Math 4250/ CS 4210: Numerical Analysis and Differential Equations

Derek Lim

Fall 2019

Instructor: Federico Fuentes

Course Description: Introduction to the fundamentals of numerical analysis: error analysis, approximation, interpolation, numerical integration. In the second half of the course, the above are used to build approximate solvers for ordinary and partial differential equations. Strong emphasis is placed on understanding the advantages, disadvantages, and limits of applicability for all the covered techniques. Computer programming is required to test the theoretical concepts throughout the course.

Textbook: Burden, Faires, and Burden, Numerical Analysis, CENGAGE Learning, 2015 (Edition: 10; ISBN: 978-1-305-25366-7)

Lecture 1: Introduction to Numerical Analysis (8/29/19)

The following is MATLAB output that shows a difference between computer arithmetic and exact arithmetic.

```
>> 170.6 + 6.9

ans =

  177.5000

>> 177.5 - (170.6 + 6.9)

ans =

     0

>> 177.5 - 170.6 - 6.9

ans =

   5.3291e-15
```

The explosion of the Ariane 5 was caused by a numerical software error (see `http://www-users.math.umn.edu/~arnold/disasters/ariane.html`).

> Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger

than 32,767, the largest integer storeable in a 16 bit signed integer, and thus the conversion failed.

Round-off errors and computer arithmetic

Numbers are stored in a finite number of bits by a computer. The amount of numbers that can be stored by a computer is finite. These numbers are called **floating point numbers**. Many computer operations with integers are exact e.g. $1+1=2$. But not many operations with non-integers are exact e.g. $\sqrt{2}^2 \neq 2$. The error produced is called **round-off error**.

Binary floating-point numbers

The IEEE standard as created in 1985, and updated in 2008, is a specific system of representing floating point numbers. For 64-bit double precision,

$$
\begin{aligned}
&b_1 \to s \quad \text{is the sign} \\
&b_2, \ldots, b_{12} \to c = b_2 2^{10} + \ldots + b_{11} 2^0 \quad \text{is the } \textbf{characteristic} \text{ or } \textbf{exponent} \\
&b_{13}, \ldots, b_{64} \to f = b_{13} \frac{1}{2^1} + \ldots + b_{64} \frac{1}{2^{52}} \quad \text{is the } \textbf{mantissa} \text{ or } \textbf{significand} \\
&(-1)^s \cdot 2^{c-1023} \cdot (1+f) \quad \text{is the resulting double precision number}
\end{aligned}
$$

Note that

$$0 \leq f < 1 \implies 1 \leq 1+f < 2$$

$$0 \leq c \leq 2047 \implies -1023 \leq c - 1023 \leq 1024$$

also, $c=0$ and $c=2047$ are special cases, which may be $0, \infty, -\infty$, or `NaN`.

The smallest positive f is called **machine epsilon**. For 64-bit double precision, $\epsilon_{\text{mach}} = 2^{-52} \approx 2.2204 \times 10^{-16}$.

The largest possible allowed number is achieved at $s=0$, $c=2046$, and $f = \left(\frac{1}{2}\right)^2 + \ldots + \left(\frac{1}{2}\right)^{52} = 1 - \left(\frac{1}{2}\right)^{52}$, and is equal to $2^{1023}(1+1-2^{-52}) \approx 1.7977 \times 10^{308}$. Numbers above this produce **overflow**.

The smallest positive number is achieved at $s=0$, $c=1$, and $f=0$, and is equal to $2^{-1022} \approx 2.251 \cdot 10^{-308}$. Values below this produce **underflow**.

Integers are represented similarly. 16-bit integers are represented as $r = b_1 2^{15} + \ldots + b_{16} 2^0$. $2^{16} - 1 - 2^{15} = 32767$ is the largest positive 16-bit integer. This limit lead to the crash of the rocket.

Given $y \in \mathbb{R}$, we denote the floating point representation of y as $fl(y)$.

Definition 0.1. If p^* is an approximation of p, then we define

- $p - p^*$ **actual error**
- $|p - p^*|$ **absolute error**
- $\frac{|p-p^*|}{|p|}$ **relative error**

Definition 0.2. p^* **approximates** p to t significant digits if

$$\frac{|p-p^*|}{|p|} \leq 5 \times 10^{-t}$$

Finite-Digit arithmetic

The operations $+, -, \times, \div$ produce round-off error. Computer addition is denoted $x \oplus y = fl(fl(x) + fl(y))$, and the other operations are defined analogously.

Cancellation errors can occur in computer arithmetic. For instance, when $x \approx y$ and both are approximated to t significant digits, then $x \ominus y$ can have very large relative errors above t signficant digits.

Example 0.1. For $x = \frac{5}{7}$ and $y = .7142857142857141$, the relative error is

$$\frac{|(x-y) - (x \ominus y)|}{|x-y|} \approx .1956$$

the 19% error is very large.

The order of operations also matters. The computer operations are not associative.

Example 0.2. Let $f(x) = x^3 - 6.1x^2 + 3.2x + 1.5$. We want to evaluate $f(4.71)$.

$$\begin{aligned} \text{Naive: } & f_1(x) = x \otimes (x \otimes x) - 6.1 \otimes x \otimes x \ldots \\ \text{Nested: } & f_2(x) = ((x \ominus 6.1) \otimes x \oplus 3.2) \otimes x \oplus 1.5 \end{aligned}$$

The relative error for nested evaluation is an order of magnitude lower.

$$\begin{aligned} \frac{|f(x) - f_1(x)|}{|f(x)|} &\approx 0.05 \\ \frac{|f(x) - f_2(x)|}{|f(x)|} &\approx 0.005 \end{aligned}$$

Example 0.3. Suppose we wish to find the roots of $ax^2 + bx + c$, where $a = c = 1$ and $b \gg 1$. Then $b^2 - 4ac \approx b^2$, so the root $\frac{-b+\sqrt{b^2-4ac}}{2a}$ faces cancellation error in the numerator.

Thus, instead we can look at

$$\begin{aligned} x_1 &= \frac{-b+\sqrt{b^2-4ac}}{2a}\left(\frac{-b-\sqrt{b^2-4ac}}{-b-\sqrt{b^2-4ac}}\right) \\ &= \frac{b^2-(b^2-4ac)}{2a(-b-\sqrt{b^2-4ac})} \\ &= \frac{2c}{-b-\sqrt{b^2-4ac}} \end{aligned}$$

Lecture 2: Interpolation (9/3)

$$\frac{1-\cos^2(x)}{x^2}$$

```
f = @(x) (1-cos(x).^2)./x^2;
x = 1e-7;
format long
fexact = .999999999
f(x)
```

$$\frac{\sin^2(x)}{x^2}$$

```
g = @(x) sin(x).^2/x.^2;
fexact
g(x)
```

$$e^x = \lim_{n\to\infty}(1+1/n)^n$$

```
clc
n = 1e15
exp(1)
e = (1+1/n)^n
```

Dividing 1 by a large number produces large error, which is amplified by raising the result to a large power.

Applying the binomial theorem to $(1+1/n)^n$ provides an accurate result.

```
s = 1;
for k = 1:20
  s = s + prod(1-(1:k)/n)./factorial(k);
end
exp(1)
s
```

Overflow/ underflow error

$$\frac{n!}{(1/2)(3/2)\cdots(n-1/2)}$$

```
n = 1000;
factorial(n)
prod(1/2:n-1/2)
```

```
eval=factorial(n)/prod(1/2:n-1/2)
eval2=prod( (1:n) ./ (1/2:n-1/2))
```

Consider the Hilbert matrix, given by

$$(H_n)_{jk} = \frac{1}{j+k-1} \quad \text{for } 1 \le j,k \le n$$

we want to solve the linear system $(H_n)v = b$ for v.

```
n = 10;
[ii, jj] = meshgrid(1:n);
H = 1./(ii+jj-1);
v = rand(n,1);
b = H*v;
v - H \ b
```

There are at least 4 digits of error in solving for v. The condition number of this matrix is on the order of 10^{13}.

```
cond(H)
```

Interpolation

Consider a function $f \in C[a,b]$. We want to approximate $f(x)$ from a sample $(x_i, f(x_i))_i$ and possibly derivative information $f'(x_i)$.

One natural idea is to use polynomials:

- Easy to compute, differentiate, integrate
- Polynomials can approximate continuous functions arbitrarily well

The second item is formally due to the following theorem

Theorem 1 (Stone-Weierstrass)**.** *Given $f \in C[a,b]$, there exists a sequence of polynomials $p_n \in \mathbb{R}[x]$ such that p_n uniformly converges to f.*

Equivalently, for all $\epsilon > 0$, there exists $p \in \mathbb{R}[x]$ such that $\|f - p\|_\infty < \epsilon$

However, this theorem does not answer an important question—what degree should p be? This depends on f and other factors.

The study of approximating functions with simpler functions is called **approximation theory**.

If we have $f(x_0), f'(x_0), \ldots, f^{(n)}(x_0)$, then we can approximate $f(x)$ by the Taylor polynomial about x_0

$$f(x) \approx \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k \quad \text{near } x_0$$

This is not a good choice for approximating $f(x)$ where x is far from x_0. In fact, in can be an arbitrarily bad choice away from x_0.

The next simplest approach: given a sample of $n+1$ points $(x_0, f(x_0)), \ldots, (x_n, f(x_n))$, we can find a unique polynomial $p_n(x)$ of degree n passing through those points.

The polynomial can be found by solving a linear system of equations.

$$\begin{aligned} f(x_0) = p_n(x_0) &= a_n x_0^n + \ldots + a_1 x_0 + a_0 \\ &\vdots \quad \vdots \\ f(x_n) = p_n(x_n) &= a_n x_n^n + \ldots + a_1 x_n + a_0 \end{aligned}$$

$$\underbrace{\begin{bmatrix} x_0^n & \ldots & x_0 & 1 \\ \vdots & \ddots & \vdots & \vdots \\ x_n^n & \ldots & x_n & 1 \end{bmatrix}}_{V} \underbrace{\begin{bmatrix} a_n \\ \vdots \\ a_0 \end{bmatrix}}_{a} = \underbrace{\begin{bmatrix} f(x_0) \\ \vdots \\ f(x_n) \end{bmatrix}}_{b}$$

It can be shown that V is invertible. Thus, $Va = b$ has unique solution $a = V^{-1}b$. However, it turns out that V is ill-conditioned. Computing $V^{-1}b$ produces large errors due to this ill-conditioning (especially as $n \to \infty$). **Conditioning** is a measure of how much a function changes with small changes in the input.

For a continuous function $g(x)$, the **relative condition number** at x is

$$\text{cond}(g) = \left| \frac{x}{g(x)} \cdot g'(x) \right|$$

For matrices, the relative condition number is

$$\text{cond}(A) = \left\| A^{-1} \right\| \|A\|$$

for symmetric matrices A,

$$\text{cond}(A) = \frac{\lambda_{\max}}{\lambda_{\min}}$$

To see why this is the definition of the condition number note that

$$\frac{g(x+\delta) - g(x)}{g(x)} \Big/ \frac{x+\delta-x}{x} = \frac{x}{g(x)} \cdot \frac{g(x+\delta) - g(x)}{\delta}$$

Lagrange interpolation

Given $x_0, \ldots, x_n$, we wish to construct polynomials of degree n, called **Lagrange polynomials** $L_0, \ldots, L_n$ satisfying

$$L_i(x_j) = \delta_{ij}$$

where δ_{ij} is the Kronecker delta.

We know the exact form of these Lagrange polynomials:

$$\begin{aligned} L_k(x) &= \frac{(x-x_0)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_0)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)} \\ &= \prod_{i \neq k} \frac{x - x_i}{x_k - x_i} \end{aligned}$$

Then given $f \in C[a,b]$, the unique interpolating polynomial of degree n passing through $(x_0, f(x_0)), \ldots, (x_n, f(x_n))$ is given by $p_n(x) = \sum_{k=0}^{n} f(x_k) L_k(x)$.

Lecture 3: Lagrange Polynomial Theory and Computation (9/5)

The following codes compute and plot Lagrange polynomials.

```
n = 4;
xk = linspace(-1, 1, n+1);
% can also choose another nodes like xk = [-.9, -.1, .2, .5]
% or xk = chebpts(n+1,1);
n = length(xk)-1;
x = -1:0.01:1;
y = mylagrange(xk,x);
figure
hold on
plot(xk, zeros(1,n+1), 'ko', xk, ones(1,n+1), 'ko')
for k=1:n+1
  plot(x, y(k,:))
end
hold off
```

The following theorem is a result on how well the function f is approximated by the polynomial interpolant P_n.

Theorem 2. *Let $f \in C^{n+1}[a,b]$, $x_0, \ldots, x_n \in [a,b]$, and P_n is the unique polynomial interpolant of degree n through these points. Then for all $x \in [a,b]$, there exists a $\xi \in [a,b]$ such that*

$$f(x) = P_n(x) + \underbrace{\frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)\cdots(x-x_n)}_{R_n(x)}$$

In particular,

$$\sup_{x\in[a,b]} |f(x) - P_n(x)| \le \sup_{y\in[a,b]} \left| \frac{f^{(n+1)}(\xi(y))}{(n+1)!}(y-x_0)\cdots(y-x_n) \right|$$

To prove this theorem, we recall the following:

Lemma 1 (Rolle Theorem)**.** *For a function $g \in C^1[a,b]$ with $g(a) = g(b) = 0$, there exists $\xi \in (a,b)$ such that $g'(\xi) = 0$. i.e. g must attain a maximum or minimum inside the domain.*

Lemma 2 (Generalized Rolle's Theorem)**.** *For a function $g \in C^{n+1}[a,b]$ that vanishes at $n+2$ points $x_{-1}, x_0, \ldots, x_n \in [a,b]$, then there exists a $\xi \in [a,b]$ such that $g^{(n+1)}(\xi) = 0$.*

Proof of theorem. If $x = x_k$ for some k, then $f(x) = f(x_k) = P_n(x_k) = P_n(x)$, and $R_n(x_k) = 0$, so the claim holds.

If $x \neq x_k$ for any k, define a new function

$$g(t) = f(t) - P_n(t) - \Big(f(x) - P_n(x)\Big) \cdot \prod_{i=0}^{n} \frac{t - x_i}{x - x_i}$$

Note that $g \in C^{n+1}[a,b]$. Moreover, $g(x_k) = f(x_k) - P_n(x_k) = 0$ for each k. Also, $g(x) = 0$. Using the generalized Rolle's theorem, there exists a $\xi \in [a,b]$ such that $g^{(n+1)}(\xi) = 0$. Thus, we have that

$$0 = g^{(n+1)}(\xi) = f^{(n+1)}(\xi) - \Big(f(x) - P_n(x)\Big)\frac{(n+1)!}{(x-x_0)\cdots(x-x_n)}$$
$$f(x) = P_n(x) + \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)\cdots(x-x_n)$$

□

We can also derive that

$$\begin{aligned}
\|f - P_n\|_\infty &\leq \sup_{y\in[a,b]}\left|\frac{f^{(n+1)}(\xi(y))}{(n+1)!}(y-x_0)\cdots(y-x_n)\right| \\
&\leq \sup_{y\in[a,b]}\left|\frac{f^{(n+1)}(\xi(y))}{(n+1)!}\right| \sup_{z\in[a,b]}\left|(z-x_0)\cdots(z-x_n)\right| \\
&= \frac{\left\|f^{(n+1)}\right\|_\infty}{(n+1)!} \sup_{z\in[a,b]}\left|(z-x_0)\cdots(z-x_n)\right|
\end{aligned}$$

Thus, the approximation can be expected to be better if the $(n+1)$th derivative is small and the sample points are distributed "well" in $[a,b]$. However, equally-spaced nodes may lead to instabilities, especially as $n \to \infty$.

Computing the Lagrange interpolation

Recall the formula for the polynomial interpolant:

$$P_n(x) = \sum_{k=0}^{n} f(x_k)L_k(x) \qquad L_k(x) = \prod_{i\neq k}\frac{x-x_i}{x_k-x_i}$$

Evaluating the polynomial interpolant at a given x naively in this manner requires $O(n^2)$ flops since each of the $n+1$ Lagrange polynomials requires $O(n)$ flops. Also, updating with a new node $(x_{n+1}, f(x_{n+1}))$ requires a new computation from scratch.

To compute this in a better way, Newton's divided differences have been used for a while. Now, the gold standard is Barycentric Lagrange Interpolation. We rewrite

$$\begin{aligned}
L_k(x) &= \prod_{i\neq k}\frac{x-x_i}{x_k-x_i}\cdot\frac{x-x_k}{x-x_k} \\
&= \frac{\prod_{i=0}^{n} x-x_i}{(x-x_k)\prod_{i\neq k}(x_k-x_i)} \\
&= L(x)\frac{w_k}{x-x_k}
\end{aligned}$$

where we define

$$L(x) = \prod_{i=0}^{n} x-x_i \qquad w_k = \prod_{i\neq k}\frac{1}{x_k-x_i}$$

therefore we have an equivalent form of the interpolant, called the **Barycentric interpolation formula in its first form**

$$P_n(x) = L(x)\sum_{k=0}^{n}\frac{w_k}{x-x_k}f(x_k)$$

This requires $O(n^2)$ flops to compute the w_k, but this only has to be done once as a preprocessing step. Then $O(n)$ flops are required to evaluate the polynomial at each x. Also, it only takes $O(n)$ flops to update with a new node. Additionally, the w_k do not depend on f. Thus, it does not take extra preprocessing to evaluate the interpolant of a different function f. Also, this does not depends on the ordering of the nodes.

However, this formula can be numerically unstable near the x_k. This issue can be fixed, using the **Barycentric formula of the second form**.

$$\begin{aligned} 1 &= \sum_{k=0}^{n} L_k(x) = L(x)\sum_{k=0}^{n}\frac{w_k}{x-x_k} \\ P_n(x) &= \frac{L(x)\sum_{k=0}^{n}\frac{w_k}{x-x_k}f(x_k)}{L(x)\sum_{k=0}^{n}\frac{w_k}{x-x_k}} \\ &= \frac{\sum_{k=0}^{n}\frac{w_k}{x-x_k}f(x_k)}{\sum_{k=0}^{n}\frac{w_k}{x-x_k}} \end{aligned}$$

although it may seem like there is cancellation error, the division cancels out the cancellation errors.

Lecture 4: Osculating Polynomials (9/10)

```
n=10;
xk=linspace(-1,1,n+1);
%xk = chebpts(n+1,1); % stable!
x=-1:0.001:1;
y=mylagrange(xk,x);
```

This gives Lagrange polynomials which take large values. The spacing between the nodes is uniform, which gives issues. Also, for interpolating along the function $\frac{1}{1+25x^2}$, we see Runge's phenomenon, where there is severe oscillation of the interpolating polynomial along the edges of the interval of approximation. Chebyshev nodes work much better.

Hermite interpolation

We will again have a function $f:[a,b]\to\mathbb{R}$ that we wish to interpolate. Now, we also assume we know the values of the derivative f', along with the value of f, at sample points $x_0,\ldots,x_n$. Our problem is to find the polynomial of minimum degree that both agrees with f and shares the same derivative as f for all points $x_0,\ldots,x_n$. This polynomial is called the **Hermite interpolant**. It is one example of an osculating polynomial.

Definition 0.3. Let $x_0,\ldots,x_n$ be distinct points in $[a,b]$. For each $i=0,\ldots,n$, let $m_i\geq 0$ be an integer. Suppose $f\in C^m[a,b]$, where $m=\max_i(m_i)$. The **osculating polynomial** $p(x)$ approximating f is the

polynomial of minimal degree satisfying that

$$p^{(k)}(x_i) = f^{(k)}(x_i) \quad \text{for } i = 0, \ldots, n,\ k = 0, \ldots, m_i$$

the degree is at most $M = (\sum_{i=0}^{n} m_i) + n$.

Note that when $m_i = 0$ for each i, the osculating polynomial is just regular polynomial interpolation. When $m_i = 1$ for each i, the osculating polynomial is the Hermite interpolant.

Theorem 3. *Let $f \in C^1[a,b]$ and $x_0, \ldots, x_n \in [a,b]$ be distinct points. The Hermite interpolant is of degree at most $2n+1$, and is given exactly by:*

$$Q_{2n+1}(x) = \sum_{k=0}^{n} f(x_k) H_k(x) + \sum_{k=0}^{n} f'(x_k) \hat{H}_k(x)$$

$$H_k(x) = \Big(1 - 2(x - x_k) L_k'(x_k)\Big) L_k^2(x)$$

$$\hat{H}_k(x) = (x - x_k) L_k^2(x)$$

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

Moreover, if $f \in C^{2n+2}[a,b]$, then for each $x \in [a,b]$,

$$f(x) = Q_{2n+1}(x) + \frac{f^{2n+2}(\xi(x))}{(2n+2)!}(x - x_0)^2 \cdots (x - x_n)^2 \quad \textit{for some } \xi(x) \in [a,b]$$

And thus the error is bounded as such:

$$\|f - Q_{2n+1}\|_\infty \leq \sup_{y \in [a,b]} \left| \frac{f^{(2n+2)}(y)}{(2n+2)!}(y - x_0)^2 \cdots (y - x_n)^2 \right|$$

Newton divided differences

Evaluating Hermite interpolant by hand is tedious. There is an easy way to evaluate them using divided differences. This is another way of interpolating polynomials.

For a function f and sample points $x_0, \ldots, x_n$, we wish to find an interpolating polynomial p_n. We rewrite $p_n(x)$ as $p_n(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \ldots + c_n(x - x_0) \cdots (x - x_{n-1})$. This is just rewriting the polynomial in a basis different from the power basis. Here, the basis is [1, $(x - x_0)$, $\ldots$, $(x - x_0) \cdots (x -$

$x_{n-1})]$. Then we have that

$$\begin{aligned}
f(x_0) &= p_n(x_0) \\
&= c_0 \\
f(x_1) &= p_n(x_1) \\
&= c_0 + c_1(x_1 - x_0) \\
&= f(x_0) + c_1(x_1 - x_0) \\
\implies c_1 &= \frac{f(x_1) - f(x_0)}{x_1 - x_0} \\
f(x_2) &= c_0 + c_1(x_1 - x_0) + c_2(x_2 - x_0)(x_2 - x_1) \\
&= f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1) \\
&= f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x_2 - x_1 + x_1 - x_0) + c_2(x_2 - x_0)(x_2 - x_1) \\
&= f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x_1 - x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x_2 - x_1) + c_2(x_2 - x_0)(x_2 - x_1) \\
&= f(x_1) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x_2 - x_1) + c_2(x_2 - x_0)(x_2 - x_1) \\
\implies c_2 &= \frac{\frac{f(x_2) - f(x_1)}{x_2 - x_1} - \frac{f(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_0}
\end{aligned}$$

We define $f[x_i, \ldots, x_{i+k}]$, the **kth divided difference**, recursively as follows:

$$\begin{aligned}
f[x_i] &= f(x_i) \\
f[x_i, x_{i+1}] &= \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} \\
f[x_i, x_{i+1}, x_{i+2}] &= \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i} \\
f[x_i, \ldots, x_{i+k}] &= \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}
\end{aligned}$$

To reconstruct the polynomial from these, use Newton's interpolation formula for $p_n(x)$

$$\begin{aligned}
&f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) \ldots + f[x_0, \ldots, x_n](x - x_0) \cdots (x - x_{n-1}) \\
&= f[x_0] + \sum_{k=1}^{n} f[x_0, \ldots, x_k] \prod_{j=0}^{k-1} (x - x_j)
\end{aligned}$$

This can be written in nested form for more efficient computation, but we will not do this. Now, we discuss how to compute the polynomials.

We use a Newton's tableau of divided differences

Lecture 5: Divided Differences, Cubic Splines (9/12)

Note that we only use the top row of the Newton tableau to compute the polynomial.

When updating a Newton tableau with a new node, we can simply add a new row, then the tableau can be updated in $O(n)$ operations, where n is the number of nodes. Note that all divided differences are independent of x, so it takes $O(n^2)$ flops to compute the tableau. Then it takes $O(n)$ flops to evaluate at each point. Thus, the divided difference method is more efficient than Lagrange interpolation. However, the coefficients in the table depend on the function. If the function changes, then the table must be recomputed. Also, the table depends on the ordering of nodes, which may lead to unstable computations. Barycentric Lagrange interpolation polynomials do not face these last two weaknesses.

For Hermite interpolation, there is an easy modification. Say we have nodes $x_0, \ldots, x_n$, and the function values $f(x_i)$ and $f'(x_i)$ at these nodes. Define $z_0, z_1, \ldots z_{2n}, z_{2n+1}$ as $z_{2j} = z_{2j+1} = x_j$ for $j = 0, \ldots, n$. Define the special divided difference $f[z_{2j}, z_{2j+1}] = f'(x_j)$. Note that the special divided difference is defined when the normal divided difference is undefined due to division by 0. Then the final Hermite polynomial is

$$Q_{2n+1}(x) = f[z_0] + \sum_{k=1}^{2n+1} f[z_0, \ldots, z_k](x - z_0) \cdots (x - z_{k-1})$$

This method also works to find the osculating polynomial when we have derivative information at some subset of the nodes.

Cubic spline interpolation

Issue with single polynomial interpolation: The choice of nodes may lead to erratic or large oscillations (like in the Runge phenomenon).

Benefits: Smooth, easy to compute, backed by theory.

We have seen many benefits and some issues with single polynomial interpolation. Another type of interpolation is piecewise linear interpolation. However, the interpolating function S is generally not differentiable at the nodes. Also, the derivative will be a step function, so $S \in C^0[a,b]$ but $S \notin C^1[a,b]$. The piecewise linear interpolation is still less prone to oscillations.

Piecewise quadratic interpolation is possible to guarantee continuous derivatives of the interpolating function. However, the end nodes are edge cases. Thus, piecewise cubic interpolation is often used.

For nodes $x_0, \ldots, x_n$, define S_0 to be the interpolating function between x_0 and x_1, S_1 to be the interpolating function between x_1 and x_2, and so on. We require the interpolating function to satisfy the following:

$$\begin{aligned}
&(a)\ S(x) \text{ restricted to } [x_j, x_{j+1}] \text{ is a cubic polynomial } S_j(x) \\
&(b)\ S_j(x_j) = f(x_j), \quad S_j(x_{j+1}) = f(x_{j+1}) \\
&(c)\ S_j(x_{j+1}) = S_{j+1}(x_{j+1}) \\
&(d)\ S_j'(x_{j+1}) = S_{j+1}'(x_{j+1}) \\
&(e)\ S_j''(x_{j+1}) = S_{j+1}''(x_{j+1})
\end{aligned}$$

The last three conditions are called **compatibility conditions**. These conditions are not enough to get a unique spline. Any one of the following **boundary conditions** is sufficient to guarantee uniqueness:

- $(f)(i)\ S''(x_0) = S''(x_n) = 0$ called a natural or **free spline**.

- $(f)(ii)$ $S'(x_0) = f'(x_0)$, $S'(x_n) = f'(x_n)$ called a **clamped spline**.

The natural or free spline is used most often, for instance in the MATLAB implementation.

Construction of cubic splines

We use this following ansatz for $S_j(x)$:

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3$$

We have n S_j's, each with a_j, b_j, c_j, d_j unknown, so we have $4n$ constants to determine. This ansatz is convenient, because $f(x_j) = S_j(x_j) = a_j$. Thus, we already know n of the constants. Using (c) with $h_j := x_{j+1} - x_j$,

$$a_{j+1} = S_{j+1}(x_{j+1}) = S_j(x_{j+1}) = a_j + b_j h_j + c_j h_j^2 + d_j h_j^3$$

using (d), we obtain

$$b_{j+1} = b_j + 2c_j h_j + 3d_j h_j^2$$

Lecture 6: Cubic Splines, Bezier Curves (9/17)

Comparing Lagrange, Hermite, and cubic splines on Runge's function with equally spaced nodes, we see that the Lagrange and Hermite interpolants both oscillate, while the cubic spline matches the curve pretty well. With not too many Chebyshev nodes, the Hermite interpolant matches the function very well, while the Lagrange interpolant still somewhat oscillates.

Note that the US Treasury yield curve is computed daily using cubic spline models! See treasury webpage here.

Construction of cubic splines (cont.)

After several manipulations, we get that

$$d_j = \frac{1}{3h_j}(c_{j+1} - c_j)$$
$$b_j = \frac{1}{h_j}(a_{j+1} - a_j) - \frac{h_j}{3}(2c_j + c_{j+1})$$

Finally,

$$h_{j-1}c_{j-1} + 2(h_{j-1} + h_j)c_j + h_j c_{j+1} = \frac{3}{h_j}(a_{j+1} - a_j) - \frac{3}{h_{j-1}}(a_j - a_{j-1}) \quad j = 1, \ldots, n-2$$

Note that we are missing $j = 0$, $j = n-1$. Adding one of the boundary conditions gives sufficient data to solve the system.

Theorem 4 (Natural splines)**.** *Let f be defined at $a = x_0 < x_1 < \ldots < x_n = b$. Then there is a unique natural spline $S \in C^2[a,b]$ interpolating f such that $S''(a) = S''(b) = 0$.*

Proof. The boundary conditions imply that $c_n := \frac{S''(b)}{2} = 0$, and $0 = S''(a) = 2c_0 + 6d_0(x_0 - x_0) = 2c_0$, so $c_0 = 0$. This gives $n+1$ equations and $n+1$ unknowns

$$\begin{bmatrix} 1 & & & & \\ h_0 & 2(h_0+h_1) & h_1 & & \\ & h_1 & 2(h_1+h_2) & h_2 & \\ & & \ddots & \ddots & \\ & & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ & & & & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{3}{h_1}(a_2 - a_1) - \frac{3}{h_0}(a_1 - a_0) \\ \vdots \\ \frac{3}{h_{n-1}}(a_n - a_{n-1}) - \frac{3}{h_{n-2}}(a_{n-1} - a_{n-2}) \\ 0 \end{bmatrix}$$

$$A_f c = b_f$$

A_f is strictly diagonally dominant, hence nonsingular, and the linear system has a unique solution. □

Theorem 5. *Let f be defined at $a = x_0 < x_1 < \ldots < x_n = b$. Then there is a unique clamped spline $S \in C^2[a,b]$ interpolating f such that $S''(a) = S''(b) = 0$.*

Proof. The proof is much the same, except the first and last rows of the linear system will be different. □

Theorem 6. *If $f \in C^4[a,b]$, $M = \left\| f^{(4)} \right\|_\infty$, and S is the unique clamped spline, then*

$$\|f - S\|_\infty \leq \frac{5M}{584} \cdot \max_{0 \leq j \leq n-1} \underbrace{(x_{j+1} - x_j)^4}_{h_j}$$

Clamped splines typically perform better than free splines, but require more information about the function.

Parametric curvers/ Bezier curves

These curves are very important in computer graphics. If we have a curve that is not a well defined function of one variable, we cannot approximate the curve by directly applying the previously covered approaches. Say we have a parametric curve $(x(t), y(t))$ with a parameter t, and nodes $(x(t_0), y(t_0))$, …, $(x(t_n), y(t_n))$. Then we can interpolate a polynomial on each set of nodes.

$$P_n^x(t) = \sum_{k=0}^{n} x(t_k) L_k(t)$$

$$P_n^y(t) = \sum_{k=0}^{n} y(t_k) L_k(t)$$

This forms an interpolating parametric curve $(P_n^x(t), P_n^y(t))$. Likewise, we can form a spline approximation.

Some issues with these approaches include the fact that an interpolating curve must be completely recomputed if one node is moved. In graphics applications, this is often unacceptable, so approximations depending on local data are often used and thus updates are based on local changes.

We seek the lowest order polynomial between two points $v_0 = (x_0, y_0)$ and $v_1 = (x_1, y_1)$ where the derivatives of the function are known. This is a Hermite interpolant (for each coordinate), where $n = 1$, so it is a cubic. Say we have **control points** $c_0 = (x_0 + \alpha_0, y_0 + \beta_0)$ and $c_1 = (x_1 - \alpha_1, y_1 - \beta_1)$. Thus, we have

$$\frac{dy}{dx}\,|_{t_0} = \frac{\beta_0}{\alpha_0} = \frac{3\beta_0}{3\alpha_0}$$

$$\begin{array}{cccc} x(0) = x_0 & x(1) = x_1 & x'(0) = 3\alpha_0 & x'(1) = 3x_1 \\ y(0) = y_0 & y(1) = y_1 & y'(0) = 3\beta_0 & y'(1) = 3y_1 \end{array}$$

That Hermite interpolant also coincides with the simplest clamped spline. After manipulation, the resulting curve is

$$v(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} (1-t)^3 x_0 + 3(1-t)^2 t(x_0 + \alpha_0) + 3(1-t)^2 t(x_1 - \alpha_1) + t^3 x_1 \\ (1-t)^3 y_0 + 3(1-t)^2 t(y_0 + \beta_0) + 3(1-t)^2 t(y_1 - \beta_1) + t^3 y_1 \end{bmatrix}$$

or, equivalently,

$$v(t) = (1-t)^3 v_0 + 3(1-t)^2 t c_0 + 3(1-t) t^2 c_1 + t^3 v_1$$

$$\text{where } v_0 = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \qquad c_0 = \begin{bmatrix} x_0 + \alpha_0 \\ y_0 + \beta_0 \end{bmatrix} \qquad c_1 = \begin{bmatrix} x_0 - \alpha_1 \\ y_0 - \beta_1 \end{bmatrix} \qquad v_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

This is called the **Bezier curve** between (x_0, y_0) and (x_1, y_1).

Note that this can be extended to higher dimensional points in the obvious way. Other curves used in computer graphics include B-splines, NURBS, and T-splines.

Lecture 7: September 19

Approximation Theory

The goal is to find simple functions that approximate a given function. We want the best way to "fit" the data. Say we have data $(x_1, y_1), \ldots, (x_m, y_m)$.

A basic example is discrete least squares approximation. Exact interpolation by polynomials for many noisy data points is likely not useful. In discrete least squares approximation, we find the best fit line $a_0 + a_1 x$. We can measure the error $e_i = |y_i - (a_0 + a_1 x_i)|$ for a single data point. There are several approaches to measure global error of the approximation:

- Best approximation in l_∞ norm.

  $$E_\infty(a_0, a_1) = \max_{1 \le i \le m} e_i = \max_{1 \le i \le m} |y_i - (a_0 + a_1) x_i|$$

  We want to find $(a_0, a_1) \subseteq \mathbb{R}^2$ that minimizes E_∞. The resulting minimax optimization problem is

  $$\min_{(a_0, a_1) \in \mathbb{R}^2} \max_{1 \le i \le m} |y_i - (a_0 + a_1 x_i)|$$

  which cannot be handled with elementary techniques.

  Note that this error is very sensitive to outliers.

- Best approximation in l_1 norm.

$$E_1(a_0, a_1) = \sum_{i=1}^{m} |y_i - (a_0 + a_1)x_i|$$

Minimizing E_1 could lead to trouble because the absolute value is not differentiable at 0. This error could place too little weight on data that is far from the trend.

- Best approximation in l_2 norm.

$$\tilde{E}_2(a_0, a_1) = \sqrt{\sum_{i=1}^{n} (y_i - (a_0 + a_1)x_i)^2}$$

minimizing this is equivalent to minimizing

$$E_2(a_0, a_1) = \sum_{i=1}^{n} (y_i - (a_0 + a_1)x_i)^2$$

This function is differentiable, which is very convenient theoreticallly. For minimization, we find roots of the partial derivatives.

$$\begin{aligned} 0 = \partial_{a_0} E &= -\sum_{i=1}^{m} 2(y_i - (a_0 + a_1)x_i) \\ &= -2\Big[\sum_{i=1}^{m} y_i - a_1 \sum_{i=1}^{m} x_i - a_0 \sum_{i=1}^{m} 1\Big] \\ 0 = \partial_{a_1} E &= \sum_{i=1}^{m} 2(y_i - (a_0 + a_1 x_i))(-x_i) \\ &= -2\Big[\sum_{i=1}^{m} y_i x_i - a_1 \sum_{i=1}^{m} x_i^2 - a_0 \sum_{i=1}^{m} x_i\Big] \end{aligned}$$

Thus, we have the **normal equations**

$$\begin{aligned} a_0 m + a_1 \sum_{i=1}^{m} x_i &= \sum_{i=1}^{m} y_i \\ a_0 \sum_{i=1}^{m} x_i + a_1 \sum_{i=1}^{m} x_i^2 &= \sum_{i=1}^{m} y_i x_i \end{aligned}$$

This is a linear system of equations with only two unknowns that can be solved for explicitly.

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \end{bmatrix}$$

Now, we consider approximation with more complicated functions, such as higher degree polynomials. In the problem of polynomial least squares, we wish to find $(a_0, \ldots, a_n)$ such that $p_n(x) = a_0 + a_1 x + \ldots + a_n x^n$ is the best least squares approximation to the data $(x_1, y_1), \ldots, (x_m, y_m)$ with $n + 1 < m$. Note that if $n + 1 \geq m$, then exact interpolation would be possible.

$$\begin{aligned}
E(a_0,\dots,a_n) &= \sum_{i=1}^{m}(y_i - p_n(x_i))^2 \\
&= \sum_{i=1}^{m} y_i^2 - 2\sum_{i=1}^{m} y_i p_n(x_i) + \sum_{i=1}^{m} p_n(x_i)^2 \\
&= \sum_{i=1}^{m} y_i^2 - 2\sum_{i=1}^{m}(\sum_{j=0}^{n} a_j x_i^j) y_i + \sum_{i=1}^{m}(\sum_{j=0}^{n} a_j x_i^j)^2 \\
&= \sum_{i=1}^{m} y_i^2 - 2\sum_{j=0}^{n} a_j (\sum_{i=1}^{m} x_i^j) y_i + \sum_{j=0}^{n}\sum_{k=0}^{n} a_j a_k \sum_{i=1}^{m} x_i^{j+k}
\end{aligned}$$

Now, we compute the roots of the partial derivatives $\partial_{a_j} E = 0$ for $j = 0,\dots,n$

$$0 = \partial_{a_j} E = -2\sum_{i=1}^{m} x_i^j y_i + 2\sum_{k=0}^{n} a_k \sum_{i=1}^{m} x_i^{j+k}$$

then we rearrange them to get the $n+1$ **normal equations**

$$\sum_{k=0}^{n} a_k \underbrace{\sum_{i=1}^{m} x_i^{j+k}}_{c_{jk}} = \underbrace{\sum_{i=1}^{m} y_i x_i^j}_{b_j} \quad j = 0,\dots,n$$

so we have a linear system of the form

$$\begin{bmatrix} c_{00} & \dots & c_{0n} \\ \vdots & \ddots & \vdots \\ c_{n0} & \dots & c_{nn} \end{bmatrix} \begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} b_0 \\ \vdots \\ b_n \end{bmatrix}$$

We can also consider other ansatzes. For instance, we can consider a best least squares approximation to $y = be^{ax}$, with error $E(a,b) = \sum_{i=1}^{m}(y_i - be^{ax_i})^2$. In general, other ansatzes give nonlinear systems of equations, which may not have a unique solution or may pose other complications such as existence of many local minima which are far from the global minimum.

Note that $y = be^{ax} \implies \ln(y) = \ln(b) + ax$. This gives a linear least squares problem. However, this may give a different solution than that of properly solving the original problem.

Lecture 8: September 24

Orthogonal polynomials and least squares approximation of functions

Previously, we wished to compute an approximation to a finite number of points $(x_1,y_1),\dots,(x_m,y_m)$. Now, say we have a function $f \in C[a,b]$, and we wish to approximate it over the entire domain.

With discrete data, we minimize l_2 error $E_2(a_0,\dots,a_n)=\sum_{i=1}^m (y_i-p_n(x))^2$. We can also write this as

$$\begin{aligned} E &= \sum_{i=1}^m (y_i-p_n(x_i))(y_i-p_n(x_i)) \\ &= (y-p_n(x))\cdot(y-p_n(x)) \\ &= \Big\langle y-p_n(x), y-p_n(x)\Big\rangle \end{aligned}$$

We have already used the L^∞ norm $\|f-p\|_\infty = \max_{a\le x\le b}|f(x)-p(x)|$. Another choice is the L^1 norm, given by $\|f-p\|_1=\int_a^b|f(x)-p(x)|\,dx$. We will use the L^2 norm,
$\|f-p\|_2=\sqrt{\int_a^b|f(x)-p(x)|^2\,dx}$.

L^2 error (squared) is given by

$$E=E(a_0,\dots,a_n)=\Big\langle f-p, f-p\Big\rangle=\int_a^b (f(x)-p(x))^2\,dx$$

where the L^2 inner product is $\langle f,g\rangle=\int_a^b f(x)g(x)\,dx$.

Now, we consider how to actually find p. First, rewrite E.

$$\begin{aligned} E &= \int_a^b (f(x)-p(x))^2\,dx \\ &= \int_a^b f(x)^2\,dx - 2\sum_{k=0}^n a_k\int_a^b x^k f(x)\,dx + \int_a^b \Big(\sum_{k=0}^n a_k x^k\Big)^2\,dx \end{aligned}$$

Since we want to minimize E, we differentiate with respect to the coefficients.

$$\partial_{a_j}E=-2\int_a^b x^j f(x)\,dx+2\sum_{k=0}^n a_k\int_a^b x^{j+k}\,dx$$

This leads to the normal equations:

$$\sum_{k=0}^n a_k \underbrace{\int_a^b x^{j+k}\,dx}_{A_{jk}} = \underbrace{\int_a^b x^j f(x)\,dx}_{b_j}, \qquad j=0,\dots,n$$

$$A\begin{bmatrix} a_0 \\ \vdots \\ a_n\end{bmatrix} = b$$

Now we rewrite the normal equations

$$\sum_{k=0}^n a_k\langle x^j,x^k\rangle=\langle x^j,f\rangle \qquad j=0,\dots,n$$

this is analogous to the normal equations for discrete data, with the difference that the inner product is now the L^2 inner product instead of the dot product. Note that

$$A_{jk}=\int_a^b x^{j+k}\,dx=\frac{b^{j+k+1}-a^{j+k+1}}{j+k+1}$$

A is a Hilbert matrix, which is notoriously ill-conditioned and difficult to invert. Instead of computing p in this way, we use orthogonal polynomials.

Theorem 7. *Suppose ϕ_j is a polynomial of order j for $j = 0, \ldots, n$. Then $\{\phi_0, \ldots, \phi_n\}$ is linearly independent.*

Proof. Suppose $p(x) = c_0\phi_0(x) + \ldots + c_n\phi_n(x) = 0$. Rewrite $p(x) = a_n x^n + \ldots + a_1 x + a_0 = 0$. This means that $a_i = 0$ for $i = 0, \ldots, n$. The only degree n term is in $c_n\phi_n(x)$. So $c_n = 0$ (otherwise $a_n \neq 0$). Proceed recursively. □

Theorem 8. $\dim \mathcal{P}_n = n + 1$

Orthogonal functions

Definition 0.4. A **weight function** is an integrable function $w(x)$ on (a, b) such that $w(x) > 0$ (except on some set of points of measure zero).

For example, $w(x) = \frac{1}{\sqrt{1-x^2}}$ is a weight function on $(-1, 1)$. This is putting more weight by the endpoints of the intervals.

The weighted L^2 inner product (or L^2_W inner product), is

$$\langle f, g\rangle_W = \int_a^b f(x)g(x)w(x)\,dx$$

the usual L^2 inner product occurs at $w = 1$.

Definition 0.5. $\{\phi_0, \ldots, \phi_n\} \subseteq C[a, b]$ is L^2_W **orthogonal** in $[a, b]$ if

$$\langle \phi_j, \phi_k\rangle_W = \int_a^b \phi_j(x)\phi_k(x)w(x)\,dx = \delta_{jk}\alpha_j$$

where $\alpha_j = \langle \phi_j, \phi_j\rangle_W$. If all the $\alpha_j = 1$, the set is called L^2_W **orthonormal**.

Theorem 9. *If $\{\phi_0, \ldots, \phi_n\}$ are L^2_W orthogonal, then the (weighted) least-squares best approximation to a function $f \in C[a, b]$ is*

$$p(x) = \sum_{j=0}^{n} a_j\phi_j(x)$$

$$\begin{aligned} a_j &= \frac{\langle \phi_j, f\rangle}{\langle \phi_j, \phi_j\rangle} \\ &= \frac{1}{\alpha_j}\int_a^b \phi_j(x)f(x)w(x)\,dx \end{aligned}$$

Proof. First, rewrite E

$$\begin{aligned} E &= \Big\langle f - \sum_{j=0}^{n} a_j\phi_j, f - \sum_{j=0}^{n} a_j\phi_j \Big\rangle_W \\ &= \Big\langle f, f\Big\rangle_W - 2\sum_{k=0}^{n} a_k\Big\langle \phi_j, f\Big\rangle_W + \sum_{k=0}^{n}\sum_{j=0}^{n} a_j a_k\Big\langle \phi_j, \phi_k\Big\rangle_W \end{aligned}$$

we differentiate with respect to the a_j in the same way as earlier, and obtain the normal equations

$$\sum_{k=0}^{n} a_k \underbrace{\langle \phi_k, \phi_j \rangle_W}_{A_{jk}} = \langle \phi_j, f \rangle_W \qquad j = 0, \ldots, n$$

Here, we have $A_{jk} = \langle \phi_j, \phi_k \rangle_W = \alpha_j \delta_{jk}$ by orthogonality. Thus,

$$A = \begin{bmatrix} \alpha_0 & & \\ & \ddots & \\ & & \alpha_n \end{bmatrix}$$

is a diagonal matrix with $\alpha_j = \int_a^b (\phi_j(x))^2 w(x)\, dx$. Therefore, the normal equations become

$$a_j \alpha_j = \langle \phi_j, f \rangle_W \qquad j = 0, \ldots, n$$

$$a_j = \frac{\langle \phi_j, f \rangle_W}{\alpha_j} \qquad j = 0, \ldots, n$$

$$a_j = \frac{\langle \phi_j, f \rangle_W}{\langle \phi_j, \phi_j \rangle_W} \qquad j = 0, \ldots, n$$

and the claim is proven. □

Lecture 9: September 26

Theorem 10. *Let $\tilde{\phi}_0, \ldots, \tilde{\phi}_n \in C[a,b]$ be linearly independent. Then the Gram-Schmidt process in L^2_W results in an orthogonal set with the same span as the original.*

$$\phi_k(x) = \tilde{\phi}_k(x) - \sum_{j=0}^{k-1} \frac{\langle \tilde{\phi}_k, \phi_j \rangle_W}{\langle \phi_j, \phi_j \rangle_W} \phi_j(x)$$

Proof. By strong induction.

$$\begin{aligned} \langle \phi_k, \phi_i \rangle &= \Big\langle \tilde{\phi}_k - \sum_{j=0}^{k-1} \frac{\langle \tilde{\phi}_k, \phi_j \rangle}{\langle \phi_j, \phi_j \rangle} \phi_j, \quad \phi_i \Big\rangle_W \\ &= \langle \tilde{\phi}_k, \phi_i \rangle - \sum_{j=0}^{k-1} \frac{\langle \tilde{\phi}_k, \phi_j \rangle}{\langle \phi_j, \phi_j \rangle} \langle \phi_j, \phi_i \rangle_W \\ &= \langle \tilde{\phi}_k, \phi_i \rangle - \frac{\langle \tilde{\phi}_k, \phi_i \rangle}{\langle \phi_i, \phi_i \rangle} \langle \phi_i, \phi_i \rangle_W \\ &= 0 \end{aligned}$$

□

Example 0.4 (Legendre polynomials)**.** For L^2 and a linearly independent set $\{1, x, \ldots, x^n\}$, Gram-Schmidt results in **Legendre polynomials**. On $[a, b] = [-1, 1]$,

$$\begin{aligned}
\phi_0(x) &= 1 \\
\phi_1(x) &= x - \frac{\langle x, \phi_0 \rangle}{\langle 1, 1 \rangle} \phi_0(x) \\
&= x \\
\phi_2(x) &= x^2 - \frac{\langle x^2, x \rangle}{\langle x, x \rangle} x - \frac{\langle x^2, 1 \rangle}{\langle 1, 1 \rangle} \cdot 1 \\
&= x^2 - \frac{1}{3} \\
\phi_3(x) &= x^3 - \frac{3}{5} x \\
\phi_4(x) &= x^4 - \frac{6}{7} x^2 + \frac{3}{35}
\end{aligned}$$

Example 0.5. When we choose weight e^{-x} and the interval to be $[0, \infty)$, Gram-Schmidt results in **Laguerre polynomials**.

Chebyshev polynomials

The **Chebyshev polynomials of the first type** $\{T_n(x)\}_{n=0}^{\infty}$ on $[-1, 1]$ result from Gram-Schmidt with the weight

$$w(x) = \frac{1}{\sqrt{1 - x^2}}$$

they are given by

$$T_n(x) = \cos(n \arccos(x)) \qquad n \geq 0$$

The Chebyshev polynomials of the second type are

$$U_{n-1}(x) = \frac{1}{n} T_n'(x) \qquad n \geq 1$$

We verify that these are indeed polynomials

$$\begin{aligned}
T_0(x) &= \cos(0) = 1 \\
T_1(x) &= \cos(\arccos(x)) = x
\end{aligned}$$

Now, we use a recurrence relationship for the rest of the polynomials. Define $\theta(x) = \arccos(x)$. Then

$$\begin{aligned}
T_n(x) &= \cos(n\theta(x)) \\
T_{n+1}(x) &= \cos((n+1)\theta(x)) \\
&= \cos(\theta)\cos(n\theta) - \sin(\theta)\sin(n\theta) \\
T_{n-1} &= \cos((n-1)\theta) \\
&= \cos(\theta)\cos(n\theta) + \sin(\theta)\sin(n\theta) \\
T_{n+1} + T_{n-1} &= 2\cos(\theta)\cos(n\theta) \\
&= 2xT_n \\
T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \qquad n \geq 1
\end{aligned}$$

We thus continue computing

$$\begin{aligned}
T_2(x) &= 2xT_1(x) - T_0(x) = 2x^2 - 1 \\
T_3(x) &= 2xT_2(x) - T_1(x) = 2x(2x^2-1) - x \\
&= 4x^3 - 3x \\
T_4(x) &= 8x^4 - 8x^2 + 1
\end{aligned}$$

the recurrence guarantees that all of these T_n are indeed polynomials. Note that T_n has leading coefficient $a_n = 2^{n-1}$ for $n \geq 1$. Now, we check that they are orthogonal

$$\begin{aligned}
\langle T_n, T_m \rangle_w &= \int_{-1}^{1} \frac{T_n(x)T_m(x)}{\sqrt{1-x^2}}\, dx \\
&= \int_{-1}^{1} \frac{\cos(n\theta)\cos(m\theta)}{\sqrt{1-x^2}}\, dx \\
\theta = \arccos(x), \qquad d\theta &= \frac{-1}{\sqrt{1-x^2}} dx \\
&= -\int_{\pi}^{0} \frac{\cos(n\theta)\cos(m\theta)}{\sqrt{1-x^2}} \sqrt{1-x^2}\, d\theta \\
&= \int_{0}^{\pi} \cos(n\theta)\cos(m\theta)\, d\theta \\
&= \int_{0}^{\pi} \frac{1}{2}(\cos((n+m)\theta) + \cos((n-m)\theta)\, d\theta
\end{aligned}$$

For $n \neq m$, we have

$$\begin{aligned}
\langle T_n, T_m \rangle_w &= \Big[\frac{1}{2}\frac{\sin((n+m)\theta)}{n+m}\Big]_0^{\pi} + \Big[\frac{1}{2}\frac{\sin((n-m)\theta)}{n-m}\Big]_0^{\pi} \\
&= 0
\end{aligned}$$

For $n = m \geq 1$, we have

$$\begin{aligned}
\langle T_m, T_m \rangle_w &= \frac{1}{2}\frac{\sin((n+m)\theta)}{n+m}\Big]_0^{\pi} + \frac{1}{2}\int_0^{\pi} 1\, d\theta \\
&= \frac{\pi}{2}
\end{aligned}$$

When $n = m = 0$, we have

$$\langle T_0, T_0 \rangle_w = \int_0^{\pi} \frac{1}{2}(1+1)\, d\theta = \pi$$

Chebyshev polynomials:

1. Are used to optimally select interpolation points to minimize interpolation error (in $\|\cdot\|_\infty$)
2. Show a way of reducing degree of an approximating polynomial with minimal loss of accuracy
3. Represent a close connection with trigonometric functions. For instance, they are used in spectral methods, and can take advantage of many fast algorithms for trigonometric functions (e.g. FFT).

Theorem 11. *The Chebyshev polynomials T_n, $n \in \mathbb{Z}_+$ have n simple zeros in $[-1,1]$ at the points*

$$\overline{x}_k = \cos\Big(\frac{2k-1}{2n}\pi\Big) \qquad k = 1,\dots,n$$

Moreover, $T_n(x)$ attains its absolute extrema at the $n+1$ points

$$\overline{x}'_k = \cos\Big(\frac{k}{n}\pi\Big) \qquad k = 0,\dots,n$$

at which $T_n(\overline{x}'_k) = (-1)^k$.

Proof. Let us manually check that $\overline{x}_k$ are zeros.

$$\begin{aligned} T_n(\overline{x}_k) &= \cos\Big(n\arccos\Big(\cos\Big(\frac{2k-1}{2n}\pi\Big)\Big)\Big) \\ &= \cos\Big(\frac{2k-1}{2}\pi\Big) \\ &= \cos\Big(k\pi - \frac{\pi}{2}\Big) \qquad k = 1,\dots,n \end{aligned}$$

These are n distinct roots of a degree n polynomial, so they must all have multiplicity one.

To find extrema (minima/ maxima), we differentiate.

$$\begin{aligned} T'_n(x) &= \frac{d}{dx}\cos(n\arccos(x)) \\ &= -\sin(n\arccos(x))n\frac{-1}{\sqrt{1-x^2}} \\ &= \frac{n\sin(n\arccos(x))}{\sqrt{1-x^2}} \\ &= nU_{n-1} \end{aligned}$$

We evaluate at each of the points

$$\begin{aligned} T'_n(\overline{x}'_k) &= \frac{n\sin(n\arccos(\cos((k/n)\pi)))}{\sqrt{1-x^2}} \\ &= \frac{n\sin(k\pi)}{\sqrt{1-x^2}} = 0 \qquad k = 1,\dots,n-1 \end{aligned}$$

The endpoints are

$$\overline{x}'_n = -1 = \cos(\pi) \qquad \overline{x}_0 = 1 = \cos(0)$$

Thus, the extrema are

$$T_n(\overline{x}'_k) = \cos(n\arccos(\cos((k/n)\pi))) = \cos(k\pi) = (-1)^k \qquad k = 0,\dots,n$$

□

Note in particular that $\|T_n\|_\infty = 1$.

Lecture 10: 10/1

The **monic Chebyshev polynomials** are given by

$$\tilde{T}_0(x) = 1 \qquad \tilde{T}_n(x) = \frac{1}{2^{n-1}} T_n$$

The recurrence to compute these polynomials is

$$\tilde{T}_2(x) = x\tilde{T}_1(x) - \frac{1}{2}\tilde{T}_0(x)$$

$$\tilde{T}_{n+1}(x) = x\tilde{T}_n(x) - \frac{1}{4}\tilde{T}_{n-1}(x) \quad n \geq 2$$

These polynomials of course have the same zeros as the Chebyshev polynomials, and its derivatives have the same zeros as the derivatives of the Chebyshev polynomials.

$$\begin{aligned} \tilde{T}_n(\overline{x}_k) &= 0 && k = 1, \ldots, n \\ \tilde{T}_n'(\overline{x}_k') &= 0 && k = 1, \ldots, n-1 \\ \tilde{T}_n(\overline{x}_k') &= \frac{(-1)^k}{2^{n-1}} && k = 0, \ldots, n \end{aligned}$$

so that $\left\|\tilde{T}_n\right\|_\infty = \frac{1}{2^{n-1}}$.

Let $\tilde{P}_n$ be the monic polynomials of degree n. Then $\tilde{T}_n$ has a special minimization property.

Theorem 12. *For all $n \in \mathbb{N}$, $\tilde{T}_n$ satsifies*

$$\frac{1}{2^{n-1}} = \left\|\tilde{T}_n\right\|_\infty \leq \|p_n\|$$

for any monic polynomial $p_n \in \tilde{P}_n$. Equality occur if and only if $p_n = \tilde{T}_n$.

Proof. Assume $p_n \in \tilde{P}_n$ has $\|p_n\| \leq \frac{1}{2^{n-1}}$. Define $Q = \tilde{T}_n - p_n$. Both $\tilde{T}_n$ and p_n are monic, so Q has degree at most $n-1$.

Moreover, at $\overline{x}_k' = \cos((k/n)\pi)$, we consider the evaluation of Q. For k odd, $Q(\overline{x}_k') \leq 0$ and for k even, $Q(\overline{x}_k') \geq 0$. Since Q is continuous, the intermediate value theorem implies that there exist roots $\xi_j \in [\overline{x}_j', \overline{x}_{j+1}']$ for $j = 0, \ldots, n-1$. Since Q is degree $n-1$ with n roots, Q is zero. □

Minimization of Lagrange interpolation error

For $f \in C^{n+1}[-1,1]$, recall that the interpolating polynomial P_n satisfies that

$$\|f - P_n\|_\infty \leq \frac{\left\|f^{(n+1)}\right\|_\infty}{(n+1)!} \|w\|_\infty$$

where $w = (x - x_0)\cdots(x - x_n) \in \tilde{P}_{n+1}$. We do not have control over f, but we can control w by choice of nodes. The minimizing $\|w\|_\infty$ (optimal) is

$$\|w\|_\infty = \left\|\tilde{T}_{n+1}\right\|_\infty = \frac{1}{2^n}$$

occurs when $w = \tilde{T}_{n+1}$. Therefore, choosing the nodes to be the roots of $\tilde{T}_n$, $x_k = \cos\left(\frac{2k+1}{2(n+1)}\pi\right)$ for $k = 0, \ldots, n$, will yield that $w = \tilde{T}_n$. Therefore, with this choice of Chebyshev nodes,

$$\|f - P_n\|_\infty \leq \frac{1}{2^n(n+1)!}\left\|f^{(n+1)}\right\|_\infty$$

Now, suppose we wish to approximate a function in another domain $[a,b]$. We simply shift the nodes linearly from $x \in [-1,1]$ to $s \in [a,b]$, where $s = \frac{1}{2}((b-a)x + (a+b))$. The final bound takes the form

$$\|f - P_n\|_\infty \leq \left(\frac{b-a}{2}\right)^{n+1} \frac{1}{2^n(n+1)!}\left\|f^{(n+1)}\right\|_\infty$$

where the sup-norm is taken over $x \in [a,b]$.

Best approximations to other polynomials

Let $Q_{n+1}(x) = a_{n+1}x^{n+1} + \ldots + a_1 x + a_0$. We would like the best approximation by another polynomial $p_n \in \mathcal{P}^n$ that minimizes $\|Q_{n+1} - p_n\|_\infty$.

Theorem 13. *The minimizing polynomial p_n satisfies*

$$\frac{|a_{n+1}|}{2^n} = \|Q_{n+1} - p_n\|_\infty \leq \|Q_{n+1} - q_n\|_\infty \quad \forall q_n \in \mathcal{P}^n$$

and is given by $p_n = Q_{n+1} - a_{n+1}\tilde{T}_{n+1} \in \mathcal{P}^n$.

Proof. For any polynomial p_n of degree at most n,

$$\|Q_{n+1} - p_n\|_\infty = |a_{n+1}|\left\|\frac{1}{a_{n+1}}(Q_{n+1} - p_n)\right\|_\infty$$

and $\frac{1}{a_{n+1}}(Q_{n+1} - p_n) \in \tilde{P}_{n+1}$. Thus, the minimal norm is

$$\frac{|a_{n+1}|}{2^n} = \|(Q_{n+1} - p_n)\|_\infty$$

and occurs when $\frac{1}{a_{n+1}}(Q_{n+1} - p_n) = \tilde{T}_{n+1}$. Rearranging this gives $p_n = Q_{n+1} - a_{n+1}\tilde{T}_{n+1}$. □

The next theorem deals with the best approximation error in the supremum (L^∞) norm.

Theorem 14 (Chebyshev's Equioscillation theorem)**.** *For $f \in C[a,b]$, the polynomial $p_n \in \mathcal{P}^n$ minimizes $\|f - p_n\|_\infty$ over $\mathcal{P}^n$ if and only if there are $n+2$ points $a \leq x_0 < \ldots < x_{n+1} \leq b$ such that*

$$f(x_j) - p_n(x_j) = \pm(-1)^j \|f - p_n\|_\infty$$

In practice, we use minimax algorithms (most commonly Remez algorithm) to approximate.

Lecture 11: 10/3

Note that the Chebyshev points are uniformly spaced in radial coordinates. Whereas equally spaced nodes in x do not work well with interpolation, these do. The Chebyshev points of the second type have a nesting property when doubled; this is taken advantage of in certain algorithms.

Trigonometric polynomial approximation

Up to this point, we have been considering orthogonal families $\phi_0, \phi_1, \ldots, \phi_n$ of polynomials. Now, we consider the family $\phi_0, \ldots, \phi_{2n-1}$ defined by

$$\begin{aligned} \phi_0(x) &= \frac{1}{2} = \frac{1}{2}\cos(0x) \\ \phi_k(x) &= \cos(kx) && k = 1, \ldots, n \\ \phi_{n+k}(x) &= \sin(kx) && k = 1, \ldots, n-1 \end{aligned}$$

for $x \in [-\pi, \pi]$. This family is L^2-orthogonal. In particular,

$$\langle \phi_i, \phi_j \rangle = \int_{-\pi}^{\pi} \phi_i(x)\phi_j(x)\, dx = \pi\delta_{ij}$$

$$\langle \phi_0, \phi_0 \rangle = \frac{\pi}{2}$$

The trigonometric polynomials are $\mathcal{T}_n = \operatorname{span}(\phi_0, \ldots, \phi_{2n-1})$. A general element of $\mathcal{T}_n$ takes the form

$$\begin{aligned} S_n(x) &= \sum_{k=0}^{n} a_k\phi_k(x) + \sum_{k=1}^{n-1} b_k\phi_{n+k}(x) \\ &= \frac{a_0}{2} + a_n\cos(nx) + \sum_{k=1}^{n-1} a_k\cos(kx) + b_k\sin(kx) \end{aligned}$$

Theorem 15. *For $f \in L^2$, the $S_n \in \mathcal{T}_n$ minimizing the L^2 error*

$$E = \|f - S_n\|_{L^2}^2 = \int_{-\pi}^{\pi} \Big(f(x) - S_n(x)\Big)^2\, dx$$

is defined by the coefficients

$$\begin{aligned} a_k &= \frac{\langle f, \phi_k \rangle}{\langle \phi_k, \phi_k \rangle} \\ &= \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos(kx)\, dx \\ b_k &= \frac{\langle f, \phi_{n+k} \rangle}{\langle \phi_{n+k}, \phi_{n+k} \rangle} \\ &= \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin(kx)\, dx \end{aligned}$$

Proof. This follows from a previous theorem and integration. □

The limit of $S_n(x)$ as $n \to \infty$ is called the **Fourier series** (for f periodic in $[-\pi, \pi]$). In this case, $E \to 0$ as $n \to \infty$. In particular, $S_\infty(x) = f(x)$ a.e for $x \in (-\pi, \pi)$. This is also true in $[-\pi, \pi]$ if $f(\pi) = f(-\pi)$.

Discrete trigonometric approximation (and interpolation)

We will consider something similar to the Fourier series but with discrete data. Let $\{(x_j, y_j)\}_{j=0}^{2m-1}$ be data points, where the x_j are equally spaced in $[-\pi, \pi]$ and take the form

$$x_j = -\pi + \Big(\frac{j}{2m}\Big)2\pi = -\pi + \Big(\frac{j}{m}\Big)\pi \qquad j = 0, \dots, 2m-1$$

note that

$$x_0 = 0, \quad x_{m+1} = 1, \quad x_{2m-1} = \pi - \frac{1}{m} < \pi$$

Consider the family of $2n$ vectors

$$\varphi_0(x) = \Big(\frac{\vec{1}}{2}\Big) = \begin{bmatrix} 1/2 \\ \vdots \\ 1/2 \end{bmatrix} \in \mathbb{R}^{2m} \qquad n < m$$

$$\varphi_k(x) = \cos(k\vec{x}) = \begin{bmatrix} \cos(kx_0) \\ \vdots \\ \cos(kx_{2m-1}) \end{bmatrix} \qquad k = 1, \dots, n$$

$$\varphi_{n+k}(x) = \sin(k\vec{x}) = \begin{bmatrix} \sin(kx_0) \\ \vdots \\ \sin(kx_{2m-1}) \end{bmatrix} \qquad k = 1, \dots, n$$

If $n = m$, this becomes interpolation, and we define

$$\varphi_m(\vec{x}) = \frac{1}{2}\cos(m\vec{x}) = \begin{bmatrix} \frac{1}{2}\cos(mx_0) \\ \vdots \\ \frac{1}{2}\cos(mx_{2m-1}) \end{bmatrix} \in \mathbb{R}^{2m}$$

In the continuous case, we were minimizing

$$\|f - S_n\|_{L^2} \qquad S_n(x) = \frac{a_0}{2} + a_n\cos(nx) + \sum_{k=1}^{n-1} a_k\cos(kx) + b_k\sin(kx)$$

in the discrete case, we minimize

$$(y - S_n)\cdot(y - S_n) = \sum_{j=0}^{2m-1} \Big(y_j - (S_n)_j\Big)^2$$

where our ansatz is

$$S_n = a_0\Big(\frac{\vec{1}}{2}\Big) + \sum_{k=1}^{n-1} a_k\cos(kx) + b_k\sin(kx) \quad \text{if } n < m$$

$$S_m = a_0\Big(\frac{\vec{1}}{2}\Big) + a_m\frac{1}{2}\cos(m\vec{x}) + \sum_{k=1}^{n-1} a_k\cos(kx) + b_k\sin(kx) \quad \text{if } n = m$$

Theorem 16. *If $n \leq m$, the discrete least squares approximation / interpolation is given by*

$$a_k = \frac{\varphi_k(x) \cdot y}{\|\varphi_k(x)\|^2} = \frac{1}{m} y \cdot \cos(k\vec{x}) = \frac{1}{m} \sum_{j=0}^{2m-1} y_j \cos(kx_j) \qquad k = 0, \ldots, n$$

$$b_k = \frac{\varphi_{n+k}(x) \cdot y}{\|\varphi_{n+k}(x)\|^2} = \frac{1}{m} y \cdot \sin(k\vec{x}) = \frac{1}{m} \sum_{j=0}^{2m-1} y_j \sin(kx_j) \qquad k = 1, \ldots, n-1$$

Proof. It suffices to prove that $\varphi_j(x) \cdot \varphi_k(x) = m\delta_{jk}$, except for φ_0 (and φ_m when $n = m$), in which case $\|\varphi_0\|_2^2 = m/2$ (and $\|\varphi_m\|_2^2 = m/2$). This amounts to showing that

$$\begin{aligned}
0 &= \left(\frac{\vec{1}}{2}\right) \cdot \cos(k\vec{x}) \\
&= \left(\frac{\vec{1}}{2}\right) \cdot \sin(k\vec{x}) \\
&= \cos(kx) \cdot \cos(lx) && k \neq l \\
&= \sin(kx) \cdot \sin(lx) && k \neq l \\
&= \cos(kx) \cdot \sin(lx) && \forall k, l
\end{aligned}$$

and that

$$m = \cos(kx) \cdot \cos(kx) = \sin(kx) \cdot \sin(kx)$$

we use the following lemma.

Lemma 3. *For the uniformly spaced $x_j = -\pi + \frac{k}{m}\pi, \quad j = 0, \ldots, 2m-1$, we have*

$$\vec{1} \cdot \sin(lx) = \sum_{j=0}^{2m-1} \sin(lx_j) = 0 \qquad l \in \mathbb{Z}$$

$$\vec{1} \cdot \cos(lx) = \sum_{j=0}^{2m-1} \cos(lx_j) = \begin{cases} 0 & l \text{ is not a multiple of } 2m \\ 2m & l \text{ is a multiple of } 2m \end{cases} \qquad l \in \mathbb{Z}$$

Proof of lemma. The trick is to consider $e^{iz} = \cos(z) + i\sin(z)$. Then we have

$$\begin{aligned}
\vec{1}\cdot(\cos(lx)+\sin(lx)) = \vec{1}\cdot(e^{ilx}) &= \sum_{j=0}^{2m-1} e^{ilx_j} \\
&= \sum_{j=0}^{2m-1} e^{il(-\pi+\frac{j}{m}\pi)} \\
&= \sum_{j=0}^{2m-1} e^{-il\pi} e^{il\frac{j}{m}\pi} \\
&= e^{-il\pi} \sum_{j=0}^{2m-1} \left(\underbrace{e^{il\frac{1}{m}\pi}}_{r}\right)^j \\
&= (-1)^l \sum_{j=0}^{2m-1} r^j \\
&= \begin{cases} (-1)^l \frac{1-r^{2m}}{1-r} & r \neq 1 \\ (-1)^l 2m & r = 1 \end{cases}
\end{aligned}$$

when $l \neq 2mk$, then $r = e^{il\frac{1}{m}\pi} \neq 1$ and $r^{2m} = \left(e^{il\frac{1}{m}\pi}\right)^{2m} = e^{i2\pi l} = 1$ thus, we have that

$$\vec{1}\cdot(\cos(lx)+\sin(lx)) = \begin{cases} 0 & l \neq 2mk \\ 2m & l = 2mk \text{ for some } k \in \mathbb{Z} \end{cases}$$

The result follows from separately considering the real and imaginary components. □

With this lemma, along with the equalities

$$\begin{aligned}
\sin(t_1)\sin(t_2) &= \frac{1}{2}(\cos(t_1-t_2) - \cos(t_1+t_2)) \\
\cos(t_1)\cos(t_2) &= \frac{1}{2}(\cos(t_1-t_2) + \cos(t_1+t_2)) \\
\sin(t_1)\cos(t_2) &= \frac{1}{2}(\sin(t_1-t_2) + \sin(t_1+t_2))
\end{aligned}$$

one can show that $\{\cos(kx), \sin(kx)\}_{k=0}^m$ are orthogonal. This is shown in Homework 6. For example,

$$\begin{aligned}
\cos(kx)\cdot\cos(lx) &= \vec{1}\cdot\frac{1}{2}\Big(\cos((k-l)x) + \cos((k+l)x)\Big) \\
&= 0 \qquad\qquad k \neq l, k, l \in \{0, \ldots, m\} \\
\cos(kx)\cdot\cos(kx) &= \vec{1}\cdot\frac{1}{2}\Big(\vec{1} + \cos(2kx)\Big) \\
&= \frac{1}{2}(2m+0) = m
\end{aligned}$$

□

Lecture 12: 10/8

Discrete Fourier Transform (DFT)

Trigonometric interpolation is closely related with the discrete Fourier transform, which is ubiquitously used in signal processing. For data $y_0, \ldots, y_{N-1}$ (generally assumed real-valued for our purposes), we define the DFT $(Y_0, \ldots, Y_{N-1})$ where

$$Y_k = \sum_{j=0}^{N-1} y_j e^{-i2\pi k \frac{j}{N}} \quad k = 0, \ldots, N-1$$

the Y_k are the **frequency components** and are in general complex-valued. In fact, for $k = 0, \ldots, m = N/2$, it holds that

$$a_k = \frac{1}{m}(-1)^k \Re(y_k)$$
$$b_k = -\frac{1}{m}(-1)^k \Im(y_k)$$

the continuous Fourier transform applies to real functions f, and the frequency components are given by

$$\hat{f}(k) = \int_{-\infty}^{\infty} f(t) e^{-i2\pi kt}\, dt$$

note that the $\frac{j}{N}$ in the DFT is analogous to time t integrated over in the Fourier transform. The Fourier series coefficients (for periodic f) are

$$\hat{f}(k) = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) e^{-ikx}\, dx$$

The inverse discrete Fourier transform (IDFT) of frequencies $Y_0, \ldots, N$ is defined

$$y_j = \frac{1}{N} \sum_{k=0}^{N-1} Y_k e^{i2\pi k \frac{j}{N}} \quad j = 0, \ldots, N$$

Computed naively, the DFT costs $O(N^2)$ operations.

Note also that DFT is polynomial evaluation. To see this, note that

$$\begin{aligned} Y_k &= \sum_{j=0}^{N-1} y_j \Big(\underbrace{e^{-i2\pi k \frac{1}{N}}}_{z_k} \Big)^j \\ &= P_{N-1}(z_k) \end{aligned}$$

where P_{N-1} is the polynomial

$$P_{N-1}(z) = \sum_{j=0}^{N-1} y_j z^j$$

More importantly, the DFT is a matrix-vector multiply

$$\begin{aligned} Y_k &= \sum_{j=0}^{N-1} y_j \Big(\underbrace{e^{-i2\pi \frac{1}{N}}}_{\omega_N} \Big)^{kj} \\ &= \sum_{j=0}^{N-1} A_{kj} y_j \qquad \text{where } A_{kj} = \omega_N^{kj} \end{aligned}$$

$$\begin{bmatrix} Y_0 \\ \vdots \\ Y_{N-1} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & \omega_N & \dots & \omega_N^{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_N^{N-1} & \dots & \omega_N^{(N-1)^2} \end{bmatrix}}_{F_N} \begin{bmatrix} y_0 \\ \vdots \\ y_{N-1} \end{bmatrix}$$

Note that F_N is symmetric, complex valued, but dense.

Fast Fourier Tranform (FFT)

Different lists of top 10 algorithms of the 20th century all include the FFT. The FFT computes the DFT in $O(N\log(N))$ operations. The idea is to write F_N as the product of sparser matrices ina recursive way.

Example 0.6. Take $N = 6$, and let $\omega = \omega_6 = e^{-i2\pi\frac{1}{6}}$. ω is the primitive 6th root of unity, $\omega^6 = 1$. Thus, $\omega^{6+l} = \omega^l$ for any l. We have our matrix

$$F_6 = \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & \omega & \dots & \omega^5 \\ \vdots & & \ddots & \\ 1 & \omega^5 & \dots & \omega^{25} \end{bmatrix}$$

For $N = 3$, we have

$$F_3 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega_3 & \omega_3^2 \\ 1 & \omega_3^2 & \omega_3^4 \end{bmatrix}$$

Thus, we can write

$$F_3 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 \\ 1 & \omega^4 & \omega^8 \end{bmatrix}$$

$$
\begin{aligned}
F_6 &= \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 \\
1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 \\
1 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \omega^{10} \\
1 & \omega^3 & \omega^6 & \omega^9 & \omega^{12} & \omega^{15} \\
1 & \omega^4 & \omega^8 & \omega^{12} & \omega^{16} & \omega^{20} \\
1 & \omega^5 & \omega^{10} & \omega^{15} & \omega^{20} & \omega^{25}
\end{bmatrix} \\
&= \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 \\
1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 \\
1 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \omega^{10} \\
1 & \omega^3 & 1 & \omega^3 & 1 & \omega^3 \\
1 & \omega^4 & \omega^2 & \omega^6 & \omega^4 & \omega^8 \\
1 & \omega^5 & \omega^4 & \omega^9 & \omega^8 & \omega^{13}
\end{bmatrix} \\
&= \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 \\
1 & \omega^2 & \omega^4 & \omega & \omega^3 & \omega^5 \\
1 & \omega^4 & \omega^8 & \omega^2 & \omega^6 & \omega^{10} \\
1 & 1 & 1 & \omega^3 & \omega^3 & \omega^3 \\
1 & \omega^2 & \omega^4 & \omega^4 & \omega^6 & \omega^8 \\
1 & \omega^4 & \omega^8 & \omega^5 & \omega^9 & \omega^{13}
\end{bmatrix}
\begin{bmatrix}
1 & & & & & \\
 & & 1 & & & \\
 & & & & 1 & \\
 & 1 & & & & \\
 & & & 1 & & \\
 & & & & & 1
\end{bmatrix} \quad \text{putting odd columns first} \\
&= \begin{bmatrix} F_3 & D_1 F_3 \\ F_3 & D_2 F_3 \end{bmatrix} P_N^{(1)} \qquad D_1 = \begin{bmatrix} 1 & & \\ & \omega & \\ & & \omega^2 \end{bmatrix} \quad D_2 = \begin{bmatrix} \omega^3 & & \\ & \omega^4 & \\ & & \omega^5 \end{bmatrix} = -D_1 \\
&= \underbrace{\begin{bmatrix} I & D_1 \\ I & -D_1 \end{bmatrix}}_{S^{(1)}} \underbrace{\begin{bmatrix} F_3 & 0 \\ 0 & F_3 \end{bmatrix}}_{F_6^{(1)}} P_N^{(1)}
\end{aligned}
$$

Note that this final multiplication is of sparse matrices, and can be done quickly.

If $N = 2^r$, then

$$
Y = F_n y = \begin{bmatrix} I_{N/2} & D_{N/2} \\ I_{N/2} & -D_{N/2} \end{bmatrix} \begin{bmatrix} F_{N/2} & \\ & F_{N/2} \end{bmatrix} P_N^{(1)} y
$$

Applying $P_N^{(1)} y$ takes N multiplications, applying $F_N^{(1)}$ takes $2 \cdot \left(\frac{N}{2}\right)^2 = \frac{N^2}{2}$ multiplications, and applying $S_N^{(1)}$ takes $2N$ multiplications. Overall, there are $3N + N^2/2$ multiplications. By recursion, we have

$$
\begin{aligned}
3N + 2\Big(3(N/2) + 2(N/4)^2\Big) &= 6N = \frac{N^2}{4} \\
6N + 4\Big(3(N/4) + 2(N/8)^2\Big) &= 9N + \frac{N^2}{8} \\
&\vdots \quad \vdots \\
r \text{ steps: } 3rN + \frac{N^2}{2^r} &= 3\log_2(N)N + \frac{N^2}{N} = N + 3\log_2(N)N
\end{aligned}
$$

If $N \neq 2^r$, modifications must be made, but the ideas are similar. There is an analogous algorithm for the IDFT, known as the IFFT.

Lecture 13: 10/10

Numerical Differentiation

We wish to compute the derivative

$$f'(x_0) = \lim_{h\to 0} \frac{f(x_0+h)-f(x_0)}{h}$$

We have an obvious approximation

$$f'(x_0) \approx \frac{f(x_0+h)-f(x_0)}{h} \qquad h \ll 1$$

This approximation is unstable, cancellation and round-off errors hurt it.

More rigorously, for numerical differentiation, we consider stencils, sets of nodes (often uniformly spaced), about x_0. We use Taylor's theorem with $f \in C^2[a,b]$ for $x_0 \in [a,b]$.

$$f(x_0+h) = f(x_0) + f'(x_0)h + \frac{1}{2}f''(\xi)h^2 \qquad \xi \text{ between } x_0 \text{ and } x_0+h$$

This gives that

$$f'(x_0) = \frac{f(x_0+h)-f(x_0)}{h} - \frac{f''(\xi)h}{2}$$

Thus, here we have the **truncation error**

$$\tau = f'(x_0) - \frac{f(x_0+h)-f(x_0)}{h} = \frac{-f''(\xi)h}{2}$$

and is bounded by

$$|\tau| \le \frac{\|f\|_{\infty,[x_0,x_0+h]}}{2}|h|$$

τ is $O(h)$. Here, we say our formula (or approximation) is $O(h)$. Clearly, $h \to 0 \implies \tau \to 0$. This is a good approximation mathematically, but faces issues when comptued on a computer. The approximation is called **forward difference** if the limit is taken from the right, and **backward difference** if the limit is taken from the left, where the approximation is

$$f'(x_0) \approx \frac{f(x_0)-f(x_0-h)}{2}$$

These are called two point formulas.

Three point formulas use a 3 point stencil. They are designed to be $O(h^2)$ (assuming higher regularity of f). If we have a stencil x_0, x_0-h, x_0+h, then we derive the midpoint formula. Assume for now that f is smooth, so

$$\begin{aligned}
f(x_0+h) &= f(x_0) + f'(x_0)h + \frac{h^2}{2}f''(x_0) + \frac{h^3}{6}f'''(x_0) + \dots \\
f(x_0-h) &= f(x_0) - f'(x_0)h + \frac{h^2}{2}f''(x_0) - \frac{h^3}{6}f'''(x_0) + \dots \\
f(x_0+h) - f(x_0-h) &= 2f'(x_0)h + +2\frac{h^3}{6}f'''(x_0) + \dots
\end{aligned}$$

If $f \in C^3[a,b]$, then (with some additional arguments to show that a ξ can be found)

$$f(x_0+h) - f(x_0-h) = 2f'(x_0)h + 2\frac{h^3}{6}f'''(\xi)$$

Rearranging, we get

$$f'(x_0) = \frac{f(x_0+h)-f(x_0-h)}{2h} - \frac{h^2}{6}f''(\xi)$$

So that our truncation error is

$$\tau = \frac{-f'''(\xi)}{6}h^2 = O(h^2)$$

This approximation is called the **3 point midpoint formula** (also called **centered differences**). It requires data on both sides of x_0.

We also have **endpoint formulas**, where the stencil is of the form x_0, x_0+h, x_0+2h. Then we have

$$f(x_0+2h) = f(x_0) + 2hf'(x_0) + \frac{4h^2}{2}f''(x_0) + \frac{8h^3}{6}f'''(x_0) + \dots$$
$$f(x_0+h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f'''(x_0) + \frac{h^3}{6}f'''(x_0) + \dots$$

Again, we want to cancel the quadratic terms. In this case, we have

$$4f(x_0+h) - f(x_0+2h) = 3f(x_0) + 2hf'(x_0) - 2\frac{h^3}{3}f'''(x_0) + \dots$$

Thus, we rearrange to get (again with the arguments for existence of ξ)

$$f'(x_0) = \frac{-3f(x_0)+4f(x_0+h)-f(x_0+2h)}{2h} + \frac{h^3}{3}f'''(\xi)$$

Thus, our **3 point endpoint formula** is

$$f'(x_0) = \frac{-3f(x_0)+4f(x_0+h)-f(x_0+2h)}{2h}$$

so our truncation error

$$\tau = \frac{h^3}{3}f'''(\xi) = O(h^3)$$

note that the centered differences truncation error has a nicer constant, that is half of this truncation error. This is due to the power of having information on both sides of the function.

The ideas behind these derivations extend to 5-point formulas and different stencils. Note the convenience of having uniformly spaced stencils, since the Taylor series at the different points of the stencil align.

Now, suppose we want an approximation of $f''(x_0)$. We want an approximation of this given only evaluations of f on a stencil (no evaluation of f').

$$f(x_0+h) = f(x_0) + f'(x_0)h + f''(x_0)\frac{h^2}{2} + f'''(x_0)\frac{h^3}{6} + f''''(x_0)\frac{h^4}{24} + \dots$$
$$f(x_0-h) = f(x_0) - f'(x_0)h + f''(x_0)\frac{h^2}{2} - f'''(x_0)\frac{h^3}{6} + f''''(x_0)\frac{h^4}{24} + \dots$$
$$f(x_0+h) + f(x_0-h) = 2f(x_0) + 2f''(x_0)\frac{h^2}{2} + f^{(4)}(x_0)\frac{h^4}{24} + \dots$$

$$f''(x_0) = \frac{f(x_0+h)-2f(x_0)+f(x_0-h)}{h^2} - \frac{h^2}{12}f^{(4)}(\xi) \qquad \xi \in [x_0-h, x_0+h]$$

The approximation on the left is called the **second derivative midpoint formula**.

Lecture 14: 10/22

Numerical differentiation by the techniques presented in the last lecture is unstable. This can be seen for instance by computing the relative error in the forward difference formula of $\sin(x)$. As h decreases from 1, the relative error decreases linearly, but then at small enough h the relative error increases as h decreases. Other approximations to the derivative using more points also face this issue, but the relative error increases at a lower rate.

Recall that theoretically, the truncation error τ converges to 0 as $h \to 0$, which suggests convergence of the approximation to $f'(x_0)$. However, this is only truly true in exact arithmetic.

Let $f(x_0) = \overline{f}(x_0) + \epsilon_1$, where $\overline{f}(x_0)$ is the computer representation, and $|\epsilon| < \epsilon_{\text{mach}}$. Likewise, say $f(x_0 + h) = \overline{f}(x_0 + h) + \epsilon_2$. Then we have that

$$f'(x_0) = \frac{\overline{f}(x_0+h) + \epsilon_2 - (\overline{f}(x_0) + \epsilon_1)}{h} + \frac{f''(\xi)}{2}h$$

Note that the approximation as computed on a computer is (we assume the computer operations are exact now for simplicity)

$$\frac{\overline{f}(x_0+h) - \overline{f}(x_0)}{h}$$

The error is then

$$\left| f'(x_0) - \frac{\overline{f}(x_0+h) - \overline{f}(x_0)}{h} \right| \leq \frac{|\epsilon_1| + |\epsilon_2|}{h} + \frac{h}{2}\left|f''(\xi)\right|$$

Thus, the upper bound on the approximation error is

$$\frac{2\epsilon}{h} + \frac{h}{2}\left\|f''\right\|_{\infty,[x_0,x_0+H]} \qquad \text{fixed } H > h$$

This goes to infinity as $h \to 0$. Thus, we have no guarantee that the approximation converges to $f'(x_0)$.

We can minimize the upper bound, by differentiating in h. The minimizing h is

$$h^* = \sqrt{\frac{4\epsilon}{\|f''\|_{\infty,[x_0,x_0+H]}}}$$

Below this h^*, the upper bound grows to infinity. Often times, the optimal h is not possible to compute, due to for instance being unable to compute the sup-norm of f.

Numerical differentiation is unstable, so we generally try to avoid it. Unfortunately, finite difference methods rely on numerical differentiation. Numerical integration does not suffer from these instabilities.

Numerical integration

The most basic way to integrate a function numerically is to use numerical quadrature. The generic approximation is

$$\int_a^b f(x)\,dx \approx \sum_{j=0}^{n} w_j f(x_j)$$

The w_j are **integration weights**, and the x_j are **integration nodes**.

Given a choice of integration nodes $x_0, \ldots, x_n$, we know that $f(x) \approx P_n(x) = \sum_{j=0}^{n} f(x_j) L_j(x)$. It makes sense to have

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \int_a^b P_n(x)\ ;dx \\ &= \sum_{j=0}^{n} f(x_j) \underbrace{\int_a^b L_j(x)\,dx}_{w_j}\end{aligned}$$

Notice that $f = P_n$ as functions if $f \in \mathbb{R}_n[x]$. Thus, the quadrature formula is exact in this case. When $f \notin \mathbb{R}_n[x]$, we use the interpolation error theorem. Say $f \in C^{n+1}[a,b]$. Recall that the theorem states

$$f(x) = \sum_{j=0}^{n} f(x_j) L_j(x) + \frac{f^{(n+1)}(\xi)}{(n_1)!} W(x) \qquad W(x) = \prod_{j=0}^{n} (x - x_j)$$

Then we have that

$$\int_a^b f(x)\,dx = \sum_{j=0}^{n} f(x_j) w_j(x) + \underbrace{\frac{1}{(n+1)!} \int_a^b f^{(n+1)}(\xi(x)) W(x)\,dx}_{E(f)}$$

Choosing equally spaced nodes for $n = 1, 2$ yield the trapezoidal and Simpson's rules, respectively. The trapezoidal variant has $x_0 = a$, $x_1 = b$, and

$$\begin{aligned}\int_a^b f(x)\,dx &= \int_a^b \frac{x-b}{a-b} f(a) + \frac{x-a}{b-a} f(b)\,dx + E(f) \\ &= \frac{(x-b)^2}{2(a-b)} \Big]_a^b f(a) + \frac{(x-a)^2}{2(b-a)} \Big]_a^b f(b) + E(f) \\ &= \underbrace{\frac{-(a-b)}{2}}_{w_0} f(a) + \underbrace{\frac{b-a}{2}}_{w_1} f(b) + E(f) \\ &= \frac{b-a}{2}\Big(f(a) + f(b)\Big) + E(f)\end{aligned}$$

Now, we compute $E(f)$ for the trapezoidal method

$$E(f) = \frac{1}{2} \int_a^b f''(\xi(x))(x-a)(x-b)\,dx$$

Note that $(x-a)(x-b)$ does not change sign in $[a,b]$. Thus, we can apply the mean-value theorem for integrals, to see that

$$\begin{aligned}E(f) &= \frac{1}{2} f''(\eta) \int_a^b (x-a)(x-b)\,dx && \eta \in [a,b] \\ &= \frac{1}{2} f''(\eta) \frac{-(b-a)^3}{6} \\ &= \frac{-h^3}{12} f''(\eta) && h = b-a\end{aligned}$$

Geometrically, the trapezoidal method integrates the trapezoid with two vertical sides at a and b, a side connecting a and b on the x-axis, and a side connecting the points of function evaluation.

Now, we do the same with Simpson's rule. The nodes are $x_0 = a, x_1 = \frac{a+b}{2} = x_0 + h, x_2 = b = x_0 + 2h$, where $h = \frac{b-a}{2}$. This looks like a stencil as used in numerical differentiation.

$$\int_a^b f(x)\,dx = \int_{x_0}^{x_2} \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_0) + L_1(x)f(x_1) + L_2(x)f(x_2)\,dx \quad + \\ \int_{x_0}^{x_2} \frac{(x-x_0)(x-x_1)(x-x_2)}{6} f'''(\xi(x))\,dx$$

After some (messy) computations, we have

$$\int_a^b f(x)\,dx = \frac{h}{3}(f(x_0) + 4f(x_1) + f(x_2)) - \underbrace{\frac{h^5}{90} f^{(4)}(\xi)}_{E(f)}$$

This error $E(f)$ is derived using Taylor expansions. Simpson's rule is the approximation

$$\frac{h}{3}(f(x_0) + 4f(x_1) + f(x_2))$$

Note that the stricter error bound $E(f)$ contains an evaluation of the fourth derivative $f^{(4)}(\xi)$. This means that not only are quadratic f integrated perfectly by this rule, but also cubic f are integrated perfectly, since the fourth derivative vanishes.

Lecture 15: Newton-Cotes and Gaussian Quadrature (10/24)

Closed Newton-Cotes formulas

Say we have $n+1$ equidistant integration nodes in $[a,b]$, including the endpoints, $a = x_0 < x_1 < \ldots < x_n = b$. $x_j = x_0 + jh$, $h = \frac{b-a}{n}$, $j = 0, \ldots, n$. Our quadrature rule is $\int_a^b f(x)\,dx = \sum_{j=0}^n w_j f(x_j)$.

At $n = 1$, we have the trapezoidal rule

$$\int_a^b f(x)\,dx = \frac{h}{2}(f(x_0) + f(x_1)) - \frac{h^3}{12} f''(\xi)$$

is exact for $f \in \mathbb{R}_1[x]$.

For $n = 2$, we have Simpson's rule

$$\int_a^b f(x)\,dx = \frac{h}{3}(f(x_0) + 4f(x_1) + f(x_2)) - \frac{h^5}{90} f^{(4)}(\xi)$$

is exact for $f \in \mathbb{R}_3[x]$.

For $n = 3$, we have Simpson's 3/8 rule

$$\int_a^b f(x)\,dx = \frac{3h}{8}(f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)) - \frac{3h^5}{80} f^{(4)}(\xi)$$

is exact for $f \in \mathbb{R}_3[x]$.

These cases motivate the definition of the **degree of accuracy** or **degree of precision** of a quadrature rule/ formula. This is defined as the largest integer $n \in \mathbb{N}$ such that the formula is exact for $f \in \mathbb{R}_n[x]$.

Theorem 17 (General closed Newton-Cotes)**.**

When n is odd and $f \in C^{n+1}[a,b]$,

$$\int_a^b f(x)\,dx = \sum_{j=0}^{n} w_j f(x_j) + \frac{h^{n+2} f^{(n+1)}(\xi)}{(n+1)!} \int_0^n t(t-1)\cdots(t-n)\,dt$$

When n is even and $f \in C^{n+2}[a,b]$,

$$\int_a^b f(x)\,dx = \sum_{j=0}^{n} w_j f(x_j) + \frac{h^{n+3} f^{(n+2)}(\xi)}{(n+2)!} \int_0^n t^2(t-1)\cdots(t-n)\,dt$$

When n is odd, the degree of precision is n, and when n is even, the degree of precision is $n+1$.

Open Newton-Cotes

Now, as integration nodes, we take $n+1$ equidistant nodes without the endpoints. $x_j = x_0 + jh$, $x_0 = a+h$, $h = \frac{b-a}{n+2}$, for $j = 0, \ldots, n$.

At $n = 0$, we have the midpoint rule

$$\int_a^b f(x)\,dx = 2hf\Big(\frac{a+b}{2}\Big) + \frac{h^3}{2} f''(\xi)$$

In practice, it is possible that $b \gg a$, in which case equidistant nodes may lead to high order interpolants that oscillate wildly (e.g. for the Runge function). This motivates the method of **composite integration**, where $[a,b]$ is broken into subintervals, and the approximate integral over $[a,b]$ is taken as the sum of approximate integrals over the subintervals. Low-order rules are used for each subinterval. For instance, the composite trapezoidal rule is of the form

$$\begin{aligned}
\int_a^b f(x)\,dx &= \sum_{j=0}^{n-1} \int_{x_j}^{x_{j+1}} f(x)\,dx \\
&= \sum_{j=0}^{n-1} \frac{h}{2}\Big(f(x_j) + f(x_{j+1})\Big) \\
&= \frac{h}{2}\Big(f(x_0) + 2\sum_{j=1}^{n-1} f(x_j) + f(x_n)\Big)
\end{aligned}$$

This is the basic idea behind `trapz` in MATLAB (although MATLAB's implementation also has optimizations such as choosing intervals of different lengths). Using the intermediate value theorem, it can be shown that there exist some $\mu \in [a,b]$ such that

$$\int_a^b f(x)\,dx = \frac{h}{2}\Big(f(x_0) + 2\sum_{j=1}^{n-1} f(x_j) + f(x_n)\Big) - \frac{(b-a)h^2}{12} f''(\mu)$$

We have a similar error bound for composite Simpson's rule. Assuming n is even,

$$\int_a^b f(x)\,dx = \frac{h}{3}\Big(f(x_0) + 2\sum_{j=1}^{n/2-1}\Big(f(x_{2j})\Big) + 4\sum_{j=1}^{n/2}\Big(f(x_{2j-1})\Big) + f(x_n)\Big) - \frac{(b-a)h^4}{180} f^{(4)}(\mu)$$

Composite integration rules are numerically stable to round-off error. Say we have $f(x_j) = \overline{f}(x_j) + \epsilon_j$, where $\overline{f}(x_j)$ is a machine representation and all $|\epsilon_j| < \epsilon$ are bounded. Then the total round-off error for the composite trapezoidal rule is

$$\leq \epsilon_T = nh\epsilon = n\frac{(b-a)}{n}\epsilon = (b-a)\epsilon$$

so that as $h \to 0$, the upper bound on the error is constant, and does not blow up. Round-off error does not increase as more points are added, meaning the procedure is stable.

Now, say we have a function that has very particular local behavior, such as when a function is mostly small but has a small neighborhood of oscillation. We can use **adaptive quadrature**, in which we repeat the computation on a given interval with twice as many nodes. By utilizing information about error bounds, one can determine which subintervals need more subdivisions in order to achieve custom overall accuracy.

Efficient algorithms exist that calculate integrals within a specified user tolerance.

Gaussian quadrature

This approach to quadrature attempts to maximize the degree of precision (DOP) of an integration rule. First, there does not exist any quadrature rule $\int_{-1}^{1} f(x)\,dx = \sum_{j=0}^{n} w_j f(x_j)$ with DOP $2n+2$ (this is proven in the homework). Gauss-Legendre quadrature achieves a DOP of $2n+1$, so it is optimal in this sense.

Theorem 18. *Given $x_0, \ldots, x_n$ be the roots of the Legendre polynomial of degree $n+1$, P_{n+1}. Then, the* ***Gauss-Legendre quadrature rule***

$$\int_{-1}^{1} f(x)\,dx = \sum_{j=0}^{n} w_j f(x_j) \qquad w_j = \int_{-1}^{1} L_j(x)\,dx$$

has optimal DOP $2n+1$.

Proof. First consider $g \in \mathbb{R}_n[x]$, so it is equal to its interpolant, meaning the Gauss-Legendre quadrature has DOP at least n, because

$$\int_{-1}^{1} \sum_j L_j(x) g(x_j)\,dx = \int_{-1}^{1} g(x)\,dx$$

Now, consider $f \in \mathbb{R}_{2n+1}[x]$. Divide by $p_{n+1}(x)$, the $n+1$th Legendre polynomial, so $f(x) = q(x)p_{n+1}(x) + r(x)$ for $q(x), r(x) \in \mathbb{R}_n[x]$. Thus, we have

$$\begin{aligned}
\int_{-1}^{1} f(x)\,dx &= \int_{-1}^{1} q(x)p_{n+1}(x) + r(x)\,dx \\
&= \int_{-1}^{1} q(x)p_{n+1}(x)\,dx + \int_{-1}^{1} r(x)\,dx \\
&= 0 + \int_{-1}^{1} r(x)\,dx && \text{orthogonality, } q \in \operatorname{span}(p_0, \ldots, p_n) \\
&= \sum_{j=0}^{n} w_j r(x_j) && r \in \mathbb{R}_n[x]
\end{aligned}$$

Moreover, we have that $p_{n+1}(x_j) = 0$, so $f(x_j) = r(x_j)$. Thus,

$$\int_{-1}^{1} f(x)\,dx = \sum_{j=0}^{n} w_j r(x_j)$$

□

Note that the Gauss-Legendre quadrature rule has the same weights as Newton-Cotes, but has differently spaced nodes.

Lastly, the error bound is very strong

$$\left| \int_a^b f(x)\,dx - \sum_{j=0}^{n} w_j f(x_j) \right| \leq \frac{(b-a)^{2n+1} n!^4}{(2n+1)!(2n)!^3} \left\| f^{(2n)} \right\|_\infty$$

Lecture 16: Clenshaw-Curtis (10/29)

We can use quadrature rules to integrate over any interval $[a,b] \subseteq \mathbb{R}$. However, usually quadrature points and weights are givne in $[-1,1]$ for convenience. To use these weights for integrating over $[a,b]$, we simply use a linear transform on our inputs $t \in [-1,1]$

$$x(t) = \frac{1}{2}\Big((a+b) + (b-a)t\Big) \in [a,b]$$

Then we have $dx = \frac{(b-a)}{2}\,dt$, so our integral is

$$\begin{aligned}
\int_a^b f(x)\,dx &= \int_{-1}^{1} f\Big(x(t)\Big)\,\frac{(b-a)}{2} dt \\
&= \int_{-1}^{1} f\Big(\frac{1}{2}((a+b)+(b-a)t)\Big)\,\frac{(b-a)}{2} dt \\
&\approx \sum_{j=0}^{n} \underbrace{\tilde{w}_j \frac{(b-a)}{2}}_{w_j} f(\underbrace{x(t_j)}_{x_j})
\end{aligned}$$

where $\tilde{w}_j$ are weights in $[-1,1]$, and t_j are integration points in $[-1,1]$.

Clenshaw-Curtis

Usual quadrature assumes that $f(x) \approx \sum_{j=0}^{n} L_j(x) f(x_j)$, meaning that f is closely approximated by its polynomial interpolant. Under this assumption, one can expect

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{j=0}^{n} \Big(\int_{-1}^{1} L_j(x)\,dx \Big) f(x_j)$$

and the formula is exact for $f \in \mathbb{R}_n[x]$. Computing the w_j and x_j generally takes $O(n^2)$.

We rewrite this formula alternatively. Given $x_0, \ldots, x_n$, we want that

$$\int_{-1}^{1} f(x)\,dx = \sum_{j=0}^{n} w_j f(x_j) \qquad \forall f \in \mathbb{R}_n[x]$$

$$\begin{bmatrix} f(x_0) & \dots & f(x_n) \end{bmatrix} \begin{bmatrix} w_0 \\ \vdots \\ w_n \end{bmatrix} = \int_{-1}^{1} f(x)\,dx \qquad \forall f \in \mathbb{R}_n[x]$$

Due to linearity of the integral, it suffices to consider a basis for $\mathbb{R}_n[x]$. For the canonical basis, the requirement is

$$\begin{bmatrix} 1 & \dots & 1 \\ x_0 & \dots & x_n \\ \vdots & & \vdots \\ x_0^n & \dots & x_n^n \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} \int_{-1}^{1} dx \\ \int_{-1}^{1} x\,dx \\ \vdots \\ \int_{-1}^{1} x^n\,dx \end{bmatrix}$$

This transposed Vandermonde system can be solved to get the weights, but this method is numerically unstable, and can also be slower than the direct formula $\int_{-1}^{1} L_j(x)\,dx$.

If we instead use the Chebyshev basis, we get stability for certain choices of points. The system is

$$\begin{bmatrix} T_0(x_0) & \dots & T_0(x_n) \\ T_1(x_0) & \dots & T_1(x_n) \\ \vdots & & \vdots \\ T_n(x_0) & \dots & T_n(x_n) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} \int_{-1}^{1} T_0\,dx \\ \int_{-1}^{1} T_1\,dx \\ \vdots \\ \int_{-1}^{1} T_n\,dx \end{bmatrix}$$

The set of nodes that make this linear system well-conditioned are the Chebyshev points of the first kind i.e. the roots of T_{n+1}. With this choice of nodes and weights, we have **Fejér quadrature**. Here we have $C^T w = b$, so $w = C^{-T} b$ which is simply an IDCT of b. Thus, the weights can be computed in $O(n \log n)$ time. Moreover, x_j can be computed in $O(n)$ time with their exact formulas.

Fejér's second quadrature takes all roots of U_{n+1} (where $U_n = \frac{T'_{n+1}}{n}$). This is the extrema of T_{n+1} besides the endpoints. **Clenshaw-Curtis** quadrature chooses Chebyshev points of the second type i.e. all extrema of T_n including endpoints. The exact form of these are $x_j = \cos\left(\frac{j}{n}\pi\right)$, $j = 0, ldots, n$.

The linear system for Clenshaw-Curtis is $\tilde{C}^T w = b$ where $\tilde{C}^T$ is almost a DCT of type 1, up to a scaling by $D = \begin{bmatrix} 1/2 & & \\ & 1 & \\ & & \ddots \end{bmatrix}$. The system is $D\tilde{C}^T w = Db$. Since DC^T is involutory, $w = DC^T Db$, which can also be computed in $O(n \log n)$.

Clenshaw-Curtis quadrature integration points are nested. This means that when refining a mesh, we can choose a refinement so that the function need not be reevaluated on all of the nodes again. This property is particularly advantageous for adaptive quadrature and higher-dimensional integrals. Integration points and weights are computed in $O(n \log n)$ which of course scales very well for large n. However, the degree of precision is only n. Thus, Gauss-Legendre is twice as good at integrating polynomials. For general functions, sometimes Gauss-Legendre and Clenshaw-Curtis are comparable (e.g. for integrating $|x|$). For analytic functions, Gauss-Legendre is often better for a given n.

Recently (Townsend 2013; Bogaert 2014) showed how to compute w_j and x_j for Gauss-Legendre in $O(n)$ operations (by using asymptotic expansions). In practice, computing the weights and nodes is not done often. For instance, using composite integration, quadrature with low n can be used for each subinterval. Also, integration points and weights are precomputed in $[-1, 1]$ and tabulated.

Finally, we briefly discuss integration in multiple dimensions. We use cubature rules. We have a basic approximation (for integrating over a rectangle in $\mathbb{R}^2$)

$$\begin{aligned}\int_c^d \int_a^b f(x,y)\,dx\,dy &\approx \int_c^d \sum_{j=0}^{n} w_j^{(a,b)} f(x_j,y)\,dy \\ &\approx \sum_{k=0}^{n}\sum_{j=0}^{n} w_k^{(c,d)} w_j^{(a,b)} f(x_j,y_k)\,dy \\ &= \sum_{l=0}^{(n+1)^2-1} w_l f(z_l)\end{aligned}$$

Integrating over non-rectangular regions is more difficult. Sometimes there are direct formulas for certain types of regions. Another way to do this is to discretize a given region into quadrilaterals. To integrate over a general quadrilateral A, use a bilinear transform to map the region into a rectangle.

$$\int_A f(x,y)\,dA = \int_a^b \int_c^d f(u,v)J(u,v)\,du\,dv$$

In a similar way, one can mesh the region by triangles, and then integrate the triangles. Another method is to directly map A by a nonlinear transform into some rectangle. These methods may be more complicated to deal with issues such as singularities at the boundaries.

Lecture 17: (10/31)

Initial value problems

Many phenomena in engineering and science can be modelled by differential equations. We have the following general form for a first order ordinary differential equation

$$\begin{aligned}\frac{dy}{dt} &= f(t,y) \qquad t \in [a,b] \\ y(a) &= \alpha\end{aligned}$$

we often think of the parameter t as time, and $f : \mathbb{R}^2 \to \mathbb{R}$ is a general function that is possibly nonlinear. $y(a) = \alpha$ is called the initial condition. We want to solve for $y(t)$, so our solution is a function of only t, meaning that the equation is a first order ODE. This structure can be generalized for systems of ODEs and high-order ODEs. We want to solve this equation numerically to obtain some approximations

$$w_0 = y(t_0),\ w_1 \approx y(t_1), \ldots,\ w_N \approx y(t_N)$$

at times $a = t_0 < t_1 < \ldots < t_N = b$. If an approximate solution is desired at an intermediate time, we can use some interpolation scheme to evaluate it. The times are often chosen to be equally spaced.

Definition 0.6. A function $f(t,y)$ is **Lipschitz** on y in $D \subseteq \mathbb{R}^2$ if there exists a $L > 0$ such that

$$|f(t,y_2) - f(t,y_1)| \le L|y_2 - y_1| \qquad \forall (t,y_1),(t,y_2) \in D$$

Example 0.7. $f(t,y) = t|y|$ for a domain $D = \{(t,y) \mid t \in [1,2],\ y \in [-3,4]\}$ satisfies

$$\begin{aligned}|f(t,y_2) - f(t,y_1)| &= |t(|y_2| - |y_1|)| \\ &\le t|y_2 - y_1| \\ &\le 2|y_2 - y_1|\end{aligned}$$

Theorem 19. *Let $f(t,y)$ be defined on a convex domain $D \subset \mathbb{R}^2$. If a constant $L > 0$ exists such that*

$$\left|\partial_y f(t,y)\right| \leq L \qquad \forall (t,y) \in D$$

then f is Lipschitz in y with Lipschitz constant L.

Theorem 20. *Let $D = \{(t,y) \mid t \in [a,b],\, y \in \mathbb{R}\} = [a,b] \times \mathbb{R}$, and $f(t,y) \in C^0(D)$ (meaning f is continuous). If f is Lipschitz in y, then the IVP*

$$\begin{aligned} y'(t) &= f(t,y) \qquad t \in [a,b] \\ y(a) &= \alpha \end{aligned}$$

has a unique solution $y(t)$ for all $t \in [a,b]$ (and is well-posed).

Example 0.8. Say we have the problem

$$\begin{aligned} y'(t) &= 1 + t\sin(ty) \qquad 0 \leq t \leq 2 \\ y(0) &= 0 \end{aligned}$$

Then we compute

$$\begin{aligned} \left|\partial_y (1 + t\sin(ty))\right| &= \left|t^2 \cos(ty)\right| \\ &\leq t^2 \\ &\leq 4 \end{aligned}$$

so that f is Lipschitz by our above theorem. This means that the IVP has a unique solution.

Well-posedness of an ODE is often defined in the sense of Hadamard

1. A unique solution exists.
2. Solution depends continuously on initial data.

   This can be taken to mean that the solution satisfies some stability bounds measuring how much the solution is perturbed when the initial data is perturbed.

Definition 0.7. The course text gives a definition—The IVP $y'(t) = f(t,y), \quad t \in [a,b]$ with $y(a) = \alpha$ is well-posed if

1. A solution $y(t)$ for $t \in [a,b]$ exists and is unique.
2. $\exists \epsilon_0 > 0$ and $\exists k > 0$ such that $\forall \epsilon \in (0, \epsilon_0)$ and $\forall \delta(t) \in C^0[a,b]$ with $\|\delta\|_\infty < \epsilon$, and $\forall \delta_0$ such that $|\delta_0| < \epsilon$, the perturbed IVT

   $$\begin{aligned} z'(t) &= f(t,z) + \delta(t) \quad t \in [a,b] \\ z(a) &= \alpha + \delta_0 \end{aligned}$$

   has a unique solution satisfying

   $$\|z - y\|_\infty < k\epsilon$$

   (note that here k is the stability constant and represents the conditioning of the problem.)

Euler's method

Euler's method is the most elementary method to solve initial value problems.

$$\begin{aligned} y'(t) &= f(t,y) \quad t \in [a,b] \\ y(a) &= \alpha \end{aligned}$$

Usually, we take an equally spaced mesh for $t \in [a,b]$.

$$\begin{aligned} t_i &= a + ih \quad i = 0, \ldots, N \\ h &= \frac{b-a}{N} = t_{i+1} - t \end{aligned}$$

h is called the step size. There are two ways of thinking about Euler's method.

$$\begin{aligned} y'(t_i) &\approx \frac{y(t_{i+1}) - y(t_i)}{h} \\ \frac{y(t_{i+1}) - y(t_i)}{h} &\approx f(t_i, y(t_i)) \\ y(t_{i+1}) &\approx y(t_i) + hf(t_i, y(t_i)) \end{aligned}$$

Note that with the initial condition, we have $y(t_0)$, so we have a valid base case. The truncation error is

$$\tau = \frac{y(t_{i+1}) - y(t_i)}{h} - f(t_i, y(t_i))$$

If $|\tau| \to 0$ as $h \to 0$, then the method is said to be **consistent**. In this case, the resulting numerical scheme converges to the original equation. Note that the truncation error measures the error in the numerical scheme, meaning only at the mesh points, and not the error of the resulting approximation of the solution to the DE.

Lecture 18: (11/5)

Here we show the consistency of Euler's method using a Taylor expansion

$$\begin{aligned} y(t_{i+1}) &= y(t_i + h) \\ &= y(t_i) + hy'(t_i) + \frac{h^2}{2} h''(\xi_i) && \xi_i \in [t_i, t_{i+1}] \\ &= y(t_i) + hf(t_i, y(t_i)) + \frac{h^2}{2} h''(\xi_i) \end{aligned}$$

Then we have a truncation error of

$$\begin{aligned} \tau &= \frac{y(t_{i+1}) - y(t_i)}{h} - f(t_i, y(t_i)) \\ &= \frac{h}{2} y''(\xi_i) \\ &= O(h) \end{aligned}$$

assuming that y'' is bounded. This means that $|\tau| \to 0$ as $h \to 0$ under the given conditions. Thus, our explicit Euler's method numerical scheme is

$$\begin{aligned} w_0 &= \alpha \\ w_{i+1} &= w_i + hf(t_i, w_i) \qquad\qquad i = 0, \ldots, N-1 \end{aligned}$$

Such schemes are called time-stepping schemes, and the formula for how to move forward is called the difference equation. We hope that $w_i \approx y(t_i)$ and that $\big|w_i - y(t_i)\big| \to 0$ as $h \to 0$.

Lemma 4. *For $x \geq -1$ and $m > 0$*

$$(1+x)^m \leq e^{mx}$$

Proof. Taylor's theorem with e^x gives

$$\begin{aligned} e^x &= 1 + x + \frac{1}{2}x^2 e^{\xi} \\ e^x &\geq 1 + x \\ e^{mx} &\geq (1+x)^m \end{aligned}$$

□

Lemma 5. *For $s, t > 0$, and $a_0, \ldots, a_k$ with $a_0 > \frac{-t}{s}$, $a_{i+1} \leq (1+s)a_i + t$ for $i = 0, \ldots, k-1$. Then*

$$a_{i+1} \leq e^{(i+1)s}\Big(a_0 + \frac{t}{s}\Big) - \frac{t}{s}$$

Proof. Note that

$$\begin{aligned} a_{i+1} &\leq (1+s)a_i + t \\ &\leq (1+s)\Big((1+s)a_{i-1} + t\Big) + t \\ &\leq (1+s)^{i+1}a_0 + \Big(1 + (1+s) + \ldots + (1+s)^i\Big)t \\ &= (1+s)^{i+1}a_0 + \frac{1-(1+s)^{i+1}}{1-(1+s)}t \\ &= (1+s)^{i+1}a_0 + \frac{(1+s)^{i+1}-1}{s}t \\ &= (1+s)^{i+1}\Big(a_0 + \frac{t}{s}\Big) - \frac{t}{s} \\ &\leq e^{(i+1)s}\Big(a_0 + \frac{t}{s}\Big) - \frac{t}{s} \qquad\qquad \text{previous lemma} \end{aligned}$$

□

Theorem 21. *Let f continuous, Lipschitz on y in the domain $D = [a,b] \times (-\infty, \infty)$. Let y be the unique solution to the IVP*

$$\begin{aligned} y'(t) &= f(t,y) \\ y(a) &= \alpha \end{aligned}$$

Suppose that $y \in C^2[a,b]$ and $\|y''\|_\infty = M < \infty$. Moreover, let $w_0, \ldots, w_N$ be the Euler approximations to $y(t_0), \ldots, y(t_N)$, where the t_j are taken evenly spaced with spacing h. Then

$$|y(t_i) - w_i| \leq \frac{hM}{2L}\Big(e^{L(t_i-a)} - 1\Big) \qquad i = 0, \ldots, N$$

where L is the Lipschitz constant of f.

Proof. There is nothing to show for $i = 0$. For $i > 0$, we have

$$\begin{aligned}
y(t_{i+1}) &= y(t_i) + hf(t_i, y(t_i)) + \frac{h^2}{2}y''(\xi) \\
w_{i+1} &= w_i + hf(t_i, w_i) \\
|y(t_{i+1}) - w_{i+1}| &\leq |y(t_i) - w_i| + h|f(t_i, y(t_i)) - f(t_i, w_i)| + \frac{h^2}{2}\big|y''(\xi_i)\big| \\
&\leq |y(t_i) - w_i| + hL|y(t_i) - w_i| + \frac{h^2}{2}\big|y''(\xi_i)\big| \\
&\leq (1 + \underbrace{hL}_{s})\underbrace{|y(t_i) - w_i|}_{a_i} + \underbrace{\frac{Mh^2}{2}}_{t}
\end{aligned}$$

Now, we apply the lemma to see

$$\begin{aligned}
|y(t_{i+1}) - w_{i+1}| &\leq e^{(i+1)hL}\Big(\frac{h^2M}{2hL}\Big) - \frac{h^2M}{2hL} \\
&\leq \frac{hM}{2L}\Big(e^{(i+1)hL} - 1\Big) \\
&\leq \frac{hM}{2L}\Big(e^{(t_{i+1}-a)L} - 1\Big) \qquad\qquad (i+1)h = t_{i+1}
\end{aligned}$$

□

Thus, as $h \to 0$, we have that $y_{t_i} \to w_i$, since the bound is linear in h. However, this derivation was done in exact arithmetic. We now take round-off error into account. Then we have steps

$$\begin{aligned}
u_0 &= \alpha + \delta_0 \\
u_{i+1} &= u_i + hf(t_i, u_i) + \delta_{i+1}
\end{aligned}$$

where $\delta_i < \delta$ for each i. Then we have the bound

$$|y(t_i) - u_i| \leq \frac{1}{L}\Big(\frac{hM}{2} + \frac{\delta}{h}\Big)\Big(e^{L(t_i-a)} - 1\Big) + \delta\Big(e^{L(t_i+a)}\Big)$$

Note that there is a term $\frac{\delta}{h}$, so as $h \to 0$ the bound explodes.

Higher order approximation

For the same initial value problem, we can look at higher order approximations by using higher order Taylor terms

$$\begin{aligned} y(t_{i+1}) &= y(t_i) + hy'(t_i) + \ldots + \frac{h^n}{n!}y^{(n)}(t_i) + \frac{h^{n+1}}{(n+1)!}y^{(n+1)}(\xi_i) \\ &= y(t_i) + hf(t_i, y_i) + \ldots + \frac{h^n}{n!}f^{(n-1)}(t_i, y_i)(t_i) + \frac{h^{n+1}}{(n+1)!}f^{(n)}(\xi_i, y(\xi_i)) \\ &= y(t_i) + hT^{(n)}(t_i, y_i) + O(h^{n+1}) \end{aligned}$$

$$\frac{y(t_{i+1}) - y(t_i)}{h} = f(t_i, y_i) + \ldots + \frac{h^{n-1}}{n!}f^{(n-1)}(t_i, y_i) + \frac{h^n}{(n+1)!}f^{(n)}(\xi_i, y(\xi_i))$$

This means that

$$\frac{y(t_{I+1}) - y(t_i)}{h} - T^{(n)}(t_i, y_i) = O(h^n)$$

so that a numerical scheme using these higher order terms converges in $O(h^n)$. A high-order Taylor method is of the form

$$\begin{aligned} w_0 &= 0 \\ w_{i+1} &= w_i + hT^{(n)}(t_i, w_i) \qquad\qquad i = 0, \ldots, N-1 \end{aligned}$$

This is Euler's method when $n = 1$. The truncation error is $O(h^n)$ for $y \in C^{n+1}([a,b])$.

However, this requires computation of the derivatives of f, which may be expensive and/or introduce significant round-off errors (recall that numerical differentiation is unstable). Thus, we desire a scheme with a rapidly decaying truncation error that does not require evaluation of derivatives of f. This brings us to Runge-Kutta methods.

Runge-Kutta methods

We make use of higher dimensional Taylor's theorem

Theorem 22 (Taylor)**.** *Suppose $f \in C^{n+1}(D)$ where $D = [a,b] \times [c,d]$. Let $(t_0, y_0) \in D$. Then for all $(t,y) \in D$, there exists some ξ between t_0 and t and μ between y_0 and y such that*

$$f(t,y) = P_n(t,y) + R_n(t,y)$$

$$\begin{aligned} P_n(t,y) = f(t_0,y_0) &+ (t-t_0)\partial_t f() + (y-y_0)\partial_y f() + \\ &\frac{(t-t_0)^2}{2}\partial_{tt} f() + (t-t_0)(y-y_0)\partial_{ty} f() + (y-y_0)^2\partial_{yy} f() + \ldots + \textit{nth derivative terms} \end{aligned}$$

$$R_n(t,y) = \frac{1}{(n+1)!}\sum_{j=0}^{n}\binom{n+1}{j}(t-t_0)^{n-j+1}(y-y_0)^j(\partial_{t^{n-j+1}y^j} f)(\xi,\mu)$$

Lecture 19: Runge-Kutta methods (11/7)

As can be shown through experiments with the simplest IVP $y'(t) = \beta y(t)$, $y(0) = \alpha$, when h is not small enough the computed solution can oscillate. Forward Euler's method is unstable in these cases. We will see that these issues can be alleviated with Runge-Kutta methods and backward Euler.

The idea behind Runge-Kutta methods is to substitute the Taylor polynomial $T^{(n)}(t_i, w_i)$ with something involving only f (not its derivatives), but still maintaining $O(h^n)$ truncation error τ. We first develop order two Runge-Kutta

$$T^{(2)}(t,y) = f(t,y) + \frac{h}{2} f'(t,y)$$

we would like this to be equal to something satisfying the ansatz

$$a_1 f(t+\alpha_1, y+\beta_1) + O(h^2)$$

First, we have

$$\begin{aligned} f'(t,y) &= \frac{d}{dt} f(t, y(t)) \\ &= \partial_t f \partial_t t + \partial_y f \partial_t y && \text{chain rule} \\ &= \partial_t f(t, y(t)) + \partial_y f(t, y(t)) \cdot f(t, y(t)) \end{aligned}$$

so that

$$T^{(2)} = f + \frac{h}{2}\partial_t f + \frac{h}{2}\partial_y f \cdot f$$

Then we Taylor expand

$$\begin{aligned} f(t+\alpha_1, y+\beta_1) &= f(t,y) + \alpha_1 \partial_t f(t,y) + \beta_1 \partial_y f(t,y) + R_1(t+\alpha_1, y+\beta_1) \\ R_1(t+\alpha_1, y+\beta_1) &= \frac{\alpha_1^2}{2}\partial_{tt} f(\xi,\mu) + \alpha_1\beta_1 \partial_{ty} f(\xi,\mu) + \frac{\beta_1^2}{2}\partial_{yy} f(\xi,\mu) \end{aligned}$$

Thus, we can solve for our parameters

$$f + \frac{h}{2}\partial_t f + \frac{h}{2}\partial_y f \cdot f = a_1 f + a_1\alpha_1 \partial_t f + a_1\beta_1 \partial_y f + a_1 R_1$$

$$\begin{aligned} a_1 &= 1 \\ \alpha_1 &= \frac{h}{2} \\ \beta_1 &= \frac{h}{2} \cdot f(t,y) \end{aligned}$$

With these choice of parameters, we can see that R_1 is $O(h^2)$ (as long as f and its derivatives are bounded). This means that

$$T^{(2)}(t,y) = f\Big(t + \frac{h}{2}, y + \frac{h}{2} f(t,y)\Big) + O(h^2)$$

Note that we have to evaluate f twice, and we need not evaluate f'. This resulting method is known as the **midpoint method**, which is of the form

$$\begin{aligned} w_0 &= \alpha \\ w_{i+1} &= w_i + hf\Big(t_i + \frac{h}{2}, w_i + \frac{h}{2} f(t_i, w_i)\Big) \end{aligned}$$

it has truncation error $\tau = O(h^2)$.

Similar ideas can be extended to higher-order Runge-Kutta methods, but the ansatzes are more complicated. For instance,

$$T^{(3)}(t,y) = f\bigg(t + \alpha_1, y + \delta_1 f\Big(t + \alpha_2, y + \delta_2 f(t,y)\Big)\bigg) + O(h^3)$$

Note the nesting of the evaluations. In practice, Butler tableaus are often computed to figure out coefficients.

Now, we show an ansatz for fourth order Runge-Kutta, which is one of the most important and most used methods of this family.

$$\frac{1}{6}\Big(\underbrace{a_1 f(t_i, w_i)}_{k_1} + 2\underbrace{a_2 f(t_i + \alpha_2, w_i + \delta_2 k_1)}_{k2}\Big) + 2\underbrace{a_3 f(t_i + \alpha_3, w_i + \delta_3 k_2)}_{k_3} + \underbrace{a_4 f(t_i + \alpha_4, w_i + \delta_4 k_3)}_{k_4}$$

Make the following choice of parameters:

$$\begin{aligned} a_1 = a_2 = a_3 = a_4 &= h \\ \alpha_2 = \alpha_3 &= \frac{h}{2} \\ \alpha_4 &= h \\ \delta_2 = \delta_3 &= \frac{1}{2} \\ \delta_4 &= 1 \end{aligned}$$

Therefore, we have the fourth order Runge-Kutta method

$$\begin{aligned} w_0 &= \alpha \\ k_1 &= hf(t_i, w_i) \\ k_2 &= hf\Big(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_1\Big) \\ k_3 &= hf\Big(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_2\Big) \\ k_4 &= hf\Big(t_i + h, w_i + k_3\Big) \\ w_{i+1} &= w_i + \frac{1}{6}\Big(k_1 + 2k_2 + 2k_3 + k_4\Big) \end{aligned}$$

The truncation error is $O(h^4)$ for $f \in C^5$. This method is often abbreviated as RK4.

Now, we go back to the question of choosing the step length. It is relatively simple to implement adaptive time stepping in Runge-Kutta methods. Time stepping methods might be used with time steps that are too large (**under resolving**) or too small (**over resolving**).

We consider RK5, then compare with RK4 to get an estimate of the error.

$$\tau_{i+1}(h) = \tilde{\tau}_{i+1}(h) + \frac{1}{h}\Big(\tilde{w}_{i+1} - w_i\Big)$$

where quantities with a tilde are from RK5, and those without are from RK4

$$\tau_{i+1}(h) = O(h^4) \approx Kh^4$$

Let $\alpha > 0$. Then

$$\tau_{i+1}(\alpha h) = K(\alpha h)^4 = \tau_{i+1}(h)$$

For h small, $\tilde{\tau}_{i+1}$ is small, so

$$\tau_{i+1}(h) \approx \frac{1}{h}\Big(\tilde{w}_{i+1} - w_{i+1}\Big)$$

$$\tau_{i+1}(h) \approx \alpha^4 \frac{1}{h}\Big(\tilde{w}_{i+1} - w_{i+1}\Big)$$

Say we want to bound the truncation error in magnitude by ϵ. Then we get

$$\begin{aligned}\left|\alpha^4 \frac{1}{h}(\tilde{w}_{i+1} - w_{i+1})\right| &= |\tau_{i+1}(\alpha h)| \\ &\leq \epsilon \\ \alpha &\leq \left(\frac{\epsilon h}{|\tilde{w}_{i+1} - w_{i+1}|}\right)^{1/4}\end{aligned}$$

More precisely, the procedure is:

1. Solve for $w_{i+1}, \tilde{w}_{i+1}$, compute $\gamma = \left(\frac{\epsilon h}{|\tilde{w}_{i+1} - w_{i+1}|}\right)^{1/4}$
2. If $\frac{1}{h}|\tilde{w}_{i+1} - w_{i+1}| > \epsilon$, repeat computation of w_{i+1} with γh the step ($\gamma h < 1$) $\Big($too big of a step $\Big)$
3. If $\frac{1}{h}|\tilde{w}_{i+1} - w_{i+1}| \leq \epsilon$, accept w_i, but change next time step to γh ($\gamma \geq 1$) $\Big($too small of a step$\Big)$

Note that the naive computation of w_{i+1} and $\tilde{w}_{i+1}$ would take over 8 function evaluations. However, some of the function evaluations can be recycled between the two. There are methods like Runge-Kutta-Fehlberg that reuse the function evaluations, allowing as few as 6 function evaluations total per step. `MATLAB`'s solver is `ode45`.

All of these ideas are applicable to systems of differential equations. A first order system IVP is

$$\frac{d}{dt}\vec{y} = \vec{f}(t, \vec{y}), \quad t \in [a, b] \quad \vec{y}(a) = \vec{\alpha}$$

Meaning that

$$\begin{aligned}\frac{d}{dt}y_1 &= f_1(t, \vec{y}) = f_1(t, y_1, \ldots, y_m) \\ &\vdots \qquad\qquad\qquad \vdots \\ \frac{d}{dt}y_m &= f_m(t, \vec{y}) = f_1(t, y_1, \ldots, y_m) \\ y_1(a) &= \alpha_1 \\ &\vdots \quad \vdots \\ y_m(a) &= \alpha_m\end{aligned}$$

Note that the methods we have learned so far generalize directly to systems of equations. We simply replace iterates with vectors.

For higher order differential equations, we can simply reformulate them as a first order system and solve with the same methods.

$$y^{(m)}(t) = f(t, y, y', \ldots, y^{(m)}), \quad t \in [a, b]$$
$$y(a) = \alpha_1, \ \ldots, \ y^{(m-1)}(a) = \alpha_m$$

The equivalent first order system is, letting $u_j = y^{(j-1)}$,

$$\begin{aligned} \frac{d}{dt}u_1 &= y'(t) = u_2(t) \\ &\vdots \quad \vdots \\ \frac{d}{dt}u_m &= y^{(m)}(t) = f(t, u_1, \ldots, u_m) \end{aligned}$$

Lecture 20: Implicit and Multistep Methods (11/12)

Recall that if h is too large, then explicit/forward Euler diverges as $t \to \infty$. Implicit methods enhance stability. Recall the forward Euler is of the form

$$\begin{aligned} w_0 &= \alpha \\ w_{i+1} &= w_i + hf(t_i, w_i) \end{aligned}$$

Here, w_{i+1} depends explicitly on known quantities $(w_i, t_i, h, f(t_i, w_i))$. In contrast, implicit Euler is of the form

$$\begin{aligned} w_0 &= \alpha \\ w_{i+1} &= w_i + hf(t_{i+1}, w_{i+1}) \end{aligned}$$

we have to solve for w_{i+1}, which depends on f.

Example 0.9. Consider the simple linear IVP

$$y'(t) = -\beta y(t)$$

which has soltuion $y(t) = y(0)e^{-\beta t}$. Then implicit Euler is of the form

$$\begin{aligned} w_0 &= \alpha \\ w_{i+1} &= w_i + h(-\beta w_{i+1}) \end{aligned}$$

solving for w_{i+1}, we have

$$w_{i+1} = \frac{1}{1 + h\beta} w_i$$

That was simple enough since f is linear, but what if $y'(t) = e^{y(t)}$. Then implicit Euler is

$$\begin{aligned} w_0 &= \alpha \\ w_{i+1} &= w_i + he^{w_{i+1}} \end{aligned}$$

this means we have to solve a nonlinear equation at each step. Newton iterations are one method to solve for these. In general implicit methods are stable but expensive.

Implicit methods are also used for systems of ODEs/ higher order ODEs. Linear equations are common, and analogously to our above example, each step would require a linear system solve, which is expensive.

For now, we have only seen one step methods, which are those of the form

$$\begin{aligned} w_0 &= \alpha \\ w_{i+1} &= w_i + h\phi(t_i, w_i, h) \end{aligned}$$

Forward Euler and RK4 are both of this form. Backward Euler is of this form with t_{i+1}, w_{i+1}, h as the parameters to ϕ. RK4 is high-order, but involves function evaluations at times between t_i and t_{i+1}. As always, these function evaluations can be very expensive.

Multi-step methods

Multi-step methods provide high-order truncation errors with function evaluations only at the w_i, $i = 0, \ldots, n$, meaning with no intermediate evaluations. They can be explicit or implicit. The general form for an m step method is

$$\begin{aligned} w_0 = \alpha,\ w_1 = \alpha_1, \ldots,\ w_{m-1} = \alpha_{m-1} \\ w_{i+1} = a_{m-1}w_i + a_{m-2}w_{i-1} + \ldots + a_0 w_{i-m+1} + hF(t_i, h, w_{i+1}, \ldots, w_{i-m+1}) \end{aligned}$$

The method is implicit if w_{i+1} is a parameter to F, and explicit if not. The local truncation error (LTE) is

$$\tau_{i+1}(h) = \frac{y(t_{i+1}) - a_{m-1}y(t_i) - \ldots - a_0 y(t_{i+1-m})}{h} - F\Big(t_i, y(t_{i+1}), \ldots, y(t_{i+1-m})\Big)$$

Here we are taking $i = m-1, \ldots, N-1$, $h = \frac{b-a}{N}$, and $t_i = a + ih$.

One-step methods have $m = 1$ and $a_0 = 1$. For $m \geq 2$, the idea is that coefficients a_i and F come from integration. We want to solve $y'(t) = f(t, y)$. Note that

$$\begin{aligned} y(t_{i+1}) - y(t_i) &= \int_{t_i}^{t_{i+1}} y'(t)\, dt \\ &= \int_{t_i}^{t_{i+1}} f(t, y(t))\, dt \end{aligned}$$

To integrate this, we use $P_m(t)$, the polynomial interpolant of $f(t_i, y(t_i)), \ldots, f(t_{i+1-m}, y(t_{i+1-m}))$. However, we now integrate only over a subinterval of the interval that the nodes reside in, usually of the form $[t_i, t_{i+1}]$. In implicit methods we also include $f(t_{i+1}, y(t_{i+1}))$.

Here we consider Simpson's implicit method. This is of the form

$$\begin{aligned} y(t_{i+1}) - y(t_{i-1}) &\approx \int_{t_{i-1}}^{t_{i+1}} P_n(t)\, dt \\ y(t_{i+1}) &\approx y(t_{i-1}) \int_{t_{i-1}}^{t_{i+1}} P_n(t)\, dt \\ &\approx y(t_{i-1}) + \frac{h}{3}\Big(f(t_{i+1}, y(t_{i+1})) + 4f(t_i, y(t_i)) + f(t_{i+1}, y(t_{i+1}))\Big) \end{aligned}$$

Thus, our method is

$$w_0 = \alpha, w_1 = \alpha_1$$
$$w_{i+1} = w_{i-1} + \frac{h}{3}\Big(f(t_{i+1}, y(t_{i+1})) + 4f(t_i, y(t_i)) + f(t_{i+1}, y(t_{i+1}))\Big)$$

This is an implicit method with $a_0 = 1, a_0 = 0$, and $m = 2$.

There are also a large class of methods known as Adams-Bashforth (explicit) and Adams-Moulton (implicit) methods. For there, we only integrate in $[t_i, t_{i+1}]$, with $a_{m-1} = 1$, $a_j = 0$, $j \neq m-1$. At $m = 4$, we have four-step (explicit) fourth-order Adams-Bashforth

$$w_0 = \alpha,\ w_1 = \alpha_1,\ w_2 = \alpha_2,\ w_3 = \alpha_3$$
$$w_{i+1} = w_i + \frac{h}{24}\Big(55f(t_i, w_i) - 59f(t_{i-1}, w_{i-1}) + 37f(t_{i-2}, w_{i-2}) - 9f(t_{i-3}, w_{i-3})\Big)$$

The truncation error can be computed using the errors from interpolation, and takes the form

$$\tau_{i+1}(h) = \frac{251}{720}y^{(5)}(\mu_i)h^4 \qquad \mu_i \in (t_{i-3}, t_{i+1})$$

At $m = 3$, we have the three-step (implicit) fourth-order Adams-Moulton

$$w_0 = \alpha,\ w_1 = \alpha_1,\ w_2 = \alpha_2$$
$$w_{i+1} = w_i + \frac{h}{24}\Big(9f(t_{i+1}, w_{i+1})) + 19f(t_i, w_i) - 5f(t_{i-1}, w_{i-1}) + f(t_{i-2}, w_{i-2})\Big)$$

The truncation error is of the form

$$\tau_{i+1}(h) = -\frac{19}{720}y^{(5)}(\mu_i)h^4$$

Note that we have a fourth-order method with three-steps.

Predictor-corrector methods

Implicit methods are difficult to use directly with nonlinear f. In practice, they are used, but with w_{i+1} on the right hand side replaced by a predictor of w_{i+1} called w^p_{i+1}. The predictor comes from an explicit method, and the "implicit" method used after (which is no longer implicit by definition) is called a corrector. For instance, a fourth-order predictor-corrector method is given by

1. $w_0 = \alpha$
2. Compute w_1, w_2, w_3 using RK4
3. Compute predictor using (explicit) fourth-order Adam-Bashforth
$$w^p_{i+1} = w_i + \frac{h}{24}(\ldots)$$
4. Use predictor in implicit fourth-order Adams-Moulton
$$w_{i+1} = w_i + \frac{h}{24}\Big(9f(t_{i+1}, w^p_{i+1}) + \ldots\Big)$$

This is generally more stable than an explicit method, while less stable than an implicit method. It does not require intermediate evaluations like Runge-Kutta on all steps would. Note that the predictor method can be iterated (a fixed point iteration) to gain some accuracy, but in practice this often does not add much accuracy.

Lecture 21: Stability of IVP Solvers (11/14)

A multi-step method for an IVP is **consistent** if

$$\lim_{h\to 0} |\tau_{i+1}(h)| = 0 \qquad i = m-1,\ldots,N$$
$$\lim_{h\to 0} |\alpha_i - y(t_i)| = 0 \qquad i = 0,\ldots,m-1$$

the method is **convergent** if

$$\lim_{h\to 0} \max_{0\le i\le N} |w_i - y(t_i)| = 0$$

Stability of the method roughly means that there is some $C > 0$ such that

$$[\text{measurement of output}] \le C \cdot [\text{measurement of input}]$$

A more rigorous definition — a method is **zero-stable** or **Dahlquist-stable** if the response of a numerical method to $f(t_i, y) = 0$ with initial value $y(a) = \alpha = w_0$ remains bounded, meaning that there exists some $C > 0$ such that

$$|w_i| \le C|y(a)| \quad \forall i$$

intuitively, this measures stability to round-off error.

We will assume that F is continuous on $(t, h, w) \in [a, b] \times [0, h_0] \times \mathbb{R}^{n+1} = D_F$ for some $h_0 > 0$. Also, F is Lipschitz in w with $F = 0$ if $f = 0$, and f continuous in $(t, y) \in [a, b] \times \mathbb{R} = D_f$.

Theorem 23. *All explicit one-step methods are*

- *Zero-stable*
- *Convergent if and only if they are consistent*
- *There exists some $\tau(h)$ such that*

$$|\tau_{i+1}(h)| \le \tau(h) \quad \forall i = 0,\ldots,N_1,\ h \in [0, h_0]$$

then

$$|y(t_i) - w_i| \le \frac{\tau(h)}{L} e^{L(t_i - a)}$$

The zero-response characteristic polynomial is $p(z) = z^m - a_{m-1}z^{m-1} - \ldots - a_1 z - a_0$, where the a_k are the coefficients of the multi-step method. $\beta_1 = 1$ is always a root of this polynomial, meaning $1 - a_{m-1} - \ldots - a_0 = 0$. When all roots are simple, it can be shown that

$$w_n = \sum_{j=1}^{m} c_j \beta_j^n$$

is the general solution to

$$w_{i+1} = a_{m-1}w_i + a_{m-2}w_{i-1} + \ldots + a_0 w_{i+1-m}$$

for stability, we do not want w_n to grow without bound, so we want the $|\beta_j| \le 1$.

Theorem 24. *The multi-step method is zero-stable if and only if $p(z)$ satisfies Dahlquist's root condition*

$$|\beta_j| \leq 1 \quad j = 1, \ldots, m$$

and if $|\beta_j| = 1$, then β_j is a simple root.

The method is said to be:

- *strongly stable if the only root with magnitude 1 is $\beta = 1$*
- *weakly stable if there is a $\beta \neq 1$ with magnitude 1*
- *unstable if Dahlquist's root condition is not met*

Moreover, if the method is consistent, then it is convergent if and only if it is zero-stable.

Example 0.10. All one-step methods have a characteristic polynomial with $p(\lambda) = \lambda - 1$, so are strongly stable.

Adams-Bashforth has $p(\lambda) = \lambda^{m-1}(\lambda - 1)$ and hence is strongly stable as well.

Example 0.11. Now, we reconsider our computational example, given by

$$y' = \lambda y \quad \lambda \in \mathbb{C},\ t \in [a, b],\ y(a) = \alpha$$

For some values of h, the method is unstable, even though it is stable to round-off error due to being zero-stable. This stability is due to the equation f. Such equations are called stiff.

We substitute $f = \lambda y$ into F.

$$\begin{aligned} w_{i+1} &= a_{m-1}w_i + \ldots + a_0 w_{i+1-m} + h\lambda(b_m w_{i+1} + \ldots + b_0 w_{i+1-m}) \\ 0 &= (1 - h\lambda b_m)w_{i+1} - (a_{m-1} + h\lambda b_{m-1})w_i - \ldots - (a_0 + h\lambda b_0)w_{i+1-m} \end{aligned}$$

Recall that $b_m = 0$ if the method is explicit. The general solution is $w_n = \sum_{j=1}^{m} c_j \beta_j$, where the β_j are roots of

$$Q(z) = (1 - h\lambda b_m)z^m - (a_{m-1} + h\lambda b_{m-1})z^{m-1} - \ldots - (a_0 + h\lambda b_0)$$

The method is absolutely stable if the response to $f = \lambda y$ for some $\lambda \in \mathbb{C}$ and $y(a) = \alpha$ is stable, meaning that there is some $C > 0$ such that

$$|w_n| \leq C\left|e^{\lambda h n}\right|$$

The region of absolute stability is defined as

$$R = \{h\lambda \in \mathbb{C} \,|\, |\beta_j| < 1,\ \forall \beta_j \in \mathbb{C} \text{ s.t. } Q(\beta_j) = 0\}$$

as expected, whenever $h\lambda \in R$, the method is absolutely stable.

A method is called **A-stable** if $\mathbb{C}_- \subseteq R$, where $\mathbb{C}_-$ is the left half plane.

Example 0.12. We take $f(t, y) = \lambda y$ and run explicit Euler on it. Then we have

$$\begin{aligned} w_{i+1} &= w_i + h\lambda w_i \\ &= (1 + h\lambda)w_i \\ Q(z) &= z - (1 + h\lambda) \end{aligned}$$

Thus, Q has one root, $1+h\lambda$. Thus, R consists of $h\lambda$ such that $|1+h\lambda|<1$. This is a circle in the plane of radius 1 centered at -1. Of course, the method is not A-stable.

For $\lambda \in \mathbb{R}$, we must have

$$\begin{aligned} |1+h\lambda| &< 1 \\ -1 < 1+h\lambda &< 1 \\ -2 < h\lambda &< 0 \\ h &< \frac{2}{|\lambda|} \end{aligned}$$

for the method to be stable.

Example 0.13. Implicit Euler with this $f(t,y)=\lambda y$ gives

$$\begin{aligned} (1-h\lambda)w_{i+1} &= w_i \\ Q(z) &= (1-h\lambda z)-1 \end{aligned}$$

so the only root is $\frac{1}{1-h\lambda}$, meaning that $|\beta<1|$ if and only if $1<|1-h\lambda|$. This means that R is the complement of the disc of radius 1 centered at $1 \in \mathbb{C}$. Thus, the method is A-stable. If λ is negative, there is no restriction on h.

Lecture 22: Boundary Value Problems (11/19)

In initial value problems, we often consider the parameter t to represent time, so we have a system with an initial state that we let evolve. In boundary value problems, we consider the parameter to be a spatial variable x, which may be higher dimensional. Here, we have boundary conditions, that specify information at the boundary of some domain. There are multiple types of boundary conditions (BCs), including:

- Dirichlet BCs, of the form $y(a)=\alpha$
- Neumann BCs, of the form $y'(a)=\alpha$
- Robin BCs , which are linear combinations of solution and derivative $Ay(a)+By'(a)=\alpha$

Other types of BCs exist, especially in higher dimensions. Physicists/ engineers sometimes have specific names for these BCs depending on the equation.

A general second order BVP with Dirichlet boundary conditions is of the form

$$\begin{aligned} y''(x) &= f(x,y(x),y'(x)) && x \in [a,b] \\ y(a) &= \alpha \\ y(b) &= \beta \end{aligned}$$

The problem is linear if there exist $p(x),q(x),r(x)$ if f can be written

$$f(x,y,y') = p(x)y' + q(x)y + r(x)$$

meaning that f is linear in y and y'.

In $1D$, we can use **shooting methods** to solve these BVPs. The basic idea is to reformulate the problem as an IVP with some manipulations, and then use numerical methods for IVPs to compute a solution. These methods may have instabilities and other limitations. Instead, we will consider finite difference methods.

Finite difference methods

These methods are simple, but limited to "easy" domains. The idea of finite difference methods is to replace derivatives with discrete finite difference approximations and solve the resulting linear system of equations.

We discretize our domain with equally spaced nodes $a = x_0 < \ldots < x_n = b$, where $x_i = a + ih,\ \ h = \frac{b-a}{n}$. We assume a general linear BVP

$$\begin{aligned} y'' &= p(x)y' + q(x)y + r(x) && x \in [a,b] \\ y(a) &= \alpha \\ y(b) &= \beta \end{aligned}$$

We use centered difference equations for y' and y'', which are $O(h^2)$

$$\begin{aligned} y''(x_i) &= \frac{y(x_{i-1}) - 2y(x_i) + y(x_{i+1})}{h^2} - \frac{h^2}{12}y^{(4)}(\xi_i) \\ y'(x_i) &= \frac{y(x_{i+1}) - y(x_{i-1})}{2h} - \frac{h^2}{6}y^{(3)}(\eta_i) \end{aligned}$$

This means that at each interior x_i, $i = 1, \ldots, n-1$, the equation is approximately (to within $O(h^2)$ error),

$$\begin{aligned} \frac{w_{i-1} - 2w_i + w_{i+1}}{h^2} &= p(x_i)\frac{w_{i+1} - w_{i-1}}{2h} + q(x_i)w_i + r(x_i) && i = 1, \ldots, n-1 \\ w_0 &= \alpha, \quad w_n = \beta \end{aligned}$$

Thus, we have $n-1$ equations in the $n-1$ unknowns w_i. Note that

$$\frac{w_{i-1} - w_i + w_{i+1}}{h^2} = -\frac{1}{h^2}\begin{bmatrix} -1 & 2 & -1 \end{bmatrix}\begin{bmatrix} w_{i-1} \\ w_i \\ w_{i+1} \end{bmatrix} \qquad 1 < i < n-1$$

and the equation is truncated at $i = 1, n-1$, so we have that this is equal to

$$\underbrace{-\frac{1}{h^2}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}}_{A}\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{n-2} \\ w_{n-1} \end{bmatrix} + \begin{bmatrix} \frac{1}{h^2}w_0 \\ 0 \\ \vdots \\ 0 \\ \frac{1}{h^2}w_n \end{bmatrix}$$

We can view the matrix A as applying the second derivative to the vector w. The next term is

$$p(x_i)\frac{w_{i+1}-w_{i-1}}{2h} = \frac{1}{2h}\underbrace{\begin{bmatrix} 0 & p(x_1) & & & & \\ -p(x_2) & 0 & p(x_2) & & & \\ & \ddots & \ddots & \ddots & & \\ & & & 0 & p(x_{n-2}) \\ & & & -p(x_{n-1}) & 0 \end{bmatrix}}_{D}\begin{bmatrix} w_1 \\ \vdots \\ w_{n-1} \end{bmatrix} + \begin{bmatrix} -p(x_1)\frac{1}{2h}w_0 \\ 0 \\ \vdots \\ 0 \\ p(x_{n-1})\frac{1}{2h}w_n \end{bmatrix}$$

We can view D as applying a derivative to the vector w. The last terms give

$$q(x_i)w_i \implies \underbrace{\begin{bmatrix} q(x_1) & & \\ & \ddots & \\ & & q(x_{n-1}) \end{bmatrix}}_{M}\begin{bmatrix} w_1 \\ \vdots \\ w_{n-1} \end{bmatrix}$$

$$r(x_i) \implies \begin{bmatrix} r(x_1) \\ \vdots \\ r(x_{n-1}) \end{bmatrix}$$

so our system is

$$(A-D-M)w = \begin{bmatrix} r(x_1) - \frac{1}{h^2}w_0 - \frac{p(x_1)}{2h}w_0 \\ r(x_2) \\ \vdots \\ r(x_{n-2}) \\ r(x_{n-1}) - \frac{1}{h^2}w_n + \frac{1}{2h}p(x_{n-1})w_n \end{bmatrix}$$

A nonlinear BVP will require Newton's method, which require matrix Jacobians. Each step requires a matrix inversion. Also, different BCs require modifications to this method. For instance, a Neumann boundary condition requires

$$y'(a) = \alpha \implies \frac{w_1 - w_0}{h} = \alpha$$

We will also consider how this process can be generalized to PDEs. Here, we will focus on $2D$ spatial problems (no time parameter). Often, these are elliptic equations. The canonical equation of this form is Poisson's equation:

$$\nabla^2 u = -f$$

For simplicity, we will take a square domain $(0,1)^2$, with an evenly spaced square grid stencil. It does not complicate matter much to choose different spacing amounts in x and y, but we keep them the same here. BCs can be full Dirichlet, part Dirichlet part Neumann, or plenty of other choices. The well-posedness of the PDE depends on the choice of BCs.

We use finite difference approximations for the partial derivatives

$$\partial_{xx}u(x_i,y_j) = \frac{u(x_{i+1},y_j) - 2u(x_i,y_j) + u(x_{i-1},y_j)}{h^2} - \frac{h^2}{12}\partial_{x^4}u(\xi_i,y_j)$$

with the analogous equations for differentiating in y. The Poisson equation then becomes

$$\nabla^2 u = -f$$
$$\implies \Big(\frac{u(x_{i+1},y_j)-2u(x_i,y_j)+u(x_{i-1},y_j)}{h^2}+\frac{u(x_i,y_{j+1})-2u(x_i,y_j)+u(x_i,y_{j-1})}{h^2}\Big) = -f(x_i,y_j)+O(h^2)$$

Lecture 23: (11/21)

We continue with the $2D$ Poisson equation $\nabla^2 u = -f$. Based on the above, our numerical scheme is

$$\frac{w_{i+1,j}-2w_{i,j}+w_{i-1,j}}{h^2}+\frac{w_{i,j+1}-2w_{i,j}+w_{i,j-1}}{h^2} = -f_{i,j}$$

for simplicity of the resulting formulas, we assume that our Dirichlet boundary conditions specify tht the solution is zero along the boundary. There are two ways of writing this equation as a matrix equation. One is as a Sylvester equation

$$AU+UA = -F$$
$$A = \frac{-1}{h^2}\begin{bmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & \ddots & \ddots & \ddots \end{bmatrix}$$
$$U = \begin{bmatrix} w_{1,1} & \dots & w_{n-1,1} \\ \vdots & & \vdots \\ w_{1,n-1} & \dots & w_{n-1,n-1} \end{bmatrix}$$
$$F = \begin{bmatrix} f_{1,1} & \dots & f_{n-1,1} \\ \vdots & & \vdots \\ f_{1,n-1} & \dots & f_{n-1,n-1} \end{bmatrix}$$

Note that the indexing of U and F are transposed from what we expect. This is due to the ordinary indexing on the plane. We multiply

$$AU = \frac{-1}{h^2}\begin{bmatrix} 2w_{1,1}-w_{1,2} & \dots \\ -w_{1,1}+2w_{1,2}-w_{1,3} & \dots \\ -w_{1,2}+2w_{1,3}-w_{1,4} & \dots \\ \vdots & \ddots \end{bmatrix}$$

The elements of AU are the finite difference approximations to $\partial_{yy}u(x_i,y_j)$. Analogously, UA are the finite difference approximations to $\partial_{xx}u(x_i,y_j)$. There are methods to solve Sylvester equations directly.

An alternative way to write the scheme is to write it as a large linear system. This is done by concatenating all columns of U in a single vector to get a system of the form

$$K\vec{u} = -\vec{f}$$

the K is a very sparse block matrix.

Thus, we have two different ways to solve this Poisson equation. Other approaches to solve the Poisson equation include using different meshes and using other finite difference approximations.

Recall that Poisson equation models heat conduction at a stable state. If we want to model evolution over time of some process, we can consider the time-dependent version of the heat equation

$$\rho c_p \partial_t u - k\nabla^2 u = \tilde{f}$$

ρ is density, c_p is specific heat, k is thermal conductivity, u is temperature, and $\tilde{f}$ is heat generation. We manipulate this to

$$\partial_t u - \alpha \nabla^2 u = \frac{1}{\rho c_p}\tilde{f}$$

where $\alpha = \frac{k}{\rho c_p} > 0$ is thermal diffusivity. We write the right side as f, so our equation is

$$\partial_t u - \alpha \nabla^2 u = f$$

This can be solved in multiple spatial dimensions plus the time dimension. We will focus on the $1D$ version. Let $\Omega = (0, L)$ the spatial domain, and $[0, T]$ the time domain. Our equation is

$$\partial_t u - \alpha \partial_{xx} u = f$$

Now, we need both initial conditions and boundary conditions:

$$\begin{aligned} u(x,0) &= u_0(x) \\ u(0,t) &= g_0(t) \\ u(L,t) &= g_L(t) \end{aligned}$$

The BCs are Dirichlet boundary conditions (aka temperature boundary conditions). Often the $g_0(t)$ and $g_L(t)$ are constant over time. Also, Neumann boundary conditions can be used to for instance model convection.

The idea to solve this problem is to discretize in time and space using finite differences. Assume for simplicity that $f = 0$, $g_0 = 0$, $g_L = 0$. We first discretize in space

$$\begin{aligned} \partial_t u &= \alpha \partial_{xx} u \\ \implies \partial_t w_i &= \alpha \frac{w_{i+1} - 2w_i + w_{i-1}}{\Delta x^2} \\ \implies \partial_t \vec{w}(t) &= \alpha A \vec{w}(t) \end{aligned}$$

Note that in time, $w_i^k \approx u(x_i, t^k)$ (superscripts are time indices), where $x_i = i\Delta x$, $t^k = k\Delta t$. We have different options to solve this by IVP solvers:

$$\begin{aligned} \text{Explicit Euler:} \quad \partial_t w &= \frac{w^{k+1} - w^k}{\Delta t} \\ \implies w^{k+1} &= w^k + \Delta t \alpha A w^k \\ \text{Implicit Euler:} \quad w^{k+1} &= w^k + \Delta t \alpha A w^{k+1} \\ w^{k+1} &= (I - \Delta t \alpha A)^{-1} w^k \end{aligned}$$

Also, the Trapezoidal method (called Crank-Nicolson here) works well in these methods

$$\begin{aligned} w^{k+1} &= w^k + \Delta t \frac{1}{2}(\alpha A w^{k+1} + \alpha A w^k) \\ w^{k+1} &= \Big(I - \Delta t \frac{1}{2}\alpha A\Big)^{-1}\Big(I + \Delta t \frac{1}{2}\alpha A\Big) w^k \end{aligned}$$

Denote $\tilde{A} = \Delta x^2 A$. Then we can write

$$w^{k+1} = \Big(I - \frac{\Delta t}{2\Delta x^2}\alpha\tilde{A}\Big)^{-1}\Big(I + \frac{\Delta t}{2\Delta x^2}\alpha A\Big)w^k$$

This method is stable if all eigenvalues of $\Big(I - \frac{\Delta t}{2\Delta x^2}\alpha\tilde{A}\Big)^{-1}\Big(I + \frac{\Delta t}{2\Delta x^2}\alpha A\Big)$ are < 1. The starting vector is known:

$$w_0 = \begin{bmatrix} u_0(x_1) \\ \vdots \\ u_0(x_{n-1}) \end{bmatrix}$$

Expanding this out, we get that

$$w_i^{k+1} = w_i^k + \frac{\Delta t}{2}\Big(\alpha\frac{w_{i+1}^{k+1} - 2w_i^{k+1} + w_{i-1}^{k+1}}{\Delta x^2} + \alpha\frac{w_{i+1}^k - 2w_i^k + w_{i-1}^k}{\Delta x^2}\Big)$$

Then our truncation error is

$$\begin{aligned}\tau_{i+1}^{k+1}(\Delta x, \Delta t) &= \frac{u(x_i, t^{k+1}) - u(x_i, t^k)}{\Delta t} - \frac{1}{2}\Big(\alpha\frac{u(x_{i+1}, t^{k+1}) - 2u(x_i, t^{k+1}) + u(x_{i-1}, t^{k+1})}{\Delta x^2} + \dots\Big) \\ &= O(\Delta t^2,\, \Delta x^2)\end{aligned}$$

as can be shown by Taylor expansion. For explicit Euler, $\tau_{i+1}^{k+1} = O(\Delta t,\, \Delta x^2)$.

Also, we can perform Von Neumann stability analysis. For this, we assume the solution is in the form of a Fourier series:

$$u(x,t) = \sum_{s\in\mathbb{Z}} \hat{u}(s,t)e^{i\frac{2\pi}{L}sx}$$

where L is the length of the spatial domain. It suffices to consider each Fourier mode independently:

$$\begin{aligned} w_j^k &\approx \underbrace{\hat{u}(s,t^k)}_{B^k} e^{i\frac{2\pi}{L}sx_j} \\ \implies w_j^{k+1} &\approx \hat{u}(s, t^k + \Delta t)e^{i\frac{2\pi}{L}sx_j}\end{aligned}$$

For stability, we want

$$\frac{\big|B^{k+1}\big|}{\big|B^k\big|} = G_k \le 1 \qquad \forall k$$

where G_k is called the growth factor or amplification factor.

Lecture 24: Properties of Finite Difference Methods (11/26)

For solving wave equations, higher frequencies require higher mesh sizes for the numerical scheme, since peaks and valleys in the solution may not be adequately captured by lower resolution mesh sizes.

We recall Von Neumann stability analysis. Replace

$$\begin{aligned} w_j^k &\approx \hat{u}(s,t^k)e^{i\frac{2\pi}{L}sx_j} = \hat{g}^k(\xi)e^{i\xi x_j} \\ w_j^{k+1} &\approx \hat{u}(s,t^{k+1})e^{i\frac{2\pi}{L}sx_j} = \hat{g}^{k+1}(\xi)e^{i\xi x_j}\end{aligned}$$

where $\xi = \frac{2\pi s}{L}$. For stability (in time), we want amplitudes under control

$$G^{(k)}(\xi) = \frac{\hat{g}^{k+1}(\xi)}{\hat{g}^k(\xi)} \qquad \left\|G^{(k)}\right\|_\infty \leq 1$$

A translation operator gives a shift in space

$$\begin{aligned} w_{j+1}^k &= \hat{g}^k(\xi) e^{i\xi(x_j + \Delta x)} \\ &= \hat{g}^k(\xi) e^{i\xi x_j} e^{i\xi \Delta x} \\ &\approx w_j^k e^{i\xi \Delta x} \end{aligned}$$

Example 0.14 (Explicit Euler on time-dependent heat equation)**.**

Consider the heat equation $\partial_t u = \alpha \partial_{xx} u$. The iterates for explicit Euler are of the form

$$w_j^{k+1} = w_j^k + \alpha \frac{\Delta t}{\Delta x^2}(w_{j+1}^k - 2w_j^k + w_{j-1}^k)$$

We perform the Von Neumann stability analysis by replacing $w_j^k = \hat{g}^k(\xi) e^{i\xi x_j}$

$$\begin{aligned} \hat{g}^{k+1}(\xi) e^{i\xi x_j} &= \hat{g}^k(\xi) e^{i\xi x_j} + \alpha \frac{\Delta t}{\Delta x^2}\Big(e^{i\xi \Delta x} - 2 + e^{-i\xi \Delta x}\Big)\hat{g}(\xi) e^{i\xi x_j} \\ &= \hat{g}^k(\xi) e^{i\xi x_j} + \alpha \frac{\Delta t}{\Delta x^2}\Big(\cos(\xi \Delta x) - 1\Big) 2\hat{g}(\xi) e^{i\xi x_j} \\ G(\xi) = \frac{\hat{g}^{k+1}(\xi)}{\hat{g}^k(\xi)} &= 1 + \underbrace{\alpha \frac{\Delta t}{\Delta x^2}}_{\sigma} 2(\cos(\xi \Delta x) - 1) \end{aligned}$$

Recall that $\sin^2(\frac{\xi \Delta x}{2}) = \frac{1}{2}(1 - \cos(\xi \Delta x)$. Thus, we have

$$G(\xi) = 1 - 4\sigma \sin^2\Big(\frac{\xi \Delta x}{2}\Big)$$

So that our stability condition is

$$|G(\xi)| \leq 1 \iff 0 \leq 4\sigma \sin^2\Big(\frac{\xi \Delta x}{2}\Big) \leq 2$$

Note that σ and $\sin^2$ are always positive, so the left inequality holds automatically. Thus, our condition is

$$\sigma \sin^2\Big(\frac{\xi \Delta x}{2}\Big) \leq \frac{1}{2}$$

Since $\sin^2$ is bounded by 1, a sufficient condition is $\sigma \leq 1/2$. This means that we should choose our discretization parameters to be related by

$$\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2\alpha}$$

Von Neumann stability analysis can be extended to multi-step methods and more complicated systems.

Finally, we relate consistency, stability, and convergence of finite difference methods for PDEs.

Recall that for the heat equation, we considered different numerical methods. Crank-Nicolson takes the form

$$w^{k+1} = \underbrace{\left(I - \frac{1}{2}\sigma\tilde{A}\right)^{-1}\left(I + \frac{1}{2}\sigma\tilde{A}\right)}_{S_{\Delta t,\Delta x}} w^k$$

This matrix S, the stepping matrix, characterizes the method.

A method is **consistent** if $\lim_{\Delta t\to 0}\left\|\vec{u}(t+\Delta t) - S_{\Delta t,\Delta x}\vec{u}(t)\right\| = 0$. Where $\vec{u}(t) = \left[u(x_1,t),\ldots,u(x_{n-1},t)\right]$ is the exact solution at time t and points in the spacial mesh $a = x_0 < \ldots < x_n = b$. This norm is roughly proportional to local truncation error.

A method is **convergent** if $\lim_{\Delta t\to 0}\left\|w^{t/\Delta t} - u(t)\right\| = 0$ for every t multiple of Δt. Intuitively, consistency means the equation is well-approximated while convergence means the solution is well-approximated.

The method is **stable** if $\frac{\|\vec{w}^{k+1}\|}{\|\vec{w}^k\|}\left\|S_{\Delta t,\Delta x}\right\| \le 1 + C\Delta t$ for some $C > 0$ and some norms.

Theorem 25 (Lax equivalence theorem)**.** *Let a finite difference scheme be a consistent approximation to a well-posed linear PDE in the form of an initial value problem ($\partial_t u = Lu$) where L is a linear operator. Then the scheme is convergent if and only if it is stable.*

Thus, for a linear PDE, it is sufficient to show that the truncation error goes to 0 and show stability in order to conclude convergence of a method.

Energy and finite element methods

These are also called Ritz methods or Rayleigh-Ritz methods. For now, we focus on the Poisson equation $-\Delta u = f$ or, in $1D$, $-u''(x) = f(x)$, $x \in [0,1]$. Also, we assume zero Dirichlet boundary conditions $u(0) = u(1) = 0$. The form of this equation as written is called the strong form (S).

The idea of these methods is to not solve the equation directly, but instead convert it into a weak form by multiplying by arbitrary test functions and integrating. For example,

$$\begin{aligned} -u'' &= f \\ -u''v &= fv \\ -\int_0^1 u''v\,dx &= \int_0^1 fv\,dx \\ \langle -u'', v\rangle_{L^2} &= \langle f, v\rangle_{L^2} \end{aligned}$$

where $v \in C_0^1[0,1]$, meaning it is continuously differentiable and satisfies the boundary conditions $v(0) = v(1) = 0$. Integrating by parts gives

$$\begin{aligned} -\int_0^1 u''v\,dx &= -u'(x)v(x)\Big]_0^1 + \int_0^1 u'(x)v'(x)\,dx \\ &= \int_0^1 u'(x)v'(x)\,dx \end{aligned}$$

so our equation is

$$\int_0^1 u'(x)v'(x)\,dx = \int_0^1 f(x)v(x)\,dx \quad \forall v \in C_0^1[0,1]$$

This is the weak form (W). It turns out to be equivalent to the strong form (S). We end by stating a lemma that we will use later:

Lemma 6. *Let $g \in C[a,b]$ with $\int_a^b g(x)v(x)\,dx = 0$ for all $v \in C_0^\infty[a,b]$. Then $g = 0$.*

Lecture 25: Energy and Finite Element Methods (12/3)

Theorem 26. $(S) \iff (W)$

Proof. We will only show this for the special case of solutions that are sufficiently smooth, meaning for $u \in C_0^2[0,1]$.

$(S) \implies (W)$ is given by the above derivation.

For $(S) \impliedby (W)$, suppose

$$\int_0^1 u'v'\,dx = \int_0^1 fv\,dx$$

for all $v \in C_0^1[0,1]$. Then we integrate by parts so

$$\begin{aligned} u'v\,|_0^1 - \int_0^1 u''v\,dx &= \int_0^1 fv\,dx && \forall v \in C_0^1[0,1] \\ -\int_0^1 u''v\,dx &= \int_0^1 fv\,dx && \forall v \in C_0^1[0,1] \\ \int_0^1 (-u'' - f)v\,dx &= 0 && \forall v \in C_0^1[0,1] \end{aligned}$$

Thus, applying the lemma gives $-u'' - f = 0$ so $-u'' = f$. □

Note that this proof can be done using just properties of the L^2 inner product. For solutions that are not sufficiently smooth, we must use more general spaces: Sobolev spaces. Recall we assume here that $\Omega = [0,1]$.

$$\begin{aligned} L^2(\Omega) &= \left\{u : \Omega \to \mathbb{R} \mid \int_0^1 |u|^2\,dx < \infty\right\} \\ H^1(\Omega) &= \left\{u \in L^2(\Omega) \mid u' \in L^2\right\} \\ H_0^1(\Omega) &= \left\{u \in H^1(\Omega) \mid u(0) = u(1) = 0\right\} \end{aligned}$$

A more general discussion of the above strong and weak formulations switches out $H_0^1[0,1]$ for each instance of $C_0^1[0,1]$.

We now consider the more general weak form equation: find $u \in H_0^1(\Omega)$ such that

$$\int_0^1 u'v'\,dx = \int_0^1 fv\,dx \qquad \forall v \in H_0^1(\Omega)$$

Define the bilinear form $b(u,v) = \int_0^1 u'v'\,dx$ and the linear form $l(v) = \int_0^1 fv\,dx$. Then we can rewrite the weak formulation as: find $u \in H_0^1(\Omega) = U$ such that

$$b(u,v) = l(v) \qquad \forall v \in H_0^1(\Omega) = V$$

U is called the **trial space**, and V is the **test space**. This is called a (linear) variational formulation. It is used to prove well-posedness of PDEs. Notice that the trial and test spaces are infinite-dimensional. In this case, formulations are called continuous variational formulations.

We consider properties of b for this equation:

- $b(u,v) = b(v,u)$, so b is symmetric.
- $b(u,u) > 0$ for all $u \neq 0$, so b is positive definite.

When b is spd, the methods used to solve the equations are sometimes called **energy methods** because it can be shown that the weak formulation is equivalent to an energy equation (E)

$$\min_{u \in U} F(u) = \frac{1}{2} b(u,u) - l(u)$$

Now, we want to use (W) to construct a numerical method to solve the equation. The idea is to discretize U and V by finding finite-dimensional subspaces called discrete trial and test spaces, given by $U_h \subseteq U$ and $V_h \subseteq V$.

Since $U = V$ here, we can choose $U_h = V_h = \text{span}(\varphi_j)_{j=1}^n$, where the φ_j are linearly independent. Thus we have

$$u_n(x) = \sum_{j=1}^{n} \alpha_j \varphi_j(x) \qquad \vec{\alpha} \in \mathbb{R}^n$$

The approximate problem becomes: find $u \in U_h$ such that

$$(G) \qquad b(u_h, v_h) = l(v_h) \qquad \forall v_h \in V_h$$

This is a discrete variational formulation, called the **Galerkin** formulation. To solve it, find $u_h = \sum_{j=1}^n \alpha_j \varphi_j(x) \in U_n$ such that

$$\begin{aligned} b(u_h, \varphi_i) &= b(\sum_j \alpha_j \varphi_j, \varphi_i) \\ &= \sum_j \alpha_j b(\varphi_j, \varphi_i) \\ &= l(\varphi_i) \end{aligned}$$

for all $i = 1, \ldots, n$. Thus, we have n equations in n unknowns

$$\sum_j \alpha_j \underbrace{b(\varphi_j, \varphi_i)}_{K_{ij}} = l(\varphi_i) \qquad i = 1, \ldots, n$$

Let $F = (l(\varphi_1), \ldots, l(\varphi_n))$, called the force vector. $\alpha = (\alpha_1, \ldots, \alpha_n)$ is the solution vector, and $K = \{K_{ij}\}_{i,j=1}^n$, so the equation is $K\alpha = F$. This is the matrix form (M), which is clearly equivalent to the Galerkin formulation (G).

To utilize this formulation, we must choose basis functions. We would like to choose φ_j so that K is easy to invert. For instance, we could choose K to have special structure so that we can use FFT-like algorithms to solve the system. Or, we could choose K to be sparse. For sparsity, we want that

$$K_{ij} = b(\varphi_j, \varphi_i) = \int_0^1 \varphi_j' \varphi_i' \, dx = 0 \qquad \text{in most cases}$$

This is easy to do by choosing φ_j with local support. Then if φ_i and φ_j have disjoint support, so do their derivatives, so $b(\varphi_i, \varphi_j) = 0$. A typical choice for φ_j are hat functions, which are piecewise linear polynomials.

First, we discretize $\Omega = [0,1]$ into elements

$$e_1 = (x_0, x_1),\ e_2 = (x_1, x_2), \ldots,\ e_n = (x_{n-1}, x_n)$$

where $0 = x_0 < x_1 < \ldots, < x_n = 1$. These are not necessarily taken to be uniformly distributed. Then we take φ_j as $\varphi_j(x_j) = 1$ and linear down to 0 as x_{j-1}, x_{j+1}, for $j = 1, \ldots, n-1$. Here, the supports satsify $\text{supp}(\varphi_j) = e_j \cup e_{j+1}$. In total, we have $n-1$ basis hat functions, with n elements. They satisfy

$$\varphi_j(x_i) = \delta_{ij}$$

Lecture 26: Finite Element Methods (12/5)

Since the hat functions satisfy $\varphi_j(x_i) = \delta_{ij}$, they have a special property:

$$\begin{aligned} u_h(x_k) &= \sum_{j=1}^{n} \alpha_j \varphi_j(x_k) \\ &= \sum_j \alpha_j \delta_{jk} \\ &= \alpha_k \end{aligned}$$

for any $k = 1, \ldots, n$. Thus, $\alpha = (u_h(x_1), \ldots, u_h(x_n))$, so the α_j actually correspond to the nodal values. This is a very nice property for the basis of hat functions.

The methods for solving PDEs are described last lecture, in the case where the basis of φ_j consists of piecewise polynomials with local support, are called **finite element methods**. When $U_h = V_h$, they are called **Bubnov-Galerkin methods**. When $U_h \neq V_h$, they are called **Petrov-Galerkin methods**.

Piecewise linear hat functions results in **convergence** of order $O(h^2)$, where h is the maximum element size. Here, we measure convergence in the L_2 norm, so $\|u_h - u\|_{L^2} = O(h^2)$. Better convergence is possible by enriching the basis with quadratic or other higher-order polynomial functions.

Example 0.15. Say we want to solve the equation

$$\begin{aligned} -u''(x) &= f(x) \qquad x \in \Omega = (0,1) \\ u(0) &= g \\ u'(1) &= q \end{aligned}$$

So now we have one Dirichlet boundary condition at $x = 0$ and one Neumann boundary condition at $x = 1$. To get this in variational form, we test with

$$v \in H^1_D(\Omega) = \{w \in H^1(\Omega) \mid w(0) = 0\}$$

so we enforce that the test functions are zero only at the Dirichlet boundary condition. Then we have

$$\begin{aligned}\int_0^1 -u''v\,dx &= \int_0^1 fv\,dx && \forall v \in H_D^1(\Omega)\\ &= -u'(x)v(x)\Big]_0^1 + \int_0^1 u'v'\,dx && \text{integrate by parts}\\ &= -u'(1)v(1) + u'(0)v(0) + \int_0^1 u'v'\,dx\\ &= -q(1)v(1) + \int_0^1 u'v'\,dx\end{aligned}$$

So we have an equation

$$\int_0^1 u'v'\,dx = \int_0^1 fv\,dx + qv(1)$$

Thus, we have the same bilinear form $b(u,v) = \int_0^1 u'v'\,dx$, but now we have a linear form

$$l_1(v) = \int_0^1 fv\,dx + qv(1)$$

Additionally, u must satisfy $u(0) = g$, so we write $u = u_0 + w$ with $u_0(0) = g$ and $w \in H_D^1(\Omega)$. Note that $H_D^1(\Omega)$ is a vector space, but the analogous space with enforcing the Dirichlet boundary condition at g is not. Therefore, the variation formulation becomes

$$\begin{cases}\text{Find } u = u_0 + w \in u_0 + U \text{ such that}\\ b(u,v) = l_1(v) & \forall v \in V\end{cases}$$

Which is equivalent to the weak formulation

$$(W) \quad \begin{cases}\text{Find } w \in U\\ b(w,v) = l_1(v) - b(u_0,v) =: l(v) & \forall v \in V\end{cases}$$

We now discretize with hat functions. Our elements are $\Omega^1, \ldots, \Omega^n$, where $\Omega^j = (x_{j-1}, x_j)$. We look for an approximation

$$u_h = u_{0,h} + w_h = g\varphi_0 + \sum_{j=1}^n \alpha_j \varphi_j$$

We choose hat functions φ_j as follows: φ_0 on its support is linear from $(x_0, 1)$ to $(x_1, 0)$. φ_n on its support is linear from $(x_{n-1}, 0)$ to $(x_n, 1)$. For j in between φ_j is linear from $(x_{j-1}, 0)$ to $(x_j, 1)$ to $(x_{j+1}, 0)$. Note that now we have these boundary hat values, since we do not have just zero boundary conditions. The derivatives φ_j' are step functions so they are easy to integrate against. Then our equations are

$$\sum_{j=1}^n \alpha_j \underbrace{b(\varphi_j, \varphi_i)}_{K_{ij}} = b(w_h, \varphi_i) = l(\varphi_i) = \underbrace{l_1(\varphi_i) - gb(\varphi_0, \varphi_i)}_{F_i} \qquad i = 1, \ldots, n$$

Note that $K_{ij} = \int_0^1 \varphi_j' \varphi_i;\, dx$, and any adjacent basis functions have disjoint support, so K is in fact symmetric tridiagonal. With a uniform grid, the diagonal of K is (since piecewise linearity gives slopes

of $\frac{1}{h}$),

$$\begin{aligned} K_{ii} &= \int_0^1 \varphi_i' \varphi_i' \, dx \\ &= \int_{\Omega^i} \frac{1}{h^2} \, dx + \int_{\Omega^{i+1}} \frac{1}{h^2} \, dx \\ &= \frac{2h}{h^2} \\ &= \frac{2}{h} \end{aligned}$$

If $i = 1, \ldots, n-1$, and $K_{nn} = \frac{1}{h}$. Now, for the sub/superdiagonal elements, let $j = i+1$. Then

$$\begin{aligned} K_{ij} &= \int_0^1 \varphi_i' \varphi_j' \, dx \\ &= \int_{\Omega^j} \frac{-1}{h^2} \, dx \\ &= \frac{-1}{h} \end{aligned}$$

Then we compute $F_i = l_1(v) - gb(\varphi_0, \varphi_i)$. We have $b(\varphi_0, \varphi_1) = \frac{-1}{h}$, and $b(\varphi_0, \varphi_i) = 0$ for $i > 1$. For the other summand,

$$\begin{aligned} l_1(\varphi_i) &= \int_0^1 f\varphi_i \, dx + q\varphi_i(1) \\ &= \underbrace{\int_0^1 f\varphi_i \, dx}_{\tilde{F}_i} + q\delta_{i,n} \\ &= \int_{\Omega^i \cup \Omega^{i+1}} f\varphi_i \, dx + q\delta_{i,n} \end{aligned}$$

The integral of course depends on f. Thus, we have

$$\begin{aligned} F_1 &= \tilde{F}_1 - g\frac{1}{h} \\ F_i &= \tilde{F}_i \qquad\qquad i = 2, \ldots, n-1 \\ F_n &= \tilde{F}_n + q \end{aligned}$$

Thus, in total we have the equation $K\alpha = F$, where

$$K = \frac{1}{h} \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & & -1 & 1 \end{bmatrix}$$

$$\alpha = \begin{bmatrix} u_h(x_1) \\ \vdots \\ u_h(x_n) \end{bmatrix}$$

$$F = \begin{bmatrix} \int_0^1 f\varphi_1 \, dx \\ \vdots \\ \int_0^1 f\varphi_n \, dx \end{bmatrix} + \begin{bmatrix} 0 \\ \vdots \\ q \end{bmatrix} + \begin{bmatrix} g\frac{1}{h} \\ \vdots \\ 0 \end{bmatrix}$$

Our method is to solve $K\alpha = F$ and recover the discrete solution

$$u_h(x) = u_{0,h} + w_h(x) = g\varphi_0(x) + \sum_{j=1}^{n} \alpha_j \varphi_j(x)$$

Computational implementation

If we do not have evenly spaced elements, then the bilinear form evaluations may not be evaluated analytically as we have done. They have computed to be computed by numerical integration.

The naive way is to form K by:

for i = 1:n

  for j =1:n

    $K(i,j) = \int_\Omega \varphi_j' \varphi_i' \, dx$

  end

  $F(i) = \int_\Omega f(x)\varphi_i(x) \, dx - q\varphi_i(1) - gK(i,0)$

end

To compute the integral requires iterating over the elements and computing each integral $\int_{\Omega^e} \varphi_j' \varphi_i' \, dx$ on the element one by one. This implementation is very inefficient, since most $K(i,j)$ are zero. Note the pattern, in which we have to compute these integrals over the elements multiple times. Instead, we want the outer loop to be over the elements.

$$\begin{aligned} K &= \begin{bmatrix} \int_\Omega \varphi_1'\varphi_1' & \cdots & \int_\Omega \varphi_n'\varphi_1' \\ \vdots & & \vdots \\ \int_\Omega \varphi_1'\varphi_n' & \cdots & \int_\Omega \varphi_n'\varphi_n' \end{bmatrix} \\ &= \sum_{e=1}^{n_e} \underbrace{\begin{bmatrix} \int_{\Omega^e} \varphi_1'\varphi_1' & \cdots & \int_{\Omega^e} \varphi_n'\varphi_1' \\ \vdots & & \vdots \\ \int_{\Omega^e} \varphi_1'\varphi_n' & \cdots & \int_{\Omega^e} \varphi_n'\varphi_n' \end{bmatrix}}_{K^e} \end{aligned}$$

K^e is called the **element stiffness matrix**. We know that $K^e_{ij} = 0$ if $\varphi_i \,|_{\Omega^e} = 0$ or $\varphi_j \,|_{\Omega^e} = 0$. Thus, these matrices have nice nonzero structure.

$$K^1 = \begin{bmatrix} * & & \\ & & \\ & & \end{bmatrix}$$

$$K^2 = \begin{bmatrix} * & * & \\ * & * & \\ & & \end{bmatrix}$$

$$K^3 = \begin{bmatrix} & & \\ & * & * \\ & * & * \end{bmatrix}$$

Only 4 nonzero elements are present in each element stiffness matrix (besides the first) which always have the same structure. These are present in the **local element stiffness matrix**.

$$k^e = \begin{bmatrix} \int_{\Omega^e} \phi_1' \phi_1' & \int_{\Omega^e} \phi_2' \varphi_1' \\ \int_{\Omega^e} \phi_1' \phi_2' & \int_{\Omega^e} \phi_2' \varphi_2' \end{bmatrix}$$
$$k^e_{ab} = \int_{\Omega^e} \phi_a' \phi_b'$$

Where the define the shape function $\phi_a = \varphi_i \mid_{\Omega^e}$ and $\phi_b = \varphi_j \mid_{\Omega^e}$ for some $i, j \in 1, \ldots, n$.

Lecture 27: Finite Element Method Implementation (12/10)

To go from local element stiffness k^e to element stiffness K^e, use a **local to global map** via the locator matrix

$$LM = \begin{bmatrix} 0 & 1 & \ldots & n-2 & n-1 \\ 1 & 2 & \ldots & n-1 & n \end{bmatrix}$$

the LM has size $n_v \times n_{el}$, meaning number of rows equal to the number of vertices in the element, and number of columns equal to number of elements. The value $LM_{a,e}$ are indices of matrix K (equation number) that the local node number a maps to in elements e. The basic algorithm to compute K is as follows:

Assembly procedure

for e = 1:n

for a = 1:2 (test function)

$i = LM(a, e)$

for b = 1:2 (trial function)

$k^e(a,b) = \int_{\Omega^e} \phi_b' \phi_a' \, dx$

$j = LM(b, e)$

if $i, j \neq 0$, $K_{ij} \leftarrow K_{ij} + k^e(a,b)$

if $i \neq 0, j = 0$, $F_i \leftarrow F_i - k^e(a,b) \cdot g$

end

$f^e(a) = \int_{\Omega^e} f \phi_a \, dx$ (local force vector)

$F_i \leftarrow F_i + f^e(a)$

if $i = n$, $F_n \leftarrow F_n + q$

end

end

When we have Dirichlet data, (0th row or 0th column), we do not update K. However, we do need to update the force vector F if we are not at the 0th test function and we are the the 0th trial function. As we can see, looping over elements saves a lot of computations. The question now is of how to compute the element integrals efficiently. The answer is to use a master element.

Here, we take our physical element Ω^e and transform it to a master element $\hat{\Omega}$. We choose the master element $\hat{\Omega} = (-1, 1)$, which is nice to integrate over. Our transformation from physical to master is $\xi(x) = \frac{2x-(x_e+x_{e-1})}{h^e}$. The backward transformation is then $x(\xi) = \frac{h^e\xi+(x_e+x_{e+1})}{2}$. Then we have

$$\frac{d\xi}{dx} = \frac{2}{h^e} \qquad \frac{dx}{d\xi} = \frac{h^e}{2}$$

The physical coordinates (x) are

$$\begin{aligned}
\phi_1(x) &= \hat{\phi}_1(\xi(x)) \\
\phi_2(x) &= \hat{\phi}_2(\xi(x)) \\
\phi_1'(x) &= \partial_x\phi_1 = \partial_\xi\hat{\phi}_1\partial_x\xi = \frac{-1}{2}\cdot\frac{2}{h^e} \\
\phi_2'(x) &= \partial_x\phi_2 = \partial_\xi\hat{\phi}_2\partial_x\xi = \frac{1}{2}\cdot\frac{2}{h^e}
\end{aligned}$$

The master element coordinates (ξ) are

$$\begin{aligned}
\hat{\phi}_1(\xi) &= \frac{1}{2}(1-\xi) = \phi_1(x(\xi)) \\
\hat{\phi}_2(\xi) &= \frac{1}{2}(1+\xi) = \phi_1(x(\xi)) \\
\hat{\phi}_1'(\xi) &= -\frac{1}{2} \\
\hat{\phi}_2'(\xi) &= \frac{1}{2}
\end{aligned}$$

When integrating, we change variables to the master domain (we are in one dimension so the Jacobian is just the derivative)

$$\int_{\Omega^e}\psi(x)\,dx = \int_{\hat{\Omega}}\psi(x(\xi))\frac{dx}{d\xi}\,d\xi$$

For example, in the problem we were working above,

$$\begin{aligned}
f_a^e &= \int_{\Omega^e} f(x)\phi_a(x)\,dx \\
&= \int_{-1}^{1} f(x(\xi))\phi_a(x(\xi))\frac{h^e}{2}\,d\xi \\
&= \int_{-1}^{1} f(x(\xi))\hat{\phi}_a(\xi)\frac{h^e}{2}\,d\xi
\end{aligned}$$

the local stiffness is

$$\begin{aligned}
k_{ab}^e &= \int_{\Omega^e}\phi_b'(x)\phi_a'(x)\,dx \\
&= \int_{-1}^{1}\phi_b'(x(\xi))\phi_a'(x(\xi))\frac{h^e}{2}\,d\xi \\
&= \int_{-1}^{1}\hat{\phi}_{a,\xi}(\xi)\frac{d\xi}{dx}\cdot\hat{\phi}_{b,\xi}(\xi)\frac{d\xi}{dx}\cdot\frac{h^e}{2}\,d\xi \\
&= \int_{-1}^{1}\hat{\phi}_{a,\xi}(\xi)\cdot\hat{\phi}_{b,\xi}(\xi)\cdot\frac{2}{h^e}\,d\xi \\
&= \frac{(-1)^{a+b}}{h^e}
\end{aligned}$$

so the local stiffness for each element is

$$k^e = \frac{1}{h^e}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

2D finite elements

Say we have a domain $\Omega \subseteq \mathbb{R}^2$, with boundary $\partial\Omega$ that we break into Γ_D, where Dirichlet boundary conditions are specified, and Γ_N, where Neumann boundary conditions are specified. We have an outward unit normal $\vec{n}$. Recall the vector calculus theorems

Lemma 7. *2D divergence theorem:* $\int_\Omega \nabla \cdot \vec{f}\, d\Omega = \int_{\partial\Omega} \vec{f} \cdot \vec{n}\, d\Gamma$

Integration by parts: $\nabla \cdot (v\vec{f}) = v\nabla \cdot \vec{f} + \nabla v \cdot \vec{f}$ *so that* $\int_\Omega (\nabla \cdot \vec{f}) v\, d\Omega = -\int_\Omega \vec{f} \cdot \nabla v\, d\Omega + \int_{\partial\Omega} v\vec{f} \cdot \vec{n}\, d\Gamma$

In two dimensions, our problem is

$$\begin{aligned} -\nabla^2 u &= r \\ u(x) &= g_D(x) \quad x \in \Gamma_D \\ \nabla u \cdot \vec{n} &= g_N(x) \quad x \in \Gamma_N \end{aligned}$$

We write the solution $u = u_0 + w$ where $u_0(x) = g_D(x)$ for $x \in \Gamma_D$ satisfies the Dirichlet conditions and $w \in H^1_D(\Omega) = U = \{w \in H^1(\Omega) \mid w\,|_{\Gamma_D} = 0\}$. The weak variational formula can be found by integrating by parts and takes the form:

$$(W) = \begin{cases} \text{Find } w \in U = V = H^1_D(\Omega) \\ b(w,v) = l(v) \quad \forall v \in V = H^1_D(\Omega) \end{cases}$$

$$\begin{aligned} b(u,v) &= \int_\Omega \nabla u \cdot \nabla v\, d\Omega \\ l(v) &= l_1(v) - b(u_0, v) \\ l_1(v) &= \int_\Omega rv\, d\Omega + \int_{\Gamma_N} g_N v\, d\Gamma_N \end{aligned}$$

We discretize U and V with hat functions associated to elements, so we have $U_h, V_h \subseteq U, V$. We break Ω into elements by triangulating it. Then hat functions are 1 at a given node, and 0 at all other nodes, decreasing linearly to adjacent nodes (geometrically, they are like pyramids). With a triangle element Ω^e, we have 3 local nodes, that we map to a master element. Call the set of all nodes η, and η_D the set of Dirichlet nodes. Then $\eta \setminus \eta_D$ contains all of the nodes with unknown values. The size of this set then measures the size of our system of equations, given by

$$\begin{aligned} \sum_{j \in \eta \setminus \eta_D} \alpha_j b(\varphi_j, \varphi_i) &= l_1(\varphi_i) \\ &= \sum_{A \in \eta_D} g_D(x_A) b(\varphi_A, \varphi_i) \end{aligned}$$

Then we get a linear system $K\alpha = F$. To implement this, we need proper arrays mapping local stiffness to indices. The element nodes array describes the element mesh. For the example given in class,

$$IEN = \begin{bmatrix} 1 & 2 & \dots \\ 2 & 3 & \dots \\ 7 & 7 & \dots \end{bmatrix}$$

IEN has size. Its columns represent elements and its rows represent local node number. $IEN_{a,e}$ is the node number in η that the ath node in element e corresponds to.

The destination array is a row vector which tells whether a node is Dirichlet or not. Any Dirichlet node is 0, and any non-Dirichlet node is indexed by which equation it is in.

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

Finally, we have a locator matrix, $LM_{a,e} = ID(IEN_{a,e})$. Thus, $LM_{a,e}$ maps the ath local node in element e to the equation number it corresponds to (with value 0 if the node is Dirichlet).