CS 370 Numerical Computation

Keven Qiu
Instructor: Christopher Batty
Winter 2024

Contents

Part I

Floating Point Numbers

Chapter 1

Floating Point Number Systems

The real numbers $\mathbb{R}$ are infinite in extent and in density. The standard (partial) solution is to use floating point numbers to approximate the reals.

Definition: Floating Point Number System

An approximate representation of $\mathbb{R}$ using a finite number of bits.

An analytic solution is an exact solution, whereas a numerical solution is the approximate solution.

We can express a real number as an infinite expansion relative to some base β. For example, in base 10

$$\frac{73}{3} = 24.3333\ldots = 2 \times 10^1 + 4 \times 10^0 + 3 \times 10^{-1} + \cdots$$

After expressing the real number in the desired base β, we multiply by a power of β to shift it into a normalized form:

$$0.d_1 d_2 d_3 d_4 \cdots \times \beta^p$$

where d_i are digits in base β, i.e., $0 \leq d_i < \beta$, normalized implies we shift to ensure $d_1 \neq 0$, and the exponent p is an integer.

Density (or precision) is bounded by limiting the number of digits, t. Extent (or range) is bounded by limiting the range of values for exponent p.

Definition: Floating Point Representation

The finite form

$$\pm 0.d_1 d_2 \cdots d_t \times \beta^p$$

for $L \leq p \leq U$ and $d_1 \neq 0$.

The four integer parameters (β, t, L, U) characterize a specific floating point system F.

Overflow/underflow errors:

- If the exponent p is too big or too small, our system cannot represent the number.
- When arithmetic operations generate such a number, this is called overflow or underflow.
- For underflow, we simply round to 0.
- For overflow, we typically produce a $\pm\infty$ or NaN.

IEEE single precision (32 bits) has $(\beta = 2, t = 24, L = -126, U = 127)$ and IEEE double precision (64 bits) has $(\beta = 2, t = 53, L = -1022, U = 1023)$.

Unlike fixed point, floating point numbers are not evenly spaced.

There are two ways to convert reals to floats:

1. Round-to-nearest: rounds to closest available number in F.
2. Truncation: rounds to next number in F towards 0.

1.1 Measuring Error

Our algorithms will compute approximate solutions to problems.

Let x_{exact} be the true analytical solution and x_{approx} be the approximate numerical solution.

Definition: Absolute Error

$$E_{abs} = |x_{exact} - x_{approx}|$$

Definition: Relative Error

$$E_{rel} = \frac{|x_{exact} - x_{approx}|}{|x_{exact}|}$$

Relative error is often more useful because it is independent of the magnitudes of the numbers involved and related the number of significant digits in the result.

A result is correct to roughly s digits if $E_{rel} \approx 10^{-s}$ or

$$0.5 \times 10^{-s} \leq E_{rel} < 5 \times 10^{-s}$$

For floating point system F, the relative error between $x \in \mathbb{R}$ and its floating point approximation, $fl(x)$, has a bound, E, such that

$$(1 - E)\,|x| \leq |fl(x)| \leq (1 + E)\,|x|$$

Definition: Machine Epsilon/Unit Round-Off Error

The maximum relative error E for converting a real into a floating point system.

It is defined as the smallest value such that $fl(1+E) > 1$ under the given floating point system.

These definitions give a rule $fl(x) = x(1+\delta)$ for some $|\delta| \leq E$. δ may be positive or negative. E is defined as positive.

For a FP system $F = (\beta, t, L, U)$:

- Rounding to nearest: $E = \frac{1}{2}\beta^{1-t}$.
- Truncation: $E = \beta^{1-t}$.

Example: Find E for $F = (\beta = 10, t = 3, L = -5, U = 5)$.

Under round to nearest:

$$E = \frac{1}{2}(10)^{1-3} = 5 \times 10^{-3}$$

Consider the smallest representable number above 1. We have $1 = 0.100 \times 10^1$ in F. The next largest is 0.101×10^1. For $fl(1+E)$ to exceed 1, we must add $0.0005 \times 10^1 = 5 \times 10^{-3}$ to get halfway to the next number, where rounding occurs.

Under truncation:

$$E = 10^{1-3} = 10^{-2}$$

1.2 Arithmetic with Floating Point

IEEE standard requires that for $w, z \in F$,

$$w \oplus z = fl(w+z) = (w+z)(1+\delta)$$

where $\oplus$ is the floating point addition.

This rule only applies to *individual* FP operations. So it is not generally true that

$$(a \oplus b) \oplus c = a \oplus (b \oplus c) = fl(a+b+c)$$

The result is order-dependent and associativity is broken.

Consider the relative error of $(a \oplus b) \oplus c$ for $a, b, c \in F$.

$$\begin{aligned}
E_{rel} &= \frac{|(a \oplus b) \oplus c - (a + b + c)|}{|a + b + c|} \\
&= \frac{|(a + b)(1 + \delta_1) \oplus c - (a + b + c)|}{|a + b + c|} \\
&= \frac{|((a + b)(1 + \delta_1) + c)(1 + \delta_2) - a - b - c|}{|a + b + c|} \\
&= \frac{|a + b + c + (a + b)\delta_1 + (a + b + c + (a + b)\delta_1)\delta_2 - a - b - c|}{|a + b + c|} \\
&= \frac{|(a + b)\delta_1 + (a + b)\delta_1\delta_2 + (a + b + c)\delta_2|}{|a + b + c|} \\
&\leq \frac{|(a + b)\delta_1| + |(a + b)\delta_1\delta_2| + |(a + b + c)\delta_2|}{|a + b + c|} && \text{(Triangle inequality)} \\
&\leq \frac{|a + b|\,|\delta_1| + |a + b|\,|\delta_1\delta_2|}{|a + b + c|} + |\delta_2| \\
&\leq \frac{|a + b|\,E + |a + b|\,E^2}{|a + b + c|} + E && (|\delta| \leq E) \\
&\leq \frac{|a + b|}{|a + b + c|}(E + E^2) + E
\end{aligned}$$

This is the bound on the relative error for $(a \oplus b) \oplus c$. We can weaken the bound slightly to make it symmetric.

$$\begin{aligned}
E_{rel} &\leq \frac{|a + b|}{|a + b + c|}(E + E^2) + \frac{|a + b + c|}{|a + b + c|}E \\
&\leq \left(\frac{|a| + |b| + |c|}{|a + b + c|}\right)(E + E^2) + \left(\frac{|a| + |b| + |c|}{|a + b + c|}\right)E \\
&\leq \left(\frac{|a| + |b| + |c|}{|a + b + c|}\right)(2E + E^2)
\end{aligned}$$

This bound will apply to both $(a \oplus b) \oplus c$ and $a \oplus (b \oplus c)$, i.e. order does not matter.

$\frac{|a|+|b|+|c|}{|a+b+c|}$ is small when the denominator is small. This occurs when quantities have differing signs and similar magnitudes, leading to cancellation.

Catastrophic cancellation occurs when subtracting of about the same magnitude, when the input numbers contain error. All significant digits cancel out, so the result will have no correct digits.

1.3 Conditioning of Problems

Problems may be ill-conditioned or well-conditioned. Consider a problem P with input I and output O.

Definition: Well-Conditioned

If a change to the input ΔI gives a small change in the output ΔO, P is well-conditioned.

1.4 Stability of Algorithms

If any initial error in the data is magnified by an algorithm, the algorithm is considered numerically unstable.

Consider the integration problem

$$I_n = \int_0^1 \frac{x^n}{x+\alpha}\, dx$$

There is a recursive algorithm to solve it. For $n \geq 0$,

$$I_0 = \log \frac{1+\alpha}{\alpha}, I_n = \frac{1}{n} - \alpha I_{n-1}$$

Stability analysis: Consider the given recursive rule for I_n. Assume there is some initial error ε_0 in I_0.

$$\varepsilon_0 = (I_0)_A - (I_0)_E$$

We will ignore any subsequent floating point error introduced during the recursion. What is ε_n, the signed error after n steps? Does ε grow or shrink?

Exact and approximate solutions both follow the recursion, so

$$\begin{aligned}
\varepsilon_n &= (I_n)_A - (I_n)_E \\
&= \left(\frac{1}{n} - \alpha(I_{n-1})_A\right) - \left(\frac{1}{n} - \alpha(I_{n-1})_E\right) \\
&= -\alpha((I_{n-1})_A - (I_{n-1})_E) \\
&= -\alpha\varepsilon_{n-1}
\end{aligned}$$

We have a simple recursion for ε, so we can find the closed form.

$$\varepsilon_n = -\alpha\varepsilon_{n-1} = (-\alpha)^2\varepsilon_{n-2} = \cdots = (-\alpha)^n\varepsilon_0$$

The initial error is scaled by $(-\alpha)^n$ for n steps, so if $|\alpha| \leq 1$, the error does not grow so it is stable. If $|\alpha| > 1$, then the error grows and it is unstable.

Part II

Interpolation and Splines

Chapter 2

Interpolation

Interpolation Problem

Given a set of data points from an unknown function $y = p(x)$, approximate p's value at other points.

2.1 Polynomial Interpolation

Theorem (Unisolvence Theorem)

Given n data pairs (x_i, y_i) with distinct x_i, there is a unique polynomial $p(x)$ of degree $\leq n-1$ that interpolates the data.

For n points, find all the coefficients c_i of the generic polynomial

$$p(x) = c_1 + c_2 x + \cdots + c_n x^{n-1}$$

Each (x_i, y_i) gives one linear equation. We then solve the $n \times n$ linear system.

2.2 Vandermonde Matrices

In general, we get a linear system

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} c_1 \\ \cdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \cdots \\ y_n \end{bmatrix}$$

or

$$Vc = y$$

Definition: Vandermonde Matrix

V in the system $Vc = y$.

2.2.1 Monomial Basis

The polynomial $p(x) = \sum_{i=1}^{n} c_i x^{i-1}$ is the monomial form. The sequence $1, x, x^2, \ldots$ is the monomial basis.

2.3 Lagrange Basis

We let the polynomial have form

$$p(x) = \sum_{k=1}^{n} y_k L_k(x)$$

where the y's are the new coefficients (we have these y from the data points).

At each (x_i, y_i) data point, activate only the ith term, $y_i L_i(x)$. So for a data point x_1, we set $L_1(x_1) = 1$ and $L_{i\neq 1} = 0$, since $p(x_1) = y_1$. Repeat this for all data points. This gives us n equations and n unknowns, so it gives a unique polynomial for each L_i.

To solve for these L_i, we can either solve them directly or use this formula:

$$L_k(x) = \frac{(x - x_1)\cdots(x - x_{k-1})(x - x_{k+1})\cdots(x - x_n)}{(x_k - x_1)\cdots(x_k - x_{k-1})(x_k - x_{k+1})\cdots(x_k - x_n)}$$

E.g. Lagrange polynomial for 2 points. Find $L_1(x), L_2(x)$ and write $p(x)$ in terms of L_i's with points $(1, 2), (-1, 4)$.

$$L_1(x) = \frac{(x - (-1))}{(1 - (-1))} = \frac{x + 1}{2}, L_2(x) = \frac{(x - 1)}{(-1 - 1)} = \frac{1 - x}{2}$$

So

$$p(x) = 2\left(\frac{x + 1}{2}\right) + 4\left(\frac{1 - x}{2}\right) = x + 1 + 2 - 2x = 3 - x$$

Runge's phenomenon says that for higher degree polynomials for many points gives more oscillation. Piecewise polynomials will be used to fix this.

2.4 Piecewise Functions

We want continuity, so we try piecewise linear functions. This is done by fitting a line per pair of adjacent points.

Often piecewise linear is not satisfactory, we want smoothness (continuity of derivatives).

2.4.1 Hermite Interpolation

The problem of fitting a polynomial given function values and its derivative.

Typically we do cubic Hermite interpolation. If we have many points, we can do one cubic per pair of adjacent points. Sharing the derivative at points ensure first derivative (C^1) continuity.

Closed Form Hermite Interpolation: Data is $x_1, y_1, s_1, x_2, y_2, s_2$. We use the polynomial form

$$p(x) = a + b(x - x_1) + c(x - x_1)^2 + d(x - x_1)^3$$

which simplifies out expressions slightly.

The derivative is

$$p'(x) = b + 2c(x - x_1) + 3d(x - x_1)^2$$

Plugging in the data points:

$$\begin{aligned}
p(x_1) &= a = y_1 \\
p'(x_1) &= b = s_1 \\
p(x_2) &= a + b(x_2 - x_1) + c(x_2 - x_1)^2 + d(x_2 - x_1)^3 = y_2 \\
p'(x_2) &= b + 2c(x_2 - x_1) + 3d(x_2 - x_1)^2 = s_2
\end{aligned}$$

Solving the system gets

$$\begin{aligned}
a &= y_1 \\
b &= s_1 \\
c &= \frac{3y_1' - 2s_1 - s_2}{\Delta x_1} \\
d &= \frac{s_2 + s_1 - 2y_1'}{(\Delta x_1)^2}
\end{aligned}$$

where $\Delta x_1 = x_2 - x_1$ and $y_1' = \frac{y_2 - y_1}{\Delta x_1}$. We would use this to get the cubic for each interval.

Definition: Knot

Point where the interpolant transitions from one polynomial/interval to another.

Definition: Node

Point where some control points/data are specified.

For Hermite interpolation, these are the same. For other curve types, they can differ.

2.5 Cubic Spline Interpolation

If we are not given derivatives and we want to fit a cubic, we need more data than just the two points.

Approach: Fit a cubic $S_i(x)$ on each interval, but now require matching first and second derivatives (C^2 continuity) between intervals.

Definition: Interpolating Conditions

On the interval $[x_i, x_{i+1}]$, $S_i(x_i) = y_i$ and $S_i(x_{i+1}) = y_{i+1}$.

Interval endpoint values match.

Definition: Derivative Conditions

At each interior point x_{i+1}, $S_i'(x_{i+1}) = S_{i+1}'(x_{i+1})$ and $S_i''(x_{i+1}) = S_{i+1}''(x_{i+1})$.

Interior knot first and second derivatives match across intervals.

Assuming n data point, and a cubic which has 4 unknowns for each $n-1$ intervals has $4n-4$ unknowns.

Assuming n data points, we have 2 endpoint interpolating conditions per interval, so $2n-2$ equations and 2 derivative conditions per interior point, so $2n-4$ equations. We have a total of $4n-6$ equations.

Since $4n-6 < 4n-4$, there is not enough information for a unique solution. We need 2 more equations, usually at domain endpoints called boundary conditions or end conditions.

Definition: Clamped Boundary Conditions

Slope set to a specific value.

Definition: Free Boundary Conditions

Second derivative is set to 0.

Basic algorithms (Gaussian elimination) for linear systems take $O(n^3)$ time for n unknowns. For the special case of cubic splines, we can do $O(n)$.

Chapter 3

Efficient Construction of Cubic Splines

3.1 Cubic Splines via Hermite Interpolation

Use Hermite interpolation as a stepping stone to build a cubic spline.

1. Express unknown polynomials with closed form Hermite equations.
2. Treat s_i as unknowns.
3. Solve for s_i that give continuous second derivatives (force it to satisfy cubic spline).
4. Given s_i, plug into closed form Hermite equations to recover polynomial coefficients a_i, b_i, c_i, d_i.

For cubic splines, we had three types of conditions:

1. Values match data at all interval endpoints.
2. First derivatives match at interior points.
3. Second derivatives match at interior points.

1 and 2 are already satisfied by using the Hermite closed form. 3 must be enforced explicitly by solving for s_i's that cause $S_i''(x) = S_{i+1}''(x)$.

Formulation: Start from the Hermite closed form on the ith interval.

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

Its 2nd derivative is

$$S_i''(x) = 2c_i + 6d_i(x - x_i) = 2\left(\frac{3y_i' - 2s_i - s_{i+1}}{\Delta x_i}\right) + 6\left(\frac{s_{i+1} + s_i - 2y_i'}{(\Delta x_i)^2}\right)(x - x_i)$$

To force matching 2nd derivatives at the knot/node between intervals i and $i+1$, set $S_i''(x_{i+1}) = S_{i+1}''(x_{i+1})$ for $i = 1$ to $n-2$.

$$S_i''(x_{i+1}) = \frac{2(3y_i' - 2s_i - s_{i+1})}{\Delta x_i} + \frac{6(s_i + s_{i+1} - 2y_i')}{(\Delta x_i)^2}(\underbrace{x_{i+1} - x_i}_{\Delta} x_i)$$

$$S_{i+1}''(x_{i+1}) = \frac{2(3y_{i+1}' - 2s_{i+1} - s_{i+2})}{\Delta x_{i+1}} + \frac{6(s_{i+1} + s_{i+2} - 2y_{i+1}')}{(\Delta x_{i+1})^2}(\underbrace{x_{i+1} - x_{i+1}}_{0})$$

Equate to get conditions on slopes s_i at knots.

$$\frac{2s_i + 4s_{i+1} - 6y_i'}{\Delta x_i} = \frac{6y_{i+1}' - 4s_{i+1} - 2s_{i+2}}{\Delta x_{i+1}}$$

Cross-multiply and bring s_i's tot he LHS.

$$\Delta x_{i+1} + 2(\Delta x_i + \Delta x_{i+1})s_{i+1} + \Delta x_i s_{i+2} = 3(\Delta x_{i+1} + y_i' + \Delta x_i y_{i+1}')$$

Reindex i to $i-1$ so we have one equation per interior node i. For $i = 2$ to $n-1$ (each inner node),

$$\Delta x_i s_{i-1} + 2(\Delta x_{i-1} + \Delta x_i)s_i + \Delta x_{i-1} s_{i+1} = 3(\Delta x_i y_{i-1}' + \Delta x_{i-1} y_i')$$

All that remains are the boundary conditions for the slopes at the endpoints.

Boundary Conditions: We need conditions for $i = 1$ and $i = n$. Since we have n unknowns (the s_i)

1. Clamped case (given slope): $S_i'(x_i) = s_i^*$ and $S_{n-1}'(x_n) = s_n^*$ where s_1^*, s_n^* are given.
2. Natural/Free: Zero 2nd derivative at ends $S_1''(x_1) = 0, S_{n-1}''(x_n) = 0$.

   Earlier expression for 2nd derivative gives us

   $$S_1''(x_1) = 2c_1 + 6d_1(x_1 - x_1) = 0$$

   So, $c_1 = \frac{3y_1' - 2s_1 - s_2}{\Delta x_1} = 0$ which gets

   $$s_1 + \frac{1}{2}s_2 = \frac{3}{2}y_1'$$

   Likewise for $S_{n-1}''(x_n) = 0$, $S_{n-1}''(x_n) = 2c_{n-1} + 6d_{n-1}(x_n - x_{n-1})$. Expand c and d and solve we get

   $$\frac{1}{2}s_{n-1} + s_n = \frac{3}{2}y_{n-1}'$$

Efficient Splines Summary

At interior nodes ($i = 1, \ldots, n-1$):

$$\Delta x_i s_{i-1} + 2(\Delta x_{i-1} + \Delta x_i)s_i + \Delta x_{i-1} s_{i+1} = 3(\Delta x_i y'_{i-1} + \Delta x_{i-1} y'_i)$$

Clamped BC ($i = 1$ or n):

$$s_1 = s_1^*, s_n = s_n^*$$

Free BC ($i = 1$ or n):

$$s_1 + \frac{s_2}{2} = \frac{3}{2} y'_1, \frac{s_{n-1}}{2} + s_n = \frac{3}{2} y'_{n-1}$$

where $\Delta x_i = x_{i+1} - x_i$ and $y'_i = \frac{y_{i+1} - y_i}{\Delta x_i}$.

Example: What is the linear system for s_i to fit a spline to the 4 points $(0, 1), (2, 1), (3, 3), (4, -1)$ with clamped BC of $S'_1(x_1) = 1$ and $S'_3(x_4) = -1$.

Compute all Δx_i and y_i' for $i = 1$ to 3 (intervals):

- $\Delta x_1 = 2 - 0 = 2, \Delta x_2 = 3 - 2 = 1, \Delta x_3 = 4 - 3 = 1$
- $y'_1 = \frac{1-1}{2} = 0, y'_2 = \frac{3-1}{1} = 2, y'_3 = \frac{-1-3}{1} = -4$

Find the equations (rows) for $i = 1$ to 4 (one per node):

- $i = 1$: $s_1 = s_1^* = S'_1(x_1) = 1$ so $T_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}, r_1 = 1$
- $i = 2$: is interior

$$\Delta x_2 s_1 + 2(\Delta x_2 + \Delta x_1)s_2 + 2s_3 = 3(\Delta x_2 y'_1 + \Delta x_1 y'_2)$$
$$s_1 + 2(1+2)s_2 + 2s_3 = 3(1 \cdot 0 + 2 \cdot 2)$$
$$s_1 + 6s_2 + 2s_3 = 12$$

  so $T_2 = \begin{bmatrix} 1 & 6 & 2 & 0 \end{bmatrix}, r_2 = 12$

Likewise for $i = 3$ and 4. The system $Ts = r$ is then

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 6 & 2 & 0 \\ 0 & 1 & 4 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} s = \begin{bmatrix} 1 \\ 12 \\ -6 \\ -1 \end{bmatrix}$$

Solve for s, then plug into Hermite closed form to recover a, b, c, d for each cubic.

The new system has one equation per node, so the matrix size is $n \times n$. The matrix is called *tridiagonal*. Only the entries on the diagonal and its two neighbours are ever non-zero. To store this, we can store three vectors, one per diagonal to save memory.

Tridiagonal systems can be solved faster than general systems. We can solve it in $O(n)$.

Chapter 4

Parametric Curves

Let x and y each be separate functions of a new parameter t. Then a point's position is given by the vector $P(t) = (x(t), y(t))$.

We say a curve is parameterized by t, the (x, y) position on the curve is dictated by parameter t.

The simple line $y = 3x + 2$ can be equivalently described by the functions $x(t) = t$ and $y(t) = 3t + 2$.

Consider a curve along a semi-circle in the upper half plant, oriented from $(1, 0)$ to $(-1, 0)$. The usual implicit equation is $x^2 + y^2 = 1$. One parametric form is $x(t) = \cos(\pi t)$ and $y(t) = \sin(\pi t)$ for $0 \leq t \leq 1$.

A given curve can be parameterized in different ways and also at different speeds/rates.

Different way: $x(t) = \cos(\pi(1 - t)), y(t) = \sin(\pi(1 - t))$.

Different speed: $x(t) = \cos(\pi t^2), y(t) = \sin(\pi t^2)$.

4.1 Arc-Length Parameterization

A common standard parameterization is to choose t as the distance along the curve. This gives a unit speed traversal of the curve.

4.2 Parametric Curves using Interpolation

We can combine all the existing interpolant types with parametric curves, by considering $x(t)$ and $y(t)$ separately.

- Use two Lagrange polynomials $x(t), y(t)$ to fit a small set of (t_i, x_i, y_i) point data.

- Use Hermite interpolation for $x(t), y(t)$ given $(t_i, x_i, y_i, sx_i, sy_i)$ point/derivative data for many points.

4.2.1 Parameterization for Piecewise Polynomials

Given only ordered (x_i, y_i) point data, we don't have a parameterization. We need data for t to form (t, x_i) and (t, y_i) pairs to fit curves to.

Option 1: Use the node index as the parameterization, i.e. $t_i = i$ at each node.

Option 2: (Approximate arc-length parameterization) Set $t_1 = 0$ at first node, then recursively compute

$$t_{i+1} = t_i + \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

Part III

Ordinary Differential Equations

Chapter 5

Introduction and Forward Euler

Definition: Ordinary Differential Equation

A relationship between the variable y and its derivative y', given by a known function f:

$$y'(t) = f(t, y(t))$$

Simple Population Model: Consider the population changes as

$$y'(t) = a \cdot y(t)$$

In this case, there is a closed-form solution for initial population $y_0 = y(t_0)$ which is

$$y(t) = y_0 e^{a(t-t_0)}$$

A more complex model is

$$y'(t) = y(t)(a - b \cdot y(t))$$

where b term expresses the effect of resource limits.

For small $y(t)$, there is exponential growth, while when $y(t) \approx \frac{a}{b}$, then $y'(t) \approx 0$. We get the closed form called logistic growth.

Definition: Logistic Growth

$$y(t) = \frac{a y_0 e^{a(t-t_0)}}{b y_0 e^{a(t-t_0)} + (a - y_0 b)}$$

Simple models often lack closed-form solutions, so we develop numerical methods to find approximate solutions.

5.1 Initial Value Problem

Definition: Initial Value Problem

The general form is a differential equation

$$y'(t) = f(t, y(t))$$

where f is specified, and the initial values are

$$y(t_0) = y_0$$

f is called the Dynamics Function and y_0 is the initial condition.

We can have systems of differential equations (more unknown variables) or higher order differential equations (higher order derivatives).

5.2 Solving ODE

For approximate solutions, the numerical solution is a discrete set of time/value pairs (t_i, y_i). y_i should approximate the true value $y(t_i)$.

5.2.1 Time-Stepping

Given initial conditions, we repeatedly step sequentially forward to the next time instant, using the derivative info, y', and a timestep, h.

Time-Stepping

Set $n = 0, t = t_0, y = y_0$

1. Compute y_{n+1}
2. Increment time $t_{n+1} = t_n + h$
3. $n = n + 1$
4. Repeat

We need to figure out how to compute y_{n+1}. There are several varieties of time-stepping:

- Single-step vs. multi-step: use information at current time or previous timesteps.
- Explicit vs. implicit: is y_{n+1} given as an explicit function to evaluate, or as an implicit equation.
- Timestep size: using a constant or variable timestep h

5.2.2 Forward Euler

It is an explicit, single-step scheme. Compute the current slope $y'_n = f(t_n, y_n)$ and step in a straight line with that slope:

$$y_{n+1} = y_n + h \cdot y'_n$$

Forward Euler

$$y_{n+1} = y_n + hf(t_n, y_n)$$

This gets a recurrence relation to approximate the functions.

Example: Consider the simple IVP $y'(t) = 2y(t)$ with initial conditions at $t_0 = 1$ of $y(t_0) = 3$.

$2y(t)$ is the dynamics function. The generic recurrence for Forward Euler with step size $h = 1$ is

$$y_{n+1} = y_n + 2y_n = 3y_n$$

n	t_n	y_n	$y(t_n)$
0	1	3	3
1	2	9	22.2
2	3	27	163.8
3	4	81	1210
4	5	243	8943

In general, a smaller timestep results in more accurate solutions.

Notation: $y(t_n)$ is the exact value and y_n is the approximate/discrete data at step n.

Note: When programming this, do not simplify the Forward Euler rule, instead leave f as a black box.

Time stepping is also known as time-integration. Forward Euler approximates the area under the derivative curve evaluated using left rectangles.

5.3 Forward Euler for Systems of Equations

We can view the systems as a vector dynamics function with 2 initial conditions. The number of rows is the number of components.

Example: Consider a particle with coordinates $(x(t), y(t))$ satisfying the ODE system

$$x'(t) = -y(t), y'(t) = x(t)$$

with $x(t_0) = 2, y(t_0) = 0$ and $t_0 = 2$ using timestep $h = 2$.

In vector form,:

$$\begin{bmatrix} x(t) \\ y(t) \end{bmatrix}' = \begin{bmatrix} -y(t) \\ x(t) \end{bmatrix}$$

and the generic Forward Euler is

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \end{bmatrix} + 2 \begin{bmatrix} -y_n \\ x_n \end{bmatrix} = \begin{bmatrix} x_n - 2y_n \\ y_n + 2x_n \end{bmatrix}$$

n	t_n	x_n	y_n
0	2	2	0
1	4	2	4
2	6	-6	8
3	8	-22	-4

5.4 Deriving Forward Euler

To justify Forward Euler more rigorously, we use Taylor series. Recall that it approximates a function in some neighbourhood using an infinite weighted sum of its derivatives.

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots$$

There are two ways to arrive at Forward Euler:

1. Finite difference.
2. Taylor series.

5.4.1 Finite Difference View

Approximate the derivative y' using a finite difference. The ODE form is $y'(t) = f(t, y(t))$. The finite difference approximation of y' gives

$$y'(t_n) \approx \frac{y_{n+1} - y_n}{t_{n+1} - t_n} = \frac{y_{n+1} - y_n}{h} = f(t_n, y_n)$$

Rearranging gets the Forward Euler

$$y_{n+1} = y_n + hf(t_n, y_n)$$

5.4.2 Taylor Series View

Truncate a Taylor series by discarding high order terms. We want to approximate $y(t_{n+1})$ based on t_n, so we set $f \to y, a \to t_n, x \to t_{n+1}$ and plug in

$$\begin{aligned} y(t_{n+1}) &= y(t_n) + y'(t_n)\frac{t_{n+1}-t_n}{n} + \frac{y''(t_n)}{2}(t_{n+1}-t_n)^2 + \frac{y'''(t_n)}{6}(t_{n+1}-t_n)^3 + \cdots \\ &= y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(t_n) + \frac{h^3}{6}f'''(t_n) + \cdots \qquad (h = t_{n+1}-t_n) \\ &\approx y_n + hf(t_n, y_n) \end{aligned}$$

We can drop all terms of order h^2 or higher since they will be smaller. Replace y' with an evaluation of the dynamics function f. This gives the Forward Euler.

5.5 Higher Order Time Stepping Schemes

5.5.1 Error of Timestepping Schemes

The absolute error at step n is $|y_n - y(t_n)|$. Since we can't measure error exactly without knowing $y(t)$, we rely on the Taylor series.

Forward Euler makes a linear approximation at each step, incurring a local error. Smaller step size h means more frequent slope estimates, which results in less error.

The Forward Euler is $y_{n+1} = y_n + hf(t_n, y_n)$ and we have the Taylor series. The error for one step is the difference between the two, assume exact data at time t_n is $y_n = y(t_n)$. The expression for Forward Euler is $y_{n+1} = y(t_n) + hy'(t_n)$. The difference is

$$y_{n+1} - y(t_{n+1}) = -\frac{h^2}{2}y''(t_n) + O(h^3) = O(h^2)$$

Definition: Local Truncation Error

$$y_{n+1} - y(t_{n+1}) = -\frac{h^2}{2}y''(t_n) + O(h^3) = O(h^2)$$

We earlier used finite difference approximation to derive Forward Euler

$$y'(t_n) \approx \frac{y_{n+1} - y_n}{h}$$

We can determine the error of this estimate by Taylor series too.

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + O(h^2)$$

Rearranging we get the derivative approximation.

Definition: Derivative Approximation Error

$$y'(t_n) = \frac{y(t_{n+1}) - y(t_n)}{h} + O(h)$$

So the derivative approximation is $O(h)$.

The Taylor expansion hints at how to derive higher order schemes. Keeping more terms in the series, so the error is higher order (i.e. smaller).

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(t_n) + O(h^3)$$

We do not know y''. Our ODE only gives the first derivative. We can use a finite difference to approximate y''.

$$y''(t_n) = \frac{y'(t_{n+1}) - y'(t_n)}{h} + O(h)$$

5.6 Trapezoidal Rule

The Taylor series is $y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(t_n) + O(h^3)$. Approximate $y''(t) = \frac{y'(t_{n+1}) - y'(t_n)}{h} + O(h)$. Plug in:

$$\begin{aligned}
y(t_{n+1}) &= y(t_n) + hy'(t_n) + \frac{h^2}{2}\left(\frac{y'(t_{n+1}) - y'(t_n)}{h} + O(h)\right) + O(h^3) \\
&= y(t_n) + hy'(t_n) + \frac{h^2}{2} \cdot \frac{y'(t_{n+1})}{h} - \frac{h^2}{2} \cdot \frac{y'(t_n)}{h} + O(h^3) \\
&= y(t_n) + \frac{h}{2}\left(y'(t_n) + y'(t_{n+1})\right) + O(h^3) \\
&= y(t_n) + \frac{h}{2}\left(f(t_n, y(t_n)) + f(t_{n+1}, y(t_{n+1}))\right) + O(h^3) \\
y_{n+1} &= y_n + \frac{h}{2}\left(f(t_n, y(t_n)) + f(t_{n+1}, y(t_{n+1}))\right) + O(h^3)
\end{aligned}$$

The local truncation error for trapezoidal rule is $O(h^3)$.

Geometric intuition: Evaluate the slope $y' = f$ at the start and end of the timestep, and step along the average slope.

Trapezoidal Example: ODE system

$$x'(t) = -y(t), y'(t) = x(t)$$

with initial conditions $x(t_0) = 2, y(t_0) = 0, t_0 = 0$ with timestep $h = 2$ to find x, y at $t = 4$.

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \end{bmatrix} + \frac{2}{2}\left(\begin{bmatrix} -y_n \\ x_n \end{bmatrix} + \begin{bmatrix} -y_{n+1} \\ x_{n+1} \end{bmatrix}\right) = \begin{bmatrix} x_n - y_n - y_{n+1} \\ y_n + x_n + x_{n+1} \end{bmatrix}$$

This is a linear system of equations to solve per timestep, since trapezoidal is implicit.

$$\begin{bmatrix} x_{n+1} + y_{n+1} \\ -x_{n+1} + y_{n+1} \end{bmatrix} = \begin{bmatrix} x_n - y_n \\ x_n + y_n \end{bmatrix}$$
$$\begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} x_n - y_n \\ x_n + y_n \end{bmatrix}$$

5.7 Explicit vs. Implicit Schemes

Forward Euler is an explicit scheme, since no y_{n+1} term appears on the RHS. Trapezoidal is an implicit scheme, since there is a y_{n+1} term on the RHS.

We can derive another scheme that avoids the implicitness of trapezoidal.

1. Evaluate the slope at start, $y'(t_n)$.
2. Take Forward Euler step to estimate the endpoint.
3. Evaluate the slope there, to approximate the true end-of-step slope $y'(t_{n+1})$.
4. Use this slope estimate in the trapezoidal formula, i.e., step along the average of start and approximate end slopes.

5.8 Improved Euler

Definition: Improved Euler

1. $y^*_{n+1} = y_n + hf(t_n, y_n)$
2. $y_{n+1} = y_n + \frac{h}{2}(f(t_n, y_n) + f(t_{n+1}, y^*_{n+1}))$

Improved Euler is an explicit scheme. Like trapezoidal, improved Euler has local truncation error $O(h^3)$.

Improved Euler derivation: Trapezoidal gave

$$y(t_{n+1}) = y(t_n) + \frac{h}{2}[f(t_n, y(t_n)) + f(t_{n+1}, y(t_{n+1}))] + O(h^3)$$

Forward Euler gave

$$y(t_{n+1}) = y(t_n) + hf(t_n, y(t_n)) + O(h^2)$$

Let $y^*_{n+1} = y(t_n) + hf(t_n, y(t_n))$, so this first sub-step has error $y(t_{n+1}) - y^*_{n+1} = O(h^2)$.

The single variable Taylor series

$$f(x+h) = f(x) + h\frac{\partial f}{\partial x} + O(h^2)$$

If we have a function of multiple variables, we can Taylor expand in one variable

$$f(x, y+h_y) = f(x,y) + h_y\frac{\partial f}{\partial y} + O(h_y^2)$$

or multiple variables:

$$f(x+h_x, y+h_x) = f(x,y) + h_x\frac{\partial f}{\partial x} + h_y\frac{\partial f}{\partial y} + O(h_x^2) + O(h_y^2) + O(h_x h_y)$$

We Taylor expand f around y^*_{n+1} to get

$$f(t_{n+1}, y(t_{n+1})) = f(t_{n+1}, y^*_{n+1}) + \frac{\partial f}{\partial y}(t_{n+1}, y^*_{n+1})\underbrace{(y(t_{n+1}) - y^*_{n+1})}_{O(h^2)} + O(\underbrace{(y(tn+1) - y_{n+1})^2}_{O(h^4)})$$

Therefore, $f(t_{n+1}, y(t_{n+1})) = f(t_{n+1}, y^*_{n+1}) + O(h^2)$. Use this approximate end of step slope in the trapezoidal expression.

$$\begin{aligned} y(t_{n+1}) &= y(t_n) + \frac{h}{2}[f(t_n, y(t_n)) + O(h^2)] + O(h^3) \\ &= y(t_n) + \frac{h}{2}[f(t_n, y(t_n)) + f(t_{n+1}, y^*_{n+1})] + O(h^3) \end{aligned}$$

5.9 Review

- Forward Euler (Explicit): Use slope at starting point, has LTE $O(h^2)$.
- Trapezoidal (Implicit): Use average of slope at start and end of step, has LTE $O(h^3)$.
- Improved Euler (Explicit): Use average of slope at start and approximate end, has LTE $O(h^3)$.

Improved Euler Example: ODE system

$$x'(t) = -y(t), y'(t) = x(t)$$

with initial conditions $x(t_0) = 2, y(t_0) = 0, t_0 = 0$ with timestep $h = 2$ to find x, y at $t = 4$.

FE-like step:

$$\begin{bmatrix} x^*_{n+1} \\ y^*_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \end{bmatrix} + 2\begin{bmatrix} -y_n \\ x_n \end{bmatrix} = \begin{bmatrix} x_n - 2y_n \\ y_n + 2x_n \end{bmatrix}$$

Trapezoidal-like step:

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \end{bmatrix} + \frac{2}{2}\left(\begin{bmatrix} -y_n \\ x_n \end{bmatrix} + \begin{bmatrix} -y^*_{n+1} \\ x^*_{n+1} \end{bmatrix}\right) = \begin{bmatrix} x_n - y_n - y^*_{n+1} \\ y_n + x_n + x^*_{n+1} \end{bmatrix}$$

Apply these for each timestep with the initial conditions.

n	t_n	x_n	y_n	x_n^*	y_n^*
0	0	2	0	NA	NA
1	2	-2	4	2	4
2	4	-6	-8	-10	0

5.10 Global Error

For a constant step h, computing from t_0 to end time t_{final}

$$\text{number of steps} = \frac{t_{final} - t_0}{h} = O(h^{-1})$$

Then for a given method we have:

$$\text{Global error} \leq \text{Local error} \cdot O(h^{-1})$$

Forward Euler: $O(h^2) \cdot O(h^{-1}) = O(h)$.

Improved Euler/Trapezoidal: $O(h^3) \cdot O(h^{-1}) = O(h^2)$.

5.11 Multistep Schemes

Multistep methods use information from earlier timesteps $(t_{n-1}, t_{n-2}, \dots)$. One way to derive them is by fitting polynomials to current and earlier data points, for better slope estimates.

5.11.1 Backwards Differentiation Formulas

This is an implicit multistep scheme (BDF1, BDF2, etc.). It generalizes backwards Euler. In fact, BDF1 is backward Euler.

BDF2: Uses current and previous step data:

$$y_{n+1} = \frac{4}{3}y_n - \frac{1}{3}y_{n-1} + \frac{2}{3}hf(t_{n+1}, y_{n+1})$$

BDF2 has local truncation error $O(h^3)$.

Deriving BDF methods via Interpolation:

1. Fit an interpolant $p(t)$ with Lagrange polynomials to the unknown point (t_{n+1}, y_{n+1}) and one or more earlier points.
2. Determine its derivative $p'(t)$ by differentiating.
3. Require end-of-step slope to match $p'(t_{n+1}) = f(t_{n+1}, y_{n+1})$.

4. Rearranging for unknown y_{n+1} gives a BDF scheme.

Deriving Backwards Euler via Interpolation: Fit a Lagrange polynomial $p(t)$ to (t_n, y_n) and (t_{n+1}, y_{n+1}).

$$p(t) = y_n \left(\frac{t - t_{n+1}}{t_n - t_{n+1}} \right) + y_{n+1} \left(\frac{t - t_n}{t_{n+1} - t_n} \right) = \frac{y_n}{-h}(t - t_{n+1}) + \frac{y_{n+1}}{h}(t - t_n)$$

Differentiating with respect to t:

$$p'(t) = \frac{y_{n+1} - y_n}{h}$$

gives the line's slope. Require it to match the end of step slope at t_{n+1} given by $f(t_{n+1}, y_{n+1})$. So

$$\frac{y_{n+1} - y_n}{h} = f(t_{n+1}, y_{n+1}) \implies y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$$

5.11.2 Adams-Bashforth

This is an explicit multistep scheme:

$$y_{n+1} = y_n + \frac{3}{2}hf(t_n, y_n) - \frac{1}{2}hf(t_{n-1}, y_{n-1})$$

5.12 Higher Order ODE

We often encounter ODEs with higher order derivatives.

Definition: Order

The order of an ODE is the highest derivative that appears.

Definition: Higher Order ODE

The general form is

$$y^{(n)}(t) = f(t, y(t), y'(t), y''(t), \ldots, y^{(n-1)}(t))$$

We can convert them to systems of first order ODEs.

5.12.1 Converting to First Order Systems

Converting a high-order ODE

For each variable y with more than a first derivative, introduce new variables

$$y_i = y^{(i-1)}$$

for $i = 1$ to n. Substituting the new variables into the original ODE leads to

1. One first order equation for each original equation.
2. One or more additional equations relating the new variables.

Example: Consider the ODE

$$y''(t) = ty(t)$$

with initial conditions $y(1) = 1, y'(1) = 2$.

We introduce variables $y_i = y^{(i-1)}$ for $i = 1$ to 2:

$$y_1(t) = y^{(0)}(t) = y(t), y_2(t) = y^{(1)}(t) = y'(t)$$

Plugging in gives us a system of first order ODEs:

$$y_2'(t) = ty_1(t)$$
$$y_1'(t) = y_2(t)$$

with initial conditions $y_1(1) = 1, y_2(1) = 2$.

Example: Convert $y^{(4)}(t) - 3y'(t)y'''(t) + \sin(ty''(t)) - 7t(y(t))^2 = e^t$ to first order.

New variables $y_i = y^{(i-1)}$ for $i = 1$ to 4.

$$y_1(t) = y(t), y_2(t) = y'(t), y_3(t) = y''(t), y_4(t) = y'''(t)$$

Plugging back in

$$y_4'(t) = 3y_2(t)y_4(t) - \sin(ty_3(t)) + 7t(y_1(t))^2 + e^t$$

with the relationships between the new variables

$$y_3'(t) = y_4(t), y_2'(t) = y_3(t), y_1'(t) = y_2(t)$$

We can define $z = \begin{bmatrix} y_1 & y_2 & y_3 & y_4 \end{bmatrix}^T$, so $z'(t) = f(t, z)$ where

$$z' = \begin{bmatrix} y_1 & y_2 & y_3 & y_4 \end{bmatrix}^{T'}$$

Example: Convert

$$x''(t) + y'(t)x(t) + 2t = 0$$
$$y''(t) + [y(t)]^2x(t)t = 0$$

We define new variables $x_i = x^{(i-1)}$ and $y_i = y^{(i-1)}$ for $i = 1$ to 2.

$$x_1(t) = x(t)$$
$$x_2(t) = x'(t)$$
$$y_1(t) = y(t)$$
$$y_2(t) = y'(t)$$

Plug in to get

$$x_2'(t) = -y_2(t)x_1(t) - 2t$$
$$y_2'(t) = -[y_1(t)]^2 x_1(t)t$$

and the new relationships

$$x_1'(t) = x_2(t)$$
$$y_1'(t) = y_2(t)$$

5.13 Stability Analysis

Errors are generally $O(h^p)$ for some p, the schemes are less accurate for large h.

Error/perturbations in initial conditions may lead to vastly different or incorrect answers, i.e. $y(0) = y_0 + \varepsilon$. If some initial error ε_0 grows exponentially for many steps, our timestepping scheme is unstable.

Definition: Test Equation

$$y'(t) = -\lambda y(t), y(0) = y_0$$

for a constant $\lambda > 0$.

The exact solution is $y(t) = y_0 e^{-\lambda t}$. This tends to 0 as $t \to \infty$.

Stability Analysis Process

1. Apply a given time stepping scheme to our test equation.
2. Find the closed form of its numerical solution and error behaviour.
3. Find the conditions on the timestep h that ensure stability (when error approaches 0 for $n \to \infty$).

Stability Analysis of Forward Euler: The forward Euler rule is $y_{n+1} = y_n + hf(t_n, y_n)$. Apply the forward Euler to the test equation:

$$y_{n+1} = y_n + h(-\lambda y_n) = y_n(1 - h\lambda)$$

The closed form is $y_n = y_0(1-h\lambda)^n$. The true solution has $y(t) = y_0 e^{-\lambda t}$, decaying to zero. The forward Euler solution goes to zero when

$$|1 - h\lambda| < 1$$

We have that $0 < h < \frac{2}{\lambda}$. Hence the numerical solution decays to zero when $0 < h < \frac{2}{\lambda}$. Otherwise, it is unstable.

If we perturb the initial solution with error ε_0, we have

$$y_0^{(p)} = y_0 + \varepsilon_0, y_n^{(p)} = (y_0 + \varepsilon_0)(1 - h\lambda)^n$$

Therefore, the error is $\varepsilon_n = y_n^{(p)} - y_n = \varepsilon_0(1 - h\lambda)^n$.

Definition: Conditionally Stable

The scheme is conditionally stable if the stability condition is satisfied.

Improved Euler Stability: We needed $\left|1 - h\lambda + \frac{h^2\lambda^2}{2}\right| < 1$ for stability.

For the left side: $-1 < 1 - h\lambda + \frac{h^2\lambda^2}{2}$ implies $0 < 2 - h\lambda + \frac{h^2\lambda^2}{2}$. This quadratic has no real roots so the inequality is always satisfied for any h.

For the right side: $1 - h\lambda + \frac{h^2\lambda^2}{2} < 1$ gives $-h\lambda + \frac{h^2\lambda^2}{2} < 0$. $h\lambda$ is always positive, so this is satisfied when $h < \frac{2}{\lambda}$ which is the same stability conditions as Forward Euler.

Forward Euler on Test Equation:

- $h > 2/\lambda$ is unstable: numerical solution rapidly blows up.
- $h = 2/\lambda$ is borderline stable: numerical solution remains constant, without decreasing to zero.
- $h < 2/\lambda$ is stable, but inaccurate: numerical solution decreases to zero, but still very inaccurate and oscillatory.
- $h \ll 2/\lambda$ is stable and more accurate: numerical solution begins to approximate the true solution.

Backward Euler on Test Equation: When $h > 2\lambda$, it is still stable.

Stability of Improved Euler: We plug in the test equation into the scheme:

$$y_{n+1}^* = y_n + h(-\lambda y_n)$$
$$y_{n+1} = y_n + \frac{h}{2}(-\lambda y_n - \lambda y_{n+1}^*)$$

Plug in the 2nd equation into 1:

$$\begin{aligned} y_{n+1} &= y_n - \frac{h\lambda}{2} y_n - \frac{h\lambda}{2}(y_n - h\lambda y_n) \\ &= y_n - h\lambda y_n + \frac{h^2\lambda^2}{2} y_n \\ &= y_n \left(1 - h\lambda + \frac{h^2\lambda^2}{2}\right) \end{aligned}$$

We have the closed form $y_n = y_0 \left(1 - h\lambda + \frac{h^2\lambda^2}{2}\right)^n$. The error will like wise follow:

$$\varepsilon_n = \varepsilon_0 \left(1 - h\lambda + \frac{h^2\lambda^2}{2}\right)^n$$

So this is stable when $\left|1 - h\lambda + \frac{h^2\lambda^2}{2}\right| < 1$ which yields $0 < h < \frac{2}{\lambda}$ for stability. This is the same as forward Euler.

Stability in General: For our linear test equation $y' = -\lambda y$, forward Euler gave a bound related to λ, or the the derivative of f with respect to y: $\lambda = \left|\frac{\partial}{\partial y}(-\lambda y)\right| = \left|\frac{\partial f}{\partial y}\right|$.

For nonlinear problems, local linear stability depends on $\frac{\partial f}{\partial y}$ evaluated at a given point in time/space.

For systems of ODEs, stability relates to the eigenvalues of the Jacobian matrix $\frac{\partial^3(f_1,\ldots,f_n)}{\partial(y_1 \partial \ldots \partial y_n)}$.

5.13.1 Truncation Error vs. Stability

1. Stability tells us what our numerical algorithm itself does to small errors/perturbations.
2. Truncation error tells us how the accuracy of our numerical solution scales with timestep h.

5.14 Local Truncation Error

We want to determine the local truncation error in general, for methods we have not seen before.

Recall that the local truncation error is

$$LTE = y(t_{n+1}) - y_{n+1}$$

where $y(t_{n+1})$ is the exact solution and y_{n+1} is the approximate solution.

Local Truncation Error Process

Given a timestepping scheme, $y_{n+1} = \text{RHS}$:

1. Replace approximations on RHS with exact versions. ($y_n \to y(t_n)$, $f(t_{n+1}, y_{n+1}) \to y'(t_{n+1})$)
2. Taylor expand all RHS quantities about time t_n.
3. Taylor expand the exact solution $y(t_{n+1})$ to compare against.
4. Compute difference $y(t_{n+1}) - y_{n+1}$. Lowest degree non-cancelling power of h determines the local truncation error.

Sometimes, we may only want the order $O(h^p)$ or we may want the actual expression $Cy^{(\beta)}(t)h^p + O(h^{p+1})$.

Forward Euler Example: $y_{n+1} = y_n + hf(t_n, y_n)$.

1. Replace y_n with $y(t_n)$ and f with y'.

$$y_{n+1} = y(t_n) + hf(t_n, y(t_n)) = y(t_n) + hy'(t_n)$$

2. Nothing to Taylor expand on RHS since everything is already at time t_n.
3. Exact solution Taylor expands to

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(t_n) + O(h^3)$$

4. Difference is $LTE = y(t_{n+1} - y_{n+1}) = \frac{h^2}{2}y''(t_n) + O(h^3) = O(h^2)$.

We may not know in advance how many Taylor series terms we will need. Try expanding up to some order; if all terms match the Taylor series, redo using more terms until no cancellation of the coefficients.

Truncation Error of Trapezoidal: Recall Trapezoidal is $y_{n+1} = y_n + \frac{h}{2}(f(t_n, y_n) + f(t_{n+1}, y_{n+1}))$.

1. Replace RHS quantities with exact counterparts:

$$y_{n+1} = y(t_n) + \frac{h}{2}(y'(t_n) + y'(t_{n+1}))$$

2. Taylor expand RHS quantities about time t_n:

$$y'(t_{n+1}) = y'(t_n) + hy''(t_n) + \frac{h^2}{2}y'''(t_n) + O(h^3)$$

Plug in:

$$\begin{aligned} y_{n+1} &= y(t_n) + \frac{h}{2}\left[y'(t_n) + y'(t_n) + hy''(t_n) + \frac{h^2}{2}y'''(t_n) + O(h^3)\right] \\ &= y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(t_n) + \frac{h^3}{4}y'''(t_n) + O(h^4) \end{aligned}$$

3. Write down exact Taylor series for $y(t_{n+1})$:

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(t_n) + \frac{h^3}{6}y'''(t_n) + O(h^4)$$

4. Difference is LTE:

$$\begin{aligned} y(t_{n+1}) - y_{n+1} &= \left(\frac{1}{6} - \frac{1}{4}\right)h^3y'''(t_n) + O(h^4) \\ &= -\frac{1}{12}h^3y'''(t_n) + O(h^4) \\ &= O(h^3) \end{aligned}$$

Truncation Error of BDF2: BDF2 is $y_{n+1} = \frac{4}{3}y_n - \frac{1}{3}y_{n-1} + \frac{2h}{3}f(t_{n+1}, y_{n+1})$.

1. Replace RHS data with exact versions:

$$y_{n+1} = \frac{4}{3}y(t_n) - \frac{1}{3}y(t_{n-1}) + \frac{2h}{3}y'(t_{n+1})$$

2. Taylor expansions on RHS:

$$y'(t_{n+1}) = y'(t_n) + hy''(t_n) + \frac{h^2}{2}y'''(t_n) + O(h^3)$$

$$y(t_{n-1}) = y(t_n) - hy'(t_n) + \frac{h^2}{2}y''(t_n) - \frac{h^3}{6}y'''(t_n) + O(h^4)$$

Plug in and simplify:

$$\begin{aligned} y_{n+1} &= \frac{4}{3}y(t_n) - \frac{1}{3}\left(y(t_n) - hy'(t_n) + \frac{h^2}{2}y''(t_n) - \frac{h^3}{6}y'''(t_n) + O(h^4)\right) \\ &= y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(t_n) + \frac{7h^3}{18}y'''(t_n) + O(h^4) \end{aligned}$$

3. True Taylor series:

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(t_n) + \frac{h^3}{6}y'''(t_n) + O(h^4)$$

4. Error:

$$y(t_{n+1}) - y_{n+1} = \left(\frac{1}{6} - \frac{7}{18}\right)h^3y'''(t_n) + O(h^4) = -\frac{2}{9}h^3y'''(t_n) + O(h^4) = O(h^3)$$

General Taylor series:

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2}f''(a) + O((x-a)^3)$$

For $y(t_{n-1})$ we have

$$y(t_{n-1}) = y(t_n) + (t_{n-1}-t_n)y'(t_n) + \frac{(t_{n-1}-t_n)^2}{2}y''(t_n) + O((t_{n-1}-t_n)^3)$$

Since $t_{n-1} - t_n = -h$, we get a series with alternating signs:

$$y(t_{n-1}) = y(t_n) - hy'(t_n) + \frac{h^2}{2}y''(t_n) - \frac{h^3}{6}y'''(t_n) + O(h^4)$$

5.15 Adaptive Time-Stepping

We can adapt the timestep during the computation to keep the error small, while minimizing wasted effort.

Key idea: run 2 methods simultaneously with different truncation error orders. Example:

1. y^A_{n+1} = method A with $O(h^4)$.
2. y^B_{n+1} = method B with $O(h^5)$.

Approximate the error as $\left|y^A_{n+1} - y^B_{n+1}\right|$. If the error is > user chosen tolerance, reduce h and recompute the step again, repeating until the bound/tolerance is satisfied.

Observe that $y^A_{n+1} = y(t_{n+1}) + O(h^p) = y(t_{n+1}) + Ch^p + O(h^{p+1})$ for some values of C and p. If B is one order more accurate, $O(h^{p+1})$, then $y^B_{n+1} = y(t_{n+1}) + O(h^{p+1})$.

Method A's true error is $\left|y^A_{n+1} - y(t_{n+1})\right| = Ch^p + O(h^{p+1})$. Our estimated error is $\left|y^A_{n+1} - y^B_{n+1}\right| = Ch^p + O(h^{p+1})$. So the dominant (low order) component of the error matches. Our error estimate is likely a decent approximation to help us choose h.

We can estimate the coefficient C as

$$C \approx \frac{\left|y^A_{n+1} - y^B_{n+1}\right|}{h^p_{old}}$$

where h_{old} is the most recent successful timestep size. If C changes slowly in time, we can estimate the next step error as

$$err_{next} = \left|y^A_{n+2} - y^B_{n+2}\right| \approx C(h_{new})^p$$

Plug in for C to get

$$err_{next} \approx \frac{\left|y^A_{n+1} - y^B_{n+1}\right|}{h^p_{old}}(h^p_{new}) = \left(\frac{h_{new}}{h_{old}}\right)^p \left|y^A_{n+1} - y^B_{n+1}\right|$$

Given a desired error tolerance tol, set $err_{next} = tol$ and solve for h_{new}:

$$h_{new} = h_{old} \left(\frac{tol}{|y^A_{n+1} - y^B_{n+1}|} \right)^{1/p}$$

To roughly compensate for our approximations, we may be conservative by scaling tol down by a factor α. This process will let our timestep grow larger when it is safe to do so.

Adaptive Time-Stepping Algorithm

1. Compute approximate solutions for one timestep with 2 schemes of different orders.
2. Estimate the error by taking their difference.
3. While (error $>$ user-defined tolerance), set $h = \frac{h}{2}$ and recompute the solutions (step 1 and 2).
4. Estimate error coefficient and predict a good next step size, h_{new}.
5. Repeat until end time is reached.

Part IV

Fourier Transforms

Chapter 6

Fourier Transforms

The basic idea of a Fourier analysis is to transform some function/data into a new form that reveals the frequencies of information in the data. We can then analyze it in this frequency domain, then transform it back.

Definition: Time Domain

Original data/function f is a function of time, $f(t)$.

Definition: Spatial Domain

Original data/function f is a function of space/position, $f(\mathbf{x})$.

Definition: Fourier Transform

Convert a time/space-dependent data into an infinite sum of increasingly high frequency sine/cosines.

$$f(t) = a_0 + a_1\cos(qt) + b_1\sin(qt) + a_2\cos(2qt) + b_2\sin(2qt) + \cdots$$

6.1 Continuous Fourier Series

Consider some continuous periodic function $f(t)$ with period T, so

$$f(t \pm T) = f(t)$$

An example is $\sin\left(\frac{2\pi kt}{T}\right), \cos\left(\frac{2\pi kt}{T}\right)$.

The goal is to represent any $f(t)$ as an infinite sume of trig functions:

$$f(t) = a_0 + \sum_{k=1}^{\infty} a_k \cos\left(\frac{2\pi kt}{T}\right) + \sum_{k=1}^{\infty} b_k \sin\left(\frac{2\pi kt}{T}\right)$$

where a_k, b_k indicate the information/amplitude for each sinusoid of a specific period $\frac{T}{k}$ or frequency $\frac{k}{T}$. Higher integer k indicates shorter period and higher wave frequency.

6.1.1 Handy Identities

Proposition (Orthogonality)

For any integers k and j,

$$\int_0^{2\pi} \cos(kt)\sin(jt)\,dt = 0$$

Proposition (More Orthogonality Relations)

- For $k \neq j$,
$$\int_0^{2\pi} \cos(kt)\cos(jt)\,dt = 0$$
- For $k \neq j$,
$$\int_0^{2\pi} \sin(kt)\sin(jt)\,dt = 0$$
- $$\int_0^{2\pi} \sin(kt)\,dt = 0$$
- $$\int_0^{2\pi} \cos(kt)\,dt = 0$$

Determining Coefficients of a Fourier Series: We want

$$f(t) = a_0 + \sum_{k=1}^{\infty} a_k \cos(kt) + \sum_{k=1}^{\infty} b_k \sin(kt)$$

Integrate over $[0, 2\pi]$ to use the orthogonality identities.

$$\int_0^{2\pi} f(t)\,dt = a_0 \int_0^{2\pi} dt + \sum_{k=1}^{\infty} a_k \underbrace{\int_0^{2\pi} \cos(kt)\,dt}_{=0} + \sum_{k=1}^{\infty} b_k \underbrace{\int_0^{2\pi} \sin(kt)\,dt}_{=0}$$

$$\begin{aligned} a_0 &= \frac{\int_0^{2\pi} f(t)\,dt}{\int_0^{2\pi} dt} \\ &= \frac{1}{2\pi} \int_0^{2\pi} f(t)\,dt \end{aligned}$$

So a_0 is the average of f over $[0, 2\pi]$.

To determine a_k, we multiply by $\cos(\ell t)$ and integrate over $[0, 2\pi]$

$$\int_0^{2\pi} f(t)\cos(\ell t)\,dt = a_0 \underbrace{\int_0^{2\pi} \cos(\ell t)\,dt}_{=0} + \sum_{k=1}^{\infty} a_k \int_0^{2\pi} \cos(kt)\cos(\ell t)\,dt + \sum_{k=1}^{\infty} b_k \underbrace{\int_0^{2\pi} \sin(kt)\cos(\ell t)\,dt}_{=0}$$

$$\begin{aligned} &= a_\ell \int_0^{2\pi} \cos^2(\ell t)\,dt \\ &= a_\ell \cdot \pi \\ a_\ell &= \frac{\int_0^{2\pi} f(t)\cos(\ell t)\,dt}{\pi} \end{aligned}$$

Definition: Coefficients of a Fourier Series

$$a_0 = \frac{\int_0^{2\pi} f(t)\,dt}{2\pi}$$

$$a_k = \frac{\int_0^{2\pi} f(t)\cos(kt)\,dt}{\int_0^{2\pi} \cos^2(kt)\,dt} = \frac{\int_0^{2\pi} f(t)\cos(kt)\,dt}{\pi}$$

$$b_k = \frac{\int_0^{2\pi} f(t)\sin(kt)\,dt}{\int_0^{2\pi} \sin^2(kt)\,dt} = \frac{\int_0^{2\pi} f(t)\sin(kt)\,dt}{\pi}$$

Chapter 7

Discrete Fourier Transform

Assume that $t \in [0, 2\pi]$, so domain length is $T = 2\pi$. We have

$$f(t) = a_0 + \sum_{k=1}^{\infty} a_k \cos(kt) + \sum_{k=1}^{\infty} b_k \sin(kt)$$

The coefficients a_k, b_k were derived in the previous chapter.

7.1 Complex Number Review

For $z \in \mathbb{C}$, we write $z = a+bi$ where $i = \sqrt{-1}$. These are visualized as points on the complex plane.

The vector length is the modulus and angle is the argument.

- Conjugate: $\overline{z} = a - bi$.
- Modulus/norm/magnitude/abs: $|z| = \sqrt{a^2 + b^2}$.
- Argument/phase/angle: $Arg(z) = atan2(b, a)$.

Theorem (Euler's Formula)

$$e^{i\theta} = \cos(\theta) + i \sin(\theta)$$

Corollary

$$e^{-i\theta} = \cos(\theta) - i \sin(\theta)$$

Corollary

$$\cos(\theta) = \frac{e^{i\theta} + e^{-i\theta}}{2}, \sin(\theta) = \frac{e^{i\theta} - e^{-i\theta}}{2i}$$

7.2 Fourier Series With Complex Exponentials

We can change our form of the sinusoid into

$$f(t) = \sum_{k=-\infty}^{\infty} c_k e^{ikt}$$

where $c_k \in \mathbb{C}$.

Definition: Coefficients of Fourier Series

$$a_0 = c_0, c_k = \frac{a_k}{2} - \frac{ib_k}{2}, c_{-k} = \frac{a_k}{2} + \frac{ib_k}{2}$$

$$c_k = \frac{1}{2\pi} \int_0^{2\pi} e^{-ikt} f(t)\, dt$$

Proposition (Relationships)

- $|a_0| = |c_0|$.
- $|c_k| = |c_{-k}| = \frac{1}{2}\sqrt{a_k^2 + b_k^2}$.

The modulus of c_k gives the *amplitude/magnitude* of a given frequency of waves.

The angle/argument of c_k gives the frequency's *phase*: $\theta = Arg(c_k)$.

7.3 Inverse Discrete Fourier Transform

Truncation: An approximation of a function could be achieved by truncating the series to a finite number of sinusoids:

$$f(t) \approx \sum_{k=-M}^{M} c_k e^{ikt}$$

Consider a vector of discrete data, $f_0, \ldots, f_{N-1}$ for N uniformly spaced data points. Assume the data are from an unknown function $f(t)$, evaluated at each $t_n = n\Delta t = n\left(\frac{T}{N}\right)$ for $n = 0, \ldots, N-1$.

For N points, we use N degrees of freedom (N coefficients) to exactly interpolate the data. Assume N is even, we can approximate with a truncated Fourier series

$$f(t) \approx \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} c_k e^{\frac{(2\pi i)kt}{T}}$$

Plugging in each of our N data points (t_n, f_n) into the expression above will give us N equations, involving unknown coefficients c_k. This leads toward the inverse discrete Fourier transform.

Deriving (Inverse) Discrete Fourier Transform: Our approximation is

$$f(t) \approx \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} c_k e^{\frac{(2\pi i)kt}{T}}$$

For the nth discrete data point $t_n = \frac{nT}{N}$, we get

$$f_n = \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} c_k e^{\frac{2\pi i k t_n}{T}} = \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} c_k e^{\frac{2\pi i k n T}{TN}}$$

For convenience, we prefer to transform this into a sum over $0, \ldots, N-1$ instead.

$$f_n = \sum_{k=0}^{\frac{N}{2}} c_k e^{\frac{2\pi i n k}{N}} + \sum_{k=-\frac{N}{2}+1}^{-1} c_k e^{\frac{2\pi i n k}{N}}$$

Apply change of variables $j = N + k$

$$\sum_{k=-\frac{N}{2}+1}^{-1} c_k e^{\frac{2\pi i n k}{N}} = \sum_{j=\frac{N}{2}+1}^{N-1} c_{j-N} e^{\frac{2\pi i n (j-N)}{N}} = \sum_{j=\frac{N}{2}+1}^{N-1} c_{j-N} e^{\frac{2\pi i n j}{N}} \underbrace{e^{\frac{-2\pi i n N}{N}}}_{=1} = \sum_{j=\frac{N}{2}+1}^{N-1} c_{j-N} e^{\frac{2\pi i n j}{N}}$$

By Euler's formula, we have $e^{-2\pi i n} = \cos(2\pi n) - i \sin(2\pi n) = 1$ for integers n. Next we define our c_j outside the range $\left[-\frac{N}{2}+1, \frac{N}{2}\right]$ to be periodic, that is, $c_{j \pm N} = c_j$.

$$\sum_{j=\frac{N}{2}+1}^{N-1} c_j e^{\frac{2\pi i n j}{N}}$$

We plug this back into f_n.

$$f_n = \sum_{k=0}^{\frac{N}{2}} c_k e^{\frac{2\pi i n k}{N}} + \sum_{j=\frac{N}{2}+1}^{N-1} c_j e^{\frac{2\pi i n j}{N}} = \sum_{k=0}^{N-1} c_k e^{\frac{2\pi i n k}{N}}$$

Definition: Inverse Discrete Fourier Transform

$$f_n = \sum_{k=0}^{N-1} F_k e^{\frac{i 2\pi n k}{N}} = \sum_{k=0}^{N-1} F_k W^{nk}$$

where $W = e^{\frac{2\pi i}{N}}$.

Definition: Nth Roots of Unity

Let $W = e^{\frac{2\pi i}{N}}$, then $W^k = e^{\frac{2\pi ik}{N}}$ are the Nth roots of unity, for all $k \in \mathbb{Z}$.

$W^N = e^{2\pi i} = 1$. Each Nth root gives a point on the unit circle. The modulus is $\left|W^k\right| = 1$ since $\cos^2(\theta) + \sin^2(\theta) = 1$.

For example, for $N = 3$, we have $W^k = e^{\frac{2\pi ik}{3}}$ and to retrieve the roots of unity, evaluate it at $k = 0, \ldots, N-1$.

For the discrete Fourier transform to be useful, we have two operations:

- Convert time-domain data f_n to frequency-domain F_k (DFT).
- Convert frequency-domain data F_k to time-domain f_n (IDFT).

7.4 Forward Discrete Fourier Transform

Definition: Kronecker Delta Function $\delta_{k,l}$

$$\delta_{k,l} = \begin{cases} 1 & \text{if } k \neq l \\ 0 & \text{if } k = l \end{cases}$$

Proposition Orthogonality Identity

$$\sum_{j=0}^{N-1} W^{jk} W^{-jl} = \sum_{j=0}^{N-1} W^{j(k-l)} = N\delta_{k,l}$$

assuming $k, l \in [0, N-1]$.

Derivation: We have two cases to consider:

- $k = l$:
$$\sum_{j=0}^{N-1} W^{j(0)} = \sum_{j=0}^{N-1} 1 = N$$
- $k \neq l$: Observe that $x^N - 1 = (x-1)(x^{N-1} + x^{N-2} + \cdots + x + 1)$.
$$\sum_{j=0}^{N-1} x^j = \frac{x^N - 1}{x - 1}$$
Apply this to our expression:
$$\sum_{j=0}^{N-1} W^{j(k-l)} = \sum_{j=0}^{N-1} (W^{k-l})^j = \frac{(W^{k-l})^N - 1}{W^{k-l} - 1} = 0$$

Since $k \neq l$ the numerator is 0 since W and W^{k-l} are Nth roots of unity, so when being raised to the power of N, we get unity/1.

Discrete Fourier Transform Derivation: Like the continuous case, use orthogonality to isolate each Fourier coefficient F_k. We have from IDFT

$$f_n = \sum_{j=0}^{N-1} F_j W^{nj}$$

To get the kth coefficient, multiply W^{-nk} and sum from 0 to $N-1$.

$$\sum_{n=0}^{N-1} f_n W^{-nk} = \sum_{n=0}^{N-1} \sum_{j=0}^{N-1} F_k W^{nj} W_{-nk} = \sum_{j=0}^{N-1} F_j \sum_{n=0}^{N-1} W^{n(j-k)} = \sum_{j=0}^{N-1} F_j N \delta_{j,k} = F_k \cdot N$$

So $F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n W^{-nk}$.

Definition: Discrete Fourier Transform Coefficients

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n W^{-nk}$$

Example: Given 4 discrete Fourier coefficients F_k, find the corresponding f_n for $n = 0, \ldots, 3$. $F = (-2, 2+i, -2, 2-i)$.

We have $W = e^{\frac{2\pi i}{N}} = e^{\frac{2\pi i}{4}} = e^{\frac{\pi i}{2}} = \cos(\pi/2) + i\sin(\pi/2)$. So $W^0 = 1, W^1 = i, W^2 = -1, W^3 = -i$ (complex plane unit circle).

- $n = 0$:
$$f_0 = \sum_{k=0}^{3} F_k W^0 = \sum_{k=0}^{3} F_k = (-2) + (2+i) + (-2) + (2-i) = 0$$
- $n = 1$:
$$\begin{aligned} f_1 &= \sum_{k=0}^{3} F_k W^k = (-2)W^0 + (2+i)W^1 + (-2)W^2 + (2-i)W^3 \\ &= (-2)(1) + (2+i)(i) + (-2)(-1) + (2-i)(-i) \\ &= -2 \end{aligned}$$
- $n = 2$:
$$f_2 = \sum_{k=0}^{3} F_k W^{2k} = -8$$

Example: Consider a set of N data points defined such that $f_n = \cos\left(\frac{2\pi n}{N}\right)$. Show that $F_1 = F_{N-1} = \frac{1}{2}$ and all other $F_k = 0$.

DFT is

$$F_k = \frac{1}{N}\sum_{n=0}^{N-1} \cos\left(\frac{2\pi n}{N}\right) W^{-nk}$$

Use Euler's formula, we had $\cos(\theta) = \frac{e^{i\theta}+e^{-i\theta}}{2}$,

$$F_k = \frac{1}{N}\sum_{n=0}^{N-1} \frac{1}{2}\left(e^{\frac{i2\pi n}{N}} + e^{\frac{-i2\pi n}{N}}\right) W^{-nk}$$

Since $W = e^{\frac{i2\pi n}{N}}$, we get

$$\begin{aligned}
F_k &= \frac{1}{2N}\sum_{n=0}^{N-1}(W^n + W^{-n})W^{-nk} \\
&= \frac{1}{2N}\sum_{n=0}^{N-1}(W^{n(1-k)} + W^{-n(1+k)}) \\
&= \frac{1}{2N}\sum_{n=0}^{N-1}(W^{n(1-k)} + W^{n(N-1-k)}) && (\times 1 = W^{Nn}) \\
&= \frac{1}{2N}(N\delta_{1,k} + N\delta_{N-1,k}) \\
&= \frac{\delta_{1,k}}{2} + \frac{\delta_{N-1,k}}{2} \\
&= \begin{cases} \frac{1}{2} & \text{if } k = 1 \\ \frac{1}{2} & \text{if } k = N-1 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

Properties of the DFT

1. The sequence $\{F_k\}$ is doubly infinite and periodic.
2. Conjugate symmetry: If data f_n is real, $F_k = \overline{F_{N-k}}$.

F_k are periodic and doubly infinite. Given F_k for integers $k \in [0, N-1]$, then for integers $k \in (-\infty, \infty)$, F_k is one of the existing values. We can express arbitrary k as $k = mN + p$ for $p \in [0, N-1]$, that is $k = p \pmod N$. Then $W^{-k} = e^{\frac{-i2\pi}{N}(mN+p)} = e^{-i2\pi m}e^{-i2\pi p/N} = W^{-p}$. So

$$F_k = \frac{1}{N}\sum_{n=0}^{N-1} f_n W^{-nk} = \frac{1}{N}\sum_{n=0}^{N-1} f_n W^{-np} = F_p$$

Conjugate symmetry: We have 3 facts

1. $W^{n-k} = e^{\frac{2\pi i}{N}(N-k)} = e^{\frac{-2\pi i}{N}} = W^{-k}$.
2. Since $e^{i\theta} = \cos(\theta) + i\sin(\theta)$ and $e^{-i\theta} = \cos(\theta) - i\sin(\theta)$, we observe $\overline{e^{i\theta}} = e^{-i\theta}$, since the sign of the imaginary part is flipped.

   Also, $\overline{W^j} = W^{-j}$.

3. For real x, $\overline{x} = x$.

Now, $\overline{F_{N-k}} = \frac{1}{N}\sum f_n W^{-n(N-k)}$ by definition of DFT.

$$\begin{aligned}\overline{F_{N-k}} &= \frac{1}{N}\sum f_n \overline{W^{-n(N-k)}} \\ &= \frac{1}{N}\sum f_n \overline{W^{-n(-k)}} \\ &= \frac{1}{N}\sum f_n W^{-nk} \\ &= F_k\end{aligned} \tag{3}$$

Discrete Data Observations:

- Coefficient F_0 is the average of data values, i.e.

$$F_0 = \frac{1}{N}\sum_{n=0}^{N-1} f_n$$

- The smooth bump had one dominant frequency/wave; therefore, one coefficient pair F_1 and F_8 had large magnitude.
- The rough bump had more irregularity, so more higher frequencies. Main hump still dominates, so low frequency contributions F_1 and F_8 remain largest.
- The Fourier F_k plots are symmetric.

SciPy uses $f_n = \frac{1}{N}\sum_{k=0}^{N-1} F_k W^{nk}$ and $F_k = \sum_{n=0}^{N-1} f_n W^{-nk}$.

7.5 Matrix View of DFT

Definition: DFT in Matrix Form

Let F and f be vectors of Fourier coefficients and input data, we can write the DFT as $F = Mf$ where M is a matrix whose kth column is

$$\frac{1}{N}\begin{bmatrix} W^0 & W^{-k} & W^{-2k} & \dots & W^{-(N-1)k}\end{bmatrix}^T$$

7.6 Matrix View of IDFT

Out orthogonality identity leads to $\overline{M^T}M = \frac{1}{N}I$. Therefore, $M^{-1} = N\overline{M^T}$.

Definition: IDFT in Matrix Form

$$f = M^{-1}F = N\overline{M^T}F$$

Example: $F = [-2, 2+i, -2, 2-i]$. Find f.

$$f = N\overline{M^T}F = 4\left(\frac{1}{4}\begin{bmatrix} W^0 & W^0 & W^0 & W^0 \\ W^0 & W^1 & W^2 & W^3 \\ W^0 & W^2 & W^4 & W^6 \\ W^0 & W^3 & W^6 & W^9 \end{bmatrix}\right)\begin{bmatrix} -2 \\ w+i \\ -2 \\ 2-i \end{bmatrix} = \begin{bmatrix} 0 \\ -2 \\ -8 \\ 2 \end{bmatrix}$$

Chapter 8

Fast Fourier Transform

A direct implementation of F_k takes $O(N^2)$ complex floating-point operations.

Definition: Fast Fourier Transform Algorithm

A divide-and-conquer algorithm:

1. Split the full DFT into two DFT's of half the length.
2. Repeat recursively.
3. Finish at the base case: DFT of individual pairs of numbers. (If $N \neq 2^m$, pad our data with zeros)

FFT Derivation: The usual DFT sequence is

$$F_k = \frac{1}{N}\sum_{n=0}^{N-1} f_n W^{-nk} = \frac{1}{N}\sum_{n=0}^{N/2-1} + \frac{1}{N}\sum_{n=N/2}^{N-1} f_n W^{-nk}$$

Assume N is even. Reindex the 2nd sum, using $m = n - \frac{N}{2}$.

$$F_k = \frac{1}{N}\sum_{n=0}^{N/2-1} f_n W^{-nk} + \frac{1}{N}\sum_{m=0}^{N/2-1} f_{m+N/2} W^{-k\left(m+\frac{N}{2}\right)}$$

Rename m back to n

$$F_k = \frac{1}{N}\sum_{n=0}^{N/2-1} f_n W^{-nk} + \frac{1}{N}\sum_{n=0}^{N/2-1} f_{n+N/2} W^{-nk} W^{-kN/2}$$

Combine sums:

$$F_k = \frac{1}{N}\sum_{n=0}^{N/2-1} (f_n + f_{n+N/2} W^{-kN/2}) W^{-nk}$$

Notice $W^{-kN/2} = (e^{2\pi i/N})^{-kN/2} = e^{-\pi ik} = (-1)^k$. Hence two cases: k is even and k is odd.

k even:

$$F_k = F_{2j} = \frac{1}{N} \sum_{n=0}^{N/2-1} (f_n + f_{n+N/2}) W^{-2nj}$$

for $j = 0$ to $\frac{N}{2} - 1$.

k odd:

$$F_k = F_{2j+1} = \frac{1}{N} \sum_{n=0}^{N/2-1} (f_n - f_{n+N/2}) W^{-n(2j+1)} = \frac{1}{N} \sum_{n=0}^{N/2-1} [(f_n - f_{n+N/2}) W^{-n}] W^{-2nj}$$

for $j = 0$ to $\frac{N}{2} - 1$.

Now define two new half-length vectors by

$$g_n = \frac{1}{2}(f_n + f_{n+N/2}), h_n = \frac{1}{2}(f_n - f_{n+N/2}) W^{-n}$$

for $n = 0$ to $\frac{N}{2} - 1$ and $M = \frac{N}{2}$.

Observe that $F_{2j} = \frac{2}{N} \sum_{n=0}^{N/2-1} g_n W_N^{-2nj} = \frac{1}{M} \sum_{n=0}^{M-1} g_n W_M^{-nj} = Gj$ and $F_{2j+1} = \frac{2}{N} \sum_{n=0}^{N/2-1} h_n W_N^{-2nj} = \frac{1}{M} \sum_{n=0}^{M} -1 h_n W_M^{-nj} = Hj$ where $W_N = e^{2\pi i/N}, W_M = e^{2\pi i/M} = e^{2\pi i/(N/2)}$ so that $W_M = W_N^2$ since $N = 2M$.

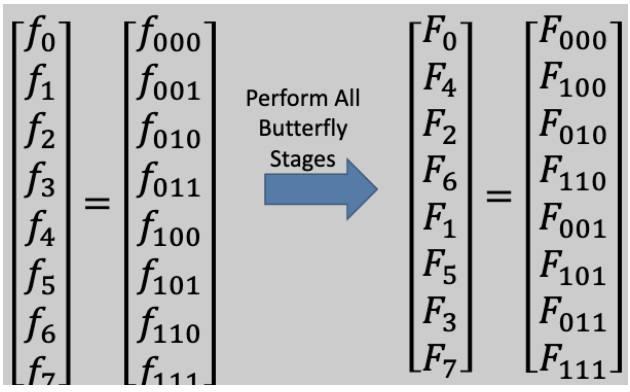
Thus, we have rewritten the entries of $F = DFT(f)$ as entries of 2 half-length vector G and H, which are the DFTs of g and h.

Definition: Fast Fourier Transform Divide Part

Let $g_n = \frac{1}{2}(f_n + f_{n+\frac{N}{2}})$ and $h_n = \frac{1}{2}(f_n - f_{n+\frac{N}{2}}) W^{-n}$, where $n \in [0, N/2 - 1]$. Then $F_{even} = DFT(g)$ and $F_{odd} = DFT(h)$.

Each step applies and even-odd index splitting, so the final vector have the indices in bit-reversed positions.

To unscramble, we reverse the bits of the index:



FFT: Spends $O(N)$ operations to split, and there are $m = \log_2 N$ stages, so total cost is $O(N \log_2 N)$.

Example: Find the sequence of vectors and the result of applying the butterfly FFT approach to $f = (4, 3, 2, 1, 4, 3, 2, 1)$.

Remember to recompute W for each new N. First one is $N = 8$.

$$g = (4, 3, 2, 1), h = (0, 0, 0, 0)$$

Now $N = 4$,

$$g_{top} = (3, 2), h_{top} = (1, -i), g_{bottom} = (0, 0), h_{bottom} = (0, 0)$$

Now $N = 2$, from top to bottom,

$$g_0, h_0 = 2.5, 0.5, \frac{1}{2} - \frac{1}{2}i, \frac{1}{2} + \frac{1}{2}i, 0, 0, 0, 0$$

Unscramble using reverse of the index bits:

$$F = (2.5, 0, \frac{1}{2} - \frac{1}{2}i, 0, 0.5, 0, \frac{1}{2} + \frac{1}{2}i, 0)$$

Example: Find the sequence of vectors and final result of applying the butterfly FFT approach to $f = (2 + i, 3, -2, 2i)$.

8.1 1D and 2D Fast Fourier Transform

We can think of a 1D array of grayscale values as a 1D image.

Compression strategy:

- Create an approximate compressed version of the image, f_n, by throwing away small Fourier coefficients: $|F_k| < tol$.
- To reconstruct the image, run the inverse DFT to get modified data, $\hat{f}_n$.
- Discard the imaginary parts of $\hat{f}_n$ to ensure new data is strictly real.

8.1.1 2D Discrete Fourier Transform

Definition: 2D Discrete Fourier Transform

$$F_{k,l} = \frac{1}{NM} \sum_{n=0}^{N-1} \sum_{j=0}^{M-1} f_{n,j} W_N^{-nk} W_M^{-jl}$$

There are two possible distinct roots of unity used:

$$W_N = e^{2\pi i/N}, W_M = e^{2\pi i/M}$$

The result is a 2D array of Fourier coefficients.

Definition: 2D Fast Fourier Transform

Transform each row separately using 1D FFTs, then transform each column of the result, using 1D FFTs.

Complexity: Performing M 1D FFTs of length N is $M \cdot O(N \log N) = O(MN \log N)$. Performing N 1D FFTs of length M is $O(MN \log M)$. So the total complexity is $O(MN(\log M + \log N))$.

Part V

Numerical Linear Algebra