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 $a, b, c \in F$.

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

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

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

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 $1, x, x^2, \dots$ 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 ($i = 1, \dots, n-1$):

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

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

$$\begin{aligned}\Delta x_2 s_1 + 2(\Delta x_2 + \Delta x_1)s_2 + 2s_3 &= 3(\Delta x_2 y'_1 + \Delta x_1 y'_2)\\ s_1 + 2(1+2)s_2 + 2s_3 &= 3(1 \cdot 0 + 2 \cdot 2)\\ s_1 + 6s_2 + 2s_3 &= 12\end{aligned}$$

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.

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.

Part IV

Fourier Transforms

Part V

Numerical Linear Algebra