University of Toronto, Scarborough Fall CSCC37
CSCC37 Finals Notes

Chapter 1 – Floating Point Arithmetic

Why we might have errors:

- Truncation (difference between true result for input and the result produced by exact arithmetic, due to approximations like truncating a series, etc.)
- Rounding (difference between result produced by exact arithmetic and result produced by using finite-precision rounded arithmetic)

Well Posed Problems

- Solution exists
- Solution is Unique
- Solutions behavior changes continuously with initial conditions

Error Systems

Absolute Error = (approx. value) – (true value)
Relative Error = (abs error) / (true value)
If a quantity has an error of 10^{-t} then the decimal representation of x^ has about *t* significant digits
(Approx. value) = (true value) x (1 + relative error)

Conditioning

Conditioning refers to the change of the solution in relation to the change in the input data.
cond = |relative change in solution| / |relative change in data|
= |(f(x)^ - f(x))/(fx)| / |(x^ - x)/x|
For x^ = (x+h) where h is the perturbed amount, we have:
Absolute error = *hf'(x)*
Relative error = *hf'(x)/f(x)*
Condition = |*xf'(x)/f(x)*|
If the condition number is too far from 1, then the system is ill-conditioned

Backward Error Analysis

How much data error in initial input is required to account for the error in the final result? (Working backwards)
Let us have $f(x) = e^x$. We would have for x = 1:
Forward Error = f(x)^ - f(x) = -0.051615
Backward Error = x^ = log(f(x)^) = -0.019171

Stability and Accuracy

Stability – If the result it produces is the exact solution to a nearby problem
Accuracy – Closeness of a computed solution to the true solution of the problem under consideration
Stability !→ Accuracy. Stable algorithms can give us a solution to a nearby problem, but that may not be relevant if the question is ill-conditioned. Thus,
Inaccuracy could be **stable algorithms** in ill-conditioned**,** or **unstable algorithms** to well-conditioned problems

Floating Point Numbers

Characterized by four integers:
B **-** Base or radix
t - Precision
[L; U] - Exponent range
So basically any number can be represented by:

$$X = +/- (d_0 + (d_1/B) + (d_2/B^2) + \dots + (d_{t-1}/B^{t-1})B^e$$

Where $0 <= d_i <= B-1$ and i = 0, …, t – 1 and L <= e <= U
Which means, that any number in the number system represented by B, t, [L; U] can be represented by having fractions of numbers of t length (mantissa length),

Normalization

A floating point system is said to be normalized when d_0 is nonzero, unless the represented number is zero. Thus,
1 <= m < B holds.
Thus, we have the number of normalized floating point numbers as:
$2(B – 1)\ B^{t-1}\ (U – L + 1) + 1$
^ A ^B ^C ^D
A – We can have two choices of sign, and B-1 options for leading mantissa digit
B – We can have B options for the next t-1 digits of the mantissa
C – We can have U-L+1 options for the exponents
D – We can have zero as a number

We have **Underflow** Level (UFL) = B^L which means that leading digit is 1, and 0s for rest of mantissa with smallest exponent
We have **Overflow** level (OFL) = $B^{U+1}(1-B^{-t})$ each mantissa digit is as large as possible, with largest possible exponent.
Floating point numbers **are not evenly distributed** within their range, but are equally spaced between successive powers of B
Real Numbers that can be represented in a given floating point system are called **machine numbers**
<u>Rounding</u>
If a given real number is not representable as a FPN, we use a nearby number represented by x = fl(x), this is called **rounding**
Roundoff Error is the error that is introduced as a result of this rounding. We have two types of rounding:
Chop: The base-B expansion is truncated after the (t-1)st digit, so fl(x) is the next number closest to zero from x.
Round: Rounds to the closest number whose last digit stored is even
<u>Machine Precision</u>
We can estimate the accuracy of a floating point system by something called **machine epsilon**
For **chopping:** $e_{mach} = B^{1-t}$
For **rounding:** $e_{mach} = ½ B^{1-t}$
The machine epsilon is important because it defines the largest possible **relative error** for representing x in a given FPN System:
$|(fl(x) – x) / x| < e_{mach}$
Or, we can also use this to say that $fl(1+e_{mach}) > 1$ (which makes sense, the FPR of a FPN + e_{mach} must be greater than 1)
So we have, for all practical FPN Systems:
$0 < UFL < e_{mach} < OFL$
**Underflow is equal to the number of digits in exponent field, Machine Epsilon is equal to the number of digits in mantissa field!
<u>Subnormals and Gradual Underflow</u>
Gradual Underflow is defined as the behavior of having un-normalized numbers, that extend the range of representable numbers (e.g. from 1.00 as smallest to 0.10)
<u>Floating Point Arithmetic</u>
For two numbers to be added, the exponents must match. If they do not, mantissa digits are shifted until they do match. We will lose some trailing digits at the end of the mantissa field.
For two numbers to be multiplied, the exponents are added, and mantissas multiplied, but then we have up to 2*t* digits, which we'll lose precision again.
Overflow is often a problem because there is no good estimation for humungous numbers, whereas zero is good for small numbers, thus for a lot of programs overflow aborts due to fatal errors whereas underflow gets silently set to zero
The reason we use backward analysis is so that we can use real arithmetic (because forward error analysis has FPN errors)
<u>Cancellation</u>
Calculating a small quantity as a difference of two large quantities is not very good, because roundoff error will account for most of the result. This is called **catastrophic cancellation**

Chapter 2 – Systems of Linear Equations

In the Matrix-Vector notation, a system of linear algebraic equations has the form of **Ax = b**
Upper triangular matrix: A matrix with only zero entries below the main diagonal. Solvable by Back- Substitution
Lower triangular matrix: A matrix with only zero entries above the main diagonal. Solvable by Forward-Substitution
<u>Permutation Matrix</u>
A matrix that has the same properties of an identity matrix with several rows interchanged
<u>Elementary Elimination Matrix</u>
The nth elementary elimination matrix has the nth column contain $m_i = a_i/a_k$ where i=k+1,...n (so basically, imagine this)
The divisor a_k is called the **pivot**
We construct it by having the kth column in the NxN identity matrix be replaced with $-m_i$ entries, like such:
Example 2.5 Elementary Elimination Matrices. If $a = [2 \quad 4 \quad -2]^T$, then

$$M_1 a = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix}, \quad \text{and} \quad M_2 a = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 0 \end{bmatrix}.$$

We can represent L_k as M_k^{-1}, which is just M_k with all of its multipliers inverted.
Products of Gaussian transforms is their union.
<u>Gaussian Elimination and LU Factorization</u>
We use M_i to eliminate the ith entries below i on the matrix A until the end. (e.g. M_1 eliminates all column entries below $M_{1,1}$ in A)
At the end, we now have the equation of $MAx = M_{n-1}M_{n-2}...M_1Ax = M_{n-1}M_{n-2}...M_1b = Mb$
We can now create the LU factorization, which would be represented as:
$L = M^{-1} = (M_{n-1}M_{n-2}...M_1)^{-1}$
$U = MA$
Therefore, we have expressed A = LU as our product.
We can then solve these equations as Ly = b (forward substitution) and Ux = y (backwards substitution)

Pivoting

If we have a row where the value is zero, we can simply row interchange with another row below it that has a non-zero pivot and continue. Otherwise, we can just have $M_k = I$ and continue onwards to the next row.

However, if we have $M_k = I$, the back-substitution will fail during the computation of Ux = y, and A is singular.

- ➔ What happens when we have an entry sufficiently small that floating-point arithmetic recognizes it as a 0?
  - We use partial pivoting, which is to say that we choose the largest magnitude on or below diagonal as our pivot
  - Essential for a numerically stable implementation of Gaussian elimination

We need to use **partial pivoting**, which uses permutation matrices:

$M = M_{n-1}P_{n-1}\ldots M_1P_1$

Instead of A = LU, we now have **PA = LU**

So now we solve Ly = Pb then we use Ux = y

Larger pivots produce smaller multipliers, and thus smaller errors

L by partial-pivoting is no longer lower triangular, but is still triangular in the general sense. We can use additional permutation matrices to permute A and L to receive a Lower-Triangular L.

Diagonally Dominant ➔ Each diagonal entry is larger in magnitude than sum of magnitudes of the other entries in its column
In this case, we do not need to partially pivot

Complexity

The LU factorization of a Gaussian Elimination process takes about $n^3/3$ floating point multiplications and additions, which we can write as

LU Factorization Work = $2N^3/3 + O(N^2)$ Flops

Forward + Back Solve = $2N^2 + O(N)$ Flops

Gauss Jordan elimination ➔ Like Gaussian elimination, except both above and below are 0s. Same $M_i = a_i/a_k$, i = 1…n, 50% expensiver

Norms and Condition Numbers

Vector Norms

$||x||_p$ = (summation of all elements in x to the p^{th} power)$^{1/p}$

$\|\|x+y\|\| <= \|\|x\|\| + \|\|y\|\|$	**Triangle Inequality**
$\|\|x=y\|\| >= \|\|x\|\| - \|\|y\|\|$	**Triangle Variation**

Matrix Norms

$||A|| = \max (||Ax||)/||x||$ (i.e. $||A||_1$ is the max absolute column sum, $||A||_{inf}$ is the max absolute row sum)

Condition Number of Matrices

$\text{Cond}(A) = ||A|| \cdot ||A^{-1}||$

The **condition number** of a matrix is how close a matrix is to being singular – larger = closer, smaller (to 1) is further from singularity

The computation of the condition number is very very expensive, so we usually just estimate it

We know that if z is the solution vector to Az = y, then we have

Residuals of Solutions

The residual of solution x^ to the NxN linear system Ax = b is defined as:

r = b – Ax^

We also have A slightly perturbed, so we have:

$$(\boldsymbol{A} + \boldsymbol{E})\hat{\boldsymbol{x}} = \boldsymbol{b},$$

then

$$||\boldsymbol{r}|| = ||\boldsymbol{b} - \boldsymbol{A}\hat{\boldsymbol{x}}|| = ||\boldsymbol{E}\hat{\boldsymbol{x}}|| \leq ||\boldsymbol{E}|| \cdot ||\hat{\boldsymbol{x}}||,$$

so that we have the inequality

$$\frac{||\boldsymbol{r}||}{||\boldsymbol{A}|| \cdot ||\hat{\boldsymbol{x}}||} \leq \frac{||\boldsymbol{E}||}{||\boldsymbol{A}||}$$

This relates the relative residual to the relative change in the matrix. Large relative residual implies large backwards error

SMALL RESIDUALS DO NOT IMPLY SMALL ERRORS (consider large condition numbers!)

We also have:

$$\frac{||\Delta\boldsymbol{x}||}{||\boldsymbol{x}||} \leq \text{cond}(\boldsymbol{A})\frac{||\Delta\boldsymbol{b}||}{||\boldsymbol{b}||}.$$

we can also rearrange it to get

$$\frac{||\Delta\boldsymbol{x}||}{||\hat{\boldsymbol{x}}||} \leq \text{cond}(\boldsymbol{A})\frac{||\boldsymbol{E}||}{||\boldsymbol{A}||}.$$

Thus we have the condition number relating the change in solution given relative change in RHS vector

Jacobi Iterative Method

Gauss-Seidel Iterative Method

Cholesky Factorization of A

Chapter 3 – Non-Linear Equations

Existence and Solutions to Non-linear equations

Often much harder to find and calculate uniqueness for.

Conditioning of Roots

Often **multiple roots** are ill conditioned. This is because for the case of a root being multiple, we would have the function almost parallel to the x-plane at times, possibly having >1 solutions for along that line.

Convergence Rates of Iterative Methods

$$\lim_{k\to\infty} \frac{\|e_{k+1}\|}{\|e_k\|^r} = C$$

If we have $r = 1$ then the convergence rate is **linear**

If we have $r > 1$ then the convergence rate is **super linear** If $r = 2$ then **quadratic**

The Bisection Method

Begins with an initial bracket [a, b] and successively reduces its length until the solution has been isolated. To find the roots, we take [a, b] as values that have f(a) and f(b) as differently signed. We take the midpoint between the two as $m = (b – a)/2$

The **number of iterations to reach a tolerance of *tol* is equivalent to $\log_2(b – a / tol)$**

The **Fixed-Point Iteration**

For a function f(x), we set it to zero, and then we set it to x = …, then we set the LHS x into x_{n+1} so that we have

$x^2 – x – 1 \Rightarrow x^2 – x = 1 \Rightarrow x(x – 1) = 1 \Rightarrow x = 1/(x – 1) \Rightarrow x_{n+1} = 1/(x – 1)$

$x_{n+1} = f(x_n)$

We know that a fixed-point iteration converges if $|g'(ROOT)| < 1$, it converges

Newton's Method

We set up the equation so that

$x_{k+1} = x_k – f(x_k)/f'(x_k)$

To study the convergence of Newton's Method, we have $g'(x) = f(x)f''(x)/(f'(x))^2$, when $g'(x^*) = 0$, then Newtons for a simple root is quadratic.

Secant Method

Similar to Newton's method, except we use

$x_{k+1} = x_k – f(x_k)(x_k – x_{k-1} / f(x_k) – f(x_{k-1}))$

The Secant method is normally superlinearly convergent, although requiring two guesses, it only requires one function evaluation, making it more efficient to evaluate than the Newton's Method, usually.

However, the **Secant Method** and the **Newton's Method** must both be started close enough to the root solution.

Chapter 4 – Interpolation

Definition – Interpolation means fitting function to given data exactly. Not approximation!

Basis of functions

Defines a family of functions for interpolating a given set of data. The interpolating polynomial is a linear combination of these

Methods of Interpolation

Given x data points, we can construct functions that interpolate to the x-1 degree.

Monomial Basis

For each pair p = (x, y) we construct a Vandermonde matrix such that it looks like:

(Given that we have 3 pairs of data points, (x_1, y_1), (x_2, y_2), and (x_3, y_3), we get the coefficients in the Ax = b solution to be the x values

$$\begin{bmatrix} x1\text{^}0 & x1\text{^}1 & x1\text{^}2 \\ x2\text{^}0 & x2\text{^}1 & x2\text{^}2 \\ x3\text{^}0 & x3\text{^}1 & x3\text{^}2 \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} y1 \\ y2 \\ y3 \end{bmatrix}$$

Wherein the x_1, x_2, x_3 values that are our solution set are applied as a multiplication to the basis of $[1, x, x^2, x^3…]$, to obtain for a solution set [-1, 5, -4] to be

$p_2(t) = -1 + 5x – 4x^2$

Solving the Vandermonde system requires **$O(n^3)$ work**

High-degree polynomials are often ill-conditioned because they are significantly less distinguishable when plotted on [0, 1]

For most choices of data points, the condition number of Vandermonde matrices grow at least exponentially with n data points.

We can use **Horner's Method** or nested evaluation to re-structure the polynomial

$p_{n-1} = x_1 + x_2t + … + x_nt^{n-1} \Rightarrow p_{n-1} = x_1 + t(x_2 + t(x_3 + t(… (x_{n-1} + x_nt) …)))$

Which only requires *n* additions and *n* multiplications, which is basically **O(n)** times, or *n* add-multiply flops

LaGrange Interpolation
We define the Lagrange interpolation as a having data points (t_1, y_1)… and having the general formula value given by:
$p_{n-1} = y_1 l_1(t) + y_2 l_2(t) + \ldots y_n l_n(t)$
Where $y_{1\ldots n}$ defines the y value of the data point, and $l_{1\ldots n}$ defines the Lagrange function of the data point $t_{1\ldots n}$
We define the $l_i(t_i)$ as being (where n is the degree), having n terms that look like:

$$l_j(t) = \frac{\prod_{k=1, k\neq j}^{n}(t-t_k)}{\prod_{k=1, k\neq j}^{n}(t_j-t_k)}.$$

In general, this means that we have:
1st term = $(x - x_2)(x - x_3)\ldots(x - x_n) / (x_1 - x_2)(x_1 - x_3)\ldots(x_1 - x_n)$
2nd term = $(x - x_1)(x - x_3)\ldots(x - x_n) / (x_2 - x_1)(x_2 - x_3)\ldots(x_2 - x_n)$
Basically, set up a n-long multiplication and division featuring the term of $(x - x_{1\ldots n})/(x_i - x_{1\ldots n})$ but remove the case where i = term # (because that will just be = 1)

Lagrange interpolation makes it easy to determine interpolating polynomial, but much more expensive to evaluate compared to monomial basis representation. Its also more difficult to integrate, differentiate, etc.

Newton Interpolation
We had the previous two methods for when the matrix A was **full (Vandermonde)** or **diagonal (LaGrange)**. Now we have the basis between the two, the **Newton interpolation**. For a set of having data points (t_1, y_1)… We can determine $x_{1\ldots n}$ by a table of div diff, then:
$p_{n-1} = x_1 + x_2(t - t1) + x_3(t - t1)(t - t2) + \ldots + x_n(t - t_1)(t - t_2)\ldots(t - t_n)$
We can use **Horner's method** of nested evaluation to rearrange this to a more efficient evaluation, as follows:
$p_{n-1}(t) = x_1 + (t - t_1)(x_2 + (t - t_2)(x_3 + (t - t_3)\ldots)$

Using the **divided differences,** we can find that the values for x_1, x_2… as stated above in the Newton Interpolation polynomial

If we have derivatives, we can simply use the rule shown in lecture that the →

$$\lim_{\substack{x_{i+j}\to x_i \\ 1\le j\le k}} y[x_{i+k}, x_{i+k-1}, \ldots, x_i] = \frac{y^{(k)}(x_i)}{k!}$$

Method of Undetermined Coefficients for Change-of-Basis
If we want to convert from Lagrange to monomial basis, we can take
Each Lagrange term, and evaluate it by multiplying through (e.g. $3x^2 + 2x + 1$)then converting into a basis $[1, x, x^2]$, being $[1, 2, 3]^T$
Putting each transposed (now column) vector into a Matrix
Setting up the equation [Lagrange matrix][coefficients of Lagrange] = $[x_1, x_2, \ldots x_n]^T$ and solving for the solution set x, the coefficients of the monomial basis
Same thing for Newtons method.

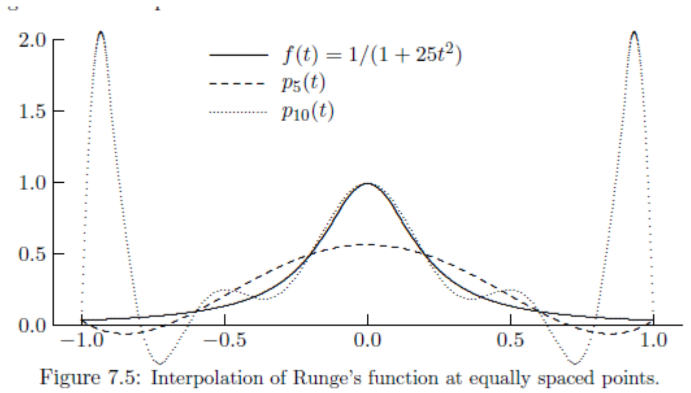

Figure 7.5: Interpolation of Runge's function at equally spaced points.

Runge's Function and Behavior – It may be that higher function degrees have giant variations in interpolation, which means for all intents and purposes, it is useless for interpolating points. We should replace this type of problem with a piecewise function for more efficient and more accurate estimation **(linear splines).** The Runge function being defined as: $\mathbf{f(t) = 1/(1+25t^2)}$

We can choose **Chebyshev points,** to ensure that they are not equally spaced out but bunched near the ends of the intervals. Using the Chebyshev points distributes error more evenly, and makes it much smoother.
However, we can't always choose Chebyshev points because we are usually given pre-existing data points to calculate from.

Piecewise Polynomial Interpolation
We would have a spline interpolation wherein for data points p_1, p_2, p_3 we have the two intervals between the functions represented as $[t_1, t_2]$ and $[t_2, t_3]$. Therefore, we can denote these two polynomials by having:
$p_1 = u_1 + u_2 t + u_3 t^2 + u_4 t^3$
$p_2 = v_1 + v_2 t + v_3 t^2 + v_4 t^3$

Linear Spline Between Several Points
We have the equation as $f(x) = y_0 + (y_1 - y_0)/(x_1 - x_0)\ (x - x_0)$
So we just construct an equation for each interval, e.g. between [a, b] for each a, b that define the piecewise polynomial intervals
Then we set up a piecewise definition that

$$P(x) = \begin{cases} f_1 & \text{if } n_0 < x < m_0 \\ f_2 & \text{if } n_1 < x < m_1 \\ f_3 & \text{if } n_2 < x < m_2 \end{cases}$$

Chapter 5 – Numerical Integration and Differentiation

Numerical Quadrature

We want to compute the quantity of the form,

$$I(f) = \int_a^b f(x)\,dx.$$

We assume that the **general interval of integration to be finite and continuous and smooth**

We seek a **single number** as an answer

We have the formula that

$$I(f) = \int_a^b f(x)\,dx = \sum_{i=1}^{n} w_i f(x_i) + R_n.$$

We understand that the points x_i are called the **nodes**

We understand that the multipliers are called the **weights**

We understand that the R_n value is the **error**

However, we can simply estimate (with removing the error)

$$I(f) \approx \sum_{i=1}^{n} w_i f(x_i),$$

Which is known as the **Quadrature Rule**

Linear Operator

We have a linear operator Q(f) if it satisfies the classic rules of a **linear transformation,** that is:

$Q(\alpha f + g) \Rightarrow \alpha Q(f) + Q(g)$

If Q(f) integrates $[1, x, x^2, \dots x^n]$, then Q(f) integrates all polynomials of degree <= n exactly.

We have the **Newton-Cotes Quadrature Rules**

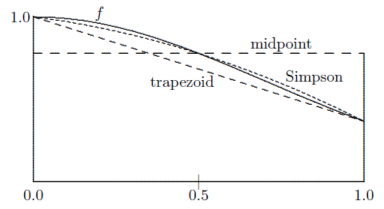

Figure 8.1: Integration of $f(x) = e^{-x^2}$ by Newton-Cotes quadrature rules.

If we have nodes x_i equally spaced in the interval [a, b] we have the following rules:

Midpoint Rule

Interpolating the function at one point, the midpoint of the interval, we have:

$$I(f) \approx M(f) = (b-a) f\left(\frac{a+b}{2}\right)$$

Trapezoid Rule

Interpolating the function at the two endpoints of the interval, we have:

$$I(f) \approx T(f) = \frac{b-a}{2}(f(a) + f(b)).$$

Simpson's Rule

Interpolating the function at three points, one midpoint and two end points, we have:

$$I(f) \approx S(f) = \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

We can prove Simpson's rule by the Method of Undetermined Coefficients, by using the Vandermonde Matrix and Gaussian Elim

Error Estimation

We have the error for the midpoint quadrature rule as very simply:

E_n = I(f) – Q_n(f) wherein this is the difference between the true value and the quadrature rule estimation for the value

We have the approximate error between the **Midpoint Rule (M)** and the **Trapezoid Rule (T)** modelled as:

$E_M = (T - M) / 3$

$E_T = f''(x)/2\ (b - a)^3$

$E_S = 1/90\ (b - a / 2)^5\ |f^4(x)|$ wherein x Is some value in [a, b]

Composite Quadrature Rules

Similar to piecewise interpolation, we should derive piecewise quadrature rules over given intervals.

Composite Midpoint Rule

$$I(f) \approx M(f) = \sum_{i=1}^{n}(x_i - x_{i-1}) f\left(\frac{x_{i-1} + x_i}{2}\right),$$

Composite Trapezoidal Rule

$$I(f) \approx T(f) = \sum_{i=1}^{n}(x_i - x_{i-1}) \frac{f(x_{i-1}) + f(x_i)}{2}.$$