.

Then, $s \in \{0, 1\}$ will always account for at most 1 bit. The other numbers f and e have lengths depending on the system. In a standard 32-bit system, f recieves 23 bits and e recieves 8 bits.

Numbers of exactly this form are called machine numbers. Some very simple decimal numbers are not machine numbers, such as 0.01. In such cases, most machines round a number to its nearest machine number (though there are other rounding systems in use). In practice, this means we compute the number x in binary to some accuracy, then truncate the expansion to get a machine number just below it, x_1. We also look at the next machine number above x_-, x_+. When the check which is smaller between $x_+ - x$ and $x - x_-$ to determine the machine number.

When arithmetic is done, the computer adds bits to make for accurate calculations. The standard for a 32-bit storage is 80 bits for internal calculations (check out the IEEE standards for more info).

Sometimes, special exponents like $e = 0$ and $e = 255$ are reserved for 0 and ∞, and the computer may return a NaN (not a number). Alternatively, arithmetic may produce numbers of exponent smaller or larger than the floating point storage allows, which are called underflow and overflow errors. There are some interesting examples of these errors causing problems in the real world.

The above process of selecting machine numbers close to our number doesn't have a consistent absolute error. Consider the machine ϵ, or the distance between machine numbers in a given range. For Marc-32 and machine numbers in $[1/2, 1)$, this ϵ is 2^{-24}. For numbers in $[1, 2)$, the ϵ is 2^{-23}. However, what we do preserve is the relative error. In Marc-32, our relative error will be at-most 2^{-24}. We may pose $fl(x) = x(1 + \delta)$ where $|\delta| \leq 2^{-24}$ to do error analyises. For example, taking the product of two floating point numbers. We have one main theorem using these:

Theorem on Relative Rounding Error in Adding: Let $x_0, x_1, ..., x_n$ be positive machine numbers in a computer whose unit roundoff error is ϵ. Then, the relative roundoff error in computing $\sum_{i=0}^{n} x_i$ in the usual way is at most $(1 + \epsilon)^n - 1 \approx n\epsilon$.

12.2 Loss of Significance, Subtraction of Nearby Quantities, Accuracy Theorem

Consider the case of $x = 2^n + 2^k$ and $y = 2^n$, so $x - y = 2^k$. However, in floating point arithmetic, if n is very large compared to k, we would get a differenc of 0 and could have large absolute error and relative error. In another situation, consider $x = 0.3721478693$ and $y = 0.3720230572$. We have $x - y = 0.0001248121$. Writing this in floating point arithmetic, $fl(x) = 0.37215$, $fl(y) = 0.37202$ and $fl(x) - fl(y) = 0.00013$, giving large relative error but also showing that we lose decimal places of accuracy in the difference, which we would write as 0.13000×10^{-3}. These are examples to display the error and loss of significance when subtracting nearbly values in floating point arithmetic. In general, we have

Theorem on Loss of Precision: If x and y are positive normalized floating-point binary machine numbers so $x > y$ and

$$2^{-q} \leq 1 - \frac{y}{x} \leq 2^{-p}$$

then at most q and at least p significant binary bits are lost in the subtraction $x - y$.

Proof: Let $x = r \times 2^n$ and $y = s \times 2^m = (s \times 2^{m-n}) \times 2^n$ (where we shift to place x and y on the same exponent, remembering $x > y$). Then,

$$x - y = (r - s \times 2^{m-n}) \times 2^n$$

where $2^{-q} \leq r - s \times 2^{m-n} = r(1 - \frac{y}{x}) \leq 2^{-p}$. Thus, we shift this mantissa left at least p bits and at most q bits, and lost at least p and at most q bits. *QED*

Thus, in cases like computing $\sqrt{1+n^2} - n$ for large n, we should perform algebraic manipulations (multiplying by the conjugate here) to eliminate the subtraction and reduce error. In other cases, we may use techniques like Taylor expansion.

12.3 Stability

[Incomplete]

13 Week 1: Calculus Review (1)

- Add IVT, MVT, Taylor's Theorem
- Add Orders of Convergence, big and little O notation, MVT for integrals, nested multiplication