CS 370: Numerical Computation

University of Waterloo

Instructor: Christopher Batty

Fall 2022

Andrew Wang

Table of Contents

Floating Point Numbers

Numerical Computation in our case is:

> A **computer approximation** of some mathematical value/problem, usually dealing with (approximate) **real numbers**

Analytical solutions are the exact solutions while **numerical** solutions are approximate solutions.

Representing Values from $\mathbb{R}$

Real numbers, $\mathbb{R}$, are

- Infinite in *extent*: $\exists x \in \mathbb{R}$ such that $|x|$ is arbitrarily large
- Infinite in *density*: $\forall a, b \in \mathbb{R}$ if $a < b$ then there are infinitely many $x \in \mathbb{R}$ that satisfy $a \leq x \leq b$

It is impossible to encode the reals with only a finite number of bits so we approximate by:

1. Multiply by power of β (base of the number representation) to shift into **normalized form**:

$$d_1 d_2 d_3 d_4 \cdots \;=\; 0.d_1 d_2 d_3 d_4 \cdots \times \beta^p \qquad \text{where } d_1 \neq 0$$

2. Convert into some specific floating point system $\mathcal{F}$ characterized by 4 integer parameters $\{\beta,\ t,\ L,\ U\}$:

$$\pm 0.d_1 d_2 \cdots d_t \times \beta^p \quad \text{for } L \leq p \leq U \text{ and } d_1 \neq 0 \text{ (special case for 0)}$$

   - *Density* (or *precision*) is bounded by limited number of digits using t
   - *Extent* (or *range*) is bounded by limiting range of value for exponent p

We are only considering two *rounding modes* when converting from $\mathbb{R}$ to floating point

- **Round-to-nearest**: round to closest number in $\mathcal{F}$ (default option)
- **Truncation**: discard digits after the t^{th}

Overflow and Underflow

When exponent p is too large ($> U$) or too small ($< L$) our system simply *cannot* represent the number.

- For underflow, simply round answer to 0
- For overflow, we produce a $\pm\infty$ (or NaN - not a number)

Floating Point Standards

The IEEE created two of the most common standard floating point systems

- **Single precision** (32 bits): $\{\beta = 2,\ t = 24,\ L = -126,\ U = 127\}$
- **Double precision** (64 bits): $\{\beta = 2,\ t = 53,\ L = -1023,\ U = 1023\}$

Arithmetic using these are always implemented directly in hardware (CPU/GPU/etc)

Floating Point Density

It is important to remember that unlike fixed point, floating point number are *not evenly spaced!*

For example: $\mathcal{F} = \{\beta = 2,\ t = 3,\ L = -1,\ U = 2\}$ will represent non-negative values spaced as

$$0,\quad 0.100 \times 2^{-1},\quad 0.101 \times 2^{-1},\quad 0.110 \times 2^{-1},\quad 0.111 \times 2^{-1},\quad 0.100 \times 2^{0},\quad 0.101 \times 2^{0},\quad \cdots,\quad 0.111 \times 2^{2}$$

Floating Point Error

We will look at two ways to measure error:

- **Absolute error**: actual distance from exact to approximation

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

- **Relative Error**: distance from exact to approximation relative to size of exact value

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

A rough rule of thumb: a result is correct to *roughly* s digits after rewriting E_{rel} as

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

Some examples:

- $E_{\text{rel}} \approx 0.4 \times 10^{-3} = 4 \times 10^{-4}$ implies 4 correct digits in x_{approx}
- $E_{\text{rel}} \approx 4 \times 10^{2}$ implies -2 correct digits (or no correct digits) in x_{approx}

Machine Epsilon

Machine Eplison E is the **largest relative error** in the conversion from real to floating point

- It is defined as the smallest E such that $fl(1 + E) > 1$
- for a FP system $\mathcal{F} = \{\beta,\ t,\ L,\ U\}$ with
  - rounding during conversion: $E = \frac{1}{2}\beta^{1-t}$
  - truncating during conversion: $E = \beta^{1-t}$
- This means we can say that for any $x \in \mathbb{R}$ in the range of $\mathcal{F}$ that

$$fl(x) = x(1 + \delta) \qquad \text{for some } |\delta| \leq E$$

Arithmetic with Floating Point

The IEEE standard *requires* that for $w, z \in \mathcal{F}$

$$w \oplus z = fl(w + z) = (w + z)(1 + \delta) \qquad \text{for some } |\delta| \leq E$$

To perform addition we add w and z as if they were real then convert to our floating point approximation.

- The result is the same as the floating point approximation of the *exact real* result
- This is done in hardware by keeping extra "guard" digits to ensure this result
- This is in the same way for other operations like $\otimes$ and $\ominus$

However this only applies to individual FP operations so it is **not true** that:

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

Since we evaluate

$$(a \oplus b) \oplus c = ((a + b)(1 + \delta_1) + c)(1 + \delta_2) \qquad a \oplus (b \oplus c) = (a + (b + c)(a + \delta_1))(1 + \delta_2)$$

This means that the result is order dependent and *associativity* is broken.

Error Bounds

for $a,\ b,\ c \in \mathcal{F}$ consider the relative error of $(a \oplus b) \oplus c$

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

We will now weaken the bound so that the error applies to both $(a \oplus b) \oplus c$ and $a \oplus (b \oplus c)$

$$\begin{aligned}
E_{\text{rel}} &\leq \frac{|a + b|(E + E^2)}{|a + b + c|} + \frac{|a + b + c|}{|a + b + c|}E \\
&\leq \frac{|a| + |b| + |c|}{|a + b + c|}(E + E^2) + \frac{|a| + |b| + |c|}{|a + b + c|}E && \text{(by triangle inequality and adding } |c|) \\
&\leq \frac{|a| + |b| + |c|}{|a + b + c|}(2E + E^2)
\end{aligned}$$

This bound is for the worst case (actual error *could* be much less)

Cancellation Errors

The worst case for the bounds is when quantities have differing signs and similar magnitudes

- If a, b, c have same signs then $\frac{|a|+|b|+|c|}{|a+b+c|} = 1$
- If a, b, c have different signs then $\frac{|a|+|b|+|c|}{|a+b+c|} \geq 1$ (can grow very large if $a + b + c \approx 0$)

Catastrophic cancellation occurs when subtracting same magnitude numbers that contain some error

$$\underbrace{173.00}_{\text{Correct digits}}\underbrace{063}_{\text{Erroneous digits}} - \underbrace{173.00}_{\text{Correct digits}}\underbrace{017}_{\text{Erroneous digits}} = \underbrace{0.00046}_{\text{Only erroneous digits left!}}$$

All *significant* digits cancelled out and we are left with no correct digits at all.

Floating Point Rule of Thumbs

Rule of thumb: prefer adding numbers of approximately the same size and sign

- adding large and small numbers can cause the smaller digits to get lost or "swamped"

Rule of thumb: try to reformulate computations to avoid cancellations

- catastrophic cancellation could occur from subtracting numbers of simular magnitude

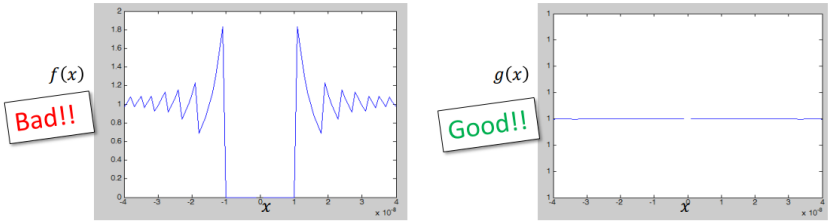
Taylor series example: to evaluate $e^{-5.5}$ in FP we have

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \quad \text{or} \quad e^{-x} = \frac{1}{e^x} = \frac{1}{1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots}$$

$e^{-5.5} = 0.0040868$ and with 5 digits of accuracy we see that RHS performs much better

- using $x = -5.5$ on the LHS we have $e^{-5.5} \approx 0.0026363$
- using $x = 5.5$ on the RHS we ahve $e^{-5.5} \approx 0.0040865$

Using IEEE double precision we compare $f(x) = \frac{1-\cos^2 x}{x^2}$ and $g(x) = \frac{\sin^2 x}{x^2}$ near $x = 0$



The true solution is to approach 1 as $x \to 0$ however $f(x)$ behaves erratically since $\cos^2 x$ approachs 1

Sources of Errors

We will mainly focus on the first two

- Round-off error: due to FP representation and arithmetic
- Truncation error: e.g. truncating a Taylor series after n terms
- Uncertainty/error in input: e.g. from measurements
- Error/approximation in mathematical model: e.g. friction

Conditioning and Stability

Conditioning of a problem:

- How sensitive is the *problem* to changes in input? less → well-conditioned more → ill-conditioned

Stability of numerical algorithm:

- How sensitive is the *algorithm* to errors/changes in input? less → stable more → unstable

Note that:

- An poor *algorithm* can be unstable even for a well-conditioned problem
- An ill-conditioned *problem* limits how well we can expect *any algorithm* to perform

Stability Analysis

We can rewrite the integration problem as a recursive rule

$$I_n = \int_0^1 \frac{x^n}{x+\alpha} dx \qquad \rightarrow \qquad I_0 = \log \frac{1+\alpha}{\alpha} \qquad I_n = \frac{1}{n} - \alpha I_{n-1}$$

Assume some initial error in E_0 in I_0 where

$$E_0 = \underbrace{(I_0)_A}_{\text{approx}} - \underbrace{(I_0)_E}_{\text{exact}}$$

then the (signed) error E_n after n steps will be

$$\begin{aligned} E_n &= (I_n)_A - (I_n)E \\ &= (\tfrac{1}{n} - \alpha(I_{n-1})_A) - (\tfrac{1}{n} - \alpha(I_{n-1})_E) \\ &= -\alpha((I_{n-1})_A - (I_{n-1})_E) \\ &= -\alpha E_{n-1} \end{aligned}$$

Thus we have $E_n = (-\alpha)^n E_0$ so we have two cases

- $|\alpha| \leq 1$ error does not grow - stable
- $|\alpha| > 1$ error grows continiually - unstable

Interpolation

Given a set of points from unknown function $y = p(x)$ how do we approximate p's values at other points

- Known: $p(x_1) = y_1$, $p(x_2) = y_2$, $\ldots$, $p(x_n) = y_n$
- Estimate: $y = p(x)$ for any point x such that $x_1 \leq x \leq x_n$

Polynomial Interpolation

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

Vandermonde Matrices

For 3 data points we find the 3 unknown coefficients a, b, c that form the quadratic

$$y = a + bx + cx^2$$

Each data point gives 1 linear equation so we a 3×3 system

$$\begin{array}{l} a + bx_1 + cx_1^2 = y_1 \\ a + bx_2 + cx_2^2 = y_2 \\ a + bx_3 + cx_3^2 = y_3 \end{array} \qquad \rightarrow \qquad \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

In general for n points we have a linear system $V\vec{c} = \vec{y}$ where V is the *Vandermonde matrix*

$$\begin{bmatrix} 1 & x_1 & \cdots & x_1^{n-1} \\ 1 & x_2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

Polynomial interpolation reduces to solving this linear system of equations.

The Monomial Basis

The form $p(x) = c_1 + c_2 x + \cdots + c_n x^{n-1}$ is called monomial form and can be written as

$$p(x) = \sum_{i=1}^{n} c_i x^{i-1}$$

- **Monomial basis**: the sequence $1,\ x,\ x^2,\ \ldots$
- **Monomial form**: sum of coefficients c_I times *basis functions* x^{i-1}

The Lagrange Basis

For the Lagrange method of interpolating polynomials we construct p by using

$$p(x) = y_1 L_1(x) + y_2 L_2(x) + \cdots + y_n L_n(x) = \sum_{k=1}^{n} y_k L_k(x)$$

Given n data points $(x_i,\ y_i)$ we define the *Lagrange basis function* $L_k(x)$ to be

$$L_k(x) = \frac{(x - x_1)\ \cdots\ (x - x_{k-1})(x - x_{k+1})\ \cdots\ (x - x_n)}{(x_k - x_1)\cdots(x_k - x_{k-1})(x_k - x_{k+1})\cdots(x_k - x_n)} \qquad \rightarrow \qquad L_i(x_j) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

This property forces $p(x)$ to interpolate every x_i since

$$\begin{aligned} p(x_i) &= y_1 L_1(x_i) + \cdots + y_i L_i(x_i) + \cdots + y_n L_n(x_i) \\ &= y_1 \times 0 \quad + \cdots + y_i \times 1 \quad + \cdots + y_n \times 0 \end{aligned}$$

This results in a polynomial that satisfies $p(x_i) = y_i$ for $i = 1, \ldots, n$ by construction

Example: Linear System and Lagrange Polynomials

Find the linear polynomial that interpolates the points $(x_1, y_1) = (1, 2)$ and $(x_2, y_2) = (-1, 4)$

- We can solve the linear system of equations $c_1 + c_2 x_1 = y_1$ and $c_1 + c_2 x_2 = y_2$

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \quad \rightarrow \quad \left[\begin{array}{cc|c} 1 & 1 & 2 \\ 1 & -1 & 4 \end{array}\right] \quad \rightarrow \quad \left[\begin{array}{cc|c} 2 & 0 & 6 \\ -1 & 1 & -4 \end{array}\right] \quad \rightarrow \quad \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$$

  We get $c_1 = 3$ and $c_2 = -1$ for the monomial form $p(x) = c_1 + c_2 x$ which results in $p(x) = 3 - x$

- We can also solve this using the Lagrange Basis

$$L_1(x) = \frac{(x - x_2)}{(x_1 - x_2)} = \frac{x - (-1)}{1 - (-1)} = \frac{x + 1}{2} \qquad L_2(x) = \frac{(x - x_1)}{(x_2 - x_1)} = \frac{x - 1}{-1 - 1} = \frac{x - 1}{-2}$$

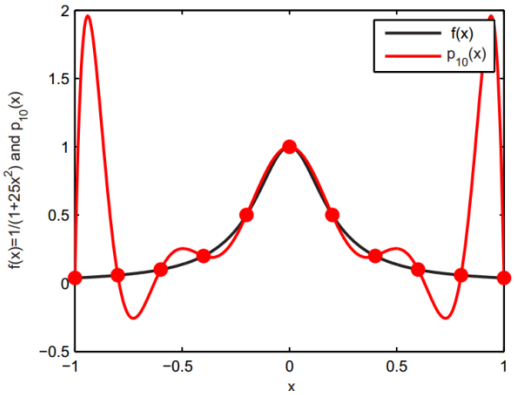
  Using these for the Lagrange form $p(x) = y_1 L_1(x) + y_2 L_2(x)$ we get

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

We find that both methods result in the same solution of $p(x) = 3 - x$. However we still prefer using the Lagrange basis to the monomial basis because it avoids needing to solve a linear system.

Polynomial Interpolation for Many Points

Consider fitting a degree 10 polynomial to 11 points sampled evenly from $f(x) = \frac{1}{1+25x^2}$:

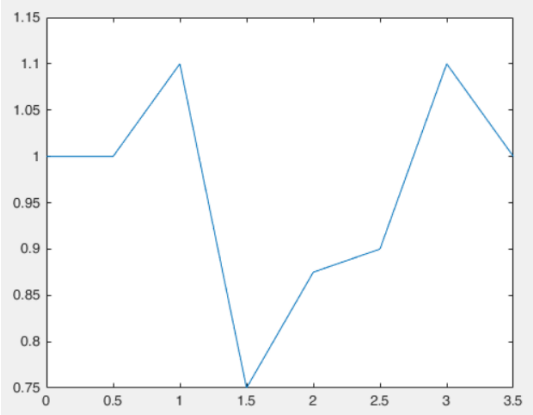


Notice the excessive wiggling at the ends, this example of overfitting is called *Runge's phenomenon.*

Instead of using a single high degree polynomial we could instead create a piecewise function consisting of multiple lower degree polynomials.

Hermite Interpolation

Although interpolation with piecewise linear can produce a continuous function it is not "smooth" enough



This function is in C^0, which is to say it is continuous but the derivatives might not be.

Hermite interpolation gets more *smoothness* by fitting the polynomial to both *values and deriatives* to achieve C^1, which is function and first derivative continuity.

Vandermonde Approach for Hermite Interpolation

For polynomials with higher derivative data we will use a simular approach to before:

1. Express polynomial $p(x)$ in general form with unknown coefficients
2. Differentiate to find expressions for needed derivatives of $p(x)$
3. Plug in the value and derivative data to give linear equations
4. Solve for the polynomial coefficients

Usage: find the coefficients of the cublic polynomial $p(x) = a + bx + cx^2 + dx^3$ using the *Hermite data*

$$p(x_1) = y_1 \quad p'(x_1) = s_1 \qquad p(x_2) = y_2 \quad p'(x_2) = s_2$$

Find $p'(x) = b + 2cx + 3dx^2$ then create a system of linear equations using the data

$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 0 & 1 & 2x_1 & 3x_1^2 \\ 0 & 1 & 2x_2 & 3x_2^2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ s_1 \\ s_2 \end{bmatrix}$$

Piecewise Hermite Interpolation: Create one cubic function *per pair* of adjacent x values. Then by construction the value and derivative of the endpoints of cubic are shared and the piecewise function will have C^1 continuity.

Closed-Form Solution for Hermite Interpolation

$$p(x_i) = y_i \quad p'(x_i) = s_i \qquad p(x_{i+1}) = y_{i+1} \quad p'(x_{i+1}) = s_{i+1}$$

We define the cubic polynomial on the i-th interval (between x_i and x_{i+1}) to be $p_i(x)$ where

$$\begin{aligned} p_i(x) &= a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \\ p_i'(x) &= b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2 \end{aligned}$$

Plugging in the data

$$\begin{aligned} y_i &= p_i(x_i) = a_i \\ s_i &= p_i'(x_i) = b_i \\ y_{i+1} &= p_i'(x_{i+1}) = a_i + b_i(x_{i+1} - x_i) + c_i(x_{i+1} - x_i)^2 + d_i(x_{i+1} - x_i)^3 \\ s_{i+1} &= p_i'(x_{i+1}) = b_i + 2c_i(x_{i+1} - x_i) + 3d_i(x_{i+1} - x_i)^2 \end{aligned}$$

Solving this system gives

$$a_i = y_i \qquad b_i = s_i \qquad c_i = \frac{3y_i' - 2s_i - s_{i+1}}{\Delta x_i} \qquad d_i = \frac{-2y_i' + s_i + s_{i+1}}{\Delta x_i^2}$$

$$\Delta x_i = x_{i+1} - x_i \qquad y_i' = \frac{y_{i+1} - y_i}{\Delta x_i}$$

We can now solve a cubic Hermite problem directly note that $p_i(x)$ will usually need to be simplified.

- **Knots**: points where interpolant transitions from one polynomial/interval to another
- **Nodes**: points where some control points/data is specified

For Hermite interpolation these are the same but for other curve types (Ex Bezier curves) they can differ

Cubic Spline Interpolation

Hermite interpolation requires info on derivative values but we want to create a curve out of a set of points. For **cubic spline interpolation** fit a cubic $S_i(x)$ on each interval while requiring matching first and second derivatives (C^2 continuity) between intervals.

Require *interpolating conditions* on each interval $[x_i, x_{i+1}]$

$$S_i(x_i) = y_i \quad S_i(x_{i+1}) = y_{i+1}$$

and *derivative conditions* are each *interior* point x_{i+1}

$$S_i'(x_{i+1}) = S_{i+1}'(x_{i+1})$$

$$S_i''(x_{i+1}) = S_{i+1}''(x_{i+1})$$

Counting Unknowns and Equations

A cubic has 4 unknowns per interval and $n-1$ intervals (for n points) so there are $4n-4$ unknowns.

- 2 interpolation (endpoint) conditions per *interval*: $2(n-1) = 2n-2$

$$S_i(x_i) = y_i \qquad S_i(s_{i+1}) = y_{i+1}$$

- 2 derivative conditions per *interior point* (between first and last points): $2(n-2) = 2n-4$

$$S_i'(x_{i+1}) = S_{i+1}'(x_{i+1}) \qquad S_i''(x_{i+1}) = S_{i+1}''(x_{i+1})$$

This gives us $4n-6$ equations which is less than the $4n-4$ unknowns so there is not enough information for a unique solution.

To solve this we need 2 more equations (constraints) which we usually put at *domain endpoints* (the beginning and end) called *boundary conditions* or *end conditions*.

Boundary Conditions (BC)

- **Clamped**: slope are set to some value (both boundaries: *complete* or *clamped spline*)

$$S'(x_1) = \textit{specified} \qquad S'(x_n) = \textit{specified}$$

- **Free**: second derivatives are set to zero (both boundaries: *natrual cubic spline*)

$$S'(x_1) = 0 \qquad S''(x_n) = 0$$

- **Periodic**: match start and end derivatives (wrap-around behaviour)

$$S'(x_1) = S'(x_n) \qquad S''(x_1) = S''(x_n)$$

- **Not-a-knot**: match 3rd derivatives on end segments (second last and second point are not knots)

$$S_1'''(x_2) = S_2'''(x_2) \qquad S_{n-2}'''(x_{n-1}) = S_{n-1}'''(x_{n-1})$$

Efficient Construction of Cubic Splines via Hermite Interpolation

For a general linear system Gaussain elimination takes $O(N^3)$ time however cubic splines produces a special linear system that can be solved in $O(N)$ time.

- Hermite Interpolation: solve $n-1$ separate systems of 4 equations
- Cubic Splines: solve 1 system of $4(n-1)$ equations

Cubic Splines via Hermite Interpolation

We use Hermite interpolation as a stepping stone to build a cubic spline

$$\begin{aligned} S_i(x) &= a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \\ S_i'(x) &= b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2 \\ S_i''(x) &= 2c_i + 6d_i(x - x_i) \end{aligned}$$

If we use the closed form Hermite and force $S_i''(x_{i+1}) = S_{i+1}''(x_{i+1})$ we can satisfy the cubic spline definition

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

$$S_{i+1}''(x_{i+1}) = 2\left(\frac{3y_{i+1}' - 2s_{i+1} - s_{i+2}}{\Delta x_{i+2}}\right) + 6\left(\frac{-2y_{i+1}' + s_{i+1} + s_{i+2}}{\Delta x_{i+2}^2}\right)\cancel{(x_{i+1} - x_{i+1})}^{\to 0}$$

$$\begin{aligned} S_i''(x_{i+1}) &= S_{i+1}''(x_{i+1}) \\ \frac{-6y_i' + 2s_i + 4s_{i+1}}{\Delta x_i} &= \frac{6y_{i+1}' - 4s_{i+1} - 2s_{i+2}}{\Delta x_{i+1}} \\ \Delta x_{i+1}s_i + 2(\Delta x_i + \Delta x_{i+1})s_{i+1} + \Delta x_i s_{i+2} &= 3(\Delta x_{i+1}y_i' + \Delta x_i y_{i+1}') \\ \Delta x_i s_{i-1} + 2(\Delta x_{i-1} + \Delta x_i)s_i + \Delta x_{i-1}s_{i+1} &= 3(\Delta x_i y_{i-1}' + \Delta x_{i-1}y_i') \qquad \text{(reindex)} \end{aligned}$$

After solving for s_i we plug them into the closed form Hermite equations to get a_i, b_i, c_i, d_i that also match the interval endpoint data and first derivative.

Remark: s_i is the *slope* at *node* i while $S_i(x)$ is the *polynomial* at *interval* i

Efficient Splines Summary

For $i = 2$ to $n - 1$ we have the equation

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

For the 2 *boundary conditions* for the ends $i = 1$ and $i = n$

- **Clamped**: slope is set to some value

$$S'_1(x_1) = \textit{specified} \qquad S'_{n-1}(x_n) = \textit{specified} \qquad \rightarrow \qquad s_1 = S'_1(x_1) \qquad s_n = S'_1(x_n)$$

- **Free**: zero for the 2nd derivative at the ends

$$S''_1(x_1) = 0 \qquad S''_{n-1}(x_n) = 0 \qquad \rightarrow \qquad s_1 + \frac{1}{2}s_2 = \frac{3}{2}y'_1 \qquad \frac{1}{2}s_{n-1} + s_n = \frac{3}{2}y'_{n-1}$$

After solving plug s_i into the closed-form Hermite expression for each interval

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

$$a_i = y_i \qquad b_i = s_i \qquad c_i = \frac{3y'_i - 2s_i - s_{i+1}}{\Delta x_i} \qquad d_i = \frac{-2y'_i + s_i + s_{i+1}}{\Delta x_i^2}$$

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

Example Problem

Fit a spline to 4 points $(0, 1), (2, 1), (3, 3), (4, -1)$ with clamped BC of $S'_1(x_1) = 1$ and $S'_3(x_4) = -1$

Computing Δx_i and y'_i for intervals $i = 1$ to 3

$$\Delta x_1 = 2 \qquad \Delta x_2 = 1 \qquad \Delta x_3 = 1 \qquad y'_1 = 0 \qquad y'_2 = 2 \qquad y'_3 = -4$$

Then we construct a system $Ts = r$

- $i = 1$ is a boundary

$$s_1 = 1 \qquad T_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \qquad r_1 = 1$$

- $i = 2$ is an interior point

$$s_1 + 2(2 + 1)s_2 + 2s_3 = 3(1 - 0 + 2 \cdot 2) \quad \rightarrow \quad s_1 + 6s_2 + 2s_3 = 12$$

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

- $i = 3$ is an interior point

$$s_2 + 2(1 + 1)s_3 + s_4 = 3(1 \cdot 2 + 1 \cdot -4) \quad \rightarrow \quad s_2 + 4s_3 + s_4 = -6$$

$$T_3 = \begin{bmatrix} 0 & 1 & 4 & 1 \end{bmatrix} \qquad r_3 = -6$$

- $i = 4$ is a boundary

$$T_4 = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \qquad r_4 = -1$$

Now we have the system

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 6 & 2 & 0 \\ 0 & 1 & 4 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 12 \\ -6 \\ -1 \end{bmatrix}$$

After solving for s_i and plugging into Hermite closed form we will have the coefficents for $S_i(x)$

Efficient Splines Matrix Structure

Notice that matrix T is *tridiagonal* meaning it only has entries on the diagonals and its neighbours.

$$\begin{bmatrix} X & X & & & \\ X & X & X & & \\ & X & X & X & \\ & & X & X & X \\ & & & X & X \end{bmatrix}$$

A tridiagonal matrix is one type of sparse matrix and we only need a vector of 3-tuples to store it. In addition, we will later figure out how to solve this system much faster than general systems.

Parametric Curves

So far we have only handled functions $y = p(x)$ but what if we want a curve that folds back over itself (multiple outputs at a single x)

Parametric curves will allow us to handle this more general case:

- Let x and y each be separate function of a new parameter t
- Then a point's position is given by a **vector**

$$\vec{P(t)} = (x(t), y(t))$$

  we say the curve is *parameterized* by t which can be thought of as time (still only plot x and y)

- Parametric curves are not a specific type of curve but a way of using curves like piecewise linear, Hermite, cubic splines, Bezier, etc

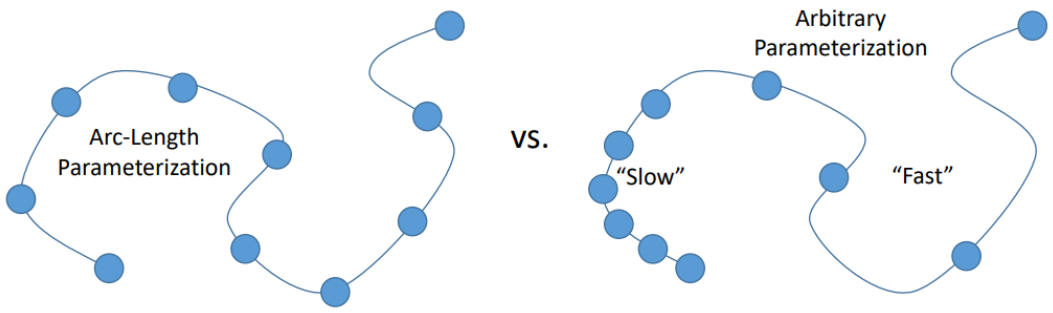
Example: for a semi-circle the implicit equation is $x^2 + y^2 = 1$ and some parametric forms are:

- $(x(t), y(t)) = (\cos(\pi t),\ \sin(\pi t))$ for $0 \le t \le 1$
- $(x(t), y(t)) = (\cos(\pi(1-t)),\ \sin(\pi(1-t)))$ for $0 \le t \le 1$
- $(x(t), y(t)) = (\cos(\pi t^2),\ \sin(\pi t^2))$ for $0 \le t \le 1$

The curve is *parameterized* in different ways that give the same result
(for full circle we just change the bounds to $0 \le t \le 2$)

Arc-Length Parameterization

A common parameterization is to choose t as the *distance along the curve*.



For piecewise polynomials we can estimate this by taking the distance between two nodes:

$$t_1 = 0 \qquad t_{i+1} = t_i + \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

Using this parameterization we then:

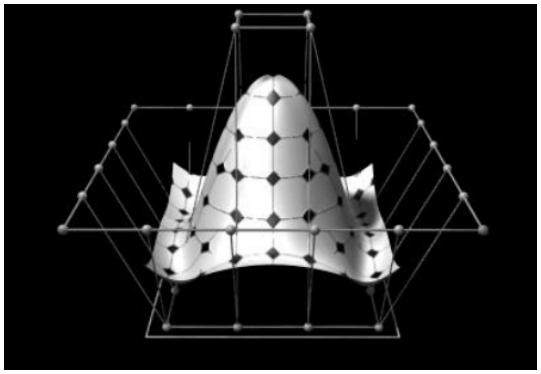
- Compute how the spline changes in the x direction over t
- Compute how the spline changes in the y direction over t
- Plot the resulting x and y values

```
# convert input points in to a numpy array then split x and y values
arr = np.array(input)
n = arr.shape[0]
x_arr = arr[:,0]
y_arr = arr[:,1]

# generate approximate arc-length parameterization
t = [0]
for i in range(1, n):
    t.append(t[i-1] + sqrt((x_arr[i] - x_arr[i-1])**2 + (y_arr[i] - y_arr[i-1])**2))

# use approximate arc-length parameterization to create splines
xspline = make_interp_spline(t, x_arr)
yspline = make_interp_spline(t, y_arr)
v = np.linspace(0, np.amax(t), 250)     # could replace np.amax(t) with t[-1]
xs = xspline(v)
ys = yspline(v)

plt.plot(xs, ys)
```

Going beyond

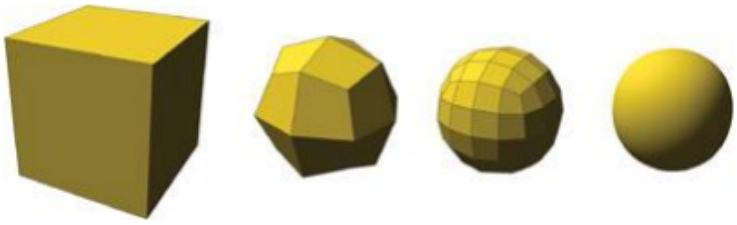
Curves to Surfaces ideas for smooth curves can be generalized to 2D *surfaces* in 3D for example some commond surface types in comptuer aided design (CAD) include:

- Non-Uniform Rational B-splines (NURB)



A NURB surface with its control mesh of nodes

- Subdivision Surfaces



Cube mesh being subdivided recursively to form smooth subdivision surface

At Waterloo Professor Steve Mann studies applications of splines in CAD and 5-axis machining.

There is also the graduate level course "CS 779 Splines" to learn lots more.

Ordinary Differential Equations (ODE)

Differential equations relate some unknown function to its derivatives. They have many applications from modeling physical systems to predicting the market. In this course we will not be solving the ODEs *analytically* instead we will attempt to use *numerical methods* to find *approximate solutions*.

In general a first order ODE will have the form:

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

At any particular time t

- know the direction and magnitude that y is going
- don't know what the value of y is

Consider the simplest ODE $y'(t) = \alpha \cdot y(t)$ this has a closed form solution

$$y(t) = y_0 \cdot e^{\alpha(t-t_0)} \qquad \leftrightarrow \qquad y(t) = Ce^{\alpha t}$$

We see that at any particular time t

- direction of mouse population is a constant α times the current number of mice $y(t)$
- number of mice cannot be said until we know number of mice at some other time

Initial Value Problems (IVP) will give the value of y at time t_0 to be some constant y_0

$$y'(t) = f(t, y(t)) \quad (f \text{ is Dynamics Function}) \qquad\qquad y(t_0) = y_0 \quad (\text{Initial Conditions})$$

System of Differential Equations: a model with *multiple* variables that interact with each other (Ex: (x, y) coordinate of some moving object)

Two dynamics functions: $\begin{bmatrix} x(t) \\ y(t) \end{bmatrix}' = \begin{bmatrix} f_x(t, x(t), y(t)) \\ f_y(t, x(t), y(t)) \end{bmatrix}$ *Two* initial conditions: $\begin{bmatrix} x(t_0) \\ y(t_0) \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$

Time Stepping: method of *numerically* solving ODEs given some initial conditions by repeatedly making steps of size h each time and using y' to update the y value.

- Begin: $n = 0$, $t = t_0$, $y = y_0$

- Repeat: compute y_{n+1} using y and y', increment time $t_{n+1} = t_n + h$, advance $n = n + 1$

There are several different varieties of time-stepping methods:

- Single-step vs Multistep: only use information from current step, or from previous timesteps as well
- Explicit vs Implicit: can we evalute y_{n+1} directly, or do we need to solve some implicit equation
- Timestep size: do we use a constant timestep h, or allow it to vary

Forward Euler Time-Stepping

Forward Euler is very simple explicit and single-step scheme that just repeats the following:

- Compute the current slope
$$y'_n = f(t_n, y_n)$$
- Step in a straight line with that slope:
$$y_{n+1} = y_n + h \cdot y'_n$$

This can be rewritten as into a **recurrence relationship** (note that y_0 should be a given):

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

This directly gives a new value y_{n+1} at the new time $t_{n+1} = t_n + h$

Note: at every step we know value y_n, time t_n, time step h, and get the slope by evaluting f

Forward Euler Examples

First Example

$$y'(t) = 2y(t) \qquad y(t_0) = y_0 = 3 \qquad t_0 = 0$$

The analytical solution for this is $y(t) = 3e^{2t}$

FE with step size $h = 1$ until $t = 3$

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

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

FE with step size $h = \frac{1}{2}$ until $t = 3$

$$y_{n+1} = y_n + (\tfrac{1}{2})(2y_n) = 2y_n$$

n	t_n	y_n	$y(t_n)$
0	0	3	3
1	0.5	6	8.15
2	1	12	22.2
3	1.5	24	60.3
4	2	48	163.8
5	2.5	96	445
6	3	192	1210

The approximate values drift away from actual values quickly and smaller step size FE has a smaller error.

Remark: When implementing FE in code leave f as a black box for generality

Remark: $y(t_n)$ is the *exact value* at time t_n while y_n is the approximate/discrete data at step n (at step n it is time t_n)

$$\begin{aligned} f \text{ evaluted on } \textit{numerical} \text{ data:} &\quad f(t_n, y_n) \\ f \text{ evaluted on } \textit{exact} \text{ value of } y\text{:} &\quad f(t_n, y(t_n)) \end{aligned}$$

Second Example:

$$x'(t) = -y(t) \qquad y'(t) = x(t) \qquad x(t_0) = 2 \qquad y(t_0) = 0 \qquad t_0 = 2$$

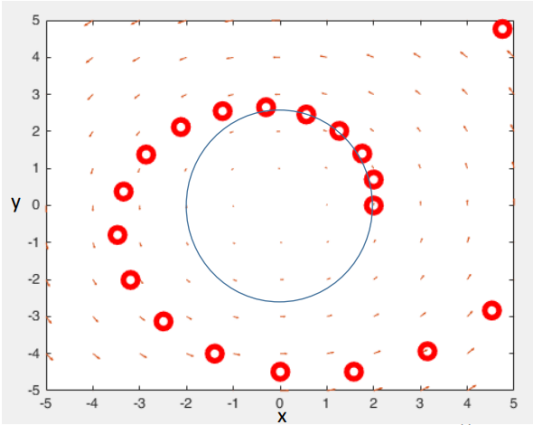
To solve numerically with $h = 2$ apply FE $y_{n+1} = y_n + hf(t_n, y_n)$ to the vector form of this IVP

$$\begin{bmatrix} x(t) \\ y(t) \end{bmatrix}' = \begin{bmatrix} -y(t) \\ x(t) \end{bmatrix} \qquad \rightarrow \qquad \begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \end{bmatrix} + 2 \begin{bmatrix} -y_n \\ x_n \end{bmatrix} = \begin{bmatrix} x_n - 2y_n \\ y_n + 2x_n \end{bmatrix}$$

Solving until $t = 8$ we can generate the table:

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

Plotting this numerical solution we notice that FE exhibits large error/drift (blue circle is correct)



Deriving Forward Euler

Recall how the Taylor series for f at some point a is defined as:

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + \cdots$$

Unrelated: function f is equal to its Taylor series if it is *analytic* (most functions are)

There are two ways we can arrive at Forward Euler:

- **Finite Differences View**: approximate the deriviative y' using a finite difference

$$y'(t_n) = f(t, y(t)) \quad \rightarrow \quad y'(t_n) \approx \frac{y_{n+1} - y_n}{t_{n+1} - t_n}$$

Since we defined $h = t_{n+1} - t_n$ we have

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

Rearranging this we get FE $y_{n+1} = y_n + hf(t_n, y_n)$

- **Taylor Series View**: Taylor series for y at t_n is (just assume y is analytic)

$$y(x) = \sum_{n=0}^{\infty} \frac{y^{(n)}(t_n)}{n!}(x - t_n)^n$$

To get $y(t_{n+1})$ we just sub $x = t_{n+1}$ and since $h = t_{n+1} - t_n$ we get

$$y(t_{n+1}) = \sum_{n=0}^{\infty} \frac{y^{(n)}(t_n)}{n!}(t_{n+1} - t_n)^n = \sum_{n=0}^{\infty} \frac{y^{(n)}(t_n)}{n!}h^n$$

assume that terms of order h^2 or higher will be small as $h \to 0$ and drop them to get FE

$$y(t_{n+1}) \approx y(t_n) + hy'(t_n) = y(t_n) + hf(t_n, y_n)$$

Error of Forward Euler

- **Local Truncation Error (LTE)**: the error incurred between step n and $n + 1$:

$$|y_{n+1} - y(t_{n+1})| \quad \text{(assuming exact data at time } t_n\text{)}$$

- **Absolute/Global Error**: is the total error incurred to get to step n:

$$|y_n - y(t_n)|$$

For Forward Euler assume y_n is exact then turning $y(t_{n+1})$ into a Taylor series we get the LTE:

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

as h decreases, LTE decreases *quadratically.* (We will find global error later)

Note: Big O in this case is used to show the dominant term is the slowest decreasing one as $h \to 0$ (typically the lowest power of h)

Example: as $x \to 0$ for $g(x) = 5x^3 - 9x^4 + 2x^9$ then

$$g(x) = O(x^3) \qquad g(x) = 5x^3 + O(x^4) \qquad g(x) \neq O(x^9)$$

Higher Order Time Stepping Schemes

Derivative Approximation

Consider the Taylor expansion for $y(t_{n+1})$ at t_n then rearrange

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

Thus we state that the *derivative approximation* has error $O(h)$

More Accurate Time-Stepping

If we add another term from the Taylor series we could $O(h^3)$ error:

$$y(t_{n+1}) = \underbrace{\underbrace{y(t_n) + hy'(t_n)}_{\text{FE: } O(h^2) \text{ error}} + \frac{h^2}{2}y''(t_n)}_{O(h^3) \text{ error}} + \frac{h^3}{3!}y'''(t_n) + \cdots$$

Notice that we don't know what y'' is so we will just approximate it using the finite difference method:

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

Trapezoidal Rule (Crank-Nicolson)

Subbing $y'(t) = f(t, y(t))$ and the finite difference for y'' into the Taylor series we get

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

replacing the exact values with approximations we get

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

The intuition is the we are averging the slopes of y at t_n and t_{n+1} to get the next step.

Explicit vs Implicit Schemes

Explicit: only y_n or earlier data appears on RHS

- Simpler and fast to compute *per step*
- Less stable so usually want to use smaller timesteps to avoid compounding errors
- Forward Euler is an *explicit scheme* and can be evaluted directly

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

Implicit: unknown y_{n+1} appears on both sides

- Often more complex and expensive to solve *per step*
- More stable so can safely use larger timesteps
- Trapezoidal is an *implicit scheme* so to solve for y_{n+1} we need to rearrange for each different f

$$y_{n+1} = y_n + \frac{h}{2}[f(t_n, y_n) + f(t_{n+1}, y_{n+1})]$$

Computation tradeoff between few expensive large steps (implicit) vs many cheap small steps (explicit).

Modified/Improved Euler

We use a Forward Euler-like step as the basis for the Trapezoidal-like step

$$y^*_{n+1} = y_n + hf(t_n, y_n)$$

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

This allows us to turn the Trapezoidal method into an explicit scheme with the same LTE of $O(h^3)$

Improved Euler Example

$$x'(t) = -y(t) \qquad y'(t) = x(t) \qquad x(t_0) = 2 \qquad y(t_0) = 0 \qquad t_0 = 0$$

Generate our improved Euler steps with $h = 2$:

- FE step:

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

- Trapzoidal step

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

Now we apply improved Euler to the IVP

- Step 1

$$\begin{bmatrix} x^*_1 \\ y^*_1 \end{bmatrix} = \begin{bmatrix} x_0 - 2y_0 \\ y_0 - 2x_0 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_0 - y_0 - y^*_1 \\ y_0 + x_0 + x^*_1 \end{bmatrix} = \begin{bmatrix} 2 - 0 - 4 \\ 0 + 2 + 2 \end{bmatrix} = \begin{bmatrix} -2 \\ 4 \end{bmatrix}$$

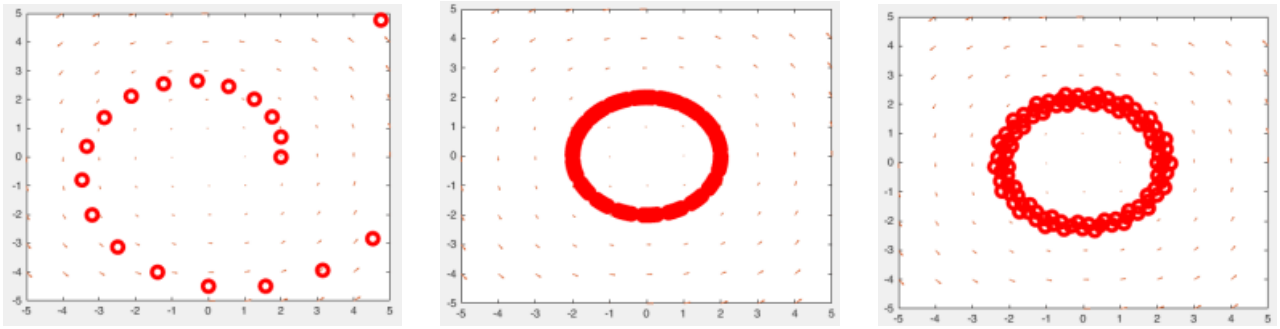
- Step 2

$$\begin{bmatrix} x^*_2 \\ y^*_2 \end{bmatrix} = \begin{bmatrix} x_1 - 2y_1 \\ y_1 - 2x_1 \end{bmatrix} = \begin{bmatrix} -10 \\ 0 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1 - y_1 - y^*_2 \\ y_1 + x_1 + x^*_2 \end{bmatrix} = \begin{bmatrix} -2 - 4 + 0 \\ 4 - 2 - 10 \end{bmatrix} = \begin{bmatrix} -6 \\ -8 \end{bmatrix}$$

Now we have the table

n	t_n	x_n	y_n	x^*_n	y^*_n
0	0	2	0		
1	2	−2	4	2	4
2	4	−6	−8	−10	0

Time Integration Methods So Far



exact values should make a circle

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

More Time Stepping Schemes

Backwords Euler Method

Similar to forward Euler, but implicit (same truncation error of $O(h^2)$)

- Forward Euler: uses the slope from the *start* of the step

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

- Backwards Euler: uses the slope from the *end* of step

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

Explicit Runge Kutta Schemes

Runge Kutta is a family of **explicit** schemes written in a specific form

- Improved Euler is rewritten as

$$y_{n+1} = y_n + \frac{k_1}{2} + \frac{k_2}{2} \qquad k_1 = hf(t_n, y_n) \qquad k_2 = hf(t_n + h, y_n + k_1)$$

- Midpiont Method: use the slope found at the approximate middle of the step (LTE $O(h^3)$)

$$y_{n+1} = y_n + k_2 \qquad k_1 = hf(t_n, y_n) \qquad k_2 = hf(t_n + \tfrac{h}{2}, y_n + \tfrac{k_1}{2})$$

There are many Runge Kutta schemes for higher error orders $O(h^\alpha)$ for $\alpha = 3, 4, 5, 6, \ldots$

4th Order Runge Kutta (RK4)

The *Classical* Runge-Kutta or RK4 with an LTE of $O(h^5)$

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \qquad \begin{array}{cc} k_1 = h \cdot f(t_n, y_n) & k_2 = h \cdot f\left(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) \\ k_3 = h \cdot f\left(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right) & k_4 = h \cdot f\left(t_n + h, y_n + k_3\right) \end{array}$$

Global Error

We previously only looked at the error for a single step but error will accumulate with each step (drift). Global error is the total error accumulated at the final time t_{final}

For constant steps h we multiply our LTE with the number of steps to get

$$\#\text{steps} = \frac{t_{final} - t_0}{h} = O(h^{-1}) \qquad \rightarrow \qquad \text{Global error} \leq \text{Local Error} \cdot O(h^{-1})$$

In other words global error is one degree lower than the LTE.

Examples:

- Forward Euler: $O(h^2) \cdot O(h^{-1}) = O(h)$ or 1st order accurate
- Improved Euler: $O(h^3) \cdot O(h^{-1}) = O(h^2)$ or 2nd order accurate
- RK4: $O(h^5) \cdot O(h^{-1}) = O(h^4)$ or 4th order accurate

Multistep Schemes

So far we have only seen *single-step* scemes which only use info from current time t_n (and forward) to solve for y_{n+1}. *Multi-step* methods also use information from earlier timesteps t_{n-1}, t_{n-2}, etc..

Implicit Multistep: Backwards Differentiation Formula (BDF) Methods

Generalize backwards Euler into the family of methods BDF1, BDF2, BDF3, etc where the number indicates order of global error.

- BDF1 is just backwards euler and is a single step scheme

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

- BDF2 uses current *and previous* step data

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

- In general, BDF schemes are written as

$$\sum_{k=-s+1}^{1} a_k y_{n+k} = hf(t_{n+1}, y_{n+1})$$

  where a_k are some coefficients

Deriving BDF coefficients via Interpolation requires 3 steps:

1. Fit $p(t)$ using Lagrange polynomials to unknown point (t_{n+1}, y_{n+1}) and one or more earlier points
2. Find $p'(t)$ by differentiating
3. Require end-of-step slope to match: $p'(t_{n+1}) = f(t_{n+1}, y_{n+1})$ then rearrange for y_{n+1}

Example: Deriving BDF2 via Interpolation:

1. Fit Lagrange polynomial to time steps: t_{n+1}, t_n, t_{n-1} and recall that $h = t_{n+1} - t_n$

$$\begin{aligned} p(t) &= y_{n+1}\frac{(t-t_n)(t-t_{n-1})}{(t_{n+1}-t_n)(t_{n+1}-t_{n-1})} + y_n\frac{(t-t_{n+1})(t-t_{n-1})}{(t_n-t_{n+1})(t_n-t_{n-1})} + y_{n-1}\frac{(t-t_{n+1})(t-t_n)}{(t_{n-1}-t_{n+1})(t_{n-1}-t_n)} \\ &= \frac{y_{n+1}}{2h^2}(t-t_n)(t-t_{n-1}) + \frac{y_n}{-h^2}(t-t_{n+1})(t-t_{n-1}) + \frac{y_{n-1}}{2h^2}(t-t_{n+1})(t-t_n) \end{aligned}$$

2. Determine $p'(t)$

$$p'(t) = \frac{y_{n+1}}{2h^2}(2t-t_n-t_{n-1}) + \frac{y_n}{-h^2}(2t-t_{n+1}-t_{n-1}) + \frac{y_{n-1}}{2h^2}(2t-t_{n+1}-t_n)$$

3. Require $p'(t_{n+1}) = f(t_{n+1}, y_{n+1})$ then rearrange for y_{n+1}

$$\begin{aligned} p'(t_{n+1}) &= \frac{y_{n+1}}{2h^2}(2t_{n+1}-t_n-t_{n-1}) + \frac{y_n}{-h^2}(2t_{n+1}-t_{n+1}-t_{n-1}) + \frac{y_{n-1}}{2h^2}(2t_{n+1}-t_{n+1}-t_n) \\ &= \frac{y_{n+1}}{2h^2}(h+2h) + \frac{y_n}{-h^2}(2h) + \frac{y_{n-1}}{2h^2}(h) \\ &= \frac{3y_{n+1}}{2h} + \frac{2y_n}{-h} + \frac{y_{n-1}}{2h} = f(t_{n+1}, y_{n+1}) \end{aligned}$$

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

Explicit Multistep: Adams-Bashforth

2nd order Adams-Bashforth:

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

We know all the data at t_n and t_{n-1} so we can directly evaluate to get y_{n+1}, has LTE of $O(h^3)$

Higher Order ODEs

We can convert a higher ODE to a lower order using

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

This allows us to convert higher order ODEs into a *system of first order ODEs.*

Method: for the term $y^{(n)}$ with the largest degree, if $n > 1$ then introduce new variables

$$y_i = y^{(i-1)} \qquad \text{for } i = 1, \ldots, n$$

substituting these new variables into the original ODE leads to:

- one first order equation for each original equation
- one more more addition equations *relating* the new variables

Example 1:

$$y''(t) = ty(t) \qquad \text{with } y(1) = 1 \text{ and } y'(t) = 2$$

- Introduce variables $y_i = y^{(i-1)}$ for $i = 1, 2$

$$y_1(t) = y(t) \qquad y_2(t) = y'(t)$$

- Plug into ODE to get us a system of 1st order ODEs

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

  with initial conditions $y_1(t) = 1$ and $y_2(1) = 2$

Example 2:

$$y^{(4)}(t) - 3y'''(t) + \sin(t)y''(t) - 7ty(t) = e^t$$

- Introduce variables $y_i = y^{(i-1)}$ for $i = 1, \ldots, 4$

$$y_1(t) = y(t) \qquad y_2(t) = y'(t) \qquad y_3(t) = y''(t) \qquad y_4(t) = y'''(t)$$

- Plug into ODE to get one first order equation:

$$y_4'(t) = 3y_4(t) - \sin(t)y_3(t) + 7ty_1(t) + e^t$$

- Add relationships between the new variables to the system

$$y_3'(t) = y_4(t) \qquad y_2'(t) = y_3(t) \qquad y_1'(t) = y_2(t)$$

  This gives us a system of 4 first order linear ODEs

Stability

Errors are generally $O(h^p)$ for some value of p so our schemes are *less accurate* for large values of h.

Stability is the measure of how much an error in the intial conditions will change the results:

$$y'(t) = f(t, y(t)) \qquad y(0) = y_0 + \epsilon_0$$

If the intial error ϵ_0 grows exponentially for many steps $n \to \infty$ our scheme is **unstable**.

To examine the stability of time-stepping schemes we will use

$$y'(t) = -\lambda y(t) \qquad y(0) = y_0 \qquad \text{where } \lambda > 0$$

as our **test equation** becuase of its simplicity. Then we have 3 steps:

1. Apply the time stepping scheme to the 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 (error approaches zero)

Explicit Scheme Stability: Forward Euler

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

Applying FE to our test equation $y'(t) = -\lambda y(t)$ and $y(0) = y_0$ with $\lambda > 0$

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

- Closed form for numerical solution is $y_n = y_0(1 - h\lambda)^n$ for n timesteps
- True solution is $y(t) = y_0 e^{-\lambda t}$ which decays to zero

Numerical solution goes to zero only when

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

- LHS says $-1 < 1 - h\lambda$ which becomes $h < \frac{2}{\lambda}$
- RHS says $1 - h\lambda < 1$ which becomes $-h\lambda < 0$ or $h > 0$ (this always holds)

Hence the numerical solution decays to 0 (like the true solution) when $h < \frac{2}{\lambda}$ otherwise it will *blow up*.

If we perturb the initial conditions with error ϵ_0 then:

$$y_0^{(p)} = y_0 + \epsilon_0 \quad \to \quad y_n^{(p)} = (y_0 + \epsilon_0)(1 - h\lambda)^n$$

Taking the difference to error $\epsilon_n = y_n^{(p)} - y_n$ becomes $\epsilon_n = \epsilon_0(1 - h\lambda)^n$. We conclude the error has the same behaviour for this test equation (decays to 0 for $h < \frac{2}{\lambda}$)

We say Forward Euler is **Conditionally Stable** (stable only when h satisfies the *stability condiiton*)

Implicit Scheme Stability: Backword Euler

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

Applying BE to our test equation $y'(t) = -\lambda y(t)$ and $y(0) = y_0$ with $\lambda > 0$

$$y_{n+1} = y_n + h(-\lambda y_{n+1}) \quad \rightarrow \quad y_{n+1} = \frac{y_n}{(1+h\lambda)} \quad \rightarrow \quad y_n = \frac{y_0}{(1+h\lambda)^n} \text{ (closed form)}$$

Since $h > 0$ and $\lambda > 0$ the value decreases in magnitude as $n \to \infty$ for any timestep h

By the same logic as for FE the error/perturbation follows the same form

$$\epsilon_n = \frac{\epsilon_0}{(1+h\lambda)^n}$$

So BE is **Unconditionally Stable** (stable for any $h > 0$)

- Generally implicit schemes are more stable than explicit schemes at the cost of being potentially more expensive to solve
- Stabiliy *does not imply* accuracy, if we take a very large timestep we will still get large errors

Explicit Scheme Stability: Improved Euler

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

Applying IE to our test equation $y'(t) = -\lambda y(t)$ and $y(0) = y_0$ with $\lambda > 0$

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

Closed form and error:

$$y_n = y_0 \left(1 - h\lambda + \frac{h^2\lambda^2}{2}\right)^n \qquad \rightarrow \qquad \epsilon_n = \epsilon_0 \left(1 - h\lambda + \frac{h^2\lambda^2}{2}\right)^n$$

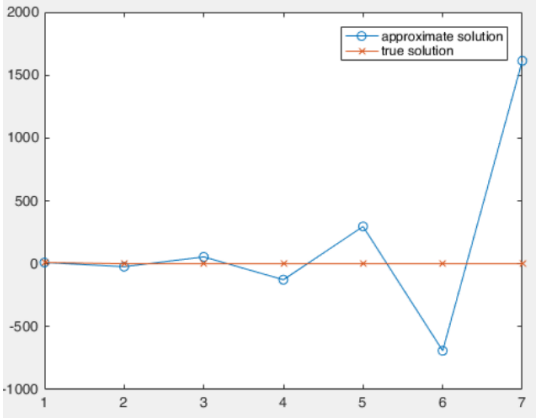
Then IE is stable when

$$\left|1 - h\lambda + \frac{h^2\lambda^2}{2}\right| < 1$$

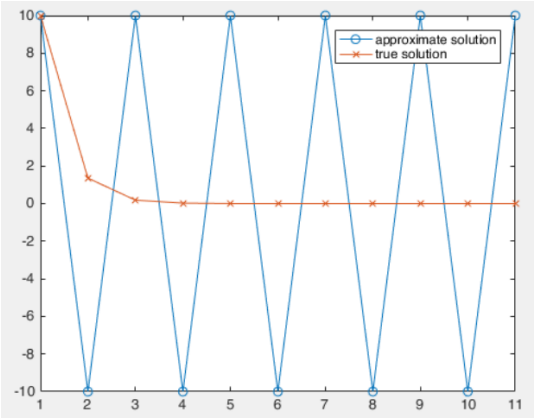
after some algebra this yields $h < \frac{2}{\lambda}$ for stability which is the same as FE.

Test Equation Stability Visualized

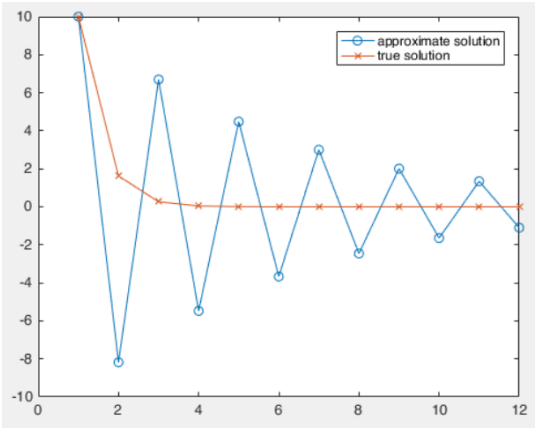
Graphs of the numerical approximation of the test equation using Forward Euler and Backword Euler.



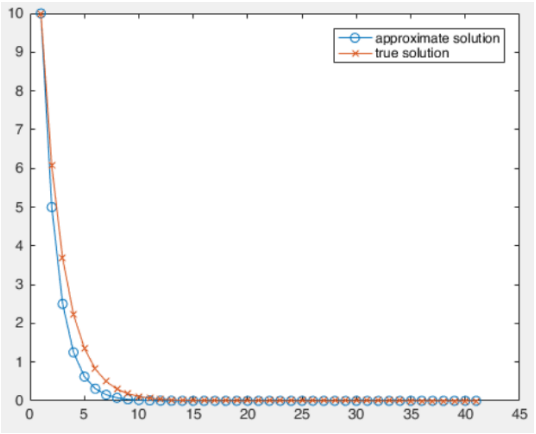
Forward Euler $h > 2/\lambda$ (unstable)



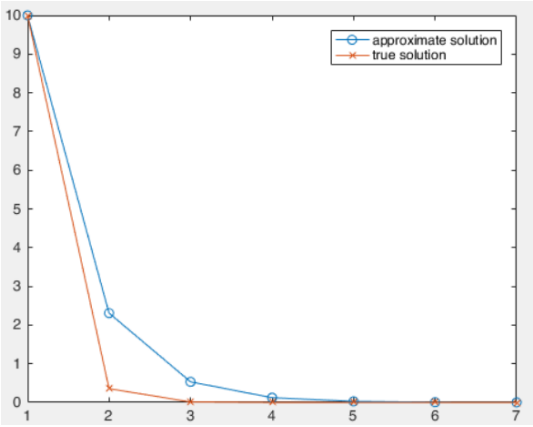
Forward Euler $h = 2/\lambda$ (borderline stable)



Forward Euler $h < 2/\lambda$ (stable but inaccurate)



Forward Euler $h \ll 2/\lambda$ (stable and accruate)



Backward Euler $h > 2/\lambda$ (stable)

- Stability: how the numerical algorithm will react to small errors
- Truncation Error: how the accuracy of our numerical solution scales with time step h (in absence of stability issues)

Both factors need to be considered to get a useful solution.

Stability in General (Beyond Test Equation)

There is a method for extrapolate the results of the test equation to other DEs however it is outside the scope of this course.

- For non-linear problems we transform it into a linear equation locally and test stability there (may need to do multiple times)
- for system of ODEs stability relates to the *eigenvalues* of the Jacobian matrix

Local Truncation Error

Previously we got truncation error as a byproduct of the derivation of the scheme (e.g. via Taylor series)

In order to find LTE for methods we don't know the derivation for recall that:

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

Assuming exact RHS data this is the error from taking one step. For time-stepping scheme $y_{n+1} = \text{RHS}$:

1. Replace approximations on RHS with *exact* versions
2. Taylor expand all RHS quantities about time t_n
3. Taylor expand the *exact* soltuion $y(t_{n+1})$ to compare against
4. Compute difference $y(t_{n+1}) - y_{n+1}$. Lowest degree non–canceling power of h gives the LTE.

Taylor Expansion:

- General Taylor series for y at time t_n:

$$y(x) = \sum_{n=0}^{\infty} \frac{y^{(n)}(t_n)}{n!}(x - t_n)^n$$

- Since $h = t_{n+1} - t_n$ if we use the above to estimate the value of y at t_{n+1} or at t_{n-1}

$$y(t_{n+1}) = \sum_{n=0}^{\infty} \frac{y^{(n)}(t_n)}{n!}h^n \qquad\qquad y(t_{n-1}) = \sum_{n=0}^{\infty} \frac{y^{(n)}(t_n)}{n!}(-h)^n$$

- Remark using Taylor series created at t_n to estimate t_n will just get $y(t_n) = y(t_n)$

Local Truncation Error of Trapezoidol

$$y_{n+1} = y_n + \frac{h}{2}\left(f(t_n, y_n) + f(t_{n+1}, y_{n+1})\right)$$

1. Replace RHS quantities with exact counterparts

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

2. Taylor expand the RHS quantities about time t_n (using $h = t_{n+1} - t_n$)

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

Then

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

3. Write down the 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. Find the difference:

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

Thus the local truncation error of Trapezoidol is $O(h^3)$

Local Truncation Error of BDF2

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

- Replace RHS data with exact versions

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

- Taylor expand RHS expressions (using $-h = t_{n-1} - t_n$)

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

Then

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

- Find difference between above and the true Taylor series:

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

Adaptive Time-Stepping

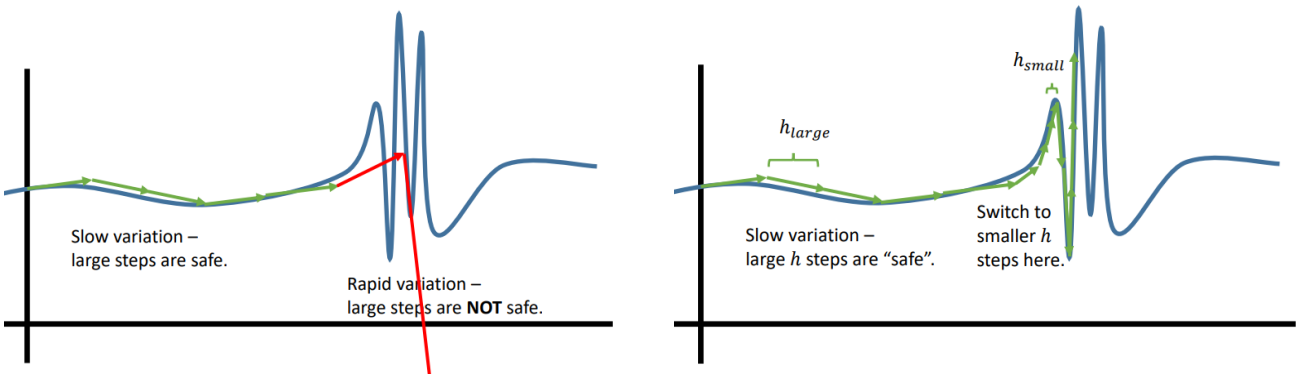
So far we had a constant h given to us. If given the choice a smaller h yields smaller error but:

- It will have a *higher computational cost*
- Too small of a time step will cause no extra benefit because of floating point errors

To minimize cost we want to the choose largest h with error still within a desired tolerance

$$\text{error} < \text{tolerance}$$

To do this with a fixed h is difficult because the rate of change of different parts can vary extremely:



Instead we can adapt the time step to minimize our effort while still getting the tolerance we want.

Basic Algorithm

The goal is to control h to always satisfy:

$$\text{error} < \text{tolerance}$$

however knowing the exact error is the same as knowing the solution.

Instead we use **two** different time-stepping schemes **together** then compare their results to estimate error and adjust h accordingly:

1. Compute the approximate solutions for one time step with 2 schemes of different orders

$$y^A_{n+1} = \text{Method A} \quad \text{with } O(h^p)$$
$$y^B_{n+1} = \text{Method B} \quad \text{with } O(h^{p+1})$$

2. Approximate the error by taking their difference

$$err = |y^A_{n+1} - y^B_{n+1}|$$

3. While error > tolerance: set $h := h/2$ then go back to step 1

4. Estimate error coefficient c and predict a good *next* step size h_{new}

$$c \approx \frac{|y^A_{n+1} - y^B_{n+1}|}{(h^p_{old})}$$

where h_{old} is the most recent timestep size. Then we use c to estimate the next time step error:

$$err_{next} = |y^A_{n+2} - y(t_{n+2})| \approx c(h^p_{new}) = \frac{|y^A_{n+1} - y^B_{n+1}|}{(h^p_{old})}(h^p_{new}) = |y^A_{n+1} - y^B_{n+1}| \left(\frac{h_{new}}{h_{old}}\right)^p$$

Then set err_{next} to our tolerance 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 be safe scale tol by some factor ℓ (Ex: $\ell = 1/2$ or $\ell = 3/4$)

5. Repeat from step 1 until end time is reached

Step 3 will shrink our step size while step 4 is allows our step size to grow larger when it is safe.

Justification: observe that for some value of p and c (notice that c is the error coefficient)

$$y^A_{n+1} = y(t_{n+1}) + ch^p + O(h^{p+1}) \qquad y^B_{n+1} = y(t_{n+1}) + O(h^{p+1})$$

- Method A's true error is:
$$|y^A_{n+1} - y(t_{n+1})| = ch^p + O(h^{p+1})$$
- Method A's estimated error is:
$$|y^A_{n+1} - y^B_{n+1}| = ch^p + O(h^{p+1})$$

Since the dominant component of the error matches we can say our estimate is a decent approximation.

Usage of Adaptive Time-Stepping

Matlab's `ODE45` and SciPy's `RK45` uses the *Dormand-Prince* scheme which uses the 4th and 5th order Runge-Kutta schemes:

$$\begin{aligned}
k_1 &= hf(t_n, y_n) \\
k_2 &= hf(t_n + \tfrac{h}{4}, y_n + \tfrac{k_1}{4}) \\
k_3 &= hf(t_n + \tfrac{3h}{8}, y_n + \tfrac{3k_1}{32} + \tfrac{9k_2}{32}) \\
k_4 &= hf(t_n + \tfrac{12h}{13}, y_n + \tfrac{1932k_1}{2197} - \tfrac{7200k_2}{2197} + \tfrac{7296k_3}{2197}) \\
k_5 &= hf(t_n + h, y_n + \tfrac{439k_1}{216} - 8k_2 + \tfrac{3680k_3}{513} - \tfrac{845k_4}{4104}) \\
k_6 &= hf(t_n + \tfrac{h}{2}, y_n - \tfrac{8k_1}{27} + 2k_2 - \tfrac{3544k_3}{2565} + \tfrac{1859k_4}{4104} - \tfrac{11k_5}{40}) \\
y^*_{n+1} &= y_n + \tfrac{25k_1}{216} + \tfrac{1408k_3}{2565} + \tfrac{2197k_4}{4104} - \tfrac{k_5}{5} \text{ with error } O(h^4) \\
y_{n+1} &= y_n + \tfrac{16k_1}{135} + \tfrac{6656k_3}{12825} + \tfrac{28561k_4}{56430} - \tfrac{9k_5}{50} + \tfrac{2k_6}{55} \text{ with error } O(h^5).
\end{aligned}$$

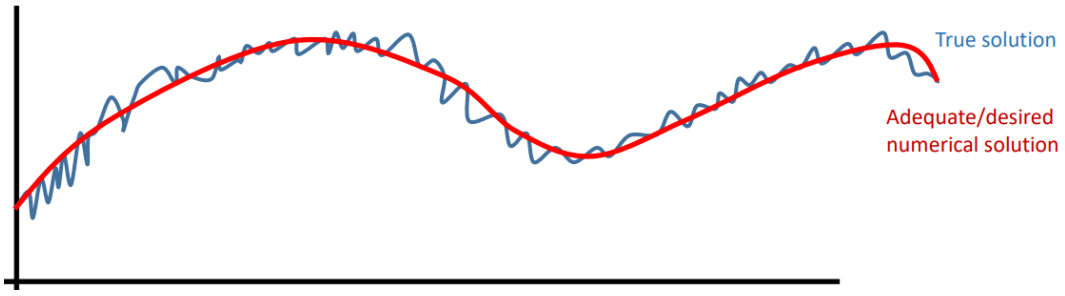
This is more efficient because we use k_1 to k_5 for both both RK4 and RK5.

Stiff Problems

Stiff problems cause some issues:

- *Interesting* aspect of the solution changes *slowly* so we want large time steps
- Stability condition of the (usually explicit) stepping method forces us to use *small* time steps

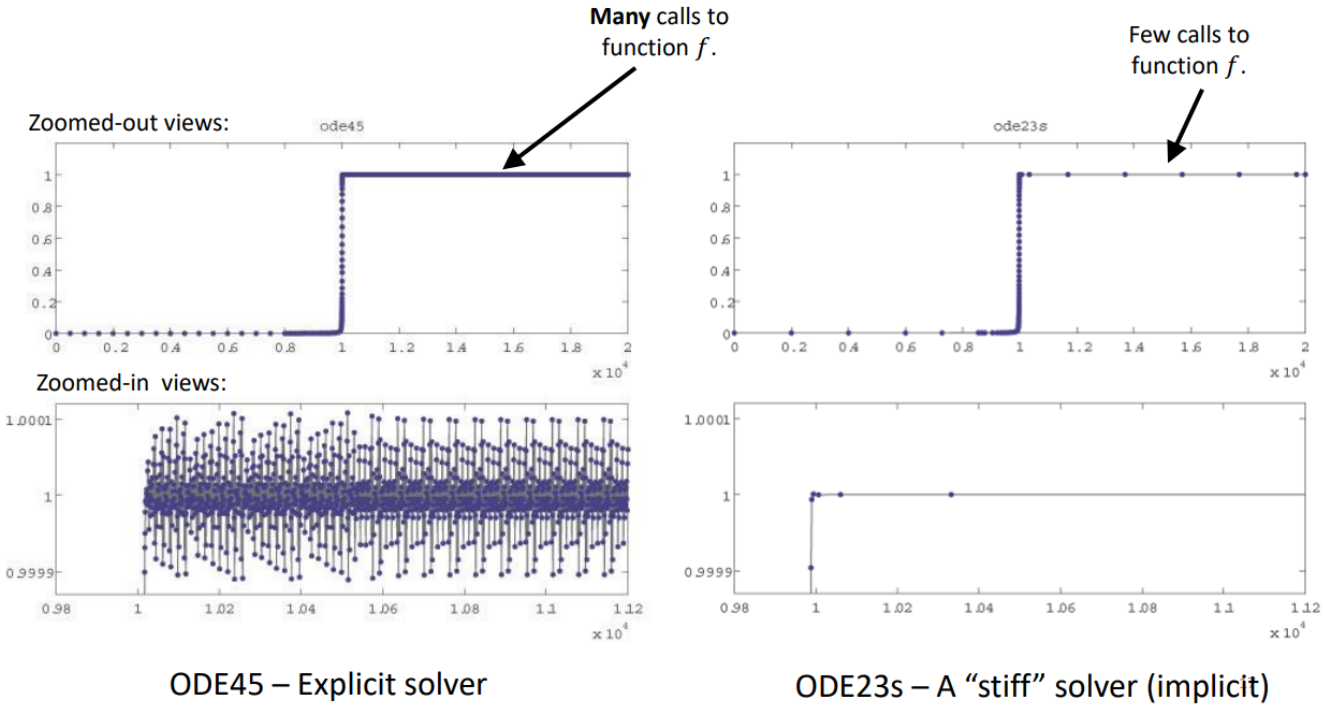
This arises when some aspects of the problem evolve at different time scales:



We only care about the trend in red but our stability condition forces us to use small steps.

1. Apply `ODE45` (with adaptive time steps) to a *stiff* problem
2. Estimated truncation error is small so `ODE45` tries to take larger step
3. Stability condition is violated which creates large errors
4. `ODE45` cuts timestep back to reduce this error

Stuck taking very small steps. Example: Mathworks Stiff Differential Equations



- `ODE45` handles this poorly as it bounces around 1 in the zoom-in view.
- `ODE23s` is the *stiff solver* and is able to deal with it well

In general implicit schemes are good since they have weaker or no stability restrictions

Matlab has a a few stiff solvers:

- **ODE23s** uses 2nd and 3rd order implicit schemes
- `ODE15s` uses 1st and 5th order implicit schemes

The Fourier Transform

The *Fourier Transform* is applied to continuous functions or discrete data to gain useful insights:

1. Transform some function/data into a form that reveals the *frequencies* of information
2. Process or analyze in this *frequency domain* form where some tasks are easier
3. Transform back

This process can be applied to a variety signal processing applications (filtering, denoising, autotune, compression, etc):

Original Bitmap: 46MB
High Quality JPEG: 4.5 MB

Low Quality JPEG: 170 KB

The goal of Fourier analysis is to convert time/space-dependent data into **Frequency domain** (infinite sum of increasingly high frequency sin or cos)

$$\text{Time domain} : f(t) \qquad \text{Space domain} : f(x)$$

$$f(t) = a_0 + a_1\cos(qt) + b_1\sin(qt) + a_2\cos(2qt) + b_2\sin(2qt) + \cdots \qquad \text{where } q = \frac{2\pi}{T}$$

now the coefficients a_i and b_i determine the function. When finding these coefficients we have two views:

- A weird non-polynomial interpolation problem (fitting to given function/data)
- A series expansion of a function as a specific kind of infinite sum (e.g. Taylor series)

Continuous Fourier Series

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

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

a_k and b_k indicate *information*/amplitude for each sinusoid of period $\frac{T}{k}$ or frequency $\frac{k}{T}$.

If we assume that the range of t is $t \in [0, 2\pi]$ then the domain length is $T = 2\pi$ and we get:

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

This can be thought of as convertion to a list of numbers a_k and b_k.

Visualization: https://bl.ocks.org/jinroh/7524988
Converting a sawtooth wave into a truncated (finite) Fourier series lets us understand why ringing artifacts occur during image compression:

Ringing artifact during image compression

Because of discontinuity at edges the Fourier series overshoots at the top and undershoots at the bottom.

Orthogonality Relations

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

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

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

Determing Coefficients of Fourier Series

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

- To find a_0 (average value fo f over $[0, 2\pi]$) we integrate over $[0, 2]$

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

$$a_0 = \frac{\int_0^{2\pi} f(t)dt}{\int_0^{2\pi} dt} = \frac{1}{2\pi}\int_0^{2\pi} f(t)dt$$

- To determine coefficients a_ℓ for integer $\ell > 0$ we multiply by $\cos(\ell t)$ and integate over $[0, 2\pi]$

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

Since $\int_0^{2\pi} \cos(kt)\cos(\ell t)dt = 0$ if $k \neq \ell$ this only leaves the case $k = \ell$

$$\int_0^{2\pi} f(t)\cos(\ell t)dt = a_\ell \int_0^{2\pi} \cos^2(\ell t)dt = a_\ell \pi$$

$$a_\ell = \frac{1}{\pi}\int_0^{2\pi} f(t)\cos(\ell t)dt$$

- Applying a similar process for b_k using sin instead of cos

$$b_\ell = \frac{1}{\pi}\int_0^{2\pi} f(t)\sin(\ell t)dt$$

Fourier Series Coefficients Summary:

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

$$a_0 = \frac{1}{2\pi}\int_0^{2\pi} f(t)dt \qquad a_k = \frac{1}{\pi}\int_0^{2\pi} f(t)\cos(kt)dt \qquad b_k = \frac{1}{\pi}\int_0^{2\pi} f(t)\sin(kt)dt$$

Complex Number Review

for $z \in \mathbb{C}$ we can write $z = a + bi$ where $i = \sqrt{-1}$ and $a, b \in \mathbb{R}$

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

Euler's Forumla: $e^{i\theta} = \cos(\theta) + i\sin(\theta)$

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

This leads to two key identities:

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

Fourier Series with Complex Exponentials

$$f(t) = a_0 + \sum_{k=1}^{\infty} a_k\cos(kt) + \sum_{k=1}^{\infty} b_k\sin(kt) \qquad \rightarrow \qquad f(t) = \sum_{k=-\infty}^{+\infty} c_k e^{ikt}$$

For $k > 0$ we can convert using

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

To find c_k directly we use the property

$$\int_0^{2\pi} e^{ikt}e^{-i\ell t}dt = \begin{cases} 0 & k \neq \ell \\ 2\pi & k = \ell \end{cases}$$

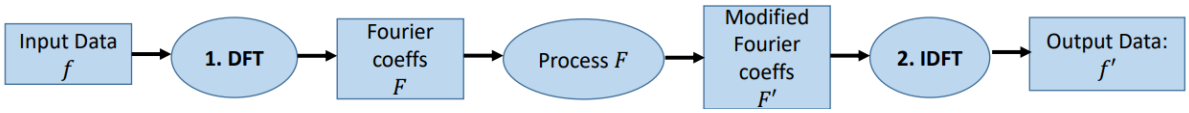
Then we have coefficient formula:

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

Discrete Fourier Transform

Consider a vector of **discrete data**: $f_0, \ldots, f_{N-1}$ for N uniformaly spaced data points (N assumed even)

Our goal is to convert these equal spaced samples of a *unknown* function into frequency domain, perform some computation, then convert back to time/space-domain and some desired modification to input data:



The Discrete Fourier Transform (DFT) and Inverse Discrete Fourier Transform (IDFT) are:

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

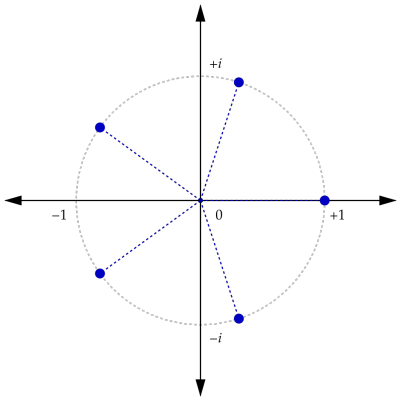
$$W = e^{\frac{2\pi i}{N}} \qquad e^{i\theta} = \cos\theta + i\sin\theta$$

Roots of Unity

$$W = e^{\frac{2\pi i}{N}} \quad \rightarrow \quad W^k = e^{\frac{2\pi ik}{N}}$$

This is the Nth roots of unity for $k \in \mathbb{Z}$

Example: for $N = 5$ roots of unity we have $W_5^k = e^{\frac{2\pi ik}{5}}$ which can be visualized as $k = 1$ going a fifth around a unit circle, $k = 2$ going 2 fifths around, and so on:



This means that for $n \in \mathbb{Z}$

$$|W^k| = 1 \qquad W_5^{5n} = 1 \qquad W_5^{k+5n} = W_5^k$$

Orthogonality Identity:

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

With *Kronecker delta* function: $\delta_{k,\ell} = \begin{cases} 1 & k = \ell \\ 0 & k \neq \ell \end{cases}$

Assume k, ℓ are integers in $[0, N-1]$

- For $k = \ell$

$$\sum_{j=0}^{N-1} W^{j(0)} = \sum_{j=0}^{N-1} 1 = N$$

- For $k \neq \ell$ we begin with the finite geometric series

$$x^N - 1 = (x-1)\underbrace{(x^{N-1} + \cdots + x + 1)}_{\sum_{j=0}^{N-1} x^j} \quad \rightarrow \quad \sum_{j=0}^{N-1} x^j = \frac{x^N - 1}{x - 1}$$

  Applying to our expression

$$\sum_{j=0}^{N-1} (W^{(k-\ell)})^j = \frac{(W^{k-\ell})^N - 1}{W^{k-\ell} - 1} = 0$$

  Since $k - \ell$ must be non-zero some integer then $(W^{k-\ell})^N = 1$ and numerator becomes 0

Remark: this is a discrete version of a simular *continuous* identity we saw earlier

$$\int_0^{2\pi} e^{ikt} e^{-i\ell t} dt = \begin{cases} 0 & k \neq \ell \\ 2\pi & k = \ell \end{cases}$$

Inverse Discrete Fourier Transform (IDFT)

Consider N equality spaced data points (t_n, f_n) over the domain $[0, T]$ for $n = 0, \ldots, N-1$ then

$$t_n = \frac{nT}{N}$$

We then truncate the Fourier series so it only contains N terms

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

Using $f(t_n) = f_n$ for any nth discrete data point $t_n = \frac{nT}{N}$

$$f_n = \sum_{-\frac{N}{2}+1}^{\frac{N}{2}} c_k e^{\frac{(2\pi i)kt_n}{T}} = \sum_{-\frac{N}{2}+1}^{\frac{N}{2}} c_k e^{\frac{(2\pi i)kn}{N}} = \sum_{k=0}^{\frac{N}{2}} c_k e^{\frac{i2\pi nk}{N}} + \sum_{k=-\frac{N}{2}+1}^{-1} c_k e^{\frac{i2\pi nk}{N}}$$

- Define c_j outside range $[-\frac{N}{2}+1, \frac{N}{2}]$ to be periodic, that is $c_{j \pm N} = c_j$
- Then apply change of variables $k = j - N$ on the 2nd term
- Notice that $e^{-i2\pi n} = \cos(2\pi n) - i\sin(2\pi n) = 1$ for any n

$$\sum_{k=-\frac{N}{2}+1}^{-1} c_k e^{\frac{i2\pi nk}{N}} \quad \rightarrow \quad \sum_{j=\frac{N}{2}+1}^{N-1} c_{j-N} e^{\frac{i2\pi nj}{N}} e^{-i2\pi n} \quad \rightarrow \quad \sum_{j=\frac{N}{2}+1}^{N-1} c_j e^{\frac{i2\pi nj}{N}}$$

Then plugging back into f_n we can recombine the sumations:

$$f_n = \sum_{k=0}^{\frac{N}{2}} c_k e^{\frac{i2\pi nk}{N}} + \sum_{k=\frac{N}{2}+1}^{N-1} c_k e^{\frac{i2\pi nk}{N}} \quad \rightarrow \quad f_n = \sum_{k=0}^{N-1} c_k e^{\frac{i2\pi nk}{N}}$$

Then we rename c_k into the N discrete Fourier coefficients F_k corresponding to our N input points f_n

$$f_n = \sum_{k=0}^{N-1} F_k W^{nk} \qquad \text{where } W = e^{\frac{i2\pi}{N}}$$

Discrete Fourier Transform (DFT)

Like the continuous case we use orthogonality to isolate the Fourier coefficients F_k, so begin with the IDFT

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

To get the k coefficient multiply both sides by W^{-nk} and sum from 0 to $N-1$

$$\begin{aligned}
\sum_{n=0}^{N-1} f_n W^{-nk} &= \sum_{n=0}^{N-1}\sum_{j=0}^{N-1} F_j W^{nj} W^{-nk} \\
&= \sum_{j=0}^{N-1} F_j \sum_{n=0}^{N-1} W^{n(j-k)} \\
&= \sum_{j=0}^{N-1} F_j N \delta_{j,k} && \text{(Roots of unity orthogonality)} \\
&= F_k \cdot N
\end{aligned}$$

Thus by reorganizing we get our Discrete Fourier Transform

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

Given data f_n (for $n = 0, \ldots, N-1$) we can find corresponding F_k (for $k = 0, \ldots, N-1$)

Properties of the DFT:

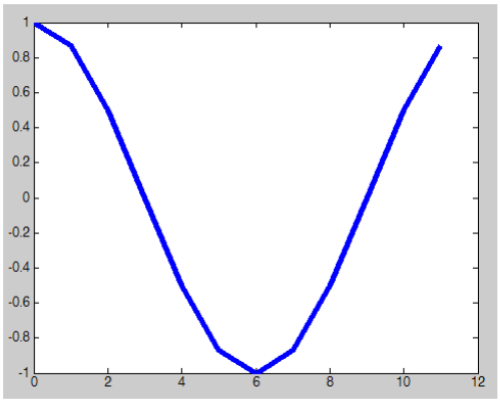
- Sequence $\{F_k\}$ is doubly infinite and periodic (if we allow $k < 0$ or $k > N-1$ then F_k repeat)
- *Conjugate Symmetry*: if data f_n is real then $F_k = \overline{F_{N-k}}$ (symmetric about $k = \frac{N}{2}$)

$$\overline{F_{N-k}} = \overline{\frac{1}{N}\sum_{n=0}^{N-1} f_n W^{-n(N-k)}} = \frac{1}{N}\sum_{n=0}^{N-1} f_n \overline{W^{-nN+nk}} = \frac{1}{N}\sum_{n=0}^{N-1} f_n \overline{W^{nk}} = \frac{1}{N}\sum_{n=0}^{N-1} f_n W^{-nk} = F_k$$

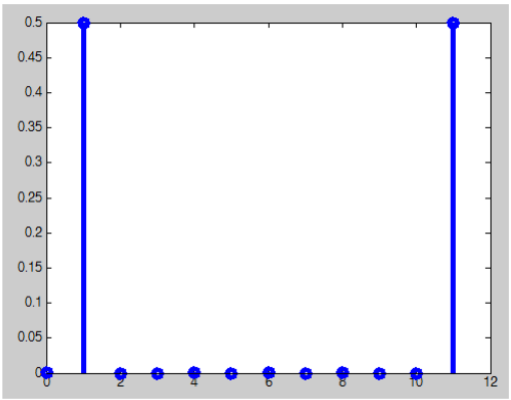
- Coefficient $F_0 = \frac{1}{N}\sum_{n=0}^{N-1} f_n$ is the average of data values
- Frequencies of corresponding waves *increase* on $k = 0$ to $\frac{N}{2}$ then *decrease* on $k = \frac{N}{2}+1$ to $N-1$

Discrete Fourier Transform Examples

$N = 12$ samples, $F_0 = 0$ average, $F_1 = F_{11} = 0.5$ (since $F_k = \overline{F_{12-k}}$) adds a wave that repeats once:



f_n



F_k

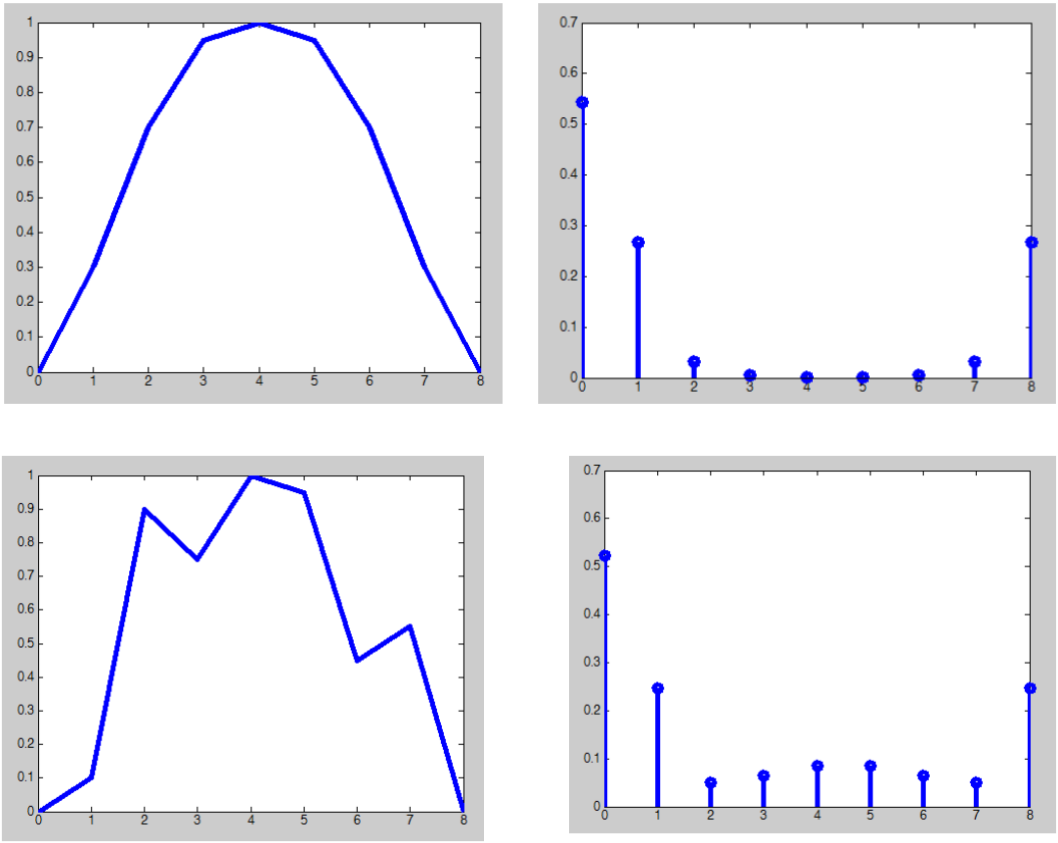
To understand why this makes sense recall Euler's identities and f_n before rearranging:

$$\cos(\theta) = \frac{e^{i\theta} + e^{-i\theta}}{2} \qquad f_n = \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} c_k e^{\frac{(2\pi i)kn}{N}}$$

When $c_1 = c_{-1} = 0.5$ (these are corresponding c_k to F_k) with all others being 0

$$f_n = 0.5e^{\frac{2\pi in}{N}} + 0.5e^{\frac{-2\pi in}{N}} = \cos\left(\frac{2\pi n}{N}\right)$$

Example: discrete data from a smooth bump vs rough bump



Notice that enough the rough bump has more irregularity the frequencies F_1 and F_8 still dominate

Remark: When Fourier coefficients are complex we plot their modulus (magnitude)

Remark: The F_0 coefficient is usually shown separately because its size can make the graph hard to read

Remark: Different definitions of DFT/IDFT can be found in literature/code

- We use the following:

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

- SciPy (and some other sources) use:

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

So be careful when coding in Jupyter that your Fourier coefficients F_k might look much larger.

Matrix Form of DFT and IDFT

$$\text{DFT} : F = Mf \qquad \text{IDFT} : f = M^{-1}F = N\overline{M^T}F$$

Where the kth column of M is

$$\frac{1}{N}\begin{bmatrix} W^0 \\ W^{-k} \\ W^{-2k} \\ W^{-3k} \\ \vdots \\ W^{-(N-1)k} \end{bmatrix}$$

Example: Using $N = 4$ then we have $W^0 = 1$, $W^1 = i$, $W^2 = -1$, $W^3 = -i$

$$F = \begin{bmatrix} -2 \\ 2+i \\ -2 \\ 2-i \end{bmatrix} \qquad M = \frac{1}{4}\begin{bmatrix} W^0 & W^0 & W^0 & W^0 \\ W^0 & W^{-1} & W^{-2} & W^{-3} \\ W^0 & W^{-2} & W^{-4} & W^{-6} \\ W^0 & W^{-3} & W^{-6} & W^{-9} \end{bmatrix} = \frac{1}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$$

Then after finding the conjugate of M then we can apply the IDFT to F

$$f = N\overline{M^T}F = 4 \cdot \frac{1}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}\begin{bmatrix} -2 \\ 2+i \\ -2 \\ 2-i \end{bmatrix} = \begin{bmatrix} 0 \\ -2 \\ -8 \\ 2 \end{bmatrix}$$

Notice if we directly apply this then DFT and IDFT will both take $O(n^2)$ complex floating-point operations.

Fast Fourier Transform

We can make DFT and IDFT faster by deploying a *divide and conquer* strategy where we split the list in half until we reach the base case of individual numbers.

Remark: This requires $N = 2^m$ for some m, so we sometimes need to pad our initial data with zeros

Dividing Up the Input

We begin by spliting the DFT of sequence f_n and assume that N is even:

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

Reindex 2nd term using $n = m + \frac{N}{2}$ then rename m back to n

$$\frac{1}{N}\sum_{n=\frac{N}{2}}^{N-1} f_n W^{-nk} \quad \to \quad \frac{1}{N}\sum_{m=0}^{\frac{N}{2}-1} f_{m+\frac{N}{2}} W^{-k(m+\frac{N}{2})} \quad \to \quad \frac{1}{N}\sum_{n=0}^{\frac{N}{2}-1} f_{n+\frac{N}{2}} W^{-nk} W^{-k\frac{N}{2}}$$

Notice $W^{\frac{-kN}{2}} = (e^{\frac{2\pi i}{N}})^{\frac{-kN}{2}} = e^{-\pi ik} = (-1)^k$. Subbing everything back into the sumation and combining:

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

Thus using $j = 0$ to $\frac{N}{2} - 1$ we can create two lists: even entries ($k = 2j$) and odd entries ($k = 2j + 1$)

$$\text{Even: } F_{2j} = \frac{1}{N} \sum_{n=0}^{\frac{N}{2}-1} \left(f_n + f_{n+\frac{N}{2}} \right) W^{-2nj}$$

$$\text{Odd: } F_{2j+1} = \frac{1}{N} \sum_{n=0}^{\frac{N}{2}-1} \left[\left(f_n - f_{n+\frac{N}{2}} \right) W^{-n} \right] W^{-2nj}$$

We call the extra W^{-n} in the odd case the *Twiddle Factor*.

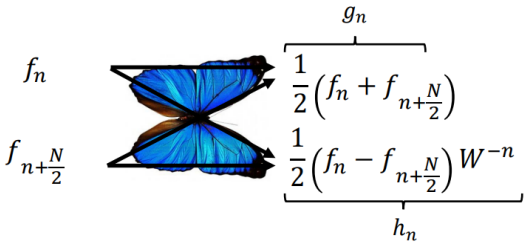
As a direct result f_n can be split into two half-length vectors

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

Notice that $n = 0$ to $\frac{N}{2} - 1$ and we update $N := \frac{N}{2}$ which combined with $W_N^{-2nj} = W_{\frac{N}{2}}^{-nj}$ ensures that

$$F_{2j} = \text{DFT}(g) \qquad F_{2j+1} = \text{DFT}(h)$$

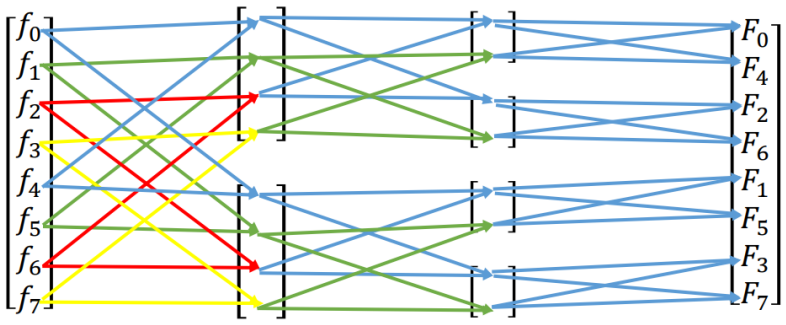
This spliting is referred to as a *Butterfly Operation*:



It mildly resembles a butterfly

Fast Fourier Transform (FFT)

To perform a Fast Fourier Transform the butteryfly operation is applied recursively until each list contains only one value (Notice that taking the DFT of a list with one element does not change the list).



Since each step performs even-odd index spltting the order of Fourier coefficients needs to be unscrambled. We observe that the binary indices of the output indices have bits in reverse order from the input.

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \end{bmatrix} = \begin{bmatrix} f_{000} \\ f_{001} \\ f_{010} \\ f_{011} \\ f_{100} \\ f_{101} \\ f_{110} \\ f_{111} \end{bmatrix} \xrightarrow{\text{Perform All Butterfly Stages}} \begin{bmatrix} F_0 \\ F_4 \\ F_2 \\ F_6 \\ F_1 \\ F_5 \\ F_3 \\ F_7 \end{bmatrix} = \begin{bmatrix} F_{000} \\ F_{100} \\ F_{010} \\ F_{110} \\ F_{001} \\ F_{101} \\ F_{011} \\ F_{111} \end{bmatrix}$$

Thus to *unscramble* the FFT output we convert the indices to binary and bit-reverse them to get index.

Inverse Fast Fourier Transform (IFFT)

The DFT and IDFT has very similar forms so we can use the same core butterfly algorithm to perform either direction if we make some minor changes.

The generic expression for IDFT and DFT is:

$$f'_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n U^{nk}$$

- DFT: $U = W^{-1}$
- IDFT: $U = W$ and we post-multiply by N

As a result for IFFT we change the twiddle factor to W^n and multiply the final result by N

Naive vs FFT Performance

Naive algorithm: $O(N^2)$ complex FP operations

FFT algorithm: $O(N \log_2 N)$ operations

- $O(N)$ operations to split the arrays in two at each stage
- There are $m = \log_2 N$ stages, since data is (or was padded to) length $N = 2^m$

FFT Example

Stage 1: split the list of $N = 8$ entries

$$f = \begin{bmatrix} 4 \\ 3 \\ 2 \\ 1 \\ 4 \\ 3 \\ 2 \\ 1 \end{bmatrix} \rightarrow \begin{matrix} \frac{1}{2} \begin{bmatrix} 4+4 \\ 3+3 \\ 2+2 \\ 1+1 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 2 \\ 1 \end{bmatrix} \\ \frac{1}{2} \begin{bmatrix} (4-4)W^{-0} \\ (3-3)W^{-1} \\ (2-2)W^{-2} \\ (1-1)W^{-3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \end{matrix}$$

Stage 2: split the lists of $N = 4$ entries (notice that $W^{-1} = -i$)

$$\begin{matrix}\begin{bmatrix}4\\3\\2\\1\end{bmatrix}\\\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\end{matrix} \rightarrow \begin{matrix}\frac{1}{2}\begin{bmatrix}4+2\\3+1\end{bmatrix} = \begin{bmatrix}3\\2\end{bmatrix}\\\frac{1}{2}\begin{bmatrix}(4-2)W^{-0}\\(3-1)W^{-1}\end{bmatrix} = \begin{bmatrix}1\\-i\end{bmatrix}\\\begin{bmatrix}0\\0\end{bmatrix}\\\begin{bmatrix}0\\0\end{bmatrix}\end{matrix}$$

Stage 3: split the lists of $N = 2$ entries

$$\begin{matrix}\begin{bmatrix}3\\2\end{bmatrix}\\\begin{bmatrix}1\\-i\end{bmatrix}\\\begin{bmatrix}0\\0\end{bmatrix}\\\begin{bmatrix}0\\0\end{bmatrix}\end{matrix} \rightarrow \begin{matrix}\frac{1}{2}\left[3+2\right] = \left[2.5\right]\\\frac{1}{2}\left[(3-2)W^{-0}\right] = \left[0.5\right]\\\frac{1}{2}\left[(1-i)\right] = \left[0.5-0.5i\right]\\\frac{1}{2}\left[(1+i)W^{-0}\right] = \left[0.5+0.5i\right]\\\begin{bmatrix}0\end{bmatrix}\\\begin{bmatrix}0\end{bmatrix}\\\begin{bmatrix}0\end{bmatrix}\\\begin{bmatrix}0\end{bmatrix}\end{matrix} \rightarrow \begin{bmatrix}2.5\\0.5\\0.5-0.5i\\0.5+0.5i\\0\\0\\0\\0\end{bmatrix}$$

Bit-reverse Stage:

$$\begin{matrix}0 & 000\\1 & 001\\2 & 010\\3 & 011\\4 & 100\\5 & 101\\6 & 110\\7 & 111\end{matrix}\begin{bmatrix}2.5\\0.5\\0.5-0.5i\\0.5+0.5i\\0\\0\\0\\0\end{bmatrix} \rightarrow \begin{matrix}0 & 000\\4 & 100\\2 & 010\\6 & 110\\1 & 001\\5 & 101\\3 & 011\\7 & 111\end{matrix}\begin{bmatrix}2.5\\0\\0.5-0.5i\\0\\0.5\\0\\0.5+0.5i\\0\end{bmatrix}$$

Notice that since the input data was all real this satisfies conjugate symmetry

$$F_k = \overline{F_{N-k}}$$

This does not apply to F_0 since F_N is outside the array of length N (max index is $N-1$)

Applications of the Fourier Transform

Image Compression

The 2D DFT is essentially one vertical 1D DFT and one horizontal 1D DFT combined:

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

Result is a 2D array of Fourier coefficients $F_{k,l}$. It uses two roots of unity (one per dimension):

$$W_N = e^{\frac{i2\pi}{N}} \qquad W_m = e^{\frac{i2\pi}{M}}$$

The 2D FFT can be computed *efficiently* by using two nested 1D FFTs:

- Transform each row (separately) using 1D FFTs
- Transform each column of the result using 1D FFTs

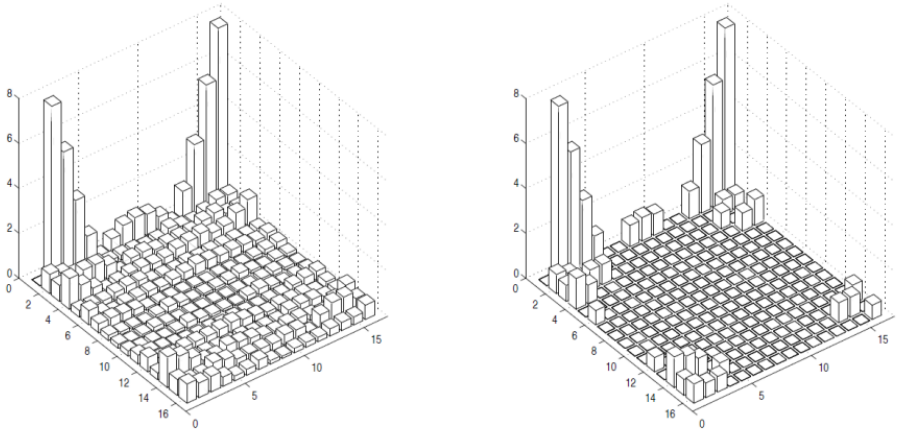
Numpy provides `fft2` and `ifft2` (watch out for $1/NM$ scaling convention differences)

Complexity of 2D FFT is $O(MN(\log_2 M + \log_2 N))$ compared to the naive algorithm's $O(MN(M+N))$

Compression Strategy:

- Take the 2D FFT of image intensities
- Create an approximate version by throwing away *small* Fourier coefficients $|F_{k,l}| < \text{tol}$
- Store the remaining and perform IFFT when you want to view the image again

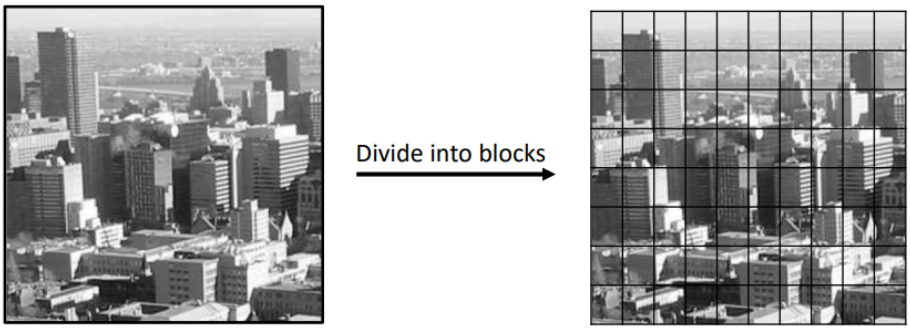
Note that the output image may contain imaginary values which we can safely ignore.



(a) Original (b) Compressed by 85%

Remark: "compressed by 85%" means we are throwing away 85% of the fourier coefficients

Block-based Image Compression: often we subdidivide an image into smaller blocks (16×16 or 8×8) then compress each block independently:



Pixels within each block often have similar/related data so there are less active coefficients.

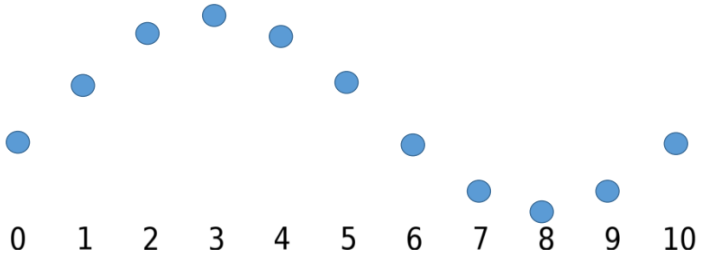
Understanding Aliasing

When capturing audio/video/images (or any kind of signal) we are often taking discrete point samples of some continuous signal.

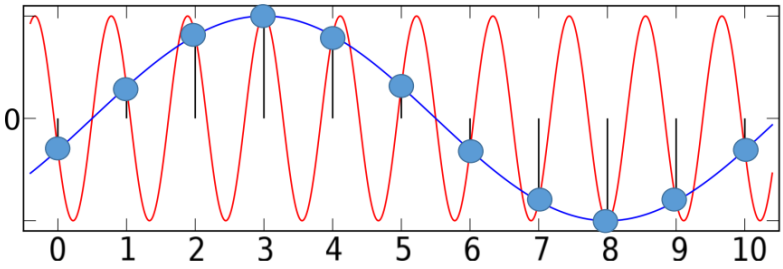
- Sampling rate: $F_s = N/T$ = number of discrete samples per second
- Fourier frequency: $\frac{k}{T} = \frac{kF_s}{N}$ = number of cycles per second ($1/s$ = Hz)

The index k of a Fourier coefficient determines the frequeny (cycles per unit time) it corresponds to.

Is this discrete data sampled from a low frequency or a high frequency continuous function?



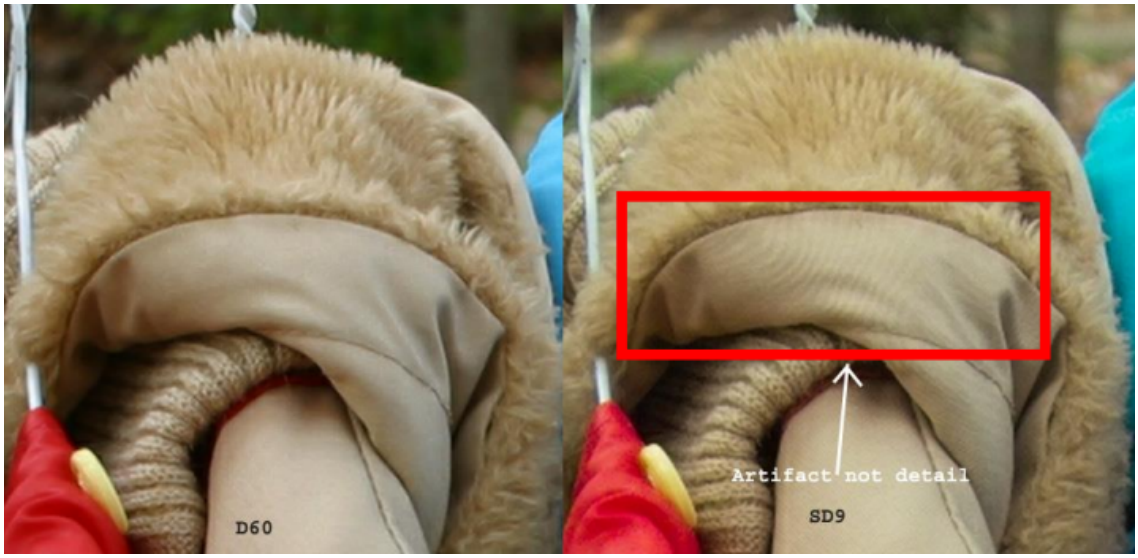
The true answer is unknowable due to *aliasing*:



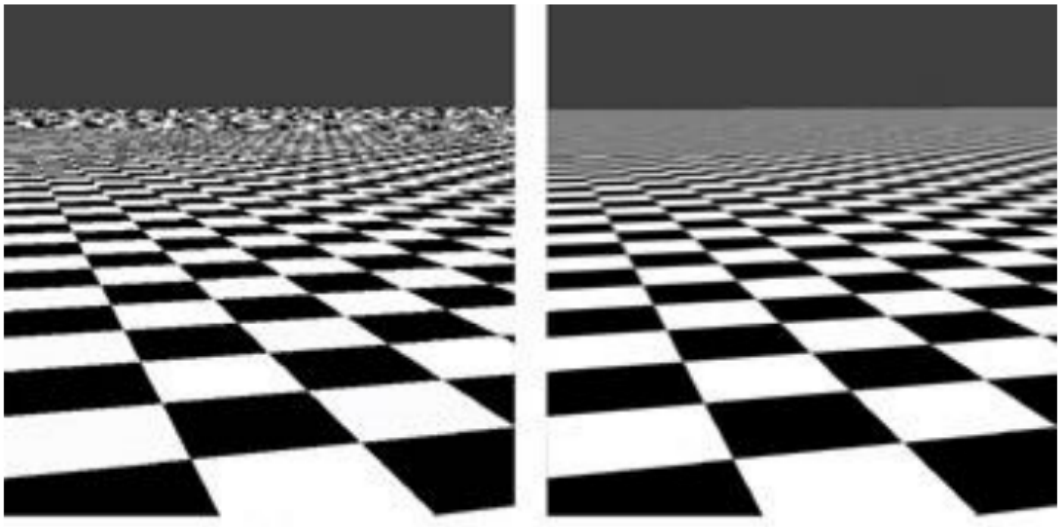
The high frequency signal (red) can *aliases as* (appear to be) a lower frequency signal (blue).

Some exmaples of aliasing in action:

- Photography: Moire patterns occur from our spaced point samples of some scene:



- Computer Graphics: a checkboard pattern gives aliasing patterns when sampled into a few pixels:



  In the right image we prevent aliasing by filtering/blurring out higher frequencies (anti-aliasing)

- Stroboscopic Effect: temporal aliasing of video frames
  - Wagon wheel effect
  - Helicopter rotor and camera shutter
  - Strobe light makes water drops fall in slow motion
  - Zoetrope animation toy
  - 3D Zoetrope animation
  - Water and vibrations

Aliasing: The Math

Consider the Fourier series of a continuous signal for a period T

$$f(t) = \sum_{k=-\infty}^{\infty} c_k e^{\frac{i2\pi kt}{T}}$$

When we sample this signal at points $t_n = n\Delta t = n\frac{T}{N}$

$$f_n = f(t_n) = \sum_{k=\infty}^{\infty} c_k e^{\frac{i2\pi kn}{N}} = \sum_{k=-\infty}^{\infty} c_k W^{nk}$$

$k \in (-\infty, \infty)$ allows arbitrarily high frequencies, comparing this to our IDFT approximation:

$$f_n = \sum_{n=-\frac{N}{2}+1}^{\frac{N}{2}} F_k W^{nk} \quad \rightarrow \quad k \in \left[-\tfrac{N}{2}+1,\ \tfrac{N}{2}\right]$$

This only uses N coefficients F_k so frequencies higher than this are *thrown away.*

To relate F_k and c_k we plug the equation containing c_k into our DFT equation

$$F_\ell = \frac{1}{N} \sum_{n=-\frac{N}{2}+1}^{\frac{N}{2}} f_n W^{-n\ell} = \frac{1}{N} \sum_{n=-\frac{N}{2}+1}^{\frac{N}{2}} \left(\sum_{k=-\infty}^{\infty} c_k W^{nk} \right) W^{-n\ell} = \sum_{k=-\infty}^{\infty} \frac{c_k}{N} \sum_{n=-\frac{N}{2}+1}^{\frac{N}{2}} W^{n(k-\ell)}$$

Notice $W^{n(k-\ell)}$ hints at using orthogonality but we will need to adapt it a bit to handle $k \in (-\infty, \infty)$

$$\sum_{p=-\frac{N}{2}+1}^{\frac{N}{2}} W^{p(k-\ell)} = N(\delta_{k,\ell} + \delta_{k,\ell+N} + \delta_{k,\ell-N} + \delta_{k,\ell+2N} + \delta_{k,\ell-2N} + \cdots)$$

When $\ell = k + jN$ (i.e. $\ell \equiv k \mod N$) then $W^{p(\ell-k)} = W^{pjN} = 1$ since W is an Nth root of unity.

Then using this orthogonality identity we have DFT coefficients

$$\begin{aligned} F_\ell &= \sum_{k=-\infty}^{\infty} \frac{c_k}{N} N(\delta_{k,\ell} + \delta_{k,\ell+N} + \delta_{k,\ell-N} + \cdots) \\ &= c_\ell + c_{\ell+N} + c_{\ell-N} + c_{\ell+2N} + c_{\ell-2N} + \cdots \end{aligned}$$

We find that F_ℓ is c_ℓ along with a bunch of higher frequncy signals $(c_{\ell+N} + c_{\ell-N} + c_{\ell+2N} + c_{\ell-2N} + \cdots)$ summed into a single term.

The high frequencies from $c_k \notin [-\frac{N}{2}+1, \frac{N}{2}]$ contribute to (*alias as*) low frequency F_k for $k \in [-\frac{N}{2}+1, \frac{N}{2}]$.

Nyquist frequency:

If the original continuous signal has frequencies $c_k \neq 0$ for

$$|k| \geq \frac{N}{2}$$

those frequences *alias* (i.e. get added) to a lower frequency.

- Once a signal is discretely sampled, there is no way to distinguish aliased (high) frequencies from real (low) frequencies
- We have to sample at **twice** the rate of the highest occuring frequency of the original signal to avoid aliasing

Treating Aliasing:

We have two partial solutions for aliasing

- Increase sampling resolution to *capture* higher frequencies
- Filter before sampling to *remove* frequencies that are too high so they don't cause alising

Digital cameras have *optical lowpass filter* that physically blur the light to filter out the frequencies the camera cannot handle, *before* it hits the image sensor.

Correlation

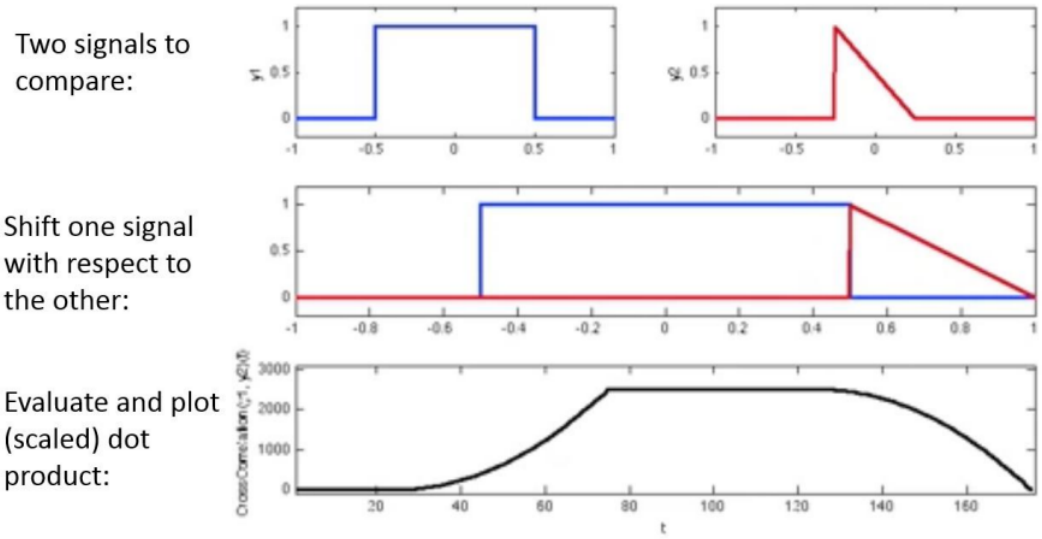
Finding similarities/relationships between different signals/data, with applications to pattern matching, synchronizing communications, signal detection, statistics, etc.

Consider two real periodic data sets y_i and z_i for $i \in [0, N-1]$, the correlation function ϕ_n defined as

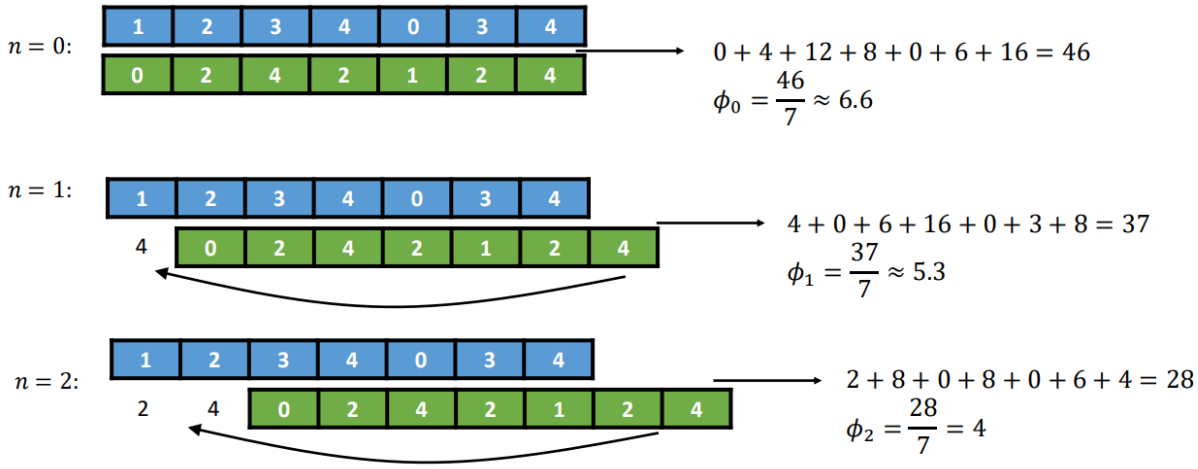
$$\phi_n = \frac{1}{N} \sum_{i=0}^{N-1} y_{i+n} z_i$$

where n is the offset. We want to find n that maximizes ϕ_n (i.e. maximum correlation between the data).

Essentially: shift one vector by n, take the dot product, and divide by N (note that data is periodic)



In a more detailed example:



- Naive algorithm uses two nested for loops of length N so $O(N^2)$
- However using the FFT we can do the same thing in $O(N \log_2 N)$

In time-domain the correlation is

$$\phi_n = \frac{1}{N} \sum_{i=0}^{N-1} y_{i+n} z_i$$

in frequency-domain there is no for loop required

$$\Phi_n = Y_k \overline{Z_k}$$

Consider $Z = FFT(z)$ and $Y = FFT(y)$ we have 4 equations:

$$\phi_n = \frac{1}{N} \sum_{\ell=0}^{N-1} y_{\ell+n} z_\ell \tag{1}$$

$$y_{\ell+n} = \sum_{m=0}^{N-1} Y_m W^{m(\ell+n)} \tag{2}$$

$$z_\ell = \sum_{r=0}^{N-1} Z_r W^{r\ell} \tag{3}$$

$$\Phi_k = \frac{1}{N} \sum_{n=1}^{N-1} \phi_n W^{-nk} \tag{4}$$

Then writing Φ_k in terms of Y and Z by subbing (1) in (4) then subbing (2) and (3) into that

$$\begin{aligned}
\Phi_k &= \frac{1}{N} \sum_{n=1}^{N-1} \left(\frac{1}{N} \sum_{\ell=0}^{N-1} y_{\ell+n} z_\ell \right) W^{-nk} \\
&= \frac{1}{N^2} \sum_n \sum_\ell \left[\left(\sum_{m=0}^{N-1} Y_m W^{m(\ell+n)} \right) \left(\sum_{r=0}^{N-1} Z_r W^{r\ell} \right) W^{-nk} \right] \\
&= \frac{1}{N^2} \sum_n \sum_\ell \sum_r \sum_m Y_m Z_r W^{m\ell} W^{mn} W^{r\ell} W^{-nk} \\
&= \frac{1}{N^2} \sum_n \sum_r \sum_m \left[Y_m Z_r W^{-n(k-m)} \sum_\ell W^{\ell(r+m)} \right] \\
&= \frac{1}{N^2} \sum_n \sum_r \sum_m Y_m Z_r W^{-n(k-m)} N \delta_{r,N-m} \\
&= \frac{1}{N} \sum_n \sum_m Y_m Z_{N-m} W^{-n(k-m)} \\
&= \frac{1}{N} \sum_m Y_m Z_{N-m} \sum_n W^{-n(k-m)} \\
&= \frac{1}{N} \sum_m Y_m Z_{N-m} N \delta_{k,m} \\
&= Y_k Z_{N-k}
\end{aligned}$$

Then assuming real input $Z_k = \overline{Z_{N-k}}$ and $Z_{N-k} = \overline{Z_k}$

$$\Phi_k = Y_k \overline{Z_k}$$

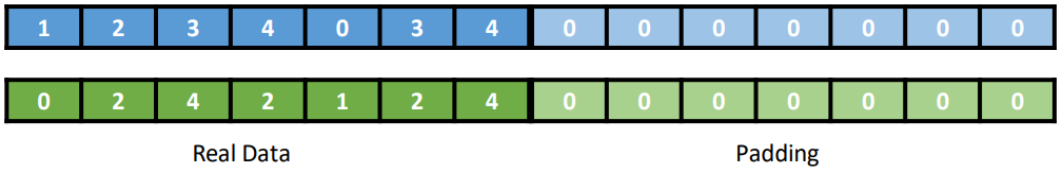
Overall algorithm for correlation via FFT:

- Apply FFT ($O(N \log_2 N)$): $Y = \text{FFT}(y)$ and $Z = \text{FFT}(z)$
- Compute the products ($O(N)$): $\Phi_k = Y_k \overline{Z}_k$ for $k = 0, \ldots, N-1$
- Apply IFFT ($O(N \log_2 N)$): $\phi = \text{IFFT}(\Phi)$

Complexity of $O(N \log_2 N)$ compared to naive two for loop $O(N^2)$

In practice input signals/data are often not actually periodic:

- In this case, *wrap-around* effect is undesired since actual signals do not wrap
- To avoid *wrap-around pollution* we pad both arrays with zeros up to twice their original length



Numerical Linear Algebra

Page Rank

Page Rank is said to be the algorithm that launched Google.

- Previous ranking strategies ranked pages by times some keyword appeared, this often gave poor results and was easy to cheat.
- Yahoo used an alternative strategy of creating a human-curated directory structure but this method was slow and hard to update.

Google's key idea: if many web pages link to your website then it must be important.

Academic paper success is measured using a similar method:

- When writing a research paper will usually cite earlier work
- If a paper gets cited many times then that paper was influential (which probably implies it is good)

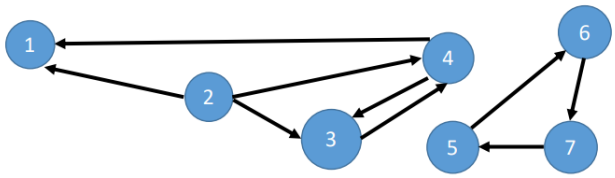
We represent the web as a directed graph

- Nodes: represent websites
- Arcs: represent links from one page to another
- Degree: number of arcs leaving that node

To store this we use an adjacency matrix

$$G_{ij} = \begin{cases} 1 & \text{if link } j \to i \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

Note that matrix G is not necessarily symmetric (if it were symmetric then the graph would be undirected)



To \ From	1	2	3	4	5	6	7
1	0	1	0	1	0	0	0
2	0	0	0	0	0	0	0
3	0	1	0	1	0	0	0
4	0	1	1	0	0	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	1	0	0
7	0	0	0	0	0	1	0

We can interpret a link from page j to page i as a vote and set outgoing links to have equal influence

$$\frac{1}{\deg(j)}$$

Example: if j has a degree of 3 and points to i then i gets $1/3$ votes (local importance)

For the global importance of page i we also need to consider how important page j is to determine how much its $1/3$ vote matters.

Random Surfer Model

Imagine a user that starts at some page and follows links at random from page to page for K steps, then restarts on some other random page repeating this R times.

They will probably end up on *important* pages more often, so we can estimate the overall importance as:

$$\text{Rank (page } i) = \frac{\text{Visits to page } i}{\text{Total visits to all pages}}$$

```
// Random Surfer Algorithm
Rank(m) = 0, m = 1,...,R
For m = 1,...,R
    j = m
    For k = 1,...,K
        Rank(j) = Rank(j) + 1
        Randomly select outlink l of page j
        j = l
    EndFor
EndFor
Rank(m) = Rank(m)/(K*R), m = 1,...,R
```

Issues with this algorihm that we will attempt to solve

- Number of real web pages is massive (over a billion) and R must be much larger
- Number of steps K must be large to get representative sample
- Dead end links leads to getting stuck on some page
- Cycles leads to getting stuck on closed subset of pages

Markov Chain Matrix

We can think of the votes as a probability that a random surfer would take some link

$$P_{ij} = \begin{cases} \frac{1}{\deg(j)} & \text{if link } j \to i \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

To build matrix P we use adjacency matrix G and divide entries of each column by the column sum.

Dead Ends: when we reach a dead end simply *teleport* to a new page at random:

$$P' = P + \frac{1}{R} e d^T$$

where $e = [1, \ldots, 1]^T$ and d^T indicates dead end columns with 1.

Cycles: to remove cycles we add a small probability $1 - \alpha$ to teleport from any page to any other page

$$M = \alpha P' + (1 - \alpha) \frac{1}{R} e e^T$$

ee^T is a matrix of all ones.

We call this combined matrix M our *Google matrix* (Google purportedly used $\alpha \approx 0.85$ originally)

Each M expresses one step of random surfing:

$$M = \alpha(P + \frac{1}{R} e d^T) + (1 - \alpha) \frac{1}{R} e e^T$$

Example of a Google matrix:

$$M = \alpha \left(\begin{bmatrix} 0 & 0 & 1/4 & 0 & 0 & 1/4 \\ 0 & 0 & 1/4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/4 \\ 0 & 1/2 & 1/4 & 0 & 1/2 & 1/4 \\ 0 & 1/2 & 0 & 0 & 0 & 1/4 \\ 0 & 0 & 1/4 & 0 & 1/2 & 0 \end{bmatrix} + \begin{bmatrix} 1/6 & 0 & 0 & 1/6 & 0 & 0 \\ 1/6 & 0 & 0 & 1/6 & 0 & 0 \\ 1/6 & 0 & 0 & 1/6 & 0 & 0 \\ 1/6 & 0 & 0 & 1/6 & 0 & 0 \\ 1/6 & 0 & 0 & 1/6 & 0 & 0 \\ 1/6 & 0 & 0 & 1/6 & 0 & 0 \end{bmatrix} \right) + \frac{(1 - \alpha)}{6} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Google matrix M is an example of a *Markov matrix*.

- In general, a Markov matrix Q has two properties:

$$0 \leq Q_{ij} \leq 1 \quad \text{and} \quad \sum_i Q_{ij} = 1$$

- A *Probability Vector* is a vector q such that

$$0 \leq q_i \leq 1 \quad \text{and} \quad \sum_i q_i = 1$$

If the surfer starts at a random page with equal probabilities then we have a probability vector $p_i = \frac{1}{R}$.

To evolve the probability vector we need:

- Probability vector describing the *initial state* p^0
- Markov matrix M describing the *transition probabilities* among pages

Then the probabilities for our surfer to be at each page after one transition is

$$p^1 = Mp^0$$

Then for any step n, the next step probabilities are $p^{n+1} = Mp^n$

To prove that p^{n+1} is a probability vector we have:

- $p_i^{n+1} \geq 0$ since it is just sums and products of probabilities ≥ 0
- $\sum_i p_i^{n+1} = 1$ since

$$\sum_i p_i^{n+1} = \sum_i \sum_j M_{ij} p_j^n = \sum_j \left(p_j^n \sum_i M_{ij} \right) = \sum_j p_j^n = 1$$

- $p_i^{n+1} \leq 1$ since there are no negative entries and it needs to sum to 1

The final question we want to check is if Page Rank will *converge* to a single answer:

$$p^\infty = \lim_{k \to \infty} (M)^k p^0$$

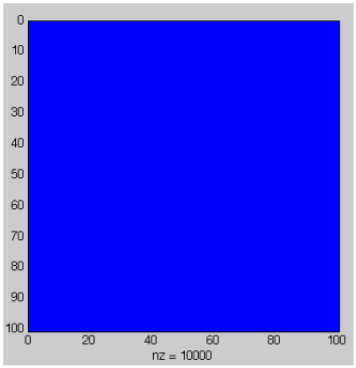
Then higher probabilities in p^∞ implies a greater importance for a page (i.e. a higher rank)

Efficient Page Rank

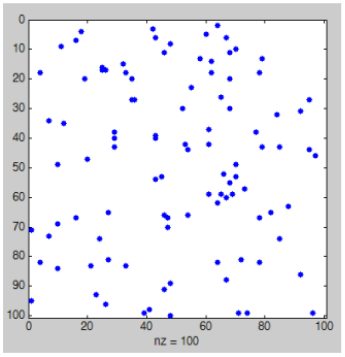
A naive implementation of Page Rank will repeatedly multiply massive matrices each search (dimensions larger than 1 billion $\times$ 1 billion)

To make it faster we exploit precomputation and sparsity:

- *Precomputation*: ranking vector p^∞ is *independent* of the query so only needs to be computed once (then recomputed once a day/week)
  - Each search will finds the subset of p^∞ that matches the keywords
- *Sparse Matrix*: the default transition matrix P is a sparse matrix (Google matrix M is fully dense)
  - Dense matrix: all entries are non-zero (stored in $N \times N$, manipulate "normally")
  - Sparse matrix: most entries are zero (able to use a sparse data structure and usually prefer algorithms that avoid "destroying" sparsity (i.e. avoid filling zero entries))



Non-zero entries (blue) in a dense matrix.



Non-zeros in a sparse matrix.

7

When performing sparse matrix-vector multiplication only non-zero matrix elements are ever accessed making it much faster.

The goal is to use linear algebra manipulations to perform the main iteration

$$p^{n+1} = Mp^n$$

without ever creating/storing M. Thus we start with the definition of the Google matrix:

$$M = \alpha\left(P + \frac{1}{R}ed^T\right) + \frac{1-\alpha}{R}ee^T$$

- M is a fully dense matrix
- P is sparse since most pages to not link to a billion other pages
- ed^T is dense in dead end columns (pages that do not link anywhere)
- ee^T is fully dense (of all ones)

$$\begin{aligned} p_{n+1} &= Mp_n \\ &= \underbrace{\alpha P p_n}_{(1)} + \underbrace{\frac{\alpha}{R}ed^T p_n}_{(2)} + \underbrace{\frac{1-\alpha}{R}ee^T p_n}_{(3)} \end{aligned}$$

1. Just perform sparse matrix-vector multiply (numpy has a built in sparse matrix data structure)
2. Perform $d^T p_n$ first (since $ed^T p_n = e(d^T p_n)$) to produce a scalar then we have
$$\frac{\alpha}{R}(d^T p_n)e$$
3. Perform $e^T p_n$ first (since $ee^T p_n = e(e^T p_n)$) to produce 1 because p_n sums to 1 and e is all ones
$$\frac{1-\alpha}{R}e$$

Putting this together we have:

$$p_{n+1} = (\alpha P)p_n + \left(\frac{(\alpha d^T)}{R}p_n + \frac{1-\alpha}{R}\right)e$$

Algorithm usually repeats until difference between p^k and p^{k-1} is smaller than some tolarence.

In modern serach have engines other factor besides pure link-based ranking:

- For example we can change standard teleportation with

$$\frac{1-\alpha}{R}ee^T \quad \rightarrow \quad (1-\alpha)ve^T$$

  where special probbability vector v places extra weight on certain sites you like (or payed you)

Review: Eigenvalues and Eigenvectors

For matrix A when we have non-zero eigenvalue λ and non-zero eigenvector x then

$$Ax = \lambda x \quad \rightarrow \quad (A - \lambda I)x = 0$$

which implies that $A - \lambda I$ must be *singular* (non-invertable) for some λ

A singular matrix A satisfies $\det A = 0$ so we solve the characteristic polynomial given by

$$\det(A - \lambda I) = 0$$

Then find any vector in the nullspace of $(A - \lambda I)$ (vector space that gets mapped to zero)

(since we want $Ax = \lambda x$ we have x is both sides)

Recall: if matrix is invertible/non-singular then $Ax = 0$ has only the solution $x = 0$

Convergence of Markov Matrices

1. Every Markov matrix Q has 1 as an eigenvalue
   - Eigenvalues of Q and Q^T are equal since $\det(Q) = \det(Q^T)$ in general, and we have

$$Q^T e = (1)e$$

     This is true because columns of Q sum to 1, so the rows of Q^T sum to 1
   - Then $\lambda = 1$ is an eigenvalue of Q^T and also an eigenvalue of Q
2. Every eigenvalue of a Markov matrix Q satisifies $|\lambda| \leq 1$. So 1 is the largest eigenvalue
   - First show that $|\lambda_i| \leq 1$ in Q^T by letting λ and x be eigenvalue/vector pair for Q

$$Q^T x = \lambda x$$

   - Let k be the index of largest magnitude entry in x such that $|x_j| \leq |x_k|$ for all j

$$(Q^T x)_k = \sum_{j=1}^{n} Q_{jk} x_j = \lambda x_k$$

- Consider

$$\begin{aligned}
|\lambda x_k| = |\lambda||x_k| &= \left|\sum_{j=1}^{n} Q_{jk}x_j\right| \\
&\leq \sum_{j=1}^{n} Q_{jk}|x_j| && \text{(Triangle inequality)} \\
&\leq \sum_{j=1}^{n} Q_{jk}|x_k| && \text{(Since } |x_j| \leq |x_k|\text{)} \\
&\leq |x_k| \sum_{j=1}^{n} Q_{jk} \\
&\leq |x_k| && \text{(Column sums of } Q \text{ are each 1)}
\end{aligned}$$

  Thus $|\lambda||x_k| \leq |x_k|$ becomes $|\lambda| \leq 1$ for Q^T. This holds for Q as well since it has the same eigenvalues.

3. A Markov matrix Q is a *positive* Markov matrix if $Q_{ij} > 0$ for all i, j (definition so no proof)
4. Q is a positive Markov matrix implies **one** linearly independent eigenvector of Q with $|\lambda| = 1$
   - Won't prove this see course notes for reference if curious
   - If Q is positive Markov matrix then eigenvector corresponding to $\lambda = 1$ is unique (Markov chain must converge to only one result)

Using all these theorems we will prove that Page Rank *will converge*.

Theorem: If M is a positive Markov matrix, then Page Rank converges to a unique vector p^∞ for initial probabilty vector p^0.

Let $\vec{x}_\ell$ be the corresponding eigenvector for λ_ℓ for all ℓ.
Assume we can write p^0 (vector of equal values that sums to 1) as a linear combination of eigenvectors $\vec{v}_\ell$

$$p^0 = \sum_{\ell} c_\ell \vec{x}_\ell \qquad \text{for scalars } c_\ell$$

Assume eigenvalues are sorted, so $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq$ etc.

Then $\vec{x}_1$ corresponds to $\lambda_1 = 1$, and as a result PageRank computes

$$\begin{aligned}
(M)^k p^0 = (M)^k \sum_{\ell-1}^{R} c_\ell \vec{x}_\ell &= \sum_{\ell=1}^{R} (M)^k c_\ell \vec{x}_\ell \\
&= \sum_{\ell=1}^{R} \lambda_\ell^k c_\ell \vec{x}_\ell \\
&= c_1\vec{x}_1 + \sum_{\ell=2}^{R} \lambda_\ell^k c_\ell \vec{x}_\ell && \text{(Since } \lambda_1 = 1\text{)}
\end{aligned}$$

We know that $|\lambda_\ell| < 1$ for $\ell > 1$ so

$$p^\infty = \lim_{k\to\infty} (M)^k p^0 = \lim_{k\to\infty} \left(c_1\vec{x}_1 + \sum_{\ell=2}^{R} \lambda_\ell^k c_\ell \vec{x}_\ell \right) = c_1\vec{x}_1 + \sum_{\ell=2}^{R} \left(\lim_{k\to\infty} \lambda_\ell^k \right) c_\ell \vec{x}_\ell = c_1\vec{x}_1$$

We see that the other non-dominant eigenvalues get scaled to 0.
(Result of PageRank is just approximating the dominant eigenvector)

If we start with a different vector $q^0 = \sum_\ell b_\ell \vec{x}_\ell$ then $q^\infty = b_1 \vec{x}_1$

Since q^∞ and p^∞ are probability vectors they both sum to 1

$$\sum_{i=1}^{R} b_1 x_1(i) = \sum_{i=1}^{R} c_1 x_1(i) = 1 \quad \rightarrow \quad b_1 \sum_{i=1}^{R} x_1(i) = c_1 \sum_{i=1}^{R} x_1(i) \quad \rightarrow \quad b_1 = c_1$$

Thus $p^\infty = q^\infty$ so PageRank converges to a unique vector

Convergence Rate

the 2nd largest eigenvalue $|\lambda_2|$ will determine how quickly PageRank will converge as it is the *slowest* decreasing *unwanted* component of p^0

$$p^k = (M)^k p^0 = c_1 \vec{x}_1 + \sum_{\ell=2}^{R} \lambda_\ell^k c_\ell \vec{x}_\ell$$

It turns out for our Google matrix $|\lambda_2| \approx \alpha$ (won't prove this)

Example: Say we have $\alpha = 0.85$ and we perform:

$$\alpha = 0.85 \quad \rightarrow \quad |\lambda_2|^{114} \approx |0.85|^{114} \approx 10^{-8}$$

This says after 114 iterations, vector components of p^0 not corresponding to eigenvalue $|\lambda_1|$ will be scaled down by 10^{-8} (or smaller)

Then we can say vector p^{114} is a good approximation the dominant eigenvector $\vec{x}_1$.

Smaller $|\lambda_2| \approx \alpha$ implies faster convergence, however $\alpha = 0$ is only random teleportation (completely ignores the link structure) so it is a trade off accuracy for efficiency.
(Changes to α change the Google matrix which changes the resulting dominant eigenvector)

PageRank as Power Iteration

PageRank is a special case of power iteration (or power method) for solving general eigenvector problems.

It has many theoretical implications and extensions and applications beyond ranking of web pages, for example as a recommender system for servieces like Facebook, Twitter, Amazon, etc.

Linear Systems of Equations

$$Ax = b$$

Fluid animations require solving linear system of more than one million unknowns for each frame.
10 seconds of video at 30 frames per second requires around 300 linear systems with size $1,000,000^2$ each.

We will use *Gaussian Elimination* however the way we will interpret it will be slightly different:

- Factor matrix A into $A = LU$ where L and U are *triangular*

- Solve $Lz = b$ for intermediate vector z (Forward Solve, L is lower triangular)
- solve $Ux = z$ for x (Backword Solve, U is upper triangular)

Example: Gaussian Elimination as LU Factorization. Consider the linear system:

$$Ax = b \quad \leftrightarrow \quad \begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & 2 \\ 1 & 2 & -1 \end{bmatrix} x = \begin{bmatrix} 0 \\ 4 \\ 2 \end{bmatrix}$$

Perform Gaussian elimination while keeping the multiplies of the row operations in the zeros:

$$R_2 := R_2 - (1)R_1 \quad \begin{bmatrix} 1 & 1 & 1 & | & 0 \\ [1] & -3 & 1 & | & 4 \\ 1 & 2 & -1 & | & 2 \end{bmatrix}$$

$$R_1 := R_3 - (1)R_1 \quad \begin{bmatrix} 1 & 1 & 1 & | & 0 \\ [1] & -3 & 1 & | & 4 \\ [1] & 1 & -2 & | & 2 \end{bmatrix}$$

$$R_3 := R_3 - (-\tfrac{1}{3})R_2 \quad \begin{bmatrix} 1 & 1 & 1 & | & 0 \\ [1] & -3 & 1 & | & 4 \\ [1] & [-1/3] & -5/3 & | & 10/8 \end{bmatrix}$$

Now we have split matrix A into its LU factorization:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ X & 1 & 0 \\ X & X & 1 \end{bmatrix} \qquad U = \begin{bmatrix} X & X & X \\ 0 & X & X \\ 0 & 0 & X \end{bmatrix} \quad \rightarrow \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & -1/3 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -3 & 1 \\ 0 & 0 & -5/3 \end{bmatrix}$$

To complete this solve we just use U and perform back substitution to find x

$$\begin{aligned} -\frac{5}{3}x_3 = \frac{10}{3} &\implies x_3 = -2 \\ -3x_2 + x_3 = 4 &\implies x_2 = -2 \\ x_1 + x_2 + x_3 = 0 &\implies x_1 = 4 \end{aligned}$$

Notice L encodes some information about how it transformed b, which may be useful for solving $Ax = b'$

In general, if we are given the factorization $A = LU$ we do

$$Ax = b \quad \rightarrow \quad LUx = b \quad \rightarrow \quad L(Ux) = b$$

Then defining $z = Ux$ we can solve using system in two steps:

- First solve $Lz = b$ for z
- Then solve $Ux = z$ for x

This is faster because L and U are both triangular.

Remark: Usually use Gaussian elimination directly on the matrix (rather than a system) to get LU.

An application of this is Polynomial fitting of $y = p(x)$, if only changing the y-values being interpolated we can factorize the Vandermonde matrix once and reuse it

$$V = \begin{bmatrix} 1 & x_1 & \cdots & x_1^{n-1} \\ 1 & x_2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^{n-1} \end{bmatrix} \qquad \vec{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

Permutation Matrix

An issue with out current methods is that numerical problems can occur when implemented in code:

```
For k = 1,...,n
    For i = k + 1,...,n
        mult := a_ik / a_kk # possible divide by (exactly or nearly) zero
        a_ik := mult
        for j = k+1,...,n
            a_ij := a_ij - mult*a_kj
        EndFor
    EndFor
EndFor
```

Even when the diagonal entry a_{kk} is close to zero then a_{ik}/a_{kk} will produce a value large in magnitude which can cause large floating errors during subtraction (or an error when $a_{kk} = 0$).

During Guassin elimination this is solved by swapping rows when its diagonal entry is zero, but for our nuermical implementations we will *always* swap to minimize floating point error incurred.

Strategy: Find a row with the *largest mangnitude entry* in the current column beneath the current row and swap those rows if larger than the current entry.

We don't make these swaps directly to A, instead we take P to be an identity matrix and swap its rows:

$$A = \begin{bmatrix} 0.5 & 2 & 3 \\ 1 & 4 & 2 \\ -3 & 2 & -1 \end{bmatrix} \qquad P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \rightarrow \quad P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \rightarrow \quad PA = \begin{bmatrix} -3 & 2 & -1 \\ 1 & 4 & 2 \\ 0.5 & 2 & 3 \end{bmatrix}$$

We call P the *permutation matrix* and now find the LU factoriation of

$$PA = LU$$

This turns the linear system we are solving into $PAx = Pb$ and we usually use $b' = Pb$ to get $LUx = b'$.

Example: LU Factoriation with Pivoting. Find $PA = LU$ factoriation for

$$\begin{bmatrix} 1 & 4 & 5 \\ -2 & 3 & 3 \\ 3 & 0 & 6 \end{bmatrix}$$

Initialize $P = I$ and notice for column 1, row 3 has the largest value, so swap row 1 and row 3:

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 3 & 0 & 6 \\ -2 & 3 & 3 \\ 1 & 4 & 5 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} 3 & 0 & 6 \\ [-2/3] & 3 & 7 \\ [1/3] & 4 & 3 \end{bmatrix}$$

For column 2, row 3 has largest value, so swap row 2 and row 3:

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 3 & 0 & 6 \\ [1/3] & 4 & 3 \\ [-2/3] & 3 & 7 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} 3 & 0 & 6 \\ [1/3] & 4 & 3 \\ [-2/3] & [3/4] & 19/4 \end{bmatrix}$$

Thus we arrive at

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ -2/3 & 3/4 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 3 & 0 & 6 \\ 0 & 4 & 3 \\ 0 & 0 & 19/4 \end{bmatrix} \qquad P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

L is lower *unit* triangular and U is upper triangular. We can verify $PA = LU$ for a sanity check.

To solve $Ax = b$ we do

1. Compute $b' = Pb$ (multiply to permute b)
2. Solve $Lz = b'$ (forward solve)
3. Solve $Ux = z$ (backward solve)

Using our $PA = LU$ factorization solve $Ax = b$ for

$$b = \begin{bmatrix} 4 \\ 1 \\ -3 \end{bmatrix}$$

1. Compute $b' = Pb$

$$Pb = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ 1 \\ -3 \end{bmatrix} = \begin{bmatrix} -3 \\ 4 \\ 1 \end{bmatrix} = b'$$

2. Solve $Lz = b'$

$$Lz = \begin{bmatrix} 1 & 0 & 0 \\ 1/3 & 1 & 0 \\ -2/3 & 3/4 & 1 \end{bmatrix} z = \begin{bmatrix} -3 \\ 4 \\ 1 \end{bmatrix} \quad \rightarrow \quad z = \begin{bmatrix} -3 \\ 4 - \frac{1}{3}(-3) = 5 \\ 1 - \frac{3}{4}(5) + \frac{2}{3}(-3) = -19/4 \end{bmatrix}$$

3. Solve $Ux = z$

$$Ux = \begin{bmatrix} 3 & 0 & 6 \\ 0 & 4 & 3 \\ 0 & 0 & 19/4 \end{bmatrix} x = \begin{bmatrix} -3 \\ 5 \\ -19/4 \end{bmatrix} \quad \rightarrow \quad x = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}$$

Notice that for step (1) we don't actually need to matrix muliply, we can just swap the rows

Scipy has a LU factorization `scipy.linalg.lu` along with a triangular solve `scipy.linalg.solve_triangular`

Cost of Solving Linear Systems

We will measure the cost in total FLOPs: *Floating Point OPerations*

True operation count be hardware dependent (Ex: FMA (*Fused Multiply Add*) can be a single operation)

Remark: FLOPS is *Floating Point OPerations per Second* (which is not what we are looking for)

We will need two sum identities:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

- FLOP count for LU factorization

$$\begin{aligned}
\sum_{k=1}^{n}\sum_{i=k+1}^{n}\left(1+\sum_{j=k+1}^{n}2\right) &= \sum_{k=1}^{n}\sum_{i=k+1}^{n}(1+(n-(k+1)+1)2) \\
&= \sum_{k=1}^{n}\sum_{i=k+1}^{n}(1+2n-2k) \\
&= \sum_{k=1}^{n}(n-k)(1+2n-2k) \\
&= \sum_{k=1}^{n}(2k^2+k(-4n-1)+(2n^2+n)) \\
&= 2\frac{n(n+1)(2n+1)}{6}+(-4n-1)\frac{n(n+1)}{2}+n(2n^2+n) \\
&= \frac{2n^3}{3}+n^2+\frac{n}{3}-2n^3-\frac{5n^2}{2}-\frac{n}{2}+2n^3+n^2 \\
&= \frac{2n^3}{3}-\frac{n^2}{2}-\frac{n}{6} \\
&= \frac{2n^3}{3}+O(n^2)
\end{aligned}$$

- FLOP count for triangular solve

$$\begin{aligned}
\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}2\right) &= n+\sum_{i=1}^{n}2(n-(i+1)+1) \\
&= n+\sum_{i=1}^{n}(2n-2i) \\
&= n+2n^2-2\frac{n(n+1)}{2} \\
&= n+2n^2-(n^2+n) \\
&= n^2
\end{aligned}$$

Thus LU factorization costs $\frac{2n^3}{3}+O(n^2)$ FLOPs

Triangular solves cost $n^2+O(n)$ FLOPs each so total is $2n^2+O(n)$

Factorization cost dominates when n is large, if given the factorization solving for RHS is relatively cheaper.

LU as Matrix Operations

Recall that the row operations can represented as matrices.

Example: row operation $R_2 := R_2 - (\frac{a_{2,1}}{a_{1,1}})R_1$ can be written as matrix

$$M = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{a_{2,1}}{a_{1,1}} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \rightarrow \quad \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ -3/2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{M} \underbrace{\begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & -1 \\ -2 & 2 & 1 \end{bmatrix}}_{A^{\text{old}}} = \underbrace{\begin{bmatrix} 2 & 3 & 4 \\ 0 & -1/2 & -7 \\ -2 & 2 & 1 \end{bmatrix}}_{A^{\text{new}}}$$

Thus the process of LU factoriation can be viewed as a sequence of matrix multiplicaitons applied to A.

These row operations create U, e.g. in the 3×3 case we have shown:

$$M^{(3)}M^{(2)}M^{(1)}A = U$$

Then we have

$$A = \left(M^{(3)}M^{(2)}M^{(1)}\right)^{-1} U = \underbrace{\left(m^{(1)}\right)^{-1}\left(m^{(2)}\right)^{-1}\left(m^{(3)}\right)^{-1}}_{L} U$$

The inverse of $M^{(i)}$ is the same matirx but with the off-diagonal entry negated

$$\begin{bmatrix} 1 & 0 & 0 \\ -3/2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 3/2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The effect of LU factoriation is to left-multiply A by matrices $M^{(k)}$ that comprise L^{-1} to produce U

$$M^{(3)}M^{(2)}M^{(1)}A = L^{-1}A = U$$

interleaving permutation matrices $P^{(k)}$ before each $M^{(k)}$ leads to the $PA = LU$ factorization.

Solving $Ax = b$ by Inverting A

An obvious alternative for solving $Ax = b$ is to invert A to get A^{-1} then find $A^{-1}b = x$

To invert A we can do

- Factor A into $A = LU$ as usual
- Find columns of A^{-1} by solving $LUx = e_i$ for $i = 1, \ldots, n$

The cost to invert is one LU factorization and n pairs of triangular solves, then we need to add the cost of multiplying $A^{-1}b$.

Not only is it slow but this method will also incur more floating point error.

As a general rule of thumb: most numerical algorithsm avoid ever computing A^{-1}

Norms and Conditioning

We will see the relationship between the norm and conditioning of a matrix.

- Norms are measurement of *size/magnitude* of vectors or matrices
- Conditioning describes how the output of a matrix changes with a change to input

Vector Norms

For a vector $x = [x_1, \ldots, x_n]^T$ some common choices of norms are:

- 1-norm (Manhatten norm):

$$||x||_1 = \sum_{i=1}^{n} |x_i|$$

- 2-norm (Euclidean norm)

$$||x||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

- ∞-norm (max norm)

$$||x||_\infty = \max_i |x_i|$$

We can collect all these the p-norm, for $p = 1, 2, \infty$, which is written as:

$$||x||_p = \left(\sum_{i=i}^{n} |x_i|^p \right)^{1/p}$$

(in the ∞ case this only holds for the *limit* of the RHS)

Some key properties of norms:

- If the norm is zreo, then the vector must be the zero-vector

$$||x|| = 0 \quad \implies \quad x_i = 0 \ \forall i$$

- The scaling the vector must scale the norm by the same amount

$$||\alpha x|| = |\alpha| \cdot ||x|| \quad \text{for scalar } \alpha$$

- The triangle inequality holds:

$$||x + y|| \leq ||x|| + ||y||$$

`numpy.linalg.norm` or `scipy.linalg.norm` implement vector (and matrix) norms

Matrix Norms

The standard matrix norms are:

- 1-norm (max absolute column sum)

$$||A||_1 = \max_j \sum_{i=1}^{n} |A_{ij}|$$

- 2-norm (*spectral* norm is related to eigenvalues)

$$||A||_2 = \sqrt{\text{max magnitude eigenvalue of } A^T A}$$

- ∞-norm (max absolute row sum)

$$||A||_\infty = \max_i \sum_{j=1}^{n} |A_{ij}|$$

- *Frobenius* norm

$$||A||_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}^2}$$

Matrix norm properties:

$$||A|| = 0 \quad \iff \quad A_{ij} = 0 \ \forall i, j$$
$$||\alpha A|| = |\alpha| \cdot ||A|| \quad \text{for scalar } \alpha$$
$$||A + B|| \leq ||A|| + ||B||$$
$$||A\vec{x}|| \leq ||A|| \cdot ||\vec{x}||$$
$$||AB|| \leq ||A|| \cdot ||B||$$
$$||I|| = 1$$

Conditioning of Linear Systems

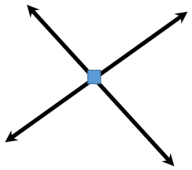
Conditioning describes how the output of changes with some change in input. It shows how difficult it is to find a *good* solution to a problem, *independent* of the algorithm/numerical method used.

When solving a linear system $Ax = b$ how much does a perturbation of b or A cause solution x to change:

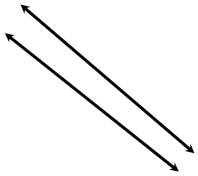
- Well-conditioned if x changes little
- Ill-conditioned if x changes lots

Example: finding the intersection between two lines

- Well-conditioned case: lines are nearly perpendicular



- Ill-conditioned case: lines are nearly parallel
  - a small change in the angle of one line can really change where the intersection is.



Conditioning derivations:

- Perturbing b in the system $Ax = b$

$$A(x + \Delta x) = b + \Delta b \quad \rightarrow \quad A\Delta x = \Delta b \quad \rightarrow \quad \Delta x = A^{-1}\Delta b$$

1. Apply norm rules to $Ax = b$

$$||b|| = ||Ax|| \leq ||A|| \cdot ||x|| \quad \rightarrow \quad \frac{1}{||x||} \leq \frac{||A||}{||b||}$$

2. Apply norm rules to $\Delta x = A^{-1}\Delta b$

$$||\Delta x|| = ||A^{-1}\Delta b|| \leq ||A^{-1}|| \cdot ||\Delta b|| \quad \rightarrow \quad ||\Delta x|| \leq ||A^{-1}|| \cdot ||\Delta b||$$

Combine (1) and (2) by multiplication

$$\frac{1}{||x||} \cdot ||\Delta x|| \leq \frac{||A||}{||b||} \cdot ||A^{-1}|| \cdot ||\Delta b|| \quad \rightarrow \quad \frac{||\Delta x||}{||x||} \leq ||A^{-1}|| \cdot ||A|| \cdot \frac{||\Delta b||}{||b||}$$

Letting $\kappa = ||A^{-1}|| \cdot ||A||$ we have

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

- Perturbing A in the system $Ax = b$

$$(A + \Delta A)(x + \Delta x) = b \quad \rightarrow \quad Ax + A\Delta x + \Delta A(x + \Delta x) = b$$

Subtract off $Ax = b$ and rearrange

$$A\Delta x = -\Delta A(x + \Delta x) \quad \rightarrow \quad \Delta x = -A^{-1}\Delta A(x + \Delta x)$$

Take the norms and apply the norm rules

$$||\Delta x|| \leq ||A^{-1}|| \cdot ||\Delta A|| \cdot ||x + \Delta x||$$

Multiply by $1 = \frac{||A||}{||A||}$, rearrange, then let $\kappa = ||A^{-1}|| \cdot ||A||$ to get

$$\frac{||\Delta x||}{||x + \Delta x||} \leq \kappa(A) \frac{||\Delta A||}{||A||}$$

The conditioning number of a matrix A is denoted $\kappa(A) = ||A|| \cdot ||A^{-1}||$

$$\kappa \approx 1 \quad \rightarrow \quad A \text{ is well-conditioned}$$

$$\kappa \gg 1 \quad \rightarrow \quad A \text{ is ill-conditioned}$$

Any norm will work, however, different norms will give a different κ (condition number).

A useful property of vector/matrix norms is *equivalence* which states they do not different from one another by more than a *constant factor*:

$$C_1||x||_a \leq ||x||_b \leq C_2||x||_a$$

for some constants C_1, C_2.

Residual

As a proxy for error we often using the *residual* r

$$r = b - A(x_{\text{approx}})$$

the exact solution ($x_{\text{approx}} = x$) should give $r = 0$

This residual r can be easily computed (unlike the error) but how does it relate to error?

- Theme $x_{\text{approx}} = x + \Delta x$ then

$$r = b - A(x + \Delta x) \quad \rightarrow \quad A(x + \Delta x) = b - r$$

- r looks like a perturbation of b, so applying the earlier bound using $\Delta b = r$ we have

$$\frac{||\Delta x||}{||x||} \leq \kappa(A)\frac{||\Delta b||}{||b||} \quad \rightarrow \quad \frac{||\Delta x||}{||x||} \leq \kappa(A)\frac{||r||}{||b||}$$

The solution's relative error $\frac{||\Delta x||}{||x||}$ is bound by κ times the *relative* size of residual r w.r.t. b

Thus if we roughly know A's condition number ($\kappa(A) = ||A|| \cdot ||A^{-1}||$) then the computed residual indicates how large the error can be (in the worst case).

- if $\kappa \approx 1$ then a small residual indicates a small relative error
- if κ is large, small residual can still mean error is large

Many algorithms for solving $Ax = b$ are *iterative* (rather than *direct* like LU/Gaussian elimination)

Itertive methods update an estimate until the residual dictates the solution is good enough.
(CS 475 for more iterative and direct schemes, entire semester on this last two topics)

Gaussian Elimination Error

$Ax = b$ is quite stable and accurate but not exact, Wilkinson [1961] showed that GE's numerical result $\hat{x}$ gives the *exact solution to a nearby problem*:

$$(A + \Delta A)\hat{x} = b$$

where E is machine epsilon and $||\Delta A|| = ||A||E$

Applying our earlier bound gets us

$$\frac{||\Delta x||}{||x + \Delta x||} \leq \kappa(A)\frac{||\Delta A||}{||A||} \quad \rightarrow \quad \frac{||x - \hat{x}||}{||\hat{x}||} \leq \kappa(A)\frac{||\Delta A||}{||A||} = \kappa(A)\frac{||A||E}{||A||} = \kappa(A)E$$

Example: if $\kappa(A) = 10^{10}$ and $E = 10^{-16}$ then relative error is around 10^{-6} or around 6 significant digits of accuracy.

Always remember that conditioning is a property of the *problem*, that is a system $Ax = b$ is well/ill-conditioned independent of how we choose to solve it.