

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. What is Computational Science?

2. Some applications

3. Some remarks

Symbolic vs. numerical computation

Operating systems

What is Computational Science?

- ▶ Also known as *Numerical Analysis* and *Scientific Computing*.
- ▶ Contains elements of mathematics and computer science.
- ▶ Main applications are in science, engineering and data science.
- ▶ Origins are old but the rapid development of computers makes it an ever-changing field. . .
- ▶ Possible definitions:

What is Computational Science?

- ▶ Also known as *Numerical Analysis* and *Scientific Computing*.
- ▶ Contains elements of mathematics and computer science.
- ▶ Main applications are in science, engineering and data science.
- ▶ Origins are old but the rapid development of computers makes it an ever-changing field. . .
- ▶ Possible definitions:
A discipline concerned with the design, implementation and use of mathematical models to analyse and solve scientific problems. Typically, the term refers to the use of computers to perform simulations or numerical analysis of a scientific system or process. (Nature);

What is Computational Science?

- ▶ Also known as *Numerical Analysis* and *Scientific Computing*.
- ▶ Contains elements of mathematics and computer science.
- ▶ Main applications are in science, engineering and data science.
- ▶ Origins are old but the rapid development of computers makes it an ever-changing field. . .
- ▶ Possible definitions:
Appropriate topics include the rigorous study of convergence of algorithms, their accuracy, their stability, and their computational complexity. (SIAM Journal on Numerical Analysis);

What is Computational Science?

- ▶ Also known as *Numerical Analysis* and *Scientific Computing*.
- ▶ Contains elements of mathematics and computer science.
- ▶ Main applications are in science, engineering and data science.
- ▶ Origins are old but the rapid development of computers makes it an ever-changing field. . .
- ▶ Possible definitions:
The process of producing and analyzing approximate, numerical solutions to mathematical or scientific problems.

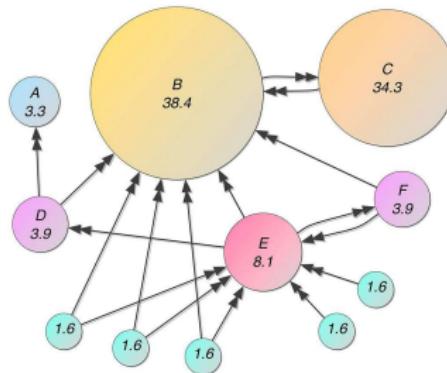
Famous CS example: PageRank.

- ▶ Ranks nodes in a network according to number and quality of links.
- ▶ Used by Google for websites.
- ▶ Can be formulated as a linear system.

To solve:

$$Lw = w$$

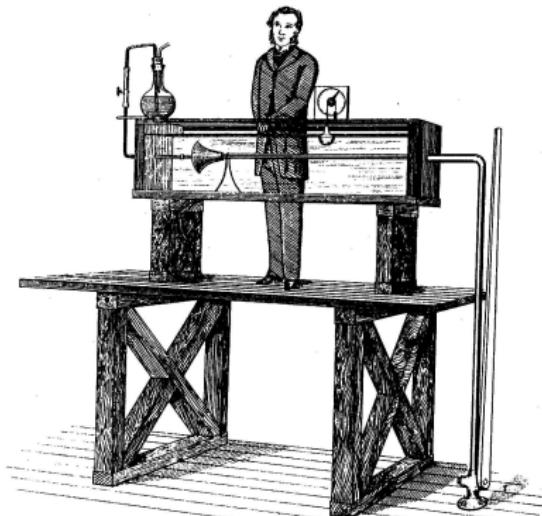
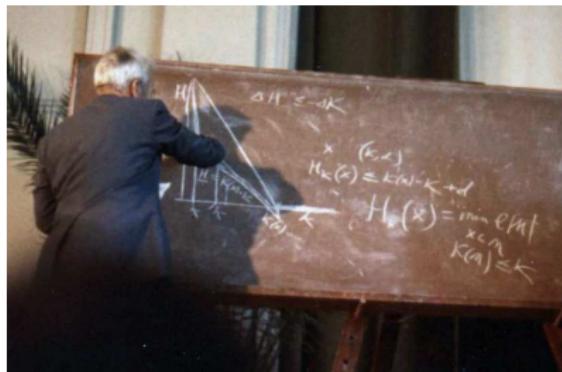
where the entry L_{ij} is related to the number of links from node j to node i .



Wikipedia, public domain.

Computational science changed physics/engineering radically.

Before the 1970s the two pillars
where *theory* and *experiment*.

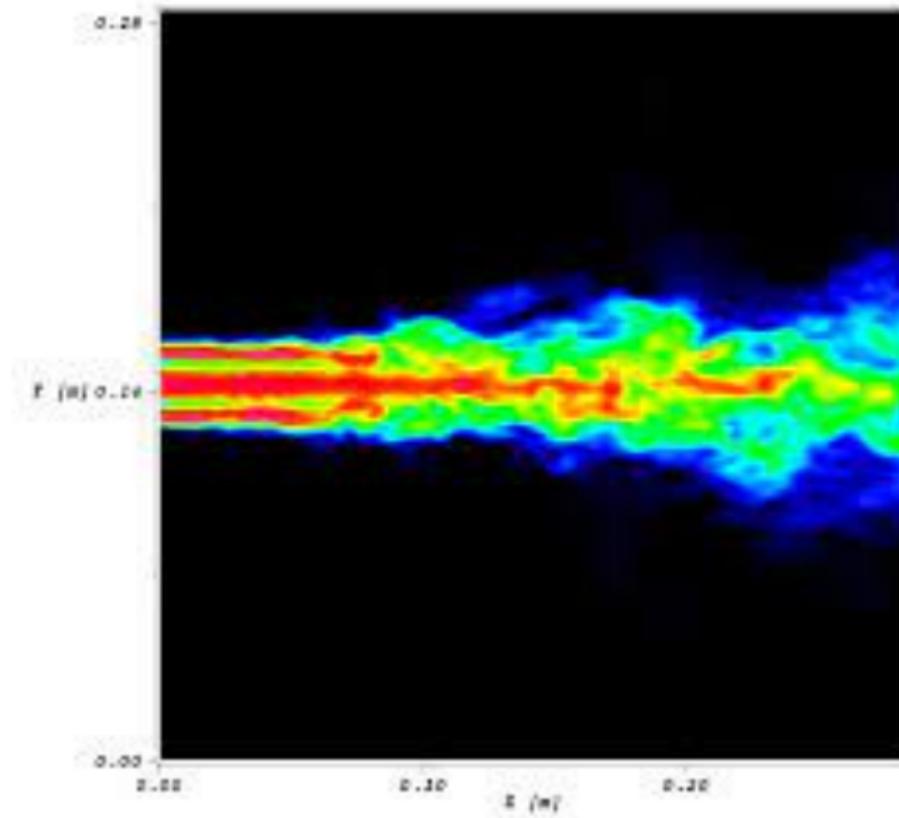


Wikipedia, public domain.

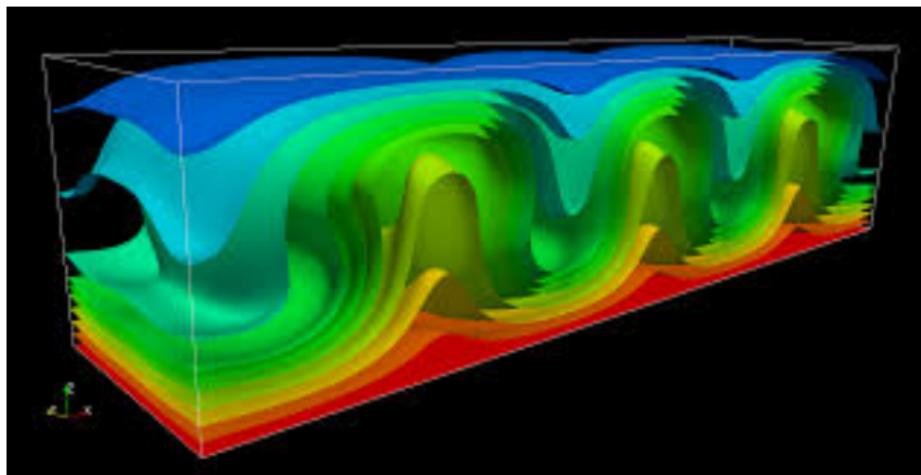
Fig. 9.1. Sketch of Reynolds's dye experiment, taken from his 1883 paper

In comparison, numerical approximation is often

- ▶ more feasible,
- ▶ easier to analyse,
- ▶ easier to manipulate,
- ▶ less expensive.



Turbulent Flow



Convective Flow



Figure 13: Collapsed steel building during January 17, 1995 Kobe Earthquake (by K. Meguro)

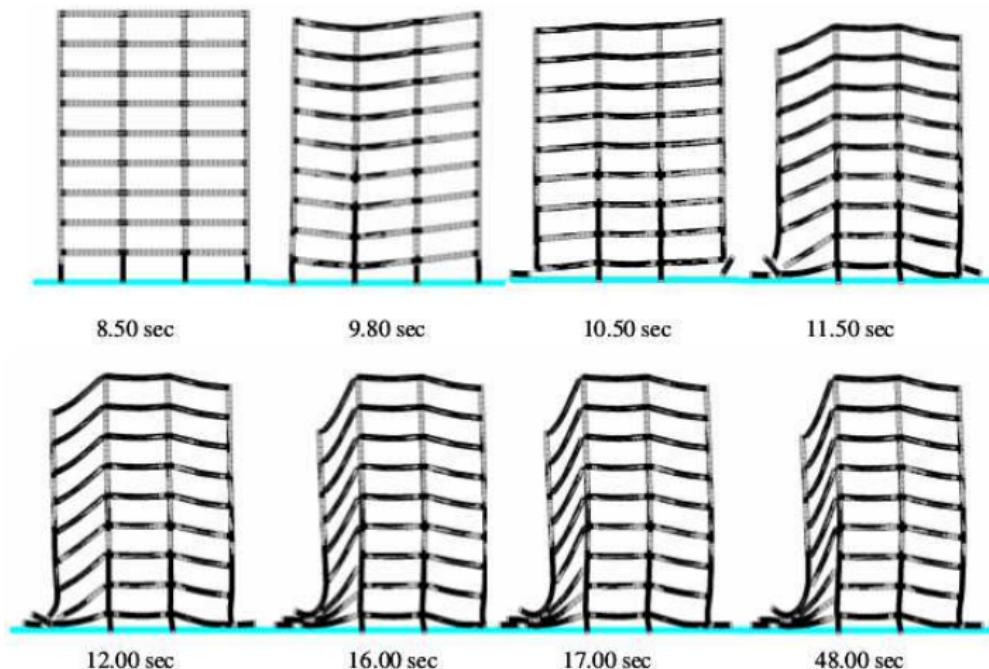
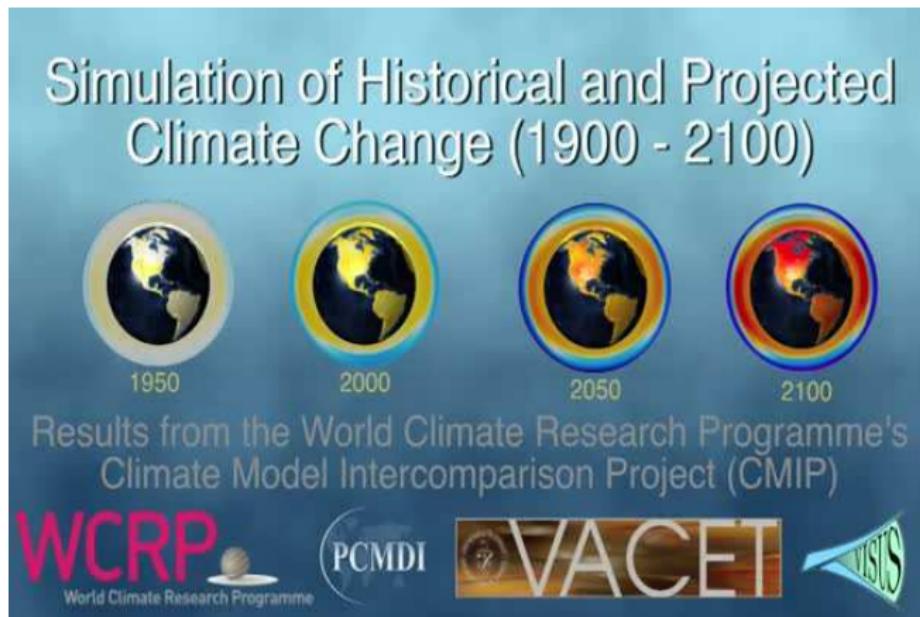


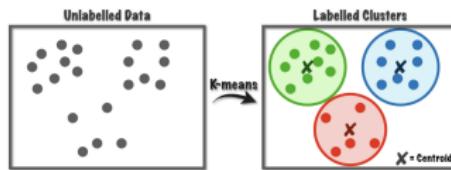
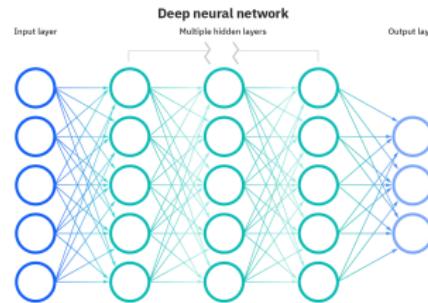
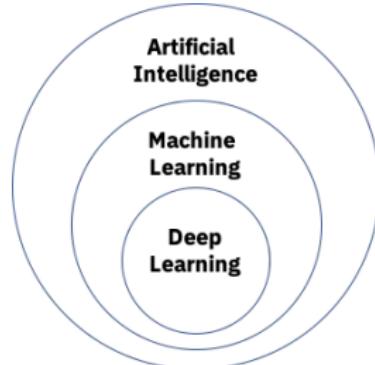
Figure 12: Ground soft-story collapse mechanism

Elkholy & Meguro, 13th World Congress on Earthquake Engineering



Safety of Nuclear Power (image by Kernkraftwerk Grafenrheinfeld - 2013, CC)





Machine Learning

Symbolic computation: application of the rules of algebra and mathematical identities to manipulate an expression involving symbols and variables.

Example: $x \in \mathbb{R}$, $a \in \mathbb{R}$

$$2 \exp(-x^2 + a) = 1 \Leftrightarrow$$

$$\ln(2) - x^2 + a = 0 \Leftrightarrow$$

$$x = \pm \sqrt{\ln(2) + a}$$

Numerical computation: performing arithmetic on (approximate) numerical quantities. The result is a (set of) number(s) of finite precision.

Example: $x \in \mathbb{R}$,

$$2 \exp(-x^2 + 2) = x \Leftrightarrow \text{approximate } x \text{ from a sequence}$$

$$x_0 = 1.\underline{5}, \quad x_1 = 1.\underline{5}1015398, \quad x_2 = 1.\underline{5}10254584, \dots$$

- ▶ MAPLE and MATHEMATICA are **symbolic computation** programs (“Computer Algebra Systems” or CAS) – but they can do some numerical computation, too.

- ▶ MAPLE and MATHEMATICA are **symbolic computation** programs (“Computer Algebra Systems” or CAS) – but they can do some numerical computation, too.
- ▶ Python and MATLAB are best suited for **numerical computation** – but can do some symbolic computation, too.

- ▶ MAPLE and MATHEMATICA are **symbolic computation** programs (“Computer Algebra Systems” or CAS) – but they can do some numerical computation, too.
- ▶ Python and MATLAB are best suited for **numerical computation** – but can do some symbolic computation, too.
- ▶ Lower-level programming languages like FORTRAN, C and C++ are commonly used for numerical computation and are generally faster than Python/Matlab.

- ▶ MAPLE and MATHEMATICA are **symbolic computation** programs (“Computer Algebra Systems” or CAS) – but they can do some numerical computation, too.
- ▶ Python and MATLAB are best suited for **numerical computation** – but can do some symbolic computation, too.
- ▶ Lower-level programming languages like FORTRAN, C and C++ are commonly used for numerical computation and are generally faster than Python/Matlab.
- ▶ Symbolic computations are **exact**.

- ▶ MAPLE and MATHEMATICA are **symbolic computation** programs (“Computer Algebra Systems” or CAS) – but they can do some numerical computation, too.
- ▶ Python and MATLAB are best suited for **numerical computation** – but can do some symbolic computation, too.
- ▶ Lower-level programming languages like FORTRAN, C and C++ are commonly used for numerical computation and are generally faster than Python/Matlab.
- ▶ Symbolic computations are **exact**.
- ▶ Numerical computations are **approximate**.

- ▶ MAPLE and MATHEMATICA are **symbolic computation** programs (“Computer Algebra Systems” or CAS) – but they can do some numerical computation, too.
- ▶ Python and MATLAB are best suited for **numerical computation** – but can do some symbolic computation, too.
- ▶ Lower-level programming languages like FORTRAN, C and C++ are commonly used for numerical computation and are generally faster than Python/Matlab.
- ▶ Symbolic computations are **exact**.
- ▶ Numerical computations are **approximate**.
- ▶ Which is appropriate depends entirely on the context.

The following tools are often used in Computational Science:

- ▶ Python
- ▶ Linux
- ▶ git
- ▶ Latex

Definition (Operating System)

The *Operating System* (OS) is the foundation systems software that controls the execution of computer programs and controls services like operating the internet connection, outputting to the screen, etc.

Examples: Unix, **Linux**, Mac OS, MS-DOS, Windows, etc.

Why Linux?

- ▶ It is free and always available.
Your work will be portable, re-usable and independent of your environment.
- ▶ It is open-source.
- ▶ It is often faster.
- ▶ It is intended for scientific computing.
Demo 1 (see the Top 500 Fastest Supercomputers).
- ▶ It strengthens your CV.
Demo 2 (job listings on Workopolis).

Definition (Compiler)

A *compiler* is a computer program that reads high-level programming language and **translates** it to a low-level (machine) language (also called object code). The object code output by a compiler is typically put into a file called an **executable** that can be run (executed) at a later time.

Definition (Interpreter)

An *interpreter* is a computer program that reads input code in some high-level programming language and **immediately executes the input program**.

A programming language executed by an interpreter is called an **interpreted programming language** (contrast with compiled programming language).

- ▶ Interpreted code generally runs slower than compiled code because the interpreter **reads and executes** each statement as it goes along while the executable machine code output from a compiler need only **execute** each statement.

- ▶ Interpreted code generally runs slower than compiled code because the interpreter **reads and executes** each statement as it goes along while the executable machine code output from a compiler need only **execute** each statement.
- ▶ When testing/developing code, interpreted languages are usually easier.

- ▶ Interpreted code generally runs slower than compiled code because the interpreter **reads and executes** each statement as it goes along while the executable machine code output from a compiler need only **execute** each statement.
- ▶ When testing/developing code, interpreted languages are usually easier.
- ▶ Programs like MATLAB & MAPLE are **interpreters**. They allow for development of scientific algorithms with lots of debugging help.

- ▶ Interpreted code generally runs slower than compiled code because the interpreter **reads and executes** each statement as it goes along while the executable machine code output from a compiler need only **execute** each statement.
- ▶ When testing/developing code, interpreted languages are usually easier.
- ▶ Programs like MATLAB & MAPLE are **interpreters**. They allow for development of scientific algorithms with lots of debugging help.
- ▶ When developing a complicated program, we often use an interpreted language first, and then translate into a lower-level language like FORTRAN or c.
- ▶ Python is mostly an interpreter, but often calls compiled libraries for the actual computations and can be fast.

2072U Computational Science I

Winter 2022

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

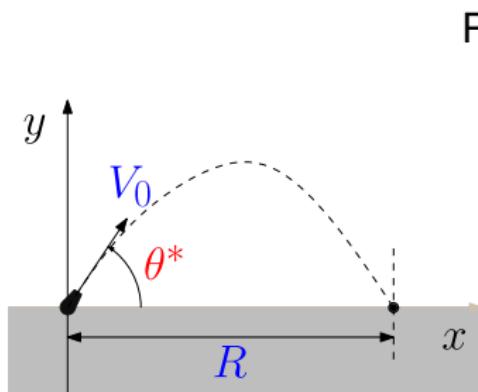
1. Root finding: introduction

2. Iterative methods

3. First method: bisection

Example 1: hitting a target with a cannon.

To what elevation should the cannon be raised to hit the target?



Parameters:

- ▶ g = gravitational acceleration (ms^{-2}): known;
- ▶ V_0 = initial speed (ms^{-1}): known;
- ▶ R = distance to target (m): known;
- ▶ θ^* = required elevation (radians): unknown.

Determine elevation θ^* needed to hit target using known values of parameters V_0 , R , and g .

- ▶ Coordinates of cannonball at time t are $(x(t), y(t))$.
- ▶ Motion of cannonball determined by Newton's 2nd law:

$$\begin{cases} x''(t) = 0, & x(0) = 0, x'(0) = V_0 \cos \theta^* \\ y''(t) = -g, & y(0) = 0, y'(0) = V_0 \sin \theta^* \end{cases}$$

- ▶ The solution to these differential equations is

$$x(t) = (V_0 \cos \theta^*) t$$

$$y(t) = (V_0 \sin \theta^*) t - \frac{1}{2} g t^2$$

- ▶ Want to find θ^* such that $x(T) = R$ and $y(T) = 0$, where T is time of flight
- ▶ Can eliminate T using $y(T) = 0$ (i.e. object is on the ground)
- ▶ If $y(T) = 0$, then $T = 0$ or $T = \frac{2V_0 \sin \theta^*}{g}$
- ▶ Reject $T = 0$, so to have $x(T) = R$, we must have

$$x(T) = (V_0 \cos \theta^*) \left(\frac{2V_0 \sin \theta^*}{g} \right) = R$$

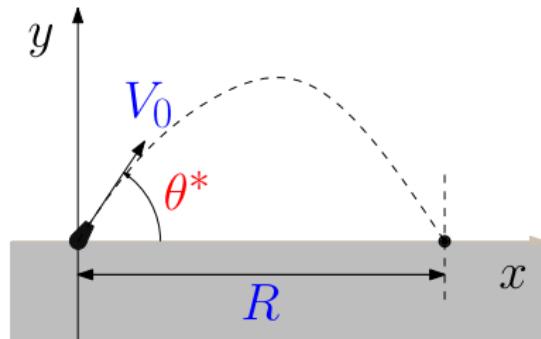
- ▶ Want to find θ^* such that $x(T) = R$ and $y(T) = 0$, where T is time of flight
- ▶ Can eliminate T using $y(T) = 0$ (i.e. object is on the ground)
- ▶ If $y(T) = 0$, then $T = 0$ or $T = \frac{2V_0 \sin \theta^*}{g}$
- ▶ Reject $T = 0$, so to have $x(T) = R$, we must have

$$x(T) = (V_0 \cos \theta^*) \left(\frac{2V_0 \sin \theta^*}{g} \right) = R$$

⇒ Zero-finding problem: find elevation θ^* such that $f(\theta^*) = 0$, where

$$f(\theta) := 2 \sin \theta \cos \theta - \frac{Rg}{V_0^2}$$

We can solve this problem analytically:



- ▶ There is a unique solution $0 < \theta^* < \pi/2$ if

$$\frac{Rg}{V_0^2} < 1$$

- ▶ Analytic formula for θ^* :

$$f(\theta^*) = 2 \sin \theta^* \cos \theta^* - \frac{Rg}{V_0^2} = 0$$

$$\theta^* = \frac{1}{2} \arcsin \left(\frac{Rg}{V_0^2} \right)$$

(uses $2 \sin \theta \cos \theta = \sin(2\theta)$)

In reality, there is **air friction**, and the equations of motion are:

$$\begin{cases} x''(t) = -c(x')^2, & x(0) = 0, x'(0) = V_0 \cos \theta^* \\ y''(t) = -g - c y' |y'|, & y(0) = 0, y'(0) = V_0 \sin \theta^* \end{cases}$$

$$x(t) = \frac{1}{c} \ln(cV_0 t \cos \theta^* + 1)$$

$$y(t) = \begin{cases} \frac{1}{c} \ln \left(\cos(\sqrt{cg}t) + \sin(\sqrt{cg}t) \sqrt{\frac{c}{g}} V_0 \sin \theta^* \right) & \text{for } t \leq t' \\ -\frac{1}{c} \ln \left(\cosh(\sqrt{cg}[t - t']) \right) \\ -\frac{1}{c} \ln \left(\cos(\arctan(\sqrt{\frac{c}{g}} V_0 \sin \theta^*)) \right) & \text{for } t > t' \end{cases}$$

where

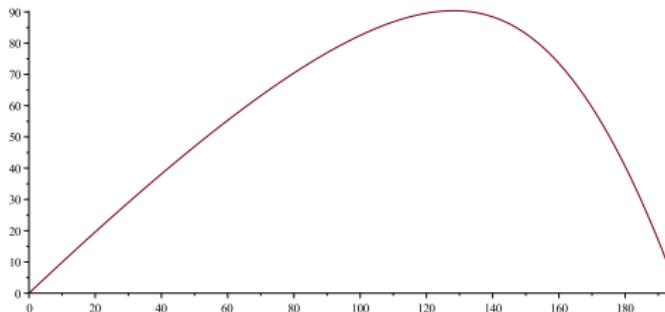
$$t' = \frac{1}{\sqrt{cg}} \arctan \left(\sqrt{\frac{c}{g}} V_0 \sin \theta^* \right)$$

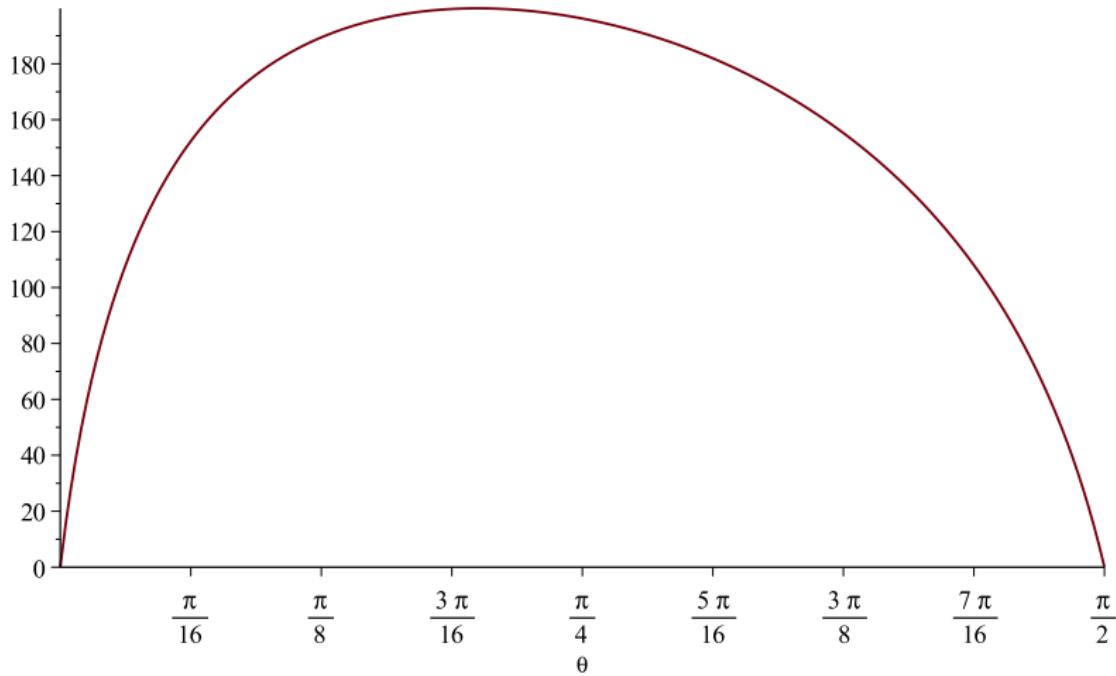
is the time at which the ball is at its highest point ($y'(t') = 0$).

1. Solve for t^* from $y(t^*) = 0$.
2. Find the distance of impact as $x(t^*)$:

$$x(t^*) = \frac{1}{c} \ln \left[\sqrt{\frac{c}{g}} V_0 \cos \theta^* \left(\arctan \left[\sqrt{\frac{c}{g}} V_0 \sin \theta^* \right] + \operatorname{arccosh} \left[\sqrt{\frac{g + cV_0^2 \sin^2 \theta^*}{g}} \right] \right) + 1 \right]$$

3. Solve for θ^* from $x(t^*) = R\dots$



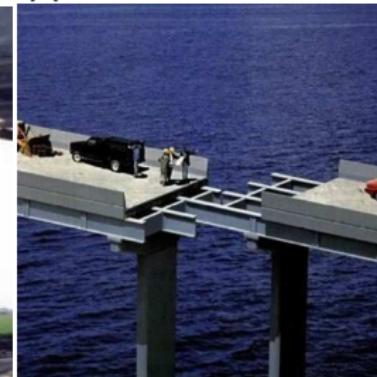


We cannot find an analytic solution anymore.
Instead, we must look for a numerical approximation like

$$\theta^* = 0.19 \pm 10^{-2}$$

where $\Delta\theta^* = 10^{-2}$ is the *absolute error*.

This error needs to be small enough for the application.



Example 2: pension payments

Suppose you pay an amount of money to a bank every year and they promise to pay you a lump sum when you retire.
Over one year, you would pay an amount

$$\nu P = \frac{1}{1+i} P \quad (i \text{ is the percentage interest rate})$$

for this service to compensate for the loss of interest.
Over n years you will pay

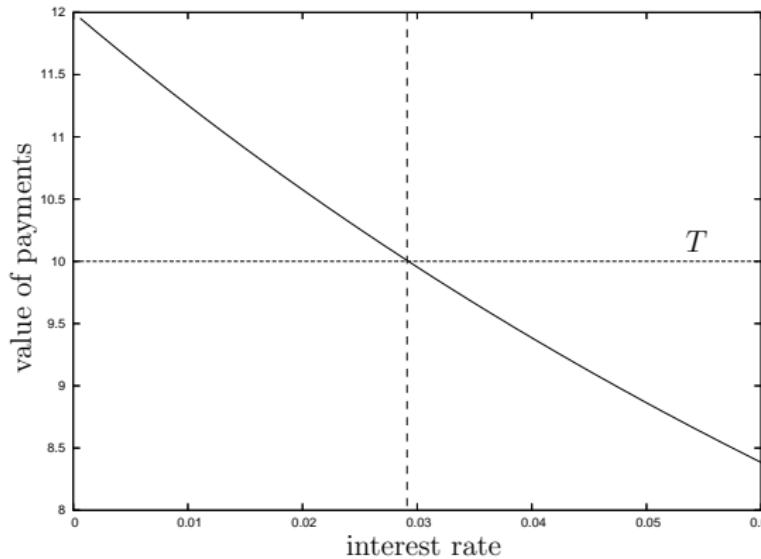
$$\nu P + \nu^2 P + \dots + \nu^n P = \nu P \frac{1 - \nu^n}{1 - \nu} = T$$

for a lump sum payment of nP .

Now suppose we *know* the total value and *want to know* the interest rate. We then have to solve the following equation:

$$\nu \frac{1 - \nu^n}{1 - \nu} = \frac{1}{i} - \frac{1}{i} \left(\frac{1}{1 + i} \right)^n = T/P$$

For example, let $n = 12$, $T/P = 10$.



In order to find the interest rate i^* we must solve a nonlinear equation

$$f(i) = \frac{1}{i} - \frac{1}{i} \left(\frac{1}{1+i} \right)^n - T/P = 0$$

to find an answer like

$$i^* = 0.0292285 \pm 2 \times 10^{-7}$$

For this end, we can use *iterative* methods.

The two methods we will study here are

1. *bisection*
2. *Newton's method* (and an adaptation called the *secant method*).

Iterative methods

- ▶ Algorithms for solving $f(x^*) = 0$ are usually **iterative**.
- ▶ Starting from $x^{(0)}$, make sequence of iterates

$$x^{(1)} = \phi(x^{(0)}),$$

$$x^{(2)} = \phi(x^{(1)}),$$

 \vdots

$$x^{(k+1)} = \phi(x^{(k)}),$$

 \vdots

ϕ function/rule generating successive iterates

- ▶ Rule $x^{(k+1)} = \phi(x^{(k)})$ for $k \geq 0$ is **recurrence relation**.

Simple example

$x^{(0)} := 1$ and $\phi : \mathbb{R} \rightarrow \mathbb{R}$ is the function given by

$$(\forall t \in \mathbb{R}) \quad \phi(t) = 2t$$

Simple example

$x^{(0)} := 1$ and $\phi : \mathbb{R} \rightarrow \mathbb{R}$ is the function given by

$$(\forall t \in \mathbb{R}) \quad \phi(t) = 2t$$

$$x^{(0)} = 1$$

Simple example

$x^{(0)} := 1$ and $\phi : \mathbb{R} \rightarrow \mathbb{R}$ is the function given by

$$(\forall t \in \mathbb{R}) \quad \phi(t) = 2t$$

$$x^{(0)} = 1$$

$$x^{(1)} = 2 \cdot x^{(0)} = 2 \cdot 1 = 2$$

Simple example

$x^{(0)} := 1$ and $\phi : \mathbb{R} \rightarrow \mathbb{R}$ is the function given by

$$(\forall t \in \mathbb{R}) \quad \phi(t) = 2t$$

$$x^{(0)} = 1$$

$$x^{(1)} = 2 \cdot x^{(0)} = 2 \cdot 1 = 2$$

$$x^{(2)} = 2 \cdot x^{(1)} = 2 \cdot 2 = 4$$

Simple example

$x^{(0)} := 1$ and $\phi : \mathbb{R} \rightarrow \mathbb{R}$ is the function given by

$$(\forall t \in \mathbb{R}) \quad \phi(t) = 2t$$

$$x^{(0)} = 1$$

$$x^{(1)} = 2 \cdot x^{(0)} = 2 \cdot 1 = 2$$

$$x^{(2)} = 2 \cdot x^{(1)} = 2 \cdot 2 = 4$$

$$x^{(3)} = 2 \cdot x^{(2)} = 2 \cdot 4 = 8$$

Simple example

$x^{(0)} := 1$ and $\phi : \mathbb{R} \rightarrow \mathbb{R}$ is the function given by

$$(\forall t \in \mathbb{R}) \quad \phi(t) = 2t$$

$$x^{(0)} = 1$$

$$x^{(1)} = 2 \cdot x^{(0)} = 2 \cdot 1 = 2$$

$$x^{(2)} = 2 \cdot x^{(1)} = 2 \cdot 2 = 4$$

$$x^{(3)} = 2 \cdot x^{(2)} = 2 \cdot 4 = 8$$

$$x^{(4)} = 2 \cdot x^{(3)} = 2 \cdot 8 = 16$$

Simple example

$x^{(0)} := 1$ and $\phi : \mathbb{R} \rightarrow \mathbb{R}$ is the function given by

$$(\forall t \in \mathbb{R}) \quad \phi(t) = 2t$$

$$x^{(0)} = 1$$

$$x^{(1)} = 2 \cdot x^{(0)} = 2 \cdot 1 = 2$$

$$x^{(2)} = 2 \cdot x^{(1)} = 2 \cdot 2 = 4$$

$$x^{(3)} = 2 \cdot x^{(2)} = 2 \cdot 4 = 8$$

$$x^{(4)} = 2 \cdot x^{(3)} = 2 \cdot 8 = 16$$

$$\vdots$$

$$x^{(k)} = 2^k$$

k	$x^{(k)}$
0	1
1	2
2	4
3	8
4	16
\vdots	\vdots
k	2^k

Less simple example

- Given $a > 0$ and $x^{(0)} > 0$, consider sequence

$$x^{(k+1)} = \phi(x^{(k)}) = \frac{1}{2} \left(x^{(k)} + \frac{a}{x^{(k)}} \right) \quad (k = 0, 1, \dots)$$

k	$x^{(k)}$
0	3.000000000000000
1	
2	
3	
4	
5	

This sequence converges to $\sqrt{5}$!

Less simple example

- Given $a > 0$ and $x^{(0)} > 0$, consider sequence

$$x^{(k+1)} = \phi(x^{(k)}) = \frac{1}{2} \left(x^{(k)} + \frac{a}{x^{(k)}} \right) \quad (k = 0, 1, \dots)$$

- e.g., with $a = 5$ and $x^{(0)} = 3$, we get

k	$x^{(k)}$
0	3.000000000000000
1	
2	
3	
4	
5	

This sequence converges to $\sqrt{5}$!

Less simple example

- Given $a > 0$ and $x^{(0)} > 0$, consider sequence

$$x^{(k+1)} = \phi(x^{(k)}) = \frac{1}{2} \left(x^{(k)} + \frac{a}{x^{(k)}} \right) \quad (k = 0, 1, \dots)$$

- e.g., with $a = 5$ and $x^{(0)} = 3$, we get

k	$x^{(k)}$
0	3.000000000000000
1	2.333333333333333
2	
3	
4	
5	

This sequence converges to $\sqrt{5}$!

Less simple example

- Given $a > 0$ and $x^{(0)} > 0$, consider sequence

$$x^{(k+1)} = \phi(x^{(k)}) = \frac{1}{2} \left(x^{(k)} + \frac{a}{x^{(k)}} \right) \quad (k = 0, 1, \dots)$$

- e.g., with $a = 5$ and $x^{(0)} = 3$, we get

k	$x^{(k)}$
0	3.000000000000000
1	2.333333333333333
2	2.23809523809524
3	
4	
5	

This sequence converges to $\sqrt{5}$!

Less simple example

- Given $a > 0$ and $x^{(0)} > 0$, consider sequence

$$x^{(k+1)} = \phi(x^{(k)}) = \frac{1}{2} \left(x^{(k)} + \frac{a}{x^{(k)}} \right) \quad (k = 0, 1, \dots)$$

- e.g., with $a = 5$ and $x^{(0)} = 3$, we get

k	$x^{(k)}$
0	3.000000000000000
1	2.333333333333333
2	2.23809523809524
3	2.23606889564336
4	
5	

This sequence converges to $\sqrt{5}$!

Less simple example

- Given $a > 0$ and $x^{(0)} > 0$, consider sequence

$$x^{(k+1)} = \phi(x^{(k)}) = \frac{1}{2} \left(x^{(k)} + \frac{a}{x^{(k)}} \right) \quad (k = 0, 1, \dots)$$

- e.g., with $a = 5$ and $x^{(0)} = 3$, we get

k	$x^{(k)}$
0	3.000000000000000
1	2.333333333333333
2	2.23809523809524
3	2.23606889564336
4	2.23606797749998
5	

This sequence converges to $\sqrt{5}$!

Less simple example

- Given $a > 0$ and $x^{(0)} > 0$, consider sequence

$$x^{(k+1)} = \phi(x^{(k)}) = \frac{1}{2} \left(x^{(k)} + \frac{a}{x^{(k)}} \right) \quad (k = 0, 1, \dots)$$

- e.g., with $a = 5$ and $x^{(0)} = 3$, we get

k	$x^{(k)}$
0	3.000000000000000
1	2.333333333333333
2	2.23809523809524
3	2.23606889564336
4	2.23606797749998
5	2.23606797749979

This sequence converges to $\sqrt{5}$!

- ▶ Circa 1800-1600 BCE
- ▶ Numbers on diagonal



$$1;24;51;10 = 1 + \frac{24}{60^1} + \frac{51}{60^2} + \frac{10}{60^3}$$
$$\simeq \sqrt{2} \text{ to 9 decimal places!}$$

- ▶ Tablet suggests Babylonians knew Pythagoras's theorem

$$a^2 + b^2 = c^2$$

relating sides of right-angled triangle

Convergence of iterations

- ▶ Any iteration $x^{(k+1)} = \phi(x^{(k)})$ generates a **sequence**

$$\left\{x^{(k)}\right\}_{k=0}^{\infty} = \left\{x^{(0)}, x^{(1)}, x^{(2)}, \dots, x^{(k-1)}, x^{(k)}, x^{(k+1)}, \dots\right\}$$

- ▶ Recall: **limit of sequence** $\left\{x^{(k)}\right\}_{k=0}^{\infty}$ is $x^* \in \mathbb{R}$ iff

$$(\forall \epsilon > 0)(\exists K \in \mathbb{N}) \quad \left[(k \geq K) \Rightarrow |x^{(k)} - x^*| < \epsilon \right]$$

- ▶ In words, **the sequence** $\left\{x^{(k)}\right\}_{k=0}^{\infty}$ **converges to** $x^* \in \mathbb{R}$ or

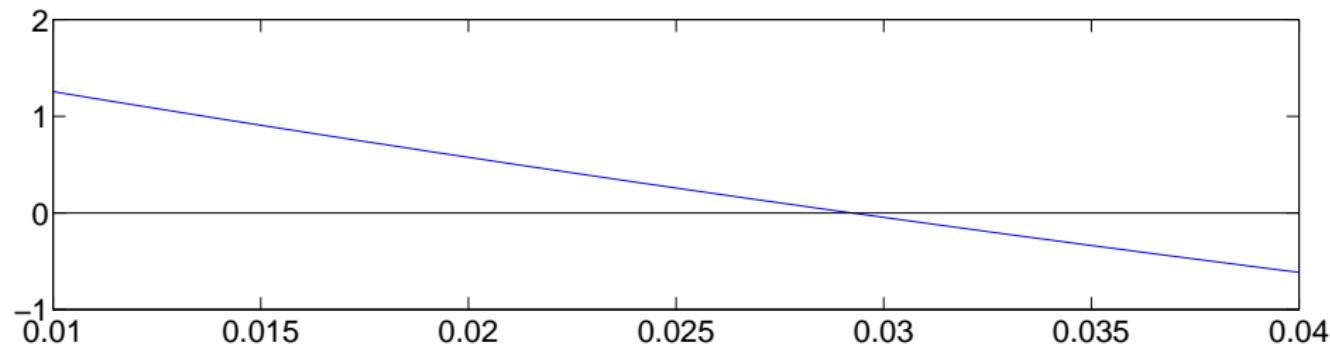
$$\boxed{\lim_{k \rightarrow \infty} x^{(k)} = x^*}$$

Bisection

Suppose we have a continuous function f on some domain $[a, b]$, and that there is a **unique** solution

$$f(x^*) = 0, \quad x^* \in [a, b]$$

then $f(a)f(b) < 0$. Example:



Or, in *pseudo-code*:

Input: f , a , b , k_{\max} , ϵ_x , ϵ_f .

1. Set $a_0 = a$, $b_0 = b$.
2. Do for $k = 1, \dots, k_{\max}$
 - ▶ Let $c_k = (a_{k-1} + b_{k-1})/2$ and $f_k = f(c_k)$.
 - ▶ If $f_k f(a_{k-1}) > 0$ then let $a_k = c_k$ and $b_k = b_{k-1}$
else let $a_k = a_{k-1}$ and $b_k = c_k$.
 - ▶ If $|b_k - a_k| < \epsilon_x$ or $|f(c_k)| < \epsilon_f$ break.
3. Return $x^* = c_k$ and $|b_k - a_k|$.

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

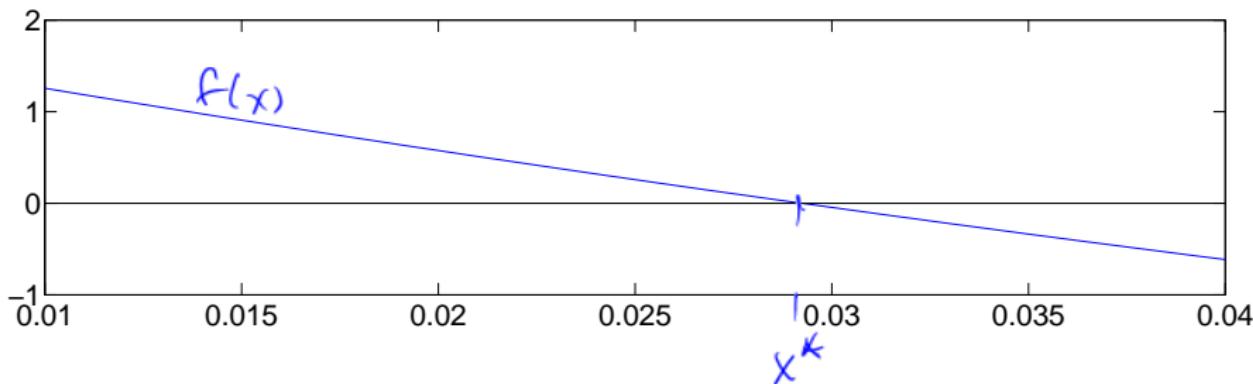
1. First method: bisection
2. Bisection
3. Newton-Raphson iteration
4. Comparing the two
5. Secant method
6. Recursion

The problem

(a.k.a root finding or solving an equation in one variable)

Suppose we have a continuous function f on some domain $[a, b]$. Find $x^* \in [a, b]$ such that

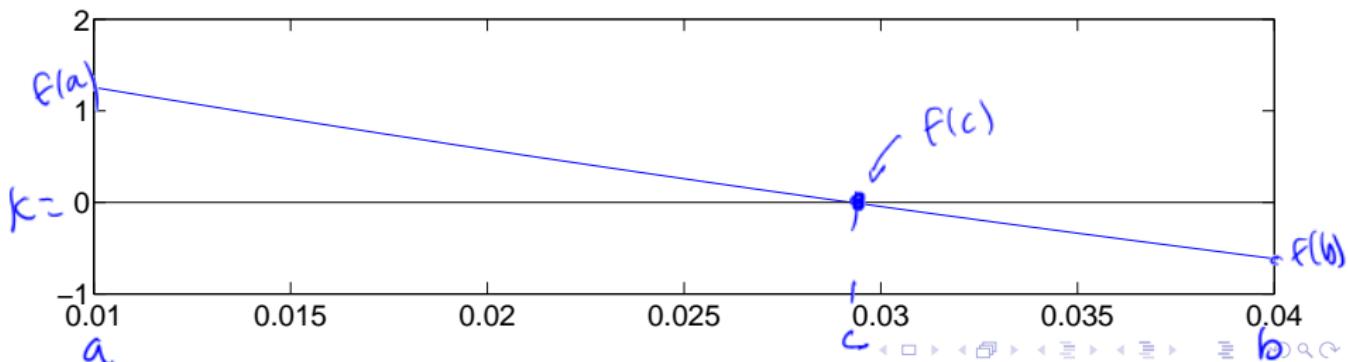
$$f(x^*) = 0.$$



Intermediate Value Theorem

Suppose we have a continuous function f on some domain $[a, b]$. Then if k is some number between $f(a)$ and $f(b)$ then there exists at least one number c in the interval $[a, b]$ such that $f(c) = k$. That is,

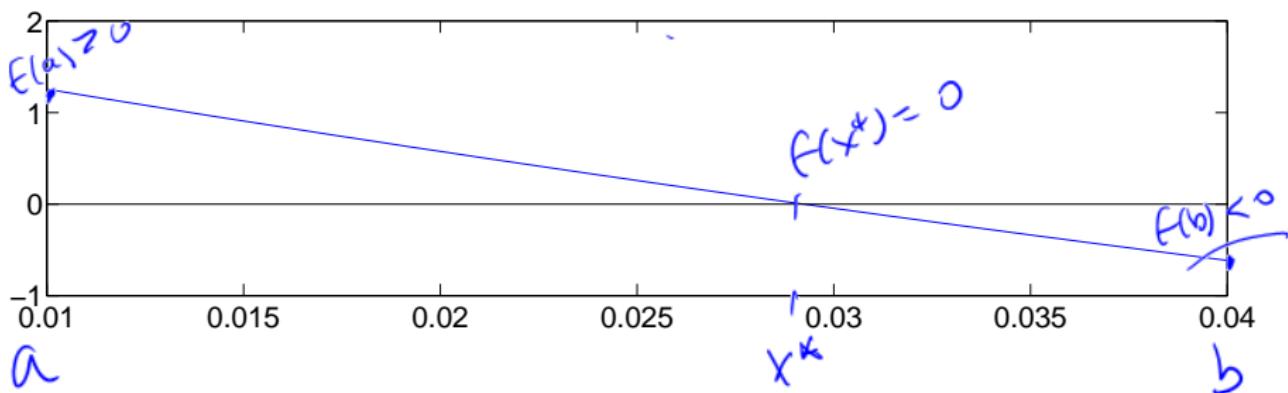
$$\underbrace{f(a) < k < f(b)}_{\text{or}} \quad \underbrace{f(a) > k > f(b)},$$
$$\rightarrow \exists c \in [a, b] \text{ s.t. } f(c) = k$$



Intermediate Value Theorem for $k = 0$

Suppose we have a continuous function f on some domain $[a, b]$. Then if $f(a)f(b) < 0$ then there exists at least one number c in the interval $[a, b]$ such that $f(c) = 0$. That is,

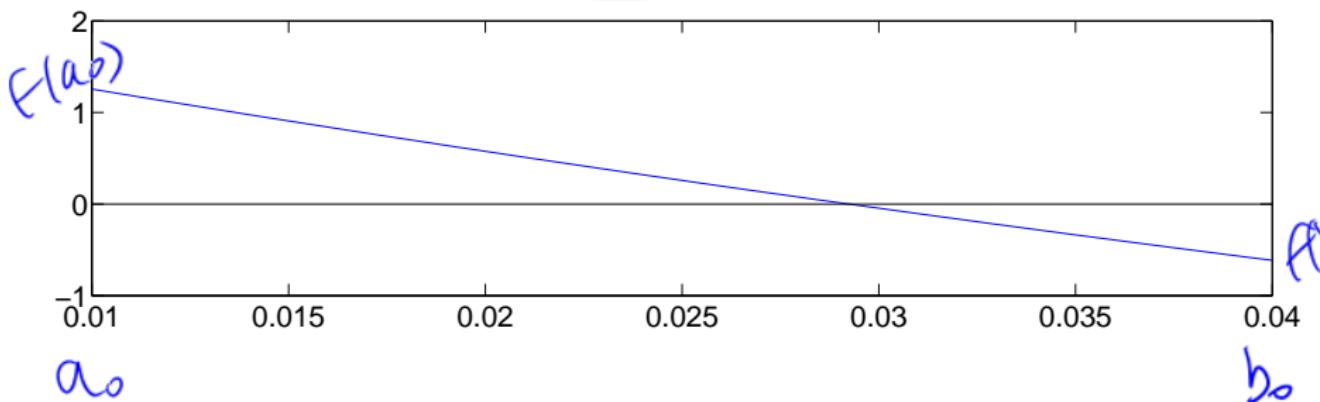
$$f(a)f(b) < 0 \rightarrow \exists c \in [a, b] \text{ s.t. } f(c) = 0$$



Bisection

Suppose we have a continuous function f on some domain $[a, b]$. Find a, b such that $f(a)f(b) < 0$, then by the Intermediate Value Theorem, there exists at least one solution $x^* \in [a, b]$ to the equation

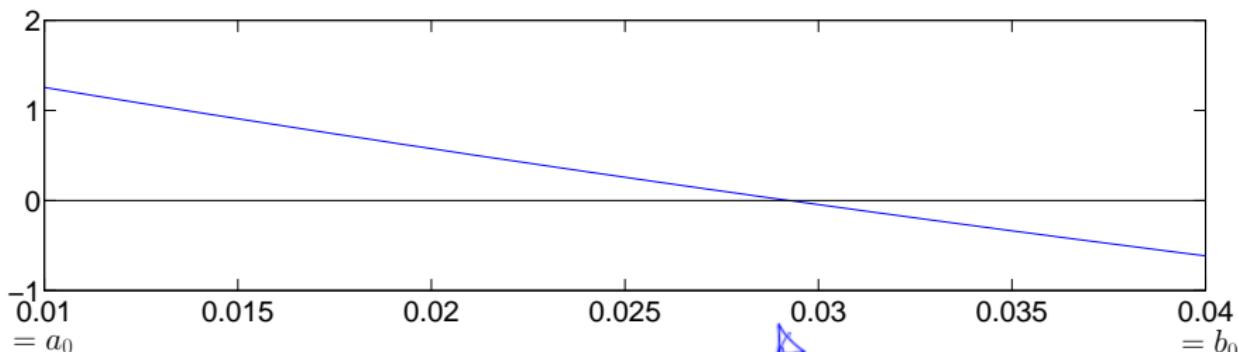
$$f(x^*) = 0.$$



Bisection in *pseudo-code*:

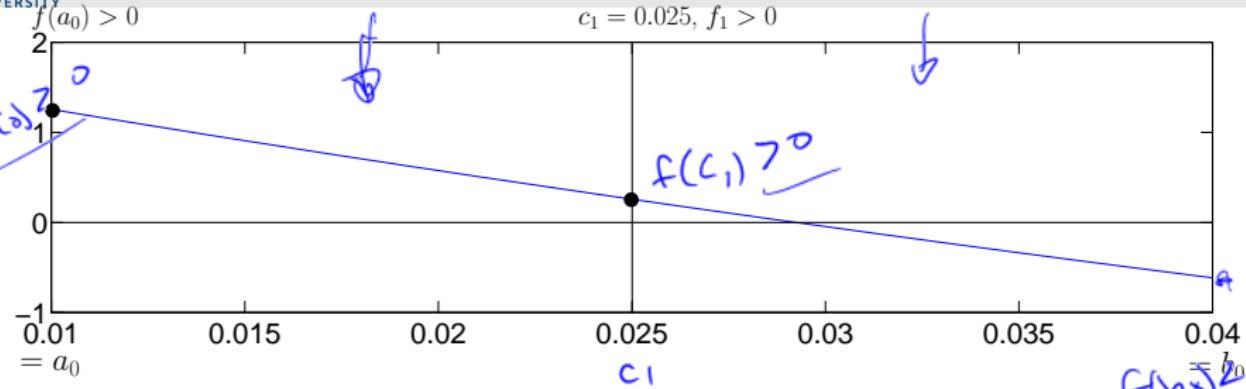
Input: f , a , b , k_{\max} , ϵ_x , ϵ_f .

1. Set $a_0 = a$, $b_0 = b$.
2. Do for $k = 1, \dots, k_{\max}$
 - ▶ Let $c_k = (a_{k-1} + b_{k-1})/2$ and $f_k = f(c_k)$.
 - ▶ If $f_k f(a_{k-1}) > 0$ then let $a_k = c_k$ and $b_k = b_{k-1}$
else let $a_k = a_{k-1}$ and $b_k = c_k$.
 - ▶ If $|b_k - a_k| < \epsilon_x$ (and/or $|f(c_k)| < \epsilon_f$) break.
3. Return $x^* = c_k$ and maximum error $|b_k - a_k|$.



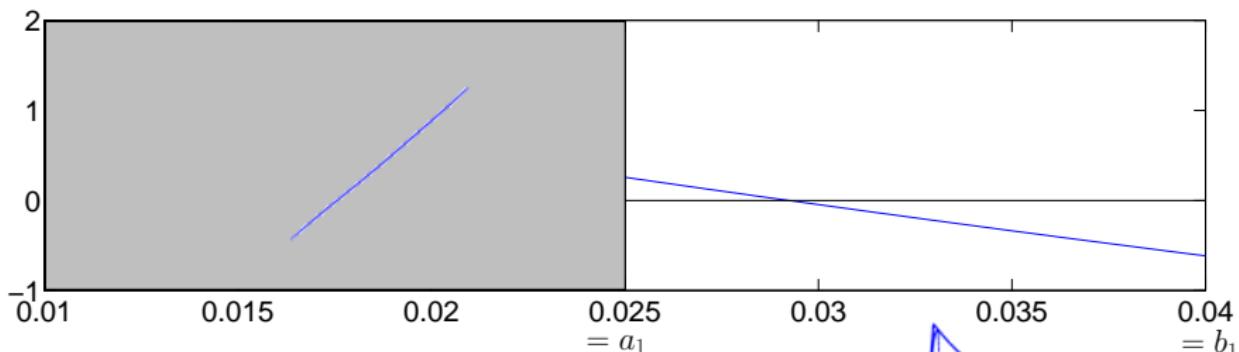
Input: f , a , b , k_{\max} , ϵ_x , ϵ_f .

1. Set $a_0 = a$, $b_0 = b$.
 2. Do for $k = 1, \dots, k_{\max}$
 - ▶ Let $c_k = (a_{k-1} + b_{k-1})/2$ and $f_k = f(c_k)$.
 - ▶ If $f_k f(a_{k-1}) > 0$ then let $a_k = c_k$ and $b_k = b_{k-1}$
else let $a_k = a_{k-1}$ and $b_k = c_k$.
 - ▶ If $|b_k - a_k| < \epsilon_x$ (or $|f(c_k)| < \epsilon_f$) break.
 3. Return $x^* = c_k$ and maximum error $|b_k - a_k|$.
- For $\epsilon_x = 0.001$, $\epsilon_f = 10^{-10}$,
- $x^* \approx 0.0296875$ with maximum error 9.375×10^{-4} .



Input: f , a , b , k_{\max} , ϵ_x , ϵ_f .

1. Set $a_0 = a$, $b_0 = b$.
 2. Do for $k = 1, \dots, k_{\max}$
 - ▶ Let $c_k = (a_{k-1} + b_{k-1})/2$ and $f_k = f(c_k)$. (1)
 - ▶ If $f_k f(a_{k-1}) > 0$ then let $a_k = c_k$ and $b_k = b_{k-1}$ (2)
else let $a_k = a_{k-1}$ and $b_k = c_k$.
 - ▶ If $|b_k - a_k| < \epsilon_x$ (or $|f(c_k)| < \epsilon_f$) break.
 3. Return $x^* = c_k$ and maximum error $|b_k - a_k|$.
- For $\epsilon_x = 0.001$, $\epsilon_f = 10^{-10}$,
- $x^* \approx 0.0296875$ with maximum error 9.375×10^{-4} .



Input: f , a , b , k_{\max} , ϵ_x , ϵ_f .

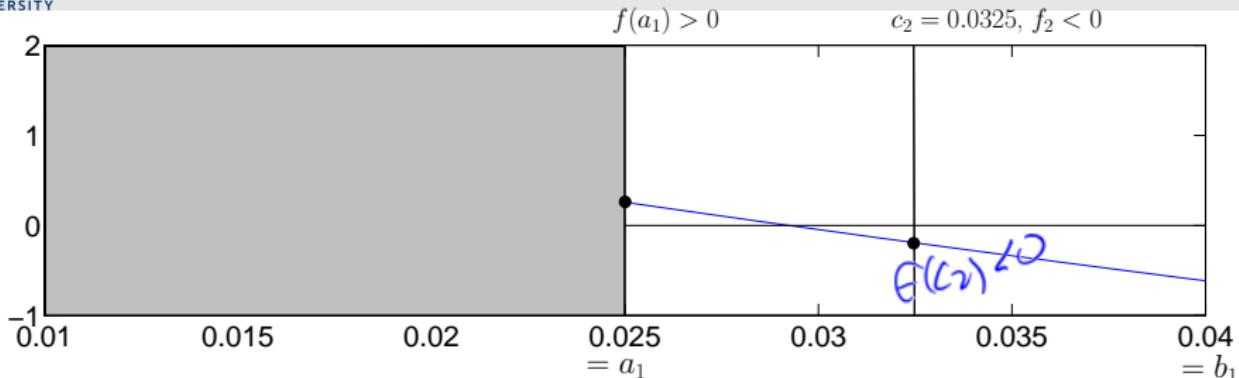
1. Set $a_0 = a$, $b_0 = b$.
2. Do for $k = 1, \dots, k_{\max}$

- ▶ Let $c_k = (a_{k-1} + b_{k-1})/2$ and $f_k = f(c_k)$.
- ▶ If $f_k f(a_{k-1}) > 0$ then let $a_k = c_k$ and $b_k = b_{k-1}$ ↗
- else let $a_k = a_{k-1}$ and $b_k = c_k$. ↙
- ▶ If $|b_k - a_k| < \epsilon_x$ (or $|f(c_k)| < \epsilon_f$) break. ↙

3. Return $x^* = c_k$ and maximum error $|b_k - a_k|$.

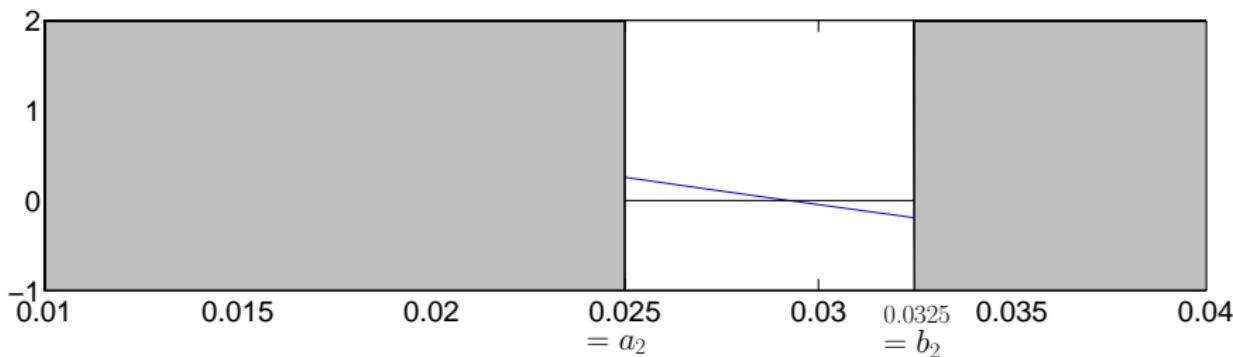
For $\epsilon_x = 0.001$, $\epsilon_f = 10^{-10}$,

$x^* \approx 0.0296875$ with maximum error 9.375×10^{-4} .



Input: f , a , b , k_{\max} , ϵ_x , ϵ_f .

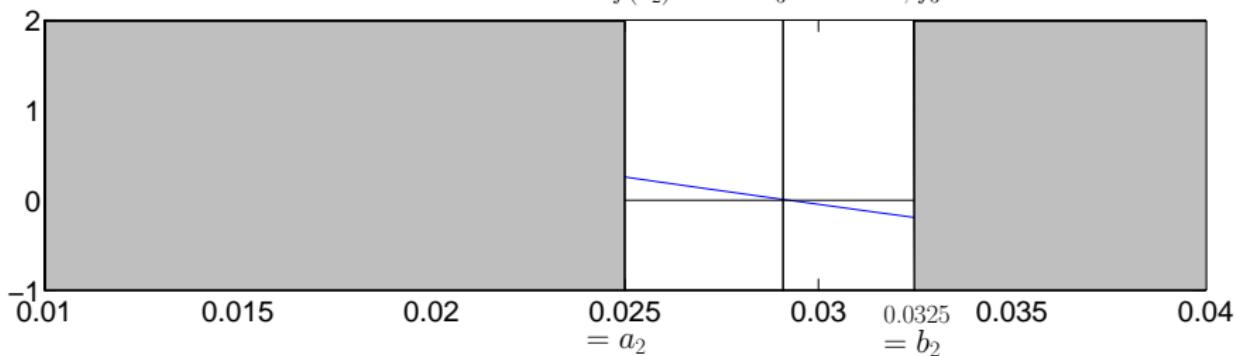
1. Set $a_0 = a$, $b_0 = b$.
 2. Do for $k = 1, \dots, k_{\max}$
 - ▶ Let $c_k = (a_{k-1} + b_{k-1})/2$ and $f_k = f(c_k)$.
 - ▶ If $f_k f(a_{k-1}) > 0$ then let $a_k = c_k$ and $b_k = b_{k-1}$
else let $a_k = a_{k-1}$ and $b_k = c_k$.
 - ▶ If $|b_k - a_k| < \epsilon_x$ (or $|f(c_k)| < \epsilon_f$) break.
 3. Return $x^* = c_k$ and maximum error $|b_k - a_k|$.
- For $\epsilon_x = 0.001$, $\epsilon_f = 10^{-10}$,
- $x^* \approx 0.0296875$ with maximum error 9.375×10^{-4} .



Input: f , a , b , k_{\max} , ϵ_x , ϵ_f .

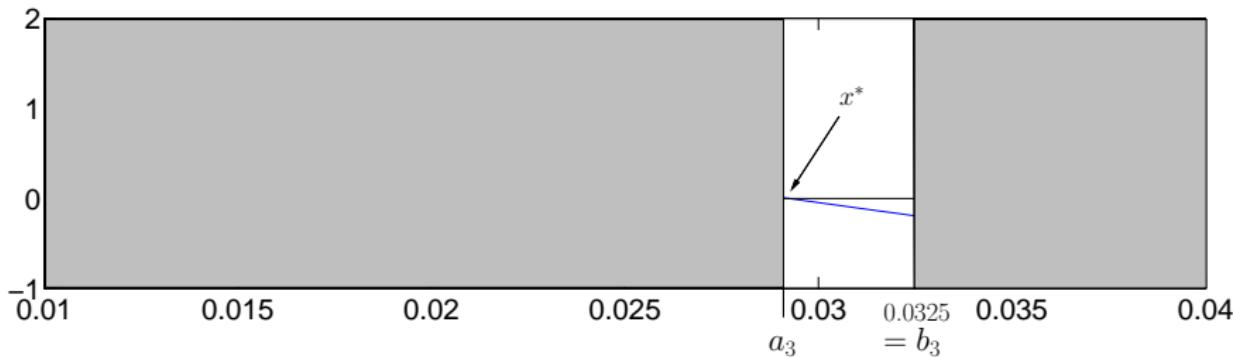
1. Set $a_0 = a$, $b_0 = b$.
 2. Do for $k = 1, \dots, k_{\max}$
 - ▶ Let $c_k = (a_{k-1} + b_{k-1})/2$ and $f_k = f(c_k)$.
 - ▶ If $f_k f(a_{k-1}) > 0$ then let $a_k = c_k$ and $b_k = b_{k-1}$
else let $a_k = a_{k-1}$ and $b_k = c_k$.
 - ▶ If $|b_k - a_k| < \epsilon_x$ (or $|f(c_k)| < \epsilon_f$) break.
 3. Return $x^* = c_k$ and maximum error $|b_k - a_k|$.
- For $\epsilon_x = 0.001$, $\epsilon_f = 10^{-10}$,
- $x^* \approx 0.0296875$ with maximum error 9.375×10^{-4} .

$$f(a_2) > 0 \quad c_3 = 0.02875, f_3 > 0$$



Input: f , a , b , k_{\max} , ϵ_x , ϵ_f .

1. Set $a_0 = a$, $b_0 = b$.
 2. Do for $k = 1, \dots, k_{\max}$
 - ▶ Let $c_k = (a_{k-1} + b_{k-1})/2$ and $f_k = f(c_k)$.
 - ▶ If $f_k f(a_{k-1}) > 0$ then let $a_k = c_k$ and $b_k = b_{k-1}$
else let $a_k = a_{k-1}$ and $b_k = c_k$.
 - ▶ If $|b_k - a_k| < \epsilon_x$ (or $|f(c_k)| < \epsilon_f$) break.
 3. Return $x^* = c_k$ and maximum error $|b_k - a_k|$.
- For $\epsilon_x = 0.001$, $\epsilon_f = 10^{-10}$,
- $x^* \approx 0.0296875$ with maximum error 9.375×10^{-4} .

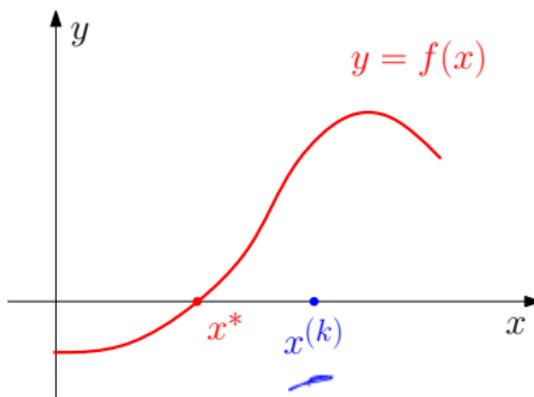


Input: f , a , b , k_{\max} , ϵ_x , ϵ_f .

1. Set $a_0 = a$, $b_0 = b$.
 2. Do for $k = 1, \dots, k_{\max}$
 - ▶ Let $c_k = (a_{k-1} + b_{k-1})/2$ and $f_k = f(c_k)$.
 - ▶ If $f_k f(a_{k-1}) > 0$ then let $a_k = c_k$ and $b_k = b_{k-1}$
else let $a_k = a_{k-1}$ and $b_k = c_k$.
 - ▶ If $|b_k - a_k| < \epsilon_x$ (or $|f(c_k)| < \epsilon_f$) break.
 3. Return $x^* = c_k$ and maximum error $|b_k - a_k|$.
- For $\epsilon_x = 0.001$, $\epsilon_f = 10^{-10}$,
- $x^* \approx 0.0296875$ with maximum error 9.375×10^{-4} .

Newton-Raphson method: derivation

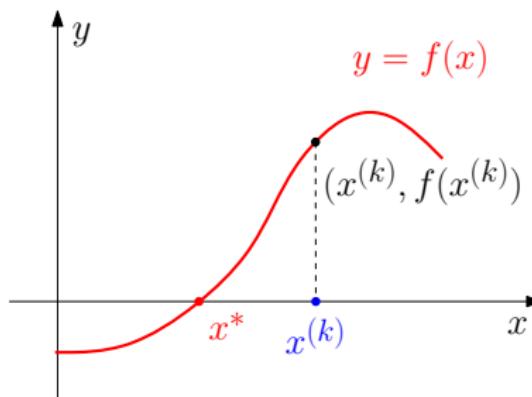
- ▶ Start with $x^{(k)}$ (intended to approximate x^* such that $f(x^*) = 0$)



Newton-Raphson method: derivation

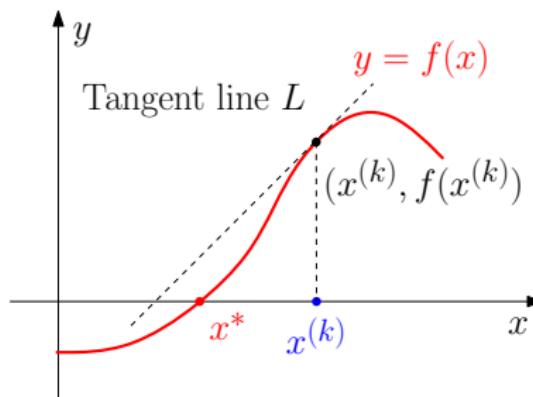
- ▶ Start with $x^{(k)}$ (intended to approximate x^* such that $f(x^*) = 0$)
- ▶ Evaluate f at $x^{(k)}$

$x^{(0)}, x^{(1)}$



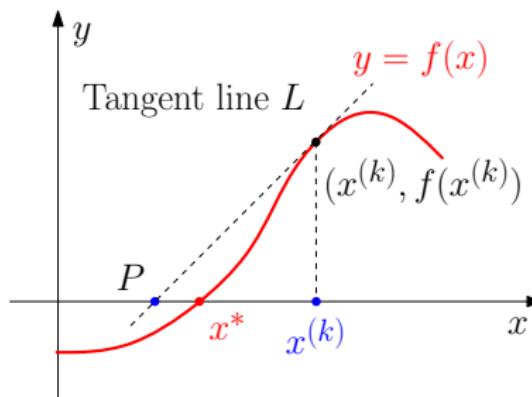
Newton-Raphson method: derivation

- ▶ Start with $x^{(k)}$ (intended to approximate x^* such that $f(x^*) = 0$)
- ▶ Evaluate f at $x^{(k)}$
- ▶ Extend **tangent** line L from $(x^{(k)}, f(x^{(k)}))$ (using $f'(x^{(k)})$)



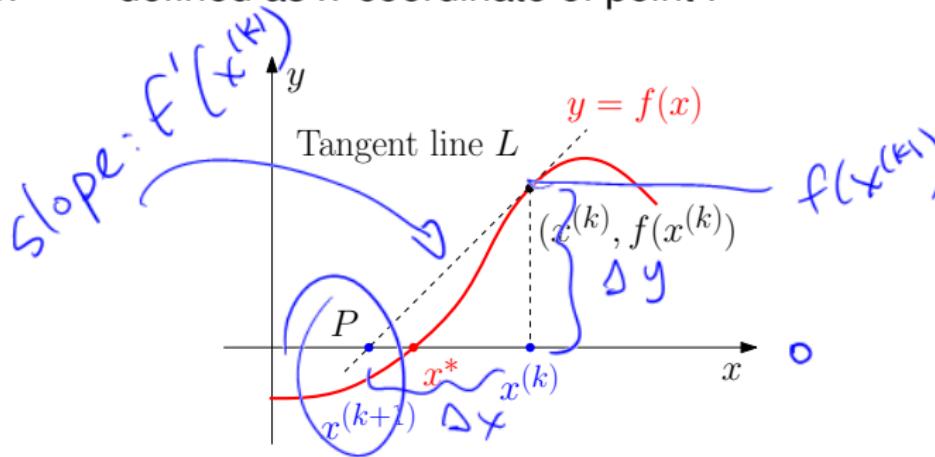
Newton-Raphson method: derivation

- ▶ Start with $x^{(k)}$ (intended to approximate x^* such that $f(x^*) = 0$)
- ▶ Evaluate f at $x^{(k)}$
- ▶ Extend **tangent** line L from $(x^{(k)}, f(x^{(k)}))$ (using $f'(x^{(k)})$)
- ▶ Follow L to P (where it cuts x -axis)



Newton-Raphson method: derivation

- ▶ Start with $x^{(k)}$ (intended to approximate x^* such that $f(x^*) = 0$)
- ▶ Evaluate f at $x^{(k)}$
- ▶ Extend **tangent** line L from $(x^{(k)}, f(x^{(k)}))$ (using $f'(x^{(k)})$)
- ▶ Follow L to P (where it cuts x -axis)
- ▶ $x^{(k+1)}$ defined as x -coordinate of point P



Newton-Raphson method: derivation

- Slope of L is $f'(x^{(k)})$ & x -intercept is $x^{(k+1)}$, so


$$f'(x^{(k)}) = \frac{f(x^{(k)}) - 0}{x^{(k)} - x^{(k+1)}}.$$

- Solving for $x^{(k+1)}$ gives the formula for Newton-Raphson method.


$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})} \quad (k \geq 0)$$

- Hopefully, $x^{(k+1)}$ closer to true zero x^* than $x^{(k)}$.

Newton-Raphson method

Given an iterate $x^{(k)}$ approximating a zero of f , the next iterate is

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})} \quad (k \geq 0)$$

- ▶ Iterative procedure to locate zeros of f .
- ▶ Requires initial iterate $x^{(0)}$ to start.
- ▶ Near true zero x^* of f , iteration converges quickly.
- ▶ Useful to define the Newton-Raphson step or *update*:

$$\delta^{(k)} = x^{(k+1)} - x^{(k)} = -\frac{f(x^{(k)})}{f'(x^{(k)})}$$

Algorithm for Newton-Raphson method

Input: $f, f', x^{(0)}$

for $k = 0, 1, 2, \dots$ **until** convergence

$$r^{(k)} \leftarrow f(x^{(k)}) \quad (\text{evaluate nonlinear residual})$$

$$\delta x^{(k)} \leftarrow -[f'(x^{(k)})]^{-1} r^{(k)} \quad (\text{compute Newton-Raphson step})$$

$$x^{(k+1)} \leftarrow x^{(k)} + \delta x^{(k)} \quad (\text{compute next iterate})$$

Test for convergence (break if necessary)

end for

Output: $x^{(k)}$

- ▶ Terminology: $r^{(k)} := f(x^{(k)})$ = **residual**
- ▶ Terminology: $\delta x^{(k)} := -[f'(x^{(k)})]^{-1} r^{(k)}$ = **update**

Example:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.75$ to find $x^{(3)}$ that approximates the real solution of $x = \cos x$.

- ▶ Define $f(x) = x - \cos x$, so $f'(x) = 1 + \sin x$

Example:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.75$ to find $x^{(3)}$ that approximates the real solution of $x = \cos x$.

- ▶ Define $f(x) = x - \cos x$, so $f'(x) = 1 + \sin x$
- ▶ Then, Newton's method is $x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$, or, in steps,

$$r^{(k)} = f(x^{(k)}) = x^{(k)} - \cos(x^{(k)}),$$

$$\delta x^{(k)} = \frac{-r^{(k)}}{f'(x^{(k)})} = \frac{-r^{(k)}}{1 + \sin(x^{(k)})},$$

$$x^{(k+1)} = x^{(k)} + \delta x^{(k)}$$

Example:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.75$ to find $x^{(3)}$ that approximates the real solution of $x = \cos x$.

- ▶ Define $f(x) = x - \cos x$, so $f'(x) = 1 + \sin x$
- ▶ Then, Newton's method is $x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$, or, in steps,

$$r^{(k)} = f(x^{(k)}) = x^{(k)} - \cos(x^{(k)}),$$

$$\delta x^{(k)} = \frac{-r^{(k)}}{f'(x^{(k)})} = \frac{-r^{(k)}}{1 + \sin(x^{(k)})},$$

$$x^{(k+1)} = x^{(k)} + \delta x^{(k)}$$

- ▶ Use formulas above 3 times starting from $k = 0$, $x^{(0)} = 0.75$ to obtain sequence $x^{(1)}, x^{(2)}, x^{(3)}$ (see following table)

Example 1:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.75$ to find $x^{(3)}$ that approximates the real solution of $x = \cos x$.

$$f(x) = x - \cos x$$

$$f'(x) = 1 + \sin x$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.75		

Example 1:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.75$ to find $x^{(3)}$ that approximates the real solution of $x = \cos X$.

$$f(x) = x - \cos x$$

$$f'(x) = 1 + \sin x$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.75	1.83111×10^{-2}	

$$x^{(0)} = 0.75,$$

$$r^{(0)} = f(x^{(0)}) = x^{(0)} - \cos(x^{(0)}) = 0.75 - \cos(0.75) = 1.83111 \times 10^{-2},$$

Example 1:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.75$ to find $x^{(3)}$ that approximates the real solution of $x = \cos X$.

$$f(x) = x - \cos x$$

$$f'(x) = 1 + \sin x$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.75	1.83111×10^{-2}	-1.08889×10^{-2}

$$x^{(0)} = 0.75,$$

$$r^{(0)} = f(x^{(0)}) = x^{(0)} - \cos(x^{(0)}) = 0.75 - \cos(0.75) = 1.83111 \times 10^{-2},$$

$$\delta x^{(0)} = -\frac{r^{(0)}}{f'(x^{(0)})} = -\frac{1.83111 \times 10^{-2}}{1 + \sin(0.75)} = -1.08889 \times 10^{-2},$$

Example 1:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.75$ to find $x^{(3)}$ that approximates the real solution of $x = \cos X$.

$$f(x) = x - \cos x$$

$$f'(x) = 1 + \sin x$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.75	1.83111×10^{-2}	-1.08889×10^{-2}
1	0.739111138752579		

$$x^{(0)} = 0.75,$$

$$r^{(0)} = f(x^{(0)}) = x^{(0)} - \cos(x^{(0)}) = 0.75 - \cos(0.75) = 1.83111 \times 10^{-2},$$

$$\delta x^{(0)} = -\frac{r^{(0)}}{f'(x^{(0)})} = -\frac{1.83111 \times 10^{-2}}{1 + \sin(0.75)} = -1.08889 \times 10^{-2},$$

$$x^{(1)} = x^{(0)} + \delta x^{(0)} = 0.75 + (-1.08889 \times 10^{-2}) = 0.739111138752579$$

Example 1:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.75$ to find $x^{(3)}$ that approximates the real solution of $x = \cos X$.

$$f(x) = x - \cos x$$

$$f'(x) = 1 + \sin x$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.75	1.83111×10^{-2}	-1.08889×10^{-2}
1	0.739111138752579	4.35234×10^{-5}	

$$x^{(1)} = 0.739111138752579,$$

$$r^{(1)} = f(x^{(1)}) = 0.739111 - \cos(0.739111) = 4.35234 \times 10^{-5},$$

Example 1:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.75$ to find $x^{(3)}$ that approximates the real solution of $x = \cos X$.

$$f(x) = x - \cos x$$

$$f'(x) = 1 + \sin x$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.75	1.83111×10^{-2}	-1.08889×10^{-2}
1	0.739111138752579	4.35234×10^{-5}	-2.60055×10^{-5}

$$x^{(1)} = 0.739111138752579,$$

$$r^{(1)} = f(x^{(1)}) = 0.739111 - \cos(0.739111) = 4.35234 \times 10^{-5},$$

$$\delta x^{(1)} = -\frac{r^{(1)}}{f'(x^{(1)})} = -\frac{4.35234 \times 10^{-5}}{1 + \sin(0.739111)} = -2.60055 \times 10^{-5},$$

Example 1:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.75$ to find $x^{(3)}$ that approximates the real solution of $x = \cos X$.

$$f(x) = x - \cos x$$

$$f'(x) = 1 + \sin x$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.75	1.83111×10^{-2}	-1.08889×10^{-2}
1	0.739111138752579	4.35234×10^{-5}	-2.60055×10^{-5}
2	0.739085133364485		

$$x^{(1)} = 0.739111138752579,$$

$$r^{(1)} = f(x^{(1)}) = 0.739111 - \cos(0.739111) = 4.35234 \times 10^{-5},$$

$$\delta x^{(1)} = -\frac{r^{(1)}}{f'(x^{(1)})} = -\frac{4.35234 \times 10^{-5}}{1 + \sin(0.739111)} = -2.60055 \times 10^{-5},$$

$$x^{(2)} = x^{(1)} + \delta x^{(1)} = 0.739111 + (-2.60055 \times 10^{-5}) = 0.73908513336448$$

Example 1:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.75$ to find $x^{(3)}$ that approximates the real solution of $x = \cos X$.

$$f(x) = x - \cos x$$

$$f'(x) = 1 + \sin x$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.75	1.83111×10^{-2}	-1.08889×10^{-2}
1	0.739111138752579	4.35234×10^{-5}	-2.60055×10^{-5}
2	0.739085133364485	2.49910×10^{-10}	

$$x^{(2)} = 0.739085133364485,$$

$$r^{(2)} = f(x^{(2)}) = 0.739085 - \cos(0.739085) = 2.49910 \times 10^{-10},$$

Example 1:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.75$ to find $x^{(3)}$ that approximates the real solution of $x = \cos X$.

$$f(x) = x - \cos x$$

$$f'(x) = 1 + \sin x$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.75	1.83111×10^{-2}	-1.08889×10^{-2}
1	0.739111138752579	4.35234×10^{-5}	-2.60055×10^{-5}
2	0.739085133364485	2.49910×10^{-10}	-1.49324×10^{-10}

$$x^{(2)} = 0.739085133364485,$$

$$r^{(2)} = f(x^{(2)}) = 0.739085 - \cos(0.739085) = 2.49910 \times 10^{-10},$$

$$\delta x^{(2)} = -\frac{r^{(2)}}{f'(x^{(2)})} = -\frac{2.49910 \times 10^{-10}}{1 + \sin(0.739085)} = -1.49324 \times 10^{-10},$$

Example 1:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.75$ to find $x^{(3)}$ that approximates the real solution of $x = \cos X$.

$$f(x) = x - \cos x$$

$$f'(x) = 1 + \sin x$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.75	1.83111×10^{-2}	-1.08889×10^{-2}
1	0.739111138752579	4.35234×10^{-5}	-2.60055×10^{-5}
2	0.739085133364485	2.49910×10^{-10}	-1.49324×10^{-10}
3	0.739085133215161	$< 10^{-14}$	$< 10^{-14}$

$$x^{(2)} = 0.739085133364485,$$

$$r^{(2)} = f(x^{(2)}) = 0.739085 - \cos(0.739085) = 2.49910 \times 10^{-10},$$

$$\delta x^{(2)} = -\frac{r^{(2)}}{f'(x^{(2)})} = -\frac{2.49910 \times 10^{-10}}{1 + \sin(0.739085)} = -1.49324 \times 10^{-10},$$

$$x^{(3)} = x^{(2)} + \delta x^{(2)} = 0.739085 + (-1.49324 \times 10^{-10}) = 0.739085133215161$$

Example 2:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.5$ to find $x^{(3)}$ that approximates a zero of the equation $xe^x = 2$.

- ▶ Define $g(x) = x \exp(x) - 2$, so $g'(x) = (x + 1) \exp(x)$

Example 2:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.5$ to find $x^{(3)}$ that approximates a zero of the equation $xe^x = 2$.

- ▶ Define $g(x) = x \exp(x) - 2$, so $g'(x) = (x + 1) \exp(x)$
- ▶ Then, Newton-Raphson method is $x^{(k+1)} = x^{(k)} - \frac{g(x^{(k)})}{g'(x^{(k)})}$, or, in steps,

$$r^{(k)} = g(x^{(k)}) = x^{(k)} \exp(x^{(k)}) - 2,$$

$$\delta x^{(k)} = \frac{-r^{(k)}}{g'(x^{(k)})} = \frac{-r^{(k)}}{(x^{(k)} + 1) \exp(x^{(k)})},$$

$$x^{(k+1)} = x^{(k)} + \delta x^{(k)}$$

Example 2:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.5$ to find $x^{(3)}$ that approximates a zero of the equation $xe^x = 2$.

- ▶ Define $g(x) = x \exp(x) - 2$, so $g'(x) = (x + 1) \exp(x)$
- ▶ Then, Newton-Raphson method is $x^{(k+1)} = x^{(k)} - \frac{g(x^{(k)})}{g'(x^{(k)})}$, or, in steps,

$$r^{(k)} = g(x^{(k)}) = x^{(k)} \exp(x^{(k)}) - 2,$$

$$\delta x^{(k)} = \frac{-r^{(k)}}{g'(x^{(k)})} = \frac{-r^{(k)}}{(x^{(k)} + 1) \exp(x^{(k)})},$$

$$x^{(k+1)} = x^{(k)} + \delta x^{(k)}$$

- ▶ Use formulas above 3 times starting from $k = 0$, $x^{(0)} = 0.5$ to obtain sequence $x^{(1)}, x^{(2)}, x^{(3)}$ (see following table)

Example 2:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.5$ to find $x^{(3)}$ that approximates a zero of the equation $xe^x = 2$.

$$g(x) = x \exp(x) - 2 \quad g'(x) = (x + 1) \exp(x)$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.5		

Example 2:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.5$ to find $x^{(3)}$ that approximates a zero of the equation $xe^x = 2$.

$$g(x) = x \exp(x) - 2 \quad g'(x) = (x + 1) \exp(x)$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.5	-1.17564	

$$x^{(0)} = 0.5,$$

$$r^{(0)} = g(x^{(0)}) = x^{(0)} \exp(x^{(0)}) - 2 = 0.5 \exp(0.5) - 2 = -1.17564,$$

Example 2:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.5$ to find $x^{(3)}$ that approximates a zero of the equation $xe^x = 2$.

$$g(x) = x \exp(x) - 2 \quad g'(x) = (x + 1) \exp(x)$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.5	-1.17564	0.475374

$$x^{(0)} = 0.5,$$

$$r^{(0)} = g(x^{(0)}) = x^{(0)} \exp(x^{(0)}) - 2 = 0.5 \exp(0.5) - 2 = -1.17564,$$

$$\delta x^{(0)} = -\frac{r^{(0)}}{g'(x^{(0)})} = -\frac{(-1.17564)}{(0.5 + 1) \exp(0.5)} = 0.475374,$$

Example 2:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.5$ to find $x^{(3)}$ that approximates a zero of the equation $xe^x = 2$.

$$g(x) = x \exp(x) - 2 \quad g'(x) = (x + 1) \exp(x)$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.5	-1.17564	0.475374
1	0.975374212950178		

$$x^{(0)} = 0.5,$$

$$r^{(0)} = g(x^{(0)}) = x^{(0)} \exp(x^{(0)}) - 2 = 0.5 \exp(0.5) - 2 = -1.17564,$$

$$\delta x^{(0)} = -\frac{r^{(0)}}{g'(x^{(0)})} = -\frac{(-1.17564)}{(0.5 + 1) \exp(0.5)} = 0.475374,$$

$$x^{(1)} = x^{(0)} + \delta x^{(0)} = 0.5 + (0.475374) = 0.975374212950178$$

Example 2:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.5$ to find $x^{(3)}$ that approximates a zero of the equation $xe^x = 2$.

$$g(x) = x \exp(x) - 2 \quad g'(x) = (x + 1) \exp(x)$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.5	-1.17564	0.475374
1	0.975374212950178	0.586848	

$$x^{(1)} = 0.975374212950178,$$

$$r^{(1)} = g(x^{(1)}) = 0.975374 \exp(0.975374) - 2 = 0.586848,$$

Example 2:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.5$ to find $x^{(3)}$ that approximates a zero of the equation $xe^x = 2$.

$$g(x) = x \exp(x) - 2 \quad g'(x) = (x + 1) \exp(x)$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.5	-1.17564	0.475374
1	0.975374212950178	0.586848	-0.112015

$$x^{(1)} = 0.975374212950178,$$

$$r^{(1)} = g(x^{(1)}) = 0.975374 \exp(0.975374) - 2 = 0.586848,$$

$$\delta x^{(1)} = -\frac{r^{(1)}}{g'(x^{(1)})} = -\frac{0.586848}{(0.975374 + 1) \exp(0.975374)} = -0.112015,$$

Example 2:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.5$ to find $x^{(3)}$ that approximates a zero of the equation $xe^x = 2$.

$$g(x) = x \exp(x) - 2 \quad g'(x) = (x + 1) \exp(x)$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.5	-1.17564	0.475374
1	0.975374212950178	0.586848	-0.112015
2	0.863359106097814		

$$x^{(1)} = 0.975374212950178,$$

$$r^{(1)} = g(x^{(1)}) = 0.975374 \exp(0.975374) - 2 = 0.586848,$$

$$\delta x^{(1)} = -\frac{r^{(1)}}{g'(x^{(1)})} = -\frac{0.586848}{(0.975374 + 1) \exp(0.975374)} = -0.112015,$$

$$x^{(2)} = x^{(1)} + \delta x^{(1)} = 0.975374212950178 + -0.112015 = 0.863359106097814$$

Example 2:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.5$ to find $x^{(3)}$ that approximates a zero of the equation $xe^x = 2$.

$$g(x) = x \exp(x) - 2 \quad g'(x) = (x + 1) \exp(x)$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.5	-1.17564	0.475374
1	0.975374212950178	0.586848	-0.112015
2	0.863359106097814	4.71213×10^{-2}	

$$x^{(2)} = 0.863359106097814,$$

$$r^{(2)} = g(x^{(2)}) = 0.863359 \exp(0.863359) - 2 = 4.71213 \times 10^{-2},$$

Example 2:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.5$ to find $x^{(3)}$ that approximates a zero of the equation $xe^x = 2$.

$$g(x) = x \exp(x) - 2 \quad g'(x) = (x + 1) \exp(x)$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.5	-1.17564	0.475374
1	0.975374212950178	0.586848	-0.112015
2	0.863359106097814	4.71213×10^{-2}	-1.06652×10^{-2}

$$x^{(2)} = 0.863359106097814,$$

$$r^{(2)} = g(x^{(2)}) = 0.863359 \exp(0.863359) - 2 = 4.71213 \times 10^{-2},$$

$$\delta x^{(2)} = -\frac{r^{(2)}}{g'(x^{(2)})} = -\frac{4.71213 \times 10^{-2}}{(0.863359 + 1) \exp(0.863359)} = -1.06652 \times 10^{-2},$$

Example 2:

Carry out Newton-Raphson method starting from $x^{(0)} = 0.5$ to find $x^{(3)}$ that approximates a zero of the equation $xe^x = 2$.

$$g(x) = x \exp(x) - 2 \quad g'(x) = (x + 1) \exp(x)$$

k	$x^{(k)}$	$r^{(k)}$	$\delta x^{(k)}$
0	0.5	-1.17564	0.475374
1	0.975374212950178	0.586848	-0.112015
2	0.863359106097814	4.71213×10^{-2}	-1.06652×10^{-2}
3	0.852693923733206	3.84285×10^{-4}	—

$$x^{(2)} = 0.863359106097814,$$

$$r^{(2)} = g(x^{(2)}) = 0.863359 \exp(0.863359) - 2 = 4.71213 \times 10^{-2},$$

$$\delta x^{(2)} = -\frac{r^{(2)}}{g'(x^{(2)})} = -\frac{4.71213 \times 10^{-2}}{(0.863359 + 1) \exp(0.863359)} = -1.06652 \times 10^{-2},$$

$$x^{(3)} = x^{(2)} + \delta x^{(2)} = 0.863359106097814 + (-1.06652 \times 10^{-2}) = 0.852693923733206$$

Now we have to answer the three essential questions:

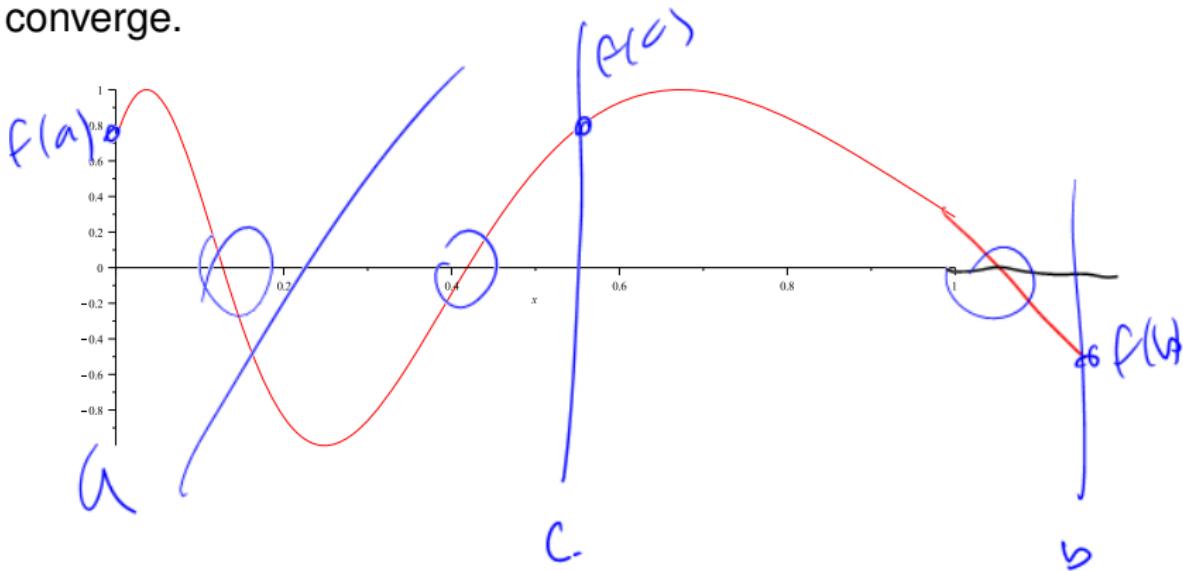
1. Under what conditions does the algorithm converge?
2. How accurate will the result be?
3. How fast does it converge?

Since bisection and Newton-Raphson iterations serve the same purpose (find x^* such that $f(x^*) = 0$) we can compare them...

1. Under what conditions does the algorithm converge?

Bisection converges to some x^* such that $f(x^*) = 0$ in $[a, b]$, if f is continuous and $f(a)f(b) < 0$.

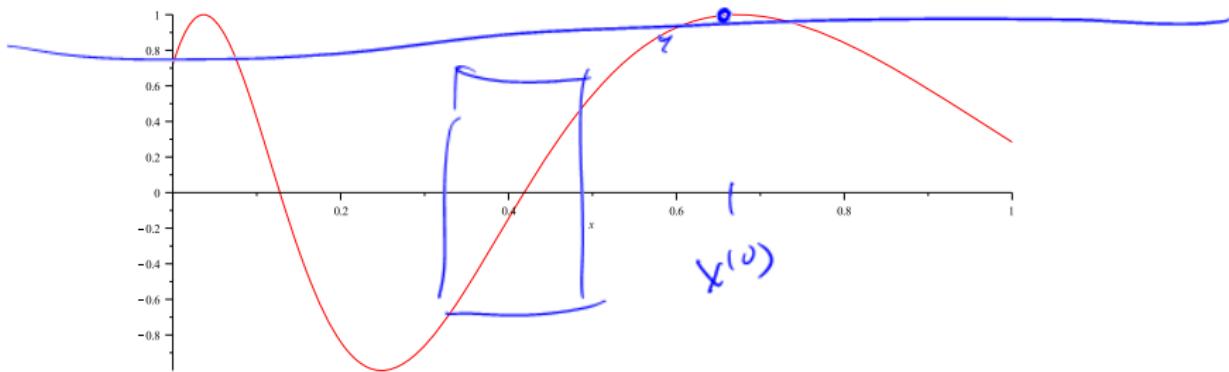
If there are two or more solutions, we don't know to which one it will converge.



1. Under what conditions does the algorithm converge?

Newton-Raphson iteration converges if x_0 is *sufficiently close* to x^* .

Usually, we do not know a priori how close is close enough and we must resort to trial and error....



2. How accurate will the result be?

Both methods can give us x^* up to machine precision.

2. How accurate will the result be?

$$\epsilon = 0.01$$
$$\epsilon^2 = 0.0001$$

Both methods can give us x^* up to machine precision.

3. How fast does it converge?

In bisection, the error $|x^* - x^{(k)}|$ decreases by a factor of 1/2 in each iteration.

In Newton-Raphson iterations, the error is approximately squared in each iteration (provided it is small enough!).

$$\epsilon_0, \frac{\epsilon_0}{2}, \frac{\epsilon_0}{4}, \frac{\epsilon_0}{8}, \dots \quad \text{vs.} \quad \epsilon_0, \epsilon_0^2, \epsilon_0^4, \epsilon_0^8, \dots$$

$\epsilon^{(k)} = \epsilon_0 2^{-k}$

$$\epsilon^{(k)} = \epsilon_0 2^{-k}$$

Newton-Raphson method converges very quickly, but requires the computation of $f'(x)$.

Sometimes, we cannot compute it, for instance if f is shorthand for some complicated procedure:



In that case, we have two options:

Newton-Raphson method converges very quickly, but requires the computation of $f'(x)$.

Sometimes, we cannot compute it, for instance if f is shorthand for some complicated procedure:



In that case, we have two options:

1. bisection, or

Newton-Raphson method converges very quickly, but requires the computation of $f'(x)$.

Sometimes, we cannot compute it, for instance if f is shorthand for some complicated procedure:



In that case, we have two options:

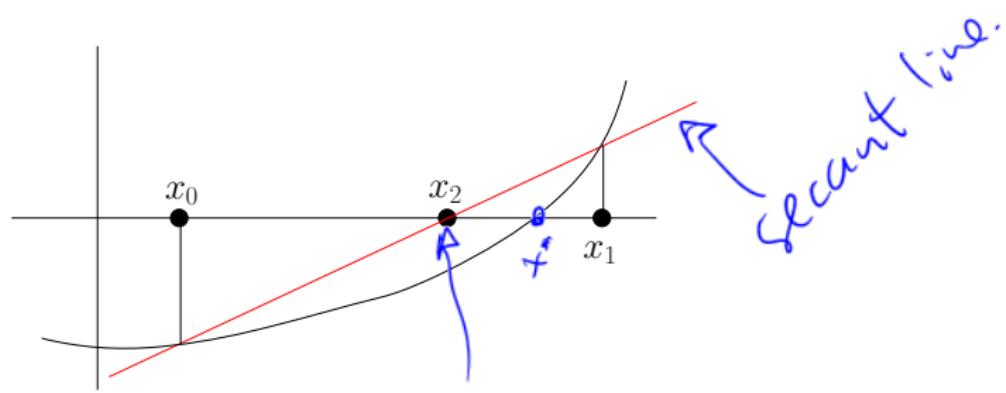
1. bisection, or
2. the secant method.

Suppose we have *two* initial points, x_0 and x_1 .
Then we can *estimate* the derivative of f at x_1 as

$$\overbrace{f'(x_1)}^{\text{approx}} \approx \frac{f(x_1) - f(x_0)}{x_1 - x_0} \quad \cancel{\text{derivative}}$$

and substitute this in the Newton-Raphson iteration:

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \approx x_1 - \frac{f(x_1)(x_1 - x_0)}{f(x_1) - f(x_0)}$$



Then we iterate to find:

Secant method

Given iterates $x^{(k)}$ and $x^{(k-1)}$ approximating a zero of f , compute

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)}) [x^{(k)} - x^{(k-1)}]}{f(x^{(k)}) - f(x^{(k-1)})} \quad (k \geq 1)$$

Secant methods needs *two* initial guesses: $x^{(0)}$ and $x^{(1)}$

Some remarks:

- ▶ This method uses a *finite difference* approximation to f' .
- ▶ Asymptotically (meaning if $|x_k - x^*|$ is small enough) the secant method converges as fast as Newton-Raphson method does.
- ▶ The secant method has extensions to problems with more than 1 unknown, but in this case Newton method tends to be less cumbersome.
- ▶ The secant method is a *second order recurrence relation*. It relates the next approximation to the *two* previous approximations.
- ▶ If we can find an a and b such that $x^* \in [a, b]$, then $x_0 = a$ and $x_1 = b$ is a good starting point.

Recurrence and *iteration* really mean procedures in which we repeat the same action over and over.

One way to program this is by using `for` and `while` loops.

We can also make the recurrent nature of the computation explicit by *making the function call itself*.

This is called *recursive* programming.

Simple example:

```
def fact(k):  
    if k == 1:  
        return 1  
    else:  
        return fact(k-1) * k
```

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. Bisection vs Newton-Raphson

2. Error plotting

3. Exercise

4. Newton iteration

Bisection

Bisection

- ▶ Evaluate function only.

Bisection

- ▶ Evaluate function only.
- ▶ Need to find a and b such that f is continuous and has unique zero on $[a, b]$.

Bisection

- ▶ Evaluate function only.
- ▶ Need to find a and b such that f is continuous and has unique zero on $[a, b]$.
- ▶ Upper bound for error:
$$|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$$

Bisection

- ▶ Evaluate function only.
- ▶ Need to find a and b such that f is continuous and has unique zero on $[a, b]$.
- ▶ Upper bound for error:
 $|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$
- ▶ Decreases as
 $\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$

Bisection

- ▶ Evaluate function only.
- ▶ Need to find a and b such that f is continuous and has unique zero on $[a, b]$.
- ▶ Upper bound for error:
$$|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$$
- ▶ Decreases as
$$\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.

Bisection

- ▶ Evaluate function only.
- ▶ Need to find a and b such that f is continuous and has unique zero on $[a, b]$.
- ▶ Upper bound for error:
 $|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$
- ▶ Decreases as
 $\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.

Bisection

- ▶ Evaluate function only.
- ▶ Need to find a and b such that f is continuous and has unique zero on $[a, b]$.
- ▶ Upper bound for error:
 $|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$
- ▶ Decreases as
 $\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.
- ▶ Works only for one unknown.

Bisection

Newton/secant

- ▶ Evaluate function only.
- ▶ Need to find a and b such that f is continuous and has unique zero on $[a, b]$.
- ▶ Upper bound for error:
$$|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$$
- ▶ Decreases as
$$\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.
- ▶ Works only for one unknown.

Bisection

- ▶ Evaluate function only.
- ▶ Need to find a and b such that f is continuous and has unique zero on $[a, b]$.
- ▶ Upper bound for error:
$$|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$$
- ▶ Decreases as
$$\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.
- ▶ Works only for one unknown.

Newton/secant

- ▶ Uses function *and* derivative/two initial points.

Bisection

- ▶ Evaluate function only.
- ▶ Need to find a and b such that f is continuous and has unique zero on $[a, b]$.
- ▶ Upper bound for error:
$$|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$$
- ▶ Decreases as
$$\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.
- ▶ Works only for one unknown.

Newton/secant

- ▶ Uses function *and* derivative/two initial points.
- ▶ Function must be *continuously differentiable* /continuous around x^* .

Bisection

- ▶ Evaluate function only.
- ▶ Need to find a and b such that f is continuous and has unique zero on $[a, b]$.
- ▶ Upper bound for error:
 $|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$
- ▶ Decreases as
 $\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.
- ▶ Works only for one unknown.

Newton/secant

- ▶ Uses function *and* derivative/two initial points.
- ▶ Function must be *continuously differentiable* /continuous around x^* .
- ▶ Need one/two initial guesses.

Bisection

- ▶ Evaluate function only.
- ▶ Need to find a and b such that f is continuous and has unique zero on $[a, b]$.
- ▶ Upper bound for error:
$$|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$$
- ▶ Decreases as
$$\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.
- ▶ Works only for one unknown.

Newton/secant

- ▶ Uses function *and derivative*/two initial points.
- ▶ Function must be *continuously differentiable* /continuous around x^* .
- ▶ Need one/two initial guesses.
- ▶ Error estimate: $\epsilon^{(k)} \approx \underline{|\delta x^{(k)}|}$.

Bisection

- ▶ Evaluate function only.
- ▶ Need to find a and b such that f is continuous and has unique zero on $[a, b]$.
- ▶ Upper bound for error:
 $|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$
- ▶ Decreases as
 $\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.
- ▶ Works only for one unknown.

Newton/secant

- ▶ Uses function *and* derivative/two initial points.
- ▶ Function must be *continuously differentiable* /continuous around x^* .
- ▶ Need one/two initial guesses.
- ▶ Error estimate: $\epsilon^{(k)} \approx |\delta x^{(k)}|$.
- ▶ Decreases as
 $\epsilon^{(k+1)} \approx (\epsilon^{(k)})^2 \dots$

Bisection

- ▶ Evaluate function only.
- ▶ Need to find a and b such that f is continuous and has unique zero on $[a, b]$.
- ▶ Upper bound for error:
$$|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$$
- ▶ Decreases as
$$\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.
- ▶ Works only for one unknown.

Newton/secant

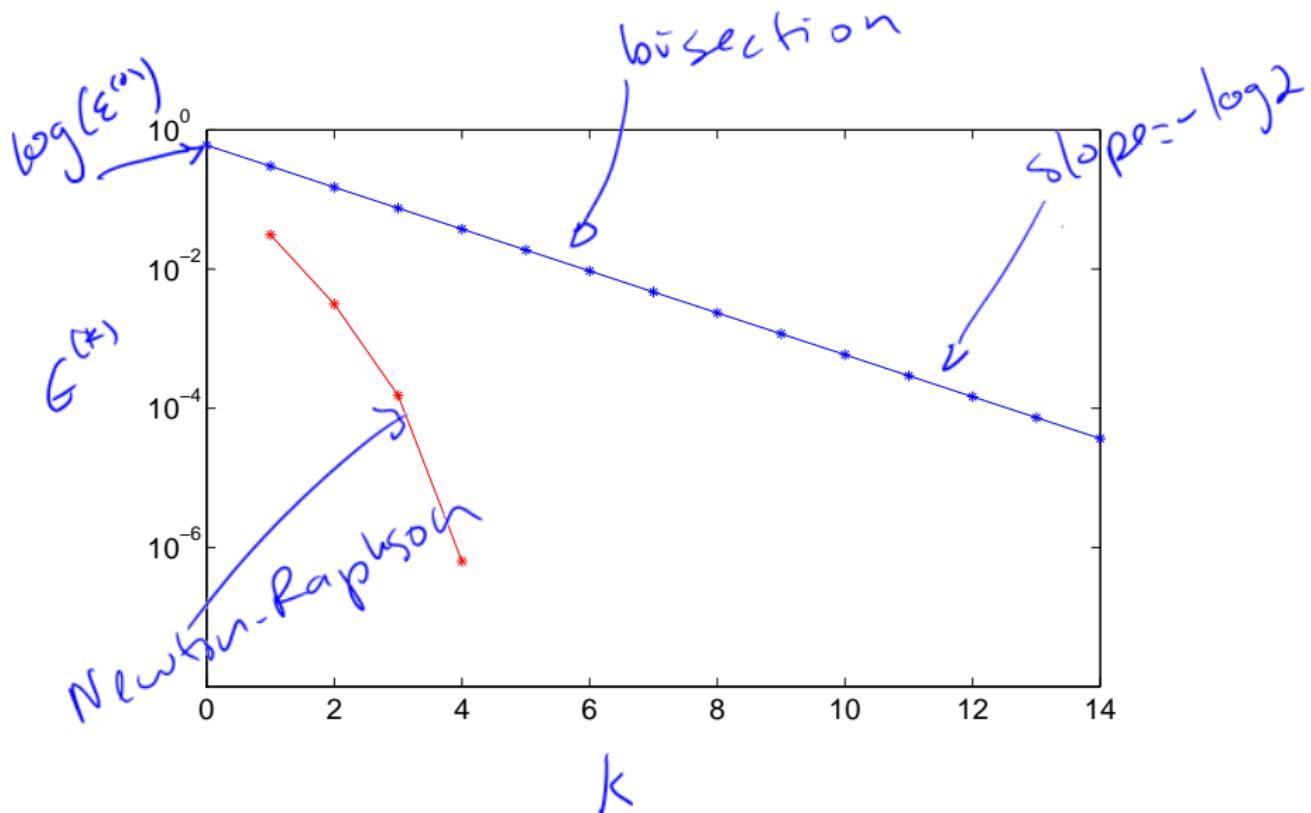
- ▶ Uses function *and derivative*/two initial points.
- ▶ Function must be *continuously differentiable* /continuous around x^* .
- ▶ Need one/two initial guesses.
- ▶ Error estimate: $\epsilon^{(k)} \approx |\delta x^{(k)}|$.
- ▶ Decreases as
$$\epsilon^{(k+1)} \approx (\epsilon^{(k)})^2 \dots$$
- ▶ ... so that $\epsilon^{(k)} \approx (\epsilon^{(0)})^{2^k}$.
aka **quadratic** convergence.

Bisection

- ▶ Evaluate function only.
- ▶ Need to find a and b such that f is continuous and has unique zero on $[a, b]$.
- ▶ Upper bound for error:
$$|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$$
- ▶ Decreases as
$$\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.
- ▶ Works only for one unknown.

Newton/secant

- ▶ Uses function *and* derivative/two initial points.
- ▶ Function must be *continuously differentiable* /continuous around x^* .
- ▶ Need one/two initial guesses.
- ▶ Error estimate: $\epsilon^{(k)} \approx |\delta x^{(k)}|$.
- ▶ Decreases as
$$\epsilon^{(k+1)} \approx (\epsilon^{(k)})^2 \dots$$
- ▶ ... so that $\epsilon^{(k)} \approx (\epsilon^{(0)})^{2^k}$.
aka **quadratic** convergence.
- ▶ Steeper than linear on semilog plot.



Bisection

- ▶ Upper bound for error:
 $|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$

Bisection

- ▶ Upper bound for error:
 $|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$
- ▶ Decreases as
 $\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$

Bisection

- ▶ Upper bound for error:
 $|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$
- ▶ Decreases as
 $\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.

Bisection

- ▶ Upper bound for error:
 $|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$
- ▶ Decreases as
 $\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.

Bisection

Newton/secant

- ▶ Upper bound for error:
 $|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$
- ▶ Decreases as
 $\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.

Bisection

- ▶ Upper bound for error:
 $|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$
- ▶ Decreases as
 $\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.

Newton/secant

- ▶ Error estimate: $\epsilon^{(k)} \approx |\delta x^{(k)}|$.
- ▶ Decreases as
 $\epsilon^{(k+1)} \approx (\epsilon^{(k)})^2 \dots$

Bisection

- ▶ Upper bound for error:
 $|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$
- ▶ Decreases as
 $\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.

Newton/secant

- ▶ Error estimate: $\epsilon^{(k)} \approx |\delta x^{(k)}|$.
- ▶ Decreases as
 $\epsilon^{(k+1)} \approx (\epsilon^{(k)})^2 \dots$
- ▶ ... so that $\epsilon^{(k)} \approx (\epsilon^{(0)})^{2^k}$.
aka **quadratic** convergence.

Bisection

- ▶ Upper bound for error:
 $|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$
- ▶ Decreases as
 $\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.

Python
matplotlib.
semi log y

Newton/secant

- ▶ Error estimate: $\epsilon^{(k)} \approx |\delta x^{(k)}|$.
- ▶ Decreases as
 $\epsilon^{(k+1)} \approx (\epsilon^{(k)})^2 \dots$
- ▶ ... so that $\epsilon^{(k)} \approx (\epsilon^{(0)})^{2^k}$.
aka **quadratic** convergence.
- ▶ Steeper than linear on semilog plot.

bisection

$$\epsilon^{(k)} = \epsilon^{(0)} / 2^k$$

$$\epsilon^{(k)} = \epsilon^{(0)} 2^{-k}$$

$$\begin{aligned}\log \epsilon^{(k)} &= \log(\epsilon^{(0)} 2^{-k}) \\ &= \log(\epsilon^{(0)}) + \log 2^{-k}\end{aligned}$$

$$\underbrace{\log \epsilon^{(k)}}_{y} = \underset{x}{\uparrow} -k \log 2 + \log(\epsilon^{(0)})$$

$$\Rightarrow y = mx + b$$

Newton-Raphson

$$\epsilon^{(k)} = [\epsilon^{(0)}]^{2^k}$$

$$\log(\epsilon^{(k)}) = 2^k \log(\epsilon^{(0)})$$

$$y = \log(\epsilon^{(k)})$$

$$x = k.$$

\Rightarrow exponential.

$$y = \log \epsilon^{(k)}, m = -\log 2, x = k, b = \log(\epsilon^{(0)})$$

Bisection

- ▶ Upper bound for error:
 $|x^{(k)} - x^*| \leq \epsilon^{(k)} = |b^{(k)} - a^{(k)}|$
- ▶ Decreases as
 $\epsilon^{(k+1)} = \epsilon^{(k)}/2 \dots$
- ▶ ... so that $\epsilon^{(k)} = \epsilon^{(0)}/2^k$.
aka **linear** convergence.
- ▶ Straight line on semilog plot.

Newton/secant

- ▶ Error estimate: $\epsilon^{(k)} \approx |\delta x^{(k)}|$.
- ▶ Decreases as
 $\epsilon^{(k+1)} \approx (\epsilon^{(k)})^2 \dots$
- ▶ ... so that $\epsilon^{(k)} \approx (\epsilon^{(0)})^{2^k}$.
aka **quadratic** convergence.
- ▶ Steeper than linear on semilog plot.

where a semilog plot shows $\log(\epsilon^{(k)})$ vs k

Exercise: Use the bisection method to find a solution x^* that satisfies the equation:

$$f(x) = x^2 - 5 = 0,$$

to a tolerance (maximum error) of $\epsilon = 10^{-6}$. Use a starting interval $[a_0, b_0] = [2, 3]$. Compare the result to $\sqrt{5}$.

Newton-Raphson iteration can be generalized to n equations with n unknowns.

Alternative derivation in 1D (Taylor expansion):

$$f(x + \delta x) \approx f(x) + f'(x)\delta x = 0 \Rightarrow \delta x = -\frac{f(x)}{f'(x)}$$

Now in 2D. We want to find x_1 and x_2 such that

$$f_1(x_1, x_2) = 0$$

$$f_2(x_1, x_2) = 0$$

Note that, in general, we need **the same number of equations and unknowns** to find (isolated) solutions...

$$f_1(x_1 + \delta x_1, x_2 + \delta x_2) \approx f_1(x_1, x_2) + \frac{\partial f_1}{\partial x_1}(x_1, x_2)\delta x_1 + \frac{\partial f_1}{\partial x_2}(x_1, x_2)\delta x_2$$
$$f_2(x_1 + \delta x_1, x_2 + \delta x_2) \approx f_2(x_1, x_2) + \frac{\partial f_2}{\partial x_1}(x_1, x_2)\delta x_1 + \frac{\partial f_2}{\partial x_2}(x_1, x_2)\delta x_2$$

which gives:

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x_1, x_2) & \frac{\partial f_1}{\partial x_2}(x_1, x_2) \\ \frac{\partial f_2}{\partial x_1}(x_1, x_2) & \frac{\partial f_2}{\partial x_2}(x_1, x_2) \end{pmatrix} \begin{pmatrix} \delta x_1 \\ \delta x_2 \end{pmatrix} = - \begin{pmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{pmatrix}$$

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. Newton iteration
2. Central questions
3. Reminder: matrices and SCIPY
4. Matrix operations
5. Matrix algebra
6. Systems of linear equations
7. Easy-to-solve systems
8. Gaussian elimination
9. LU decomposition

Newton-Raphson iteration can be generalized to n equations with n unknowns.

Alternative derivation in 1D:

$$f(x + \delta x) \approx f(x) + f'(x)\delta x = 0 \Rightarrow \delta x = -\frac{f(x)}{f'(x)}$$

Now in 2D. We want to find x_1 and x_2 such that

$$f_1(x_1, x_2) = 0$$

$$f_2(x_1, x_2) = 0$$

Note that, in general, we need **the same number of equations and unknowns** to find (isolated) solutions...

$$f_1(x_1 + \delta x_1, x_2 + \delta x_2) \approx f_1(x_1, x_2) + \frac{\partial f_1}{\partial x_1}(x_1, x_2)\delta x_1 + \frac{\partial f_1}{\partial x_2}(x_1, x_2)\delta x_2$$
$$f_2(x_1 + \delta x_1, x_2 + \delta x_2) \approx f_2(x_1, x_2) + \frac{\partial f_2}{\partial x_1}(x_1, x_2)\delta x_1 + \frac{\partial f_2}{\partial x_2}(x_1, x_2)\delta x_2$$

In matrix form:

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x_1, x_2) & \frac{\partial f_1}{\partial x_2}(x_1, x_2) \\ \frac{\partial f_2}{\partial x_1}(x_1, x_2) & \frac{\partial f_2}{\partial x_2}(x_1, x_2) \end{pmatrix} \begin{pmatrix} \delta x_1 \\ \delta x_2 \end{pmatrix} = - \begin{pmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{pmatrix}$$

To find the update, we need to solve a system of linear equations of the form:

$$a_{1,1}x_1 + a_{1,2}x_2 = b_1$$

$$a_{2,1}x_1 + a_{2,2}x_2 = b_2$$

or in matrix form:

$$Ax = \mathbf{b}$$

where $A \in \mathbb{R}^{2 \times 2}$ is a 2×2 matrix, $\mathbf{x} \in \mathbb{R}^2$ are the unknowns, and $\mathbf{b} \in \mathbb{R}^2$, i.e.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

In general, we need to solve a system of linear equations of the form:

$$a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = b_1$$

$$a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n = b_2$$

 \vdots \vdots

$$a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n = b_n$$

or in matrix form:

$$A\mathbf{x} = \mathbf{b}$$

where $A \in \mathbb{R}^{n \times n}$ is an $n \times n$ matrix with entries $a_{i,j}$, $\mathbf{x} \in \mathbb{R}^n$ is the vector of unknowns x_j , and $\mathbf{b} \in \mathbb{R}^n$ is the vector with entries b_i .

Example for $n = 4$ (i.e. with 4 unknowns: x_1, x_2, x_3, x_4 .)

$$x_1 + 2x_2 - 4x_3 + x_4 = 1$$

$$3x_1 - x_2 + x_3 + 4x_4 = 3$$

$$x_1 - 2x_2 + 3x_3 - x_4 = -1$$

$$2x_1 - x_2 - x_3 + 3x_4 = 2$$

or in matrix form:

$$A\mathbf{x} = \mathbf{b}$$

where

$$A = \begin{bmatrix} 1 & 2 & -4 & 1 \\ 3 & -1 & 1 & 4 \\ 1 & -2 & 3 & -1 \\ 2 & -1 & -1 & 3 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 3 \\ -1 \\ 2 \end{bmatrix}.$$

How do we solve such a system of linear equations?

Central questions:

- ▶ What is Gaussian elimination? LU decomposition?
- ▶ How is LU decomposition related to Gaussian elimination?
- ▶ How is an LU decomposition $A = LU$ computed?
- ▶ For any square $A \in \mathbb{R}^{n \times n}$, does a decomposition $A = LU$ exist?

In Python (use **NUMPY** and **SCIPY**):
`import numpy as np,
import scipy.linalg`

Matrices

- Matrix $A \in \mathbb{R}^{m \times n}$ is rectangular array of numbers

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n-1} & a_{2,n} \\ \vdots & \vdots & & \vdots & \vdots \\ a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,n-1} & a_{m-1,n} \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n-1} & a_{m,n} \end{pmatrix}$$

- Numbers $a_{i,j}$ = **elements of A** = **entries of A** .
- First index (i) of element $a_{i,j}$ = **row index**.
- Second index (j) of element $a_{i,j}$ = **column index**.

Example: $A = \text{np.array}([[1, 2], [3, 4]])$

Vectors

- ▶ n -vector: “skinny” matrix (dimension $n \times 1$ or $1 \times n$)

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} \text{ or } \mathbf{x}^T = (x_1, x_2, \dots, x_{n-1}, x_n)$$

- ▶ Elements x_i = components of \mathbf{x} .
- ▶ Convention: vectors generically **column** vectors
assume $\mathbf{x} \in \mathbb{R}^n$ means $\mathbf{x} \in \mathbb{R}^{n \times 1}$.
- ▶ To NumPy, scalars are vectors of length 1 and also matrices of dimension 1×1 .

Special matrices

Zero matrix $0 \in \mathbb{R}^{m \times n}$

$$\forall A \in \mathbb{R}^{m \times n} \quad A + 0 = 0 + A = A, \text{ where } 0 = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

Identity matrix $I \in \mathbb{R}^{n \times n}$

$$A \in \mathbb{R}^{n \times n} \quad AI = IA = A, \text{ where } I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

Examples: `np.zeros((3, 2))`, `np.identity(3)`

Special vectors

- ▶ Coordinate vectors: all 0s except one 1.
- ▶ k^{th} -coordinate vector is

$$\mathbf{e}_k := I_{:,k} \in \mathbb{R}^{n \times 1},$$

i.e., k^{th} column of $I \in \mathbb{R}^{n \times n}$.

- ▶ Convenient notation for matrix algorithms.

$$\mathbf{e}_k = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Example with $n = 4$:

```
I=np.identity(4)
e1=I[:,[0]], e2=I[:,[1]]
e3=I[:,[2]], e4=I[:,[3]]
```

Matrix transpose

If $A \in \mathbb{R}^{m \times n}$, $C = A^T \in \mathbb{R}^{n \times m}$ is

$$c_{i,j} = a_{j,i} \quad (1 \leq i \leq n, 1 \leq j \leq m)$$

e.g., $\begin{bmatrix} -7 & -5 & 6 \\ -1 & -8 & 10 \end{bmatrix}^T = \begin{bmatrix} -7 & -1 \\ -5 & -8 \\ 6 & 10 \end{bmatrix}$

- ▶ Use `np.transpose` or `.T` for the transpose of matrices:

```
A = np.array([[-7, -5, 6], [-1, -8, 10]])
```

```
C = np.transpose(A)
```

```
C = A.T
```

- ▶ If $A \in \mathbb{R}^{n \times n}$ satisfies $A = A^T$, A is said to be **symmetric**.

Scalar multiplication

If $\mu \in \mathbb{R}$ and $A \in \mathbb{R}^{m \times n}$, $C = \mu A \in \mathbb{R}^{m \times n}$ is

$$c_{i,j} = \mu a_{i,j} \quad (i = 1 : m, j = 1 : n)$$

e.g., $3 \begin{bmatrix} 1 & -2 \\ -3 & 1/2 \end{bmatrix} = \begin{bmatrix} 3 & -6 \\ -9 & 3/2 \end{bmatrix}$

- ▶ In PYTHON, scalar multiplication uses operator *

```
A=np.array([[1,-2],[-3,0.5]])
```

```
B=3*A
```

Matrix addition

If $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{m \times n}$, **matrix sum** $C = A + B \in \mathbb{R}^{m \times n}$ is

$$c_{i,j} = a_{i,j} + b_{i,j} \quad (i = 1 : m, j = 1 : n)$$

e.g.,

$$\begin{bmatrix} -2 & -3 & 3 \\ 4 & -5 & -3 \end{bmatrix} + \begin{bmatrix} 7 & 5 & 2 \\ -9 & -3 & 8 \end{bmatrix} = \begin{bmatrix} 5 & 2 & 5 \\ -5 & -8 & 5 \end{bmatrix}$$

- Matrix addition uses operator +

```
A=np.array([[ -2.0, -3, 3], [4, -5, -3]])
```

```
B=np.array([[ 7.0, 5, 2], [-9, -3, 8]])
```

```
C=A+B
```

- Matrices **must** be conformable (same shape) for addition.

Matrix multiplication

If $A \in \mathbb{R}^{m \times s}$, $B \in \mathbb{R}^{s \times n}$, matrix product $C = AB \in \mathbb{R}^{m \times n}$ is

$$c_{i,j} = \sum_{k=1}^s a_{i,k} b_{k,j} \quad (i = 1 : m, j = 1 : n)$$

e.g.,

$$\begin{bmatrix} -1 & 5 & -4 \\ -4 & 1 & 2 \end{bmatrix} \begin{bmatrix} -2 & -1 & 0 \\ 3 & 3 & 2 \\ -1 & -1 & 2 \end{bmatrix} = \begin{bmatrix} 21 & 20 & 2 \\ 9 & 5 & 6 \end{bmatrix}$$

- ▶ In PYTHON: `np.matmul(A, B)` or `A @ B`
- ▶ Requires A and B satisfies
`np.shape(A) [1] == np.shape(B) [0]`.
- ▶ Note: $AB \neq BA$ in general!

Matrix inverse

Square matrix $A \in \mathbb{R}^{n \times n}$ is **invertible** (or **regular** or **nonsingular**) if there exists $B \in \mathbb{R}^{n \times n}$ such that

$$AB = BA = I$$

Inverse of A is unique and denoted A^{-1} ; A must be square,

e.g.,
$$\begin{bmatrix} -2 & -2 & 4 \\ 1 & -3 & 0 \\ -4 & 4 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1/8 & -3/4 & -1/2 \\ 1/24 & -7/12 & -1/6 \\ 1/3 & -2/3 & -1/3 \end{bmatrix}$$

- ▶ Use routine `scipy.linalg.inv` for computing matrix inverse:

```
A=np.array([[ -2, -3, 3], [4, -5, -3]])  
B=scipy.linalg.inv(A)
```

Algebra

For any scalars $\mu \in \mathbb{R}$:

1. $A + 0 = 0 + A = A$
2. $IA = AI = A$
3. $A(B + C) = AB + AC$ for any $A \in \mathbb{R}^{m \times s}$; $B, C \in \mathbb{R}^{s \times n}$
4. $(AB)C = A(BC)$ for any $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times l}$, $C \in \mathbb{R}^{l \times n}$
5. $\mu(AB) = (\mu A)B = A(\mu B)$ for any $A \in \mathbb{R}^{m \times s}$, $B \in \mathbb{R}^{s \times n}$
6. $(\mu A)^T = \mu A^T$
7. $(A + B)^T = A^T + B^T$ for any matrices $A, B \in \mathbb{R}^{m \times n}$
8. $(AB)^T = B^T A^T$ for any $A \in \mathbb{R}^{m \times s}$, $B \in \mathbb{R}^{s \times n}$
9. $(AB)^{-1} = B^{-1}A^{-1}$ for any invertible $A, B \in \mathbb{R}^{n \times n}$

Theorem (Nonsingular matrix properties)

For $A \in \mathbb{R}^{n \times n}$, the following properties are equivalent:

1. The inverse of A exists; i.e., A is nonsingular
2. $\det(A) \neq 0$
3. For every $\mathbf{b} \in \mathbb{R}^n$, system $A\mathbf{x} = \mathbf{b}$ has unique solution $\mathbf{x} \in \mathbb{R}^n$
4. $A\mathbf{x} = \mathbf{0} \Rightarrow \mathbf{x} = \mathbf{0}$
5. The rows of A form a basis for \mathbb{R}^n
6. The columns of A form a basis for \mathbb{R}^n
7. The map $\{A : \mathbb{R}^n \text{ into } \mathbb{R}^n\}$ is one-to-one (injective)
8. The map $\{A : \mathbb{R}^n \text{ into } \mathbb{R}^n\}$ is onto (surjective)
9. 0 is not an eigenvalue of A

- ▶ Rule for matrix multiplication permits representation of linear systems of equations using matrices and vectors.
- ▶ e.g., linear system of equations

$$2x_1 + x_2 + x_3 = 4$$

$$4x_1 + 3x_2 + 3x_3 + x_4 = 11$$

$$8x_1 + 7x_2 + 9x_3 + 5x_4 = 29$$

$$6x_1 + 7x_2 + 9x_3 + 8x_4 = 30$$

can be written as $A\mathbf{x} = \mathbf{b}$ with

$$\underbrace{\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}}_A \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_x = \underbrace{\begin{bmatrix} 4 \\ 11 \\ 29 \\ 30 \end{bmatrix}}_{\mathbf{b}}$$

We can solve linear systems of equations in SciPY with the linalg module using `scipy.linalg.solve`.

► Simplest use:

```
In [1]: import scipy.linalg
In [2]: A = np.array([[7.0,5.0,2.0],
                  [-3.0,1.0,0.0],[0.0,12.0,-3.0]])
In [3]: b = np.array([[3.0],[-4.0],[0.0]])
In [4]: x = scipy.linalg.solve(A,b)
```

► `scipy.linalg` actually calls the LAPACK and BLAS routines - optimized for your hardware under Linux.

- ▶ Present goal: to understand what `scipy.linalg.solve` does:
 - ▶ Gaussian elimination,
 - ▶ *LU* decomposition,
 - ▶ pivoting.

- ▶ Present goal: to understand what `scipy.linalg.solve` does:
 - ▶ Gaussian elimination,
 - ▶ *LU* decomposition,
 - ▶ pivoting.
- ▶ See `docs.scipy.org` reference guide.

- ▶ Present goal: to understand what `scipy.linalg.solve` does:
 - ▶ Gaussian elimination,
 - ▶ LU decomposition,
 - ▶ pivoting.
- ▶ See `docs.scipy.org` reference guide.

Solution of $A\mathbf{x} = \mathbf{b}$

Never solve linear systems by computing A^{-1} and $\mathbf{x} = A^{-1}\mathbf{b}$!

Use SciPy's built-in solvers that avoid inverting matrices.

We will see that computing A^{-1} explicitly is *slow* and often leads to *large numerical error*.

Diagonal systems:

- Given vector $\mathbf{b} = (b_1, \dots, b_n)^T \in \mathbb{R}^n$, and diagonal matrix D , wish to solve linear system of equations $D\mathbf{x} = \mathbf{b}$, i.e.,

$$\begin{bmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

- Solution of $D\mathbf{x} = \mathbf{b}$ directly computable:

$$x_k = \frac{b_k}{d_k} \quad (d_k \neq 0, k = 1 : n)$$

Solve the linear system of equations

$$\begin{bmatrix} 2 & 3 & -4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 9 \\ 1 \end{bmatrix}$$

$$2x_1 = 5 \Rightarrow x_1 = \frac{5}{2}$$

$$3x_2 = 9 \Rightarrow x_2 = \frac{9}{3} = 3$$

$$-4x_3 = 1 \Rightarrow x_3 = -\frac{1}{4}$$

Equations are completely decoupled.

Upper triangular systems:

- Given $\mathbf{b} = (b_1, \dots, b_n)^T \in \mathbb{R}^n$ and U upper triangular, wish to solve linear system of equations $U\mathbf{x} = \mathbf{b}$, i.e.,

$$\begin{bmatrix} U_{1,1} & U_{1,2} & \cdots & U_{1,n} \\ U_{2,1} & U_{2,2} & \cdots & U_{2,n} \\ \ddots & & \ddots & \\ & & & U_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

- Solution of $U\mathbf{x} = \mathbf{b}$ through **backward substitution**:

$$x_k = \frac{1}{U_{k,k}} \left(b_k - \sum_{j=k+1}^n U_{k,j} x_j \right) \quad (k = 1 : n)$$

Solve the linear system of equations

$$\begin{bmatrix} 2 & 3 & -2 \\ 0 & 3 & 5 \\ 0 & 0 & -4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 9 \\ 1 \end{bmatrix}$$

$$-4x_3 = 1 \Rightarrow x_3 = -\frac{1}{4}$$

$$3x_2 + 5x_3 = 9 \Rightarrow x_2 = \frac{1}{3} \left(9 - 5 \left(-\frac{1}{4} \right) \right) = \frac{41}{12}$$

$$2x_1 + 3x_2 - 2x_3 = 5 \Rightarrow x_1 = \frac{1}{2} \left(5 - 3 \left(\frac{41}{12} \right) + 2 \left(-\frac{1}{4} \right) \right) = -\frac{23}{8}$$

Lower triangular systems:

- Given $\mathbf{b} = (b_1, \dots, b_n)^T \in \mathbb{R}^n$ and L lower triangular, wish to solve linear system of equations $L\mathbf{x} = \mathbf{b}$, i.e.,

$$\begin{bmatrix} L_{1,1} & & & \\ L_{2,1} & L_{2,2} & & \\ \vdots & \vdots & \ddots & \\ L_{n,1} & L_{n,2} & \cdots & L_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

- Solution of $L\mathbf{x} = \mathbf{b}$ through **forward substitution**:

$$x_k = \frac{1}{L_{k,k}} \left(b_k - \sum_{j=1}^{k-1} L_{k,j}x_j \right) \quad (k = 1 : n)$$

Solve the linear system of equations

$$\begin{bmatrix} 2 & & \\ 3 & 3 & \\ -2 & 5 & -4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 9 \\ 1 \end{bmatrix}$$

$$2x_1 = 5 \quad \Rightarrow \quad x_1 = \frac{5}{2}$$

$$3x_1 + 3x_2 = 9 \quad \Rightarrow \quad x_2 = \frac{1}{3} \left(9 - 3 \left(\frac{5}{2} \right) \right) = \frac{1}{2}$$

$$-2x_1 + 5x_2 - 4x_3 = 1 \quad \Rightarrow \quad x_3 = -\frac{1}{4} \left(1 + 2 \left(\frac{5}{2} \right) - 5 \left(\frac{1}{2} \right) \right) = -\frac{7}{8}$$

Gaussian elimination

Gaussian elimination transforms a general system $A\mathbf{x} = \mathbf{b}$ into an easy-to-solve system.

- ▶ Elementary row operations:
 - ▶ Interchanging two equations: $R_i \leftrightarrow R_j$
 - ▶ Multiplying an equation by a nonzero scalar: $R_i \leftarrow \lambda R_i$
 - ▶ Adding a multiple of an equation to another: $R_i \leftarrow R_i + \lambda R_j$
- ▶ Applying elementary row operations to linear system of equations preserves solution of original system

Gaussian elimination

Gaussian elimination transforms a general system $A\mathbf{x} = \mathbf{b}$ into an easy-to-solve system.

- ▶ Elementary row operations:
 - ▶ Interchanging two equations: $R_i \leftrightarrow R_j$
 - ▶ Multiplying an equation by a nonzero scalar: $R_i \leftarrow \lambda R_i$
 - ▶ Adding a multiple of an equation to another: $R_i \leftarrow R_i + \lambda R_j$
- ▶ Applying elementary row operations to linear system of equations preserves solution of original system

Central Idea

Reduce square system of linear equations to upper triangular system by sequence of elementary row operations.

Example:

- ▶ Consider solving linear system of equations

$$2x_1 + x_2 + x_3 = 4$$

$$4x_1 + 3x_2 + 3x_3 + x_4 = 11$$

$$8x_1 + 7x_2 + 9x_3 + 5x_4 = 29$$

$$6x_1 + 7x_2 + 9x_3 + 8x_4 = 30$$

Example:

- ▶ Consider solving linear system of equations

$$\begin{aligned} 2x_1 + x_2 + x_3 &= 4 \\ 4x_1 + 3x_2 + 3x_3 + x_4 &= 11 \\ 8x_1 + 7x_2 + 9x_3 + 5x_4 &= 29 \\ 6x_1 + 7x_2 + 9x_3 + 8x_4 &= 30 \end{aligned}$$

- ▶ Write system as $A\mathbf{x} = \mathbf{b}$ with

$$A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 4 \\ 11 \\ 29 \\ 30 \end{bmatrix}$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right]$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \quad \leftarrow R_2 - (4/2)R_1$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \quad \leftarrow R_2 - (4/2)R_1$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \end{array} \right]$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \quad \begin{matrix} \leftarrow R_2 - (4/2)R_1 \\ \leftarrow R_3 - (8/2)R_1 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 1 & 3 \end{array} \right]$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \quad \begin{matrix} \leftarrow R_2 - (4/2)R_1 \\ \leftarrow R_3 - (8/2)R_1 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 1 & 3 \\ 3 & 5 & 5 & 13 \end{array} \right]$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \quad \begin{matrix} \leftarrow R_2 - (4/2)R_1 \\ \leftarrow R_3 - (8/2)R_1 \\ \leftarrow R_4 - (6/2)R_1 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 3 & 5 & 5 & 13 & 13 \end{array} \right]$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \quad \begin{matrix} \leftarrow R_2 - (4/2)R_1 \\ \leftarrow R_3 - (8/2)R_1 \\ \leftarrow R_4 - (6/2)R_1 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 \\ 3 & 5 & 5 & 13 \\ 4 & 6 & 8 & 18 \end{array} \right]$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \begin{matrix} \leftarrow R_2 - (4/2)R_1 \\ \leftarrow R_3 - (8/2)R_1 \\ \leftarrow R_4 - (6/2)R_1 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 3 & 5 & 5 & 13 & 13 \\ 4 & 6 & 8 & 18 & 18 \end{array} \right]$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \quad \begin{matrix} \leftarrow R_2 - (4/2)R_1 \\ \leftarrow R_3 - (8/2)R_1 \\ \leftarrow R_4 - (6/2)R_1 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 3 & 5 & 5 & 13 & 13 \\ 4 & 6 & 8 & 18 & 18 \end{array} \right] \quad \begin{matrix} \\ \\ \leftarrow R_3 - (3/1)R_2 \\ \end{matrix}$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \quad \begin{matrix} \leftarrow R_2 - (4/2)R_1 \\ \leftarrow R_3 - (8/2)R_1 \\ \leftarrow R_4 - (6/2)R_1 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 3 & 5 & 5 & 13 & 13 \\ 4 & 6 & 8 & 18 & 18 \end{array} \right] \quad \leftarrow R_3 - (3/1)R_2$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 2 & 2 & 2 & 4 & 4 \end{array} \right]$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \quad \begin{matrix} \leftarrow R_2 - (4/2)R_1 \\ \leftarrow R_3 - (8/2)R_1 \\ \leftarrow R_4 - (6/2)R_1 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 3 & 5 & 5 & 13 & 13 \\ 4 & 6 & 8 & 18 & 18 \end{array} \right] \quad \begin{matrix} \leftarrow R_3 - (3/1)R_2 \\ \leftarrow R_4 - (4/1)R_2 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 2 & 2 & 2 & 4 & 4 \end{array} \right]$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \quad \begin{matrix} \leftarrow R_2 - (4/2)R_1 \\ \leftarrow R_3 - (8/2)R_1 \\ \leftarrow R_4 - (6/2)R_1 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 3 & 5 & 5 & 13 & 13 \\ 4 & 6 & 8 & 18 & 18 \end{array} \right] \quad \begin{matrix} \leftarrow R_3 - (3/1)R_2 \\ \leftarrow R_4 - (4/1)R_2 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 2 & 2 & 2 & 4 & 4 \\ 2 & 4 & 4 & 6 & 6 \end{array} \right]$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \quad \begin{matrix} \leftarrow R_2 - (4/2)R_1 \\ \leftarrow R_3 - (8/2)R_1 \\ \leftarrow R_4 - (6/2)R_1 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 1 & 3 \\ 3 & 5 & 5 & 5 & 13 \\ 4 & 6 & 8 & 8 & 18 \end{array} \right] \quad \begin{matrix} \leftarrow R_3 - (3/1)R_2 \\ \leftarrow R_4 - (4/1)R_2 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 1 & 3 \\ 2 & 2 & 2 & 2 & 4 \\ 2 & 4 & 4 & 4 & 6 \end{array} \right]$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \quad \leftarrow R_2 - (4/2)R_1 \\ \leftarrow R_3 - (8/2)R_1 \\ \leftarrow R_4 - (6/2)R_1$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 1 & 3 \\ 3 & 5 & 5 & 13 & 13 \\ 4 & 6 & 8 & 18 & 18 \end{array} \right] \quad \leftarrow R_3 - (3/1)R_2 \\ \leftarrow R_4 - (4/1)R_2$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 1 & 3 \\ 2 & 2 & 2 & 4 & 4 \\ 2 & 4 & 6 & 6 & 6 \end{array} \right] \quad \leftarrow R_4 - (2/2)R_3$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \quad \begin{matrix} \leftarrow R_2 - (4/2)R_1 \\ \leftarrow R_3 - (8/2)R_1 \\ \leftarrow R_4 - (6/2)R_1 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 3 & 5 & 5 & 13 & 13 \\ 4 & 6 & 8 & 18 & 18 \end{array} \right] \quad \begin{matrix} \leftarrow R_3 - (3/1)R_2 \\ \leftarrow R_4 - (4/1)R_2 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 2 & 2 & 2 & 4 & 4 \\ 2 & 4 & 6 & 6 & 6 \end{array} \right] \quad \leftarrow R_4 - (2/2)R_3$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 2 & 2 & 2 & 4 & 4 \\ 2 & 2 & 2 & 2 & 2 \end{array} \right]$$

Form augmented system and carry out elimination

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 4 & 3 & 3 & 1 & 11 \\ 8 & 7 & 9 & 5 & 29 \\ 6 & 7 & 9 & 8 & 30 \end{array} \right] \quad \begin{matrix} \leftarrow R_2 - (4/2)R_1 \\ \leftarrow R_3 - (8/2)R_1 \\ \leftarrow R_4 - (6/2)R_1 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 3 & 5 & 5 & 13 & 13 \\ 4 & 6 & 8 & 18 & 18 \end{array} \right] \quad \begin{matrix} \leftarrow R_3 - (3/1)R_2 \\ \leftarrow R_4 - (4/1)R_2 \end{matrix}$$

$$\left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 2 & 2 & 2 & 4 & 4 \\ 2 & 4 & 6 & 6 & 6 \end{array} \right] \quad \leftarrow R_4 - (2/2)R_3 \quad \left[\begin{array}{cccc|c} 2 & 1 & 1 & 0 & 4 \\ 1 & 1 & 1 & 3 & 3 \\ 2 & 2 & 2 & 4 & 4 \\ 2 & 2 & 2 & 2 & 2 \end{array} \right]$$

We arrive at upper triangular system $U\mathbf{x} = \mathbf{c}$ to solve.

Observations:

- ▶ Pivot element on diagonal used to zero out entries

$$\text{pivot} = A_{k,k} \quad (k = 1 : n - 1)$$

- ▶ Multiplier for eliminating $A_{k,\ell}$ with pivot element $A_{k,k}$ is

$$m_{k,\ell} := A_{k,\ell}/A_{k,k} \quad (k = 1 : n - 1, \ell = k + 1 : n)$$

- ▶ Multiply k th row by $-m_{k,\ell}$ and add to ℓ th row
- ▶ Zeros out k th column below diagonal pivot element.
- ▶ For the moment, assume no row interchanges.

Observations:

- ▶ Pivot element on diagonal used to zero out entries

$$\text{pivot} = A_{k,k} \quad (k = 1 : n - 1)$$

- ▶ Multiplier for eliminating $A_{k,\ell}$ with pivot element $A_{k,k}$ is

$$m_{k,\ell} := A_{k,\ell}/A_{k,k} \quad (k = 1 : n - 1, \ell = k + 1 : n)$$

- ▶ Multiply k th row by $-m_{k,\ell}$ and add to ℓ th row
- ▶ Zeros out k th column below diagonal pivot element.
- ▶ For the moment, assume no row interchanges.

Key Observation

Each stage of elimination amounts to multiplying A on the left by unit lower triangular matrix with negatives of multipliers in pivot column.

In our example:

$$\begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 3 & 5 & 5 \\ 0 & 4 & 6 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -4 & 0 & 1 & 0 \\ -3 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{pmatrix} = L_1 A$$

$$\begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 2 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -3 & 1 & 0 \\ 0 & -4 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 3 & 5 & 5 \\ 0 & 4 & 6 & 8 \end{pmatrix} = L_2 L_1 A$$

$$\begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 2 & 4 \end{pmatrix} = L_3 L_2 L_1 A$$

So that, finally,

$$L_3 L_2 L_1 A = U \text{ or } A = (L_3 L_2 L_1)^{-1} U = L_1^{-1} L_2^{-1} L_3^{-1} U = LU$$

where

$$L = L_1^{-1} L_2^{-1} L_3^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 4 & 3 & 1 & 0 \\ 3 & 4 & 1 & 1 \end{pmatrix}$$

If two matrices are lower (upper) triangular, then so is their product and their inverse!

Note, because L_1 , L_2 , and L_3 are matrix representations of elementary row operations, their inverses are easy to find, and thus L is easy to find.

Example: Inverses of elementary matrices (i.e matrix representations of elementary row operations) are easy to find.

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -m & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \rightarrow L^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ m & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

It's easy to check that $LL^{-1} = I$.

They're also easy to multiply.

$$L_{21}L_{31} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ m_1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ m_2 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ m_1 & 1 & 0 & 0 \\ m_2 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

LU decomposition:

Key principle

Gaussian elimination is **equivalent** to finding L & U such that

LU decomposition:

Key principle

Gaussian elimination is **equivalent** to finding L & U such that

- ▶ L is unit lower triangular matrix (ones on diagonal),

LU decomposition:

Key principle

Gaussian elimination is **equivalent** to finding L & U such that

- ▶ L is unit lower triangular matrix (ones on diagonal),
- ▶ U is upper triangular matrix,

LU decomposition:

Key principle

Gaussian elimination is **equivalent** to finding L & U such that

- ▶ L is unit lower triangular matrix (ones on diagonal),
- ▶ U is upper triangular matrix,
- ▶ $A = LU$.

LU decomposition:

Key principle

Gaussian elimination is **equivalent** to finding L & U such that

- ▶ L is unit lower triangular matrix (ones on diagonal),
- ▶ U is upper triangular matrix,
- ▶ $A = LU$.

LU Decomposition

A pair of matrices L and U with the properties above is an **LU decomposition** (or **LU factorisation** or **Gauss factorisation**) of A .

Procedure for LU decomposition

1. Start by writing down $n \times n$ matrix A and identity matrix.
 2. Carry out steps of Gaussian elimination, transforming A to upper triangular (“row echelon”) form.
 3. At each stage of elimination, write multiplier $m_{k,\ell}$ in (k, ℓ) position of identity matrix ($k = 1 : n - 1$, $\ell = k + 1 : n$).
 4. At end, result is upper triangular U and unit lower triangular L .
-
- ▶ Even if A invertible, procedure above may not work.
 - ▶ Pivoting required for some matrices (see Lec 6).

Example: Start from square matrix A and an identity matrix

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$$

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$$

Example: Start from square matrix A and an identity matrix

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \leftarrow R_2 - (4/2)R_1, \quad m_{2,1} := 2$$

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$$

Example: Start from square matrix A and an identity matrix

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \quad \leftarrow R_2 - (4/2)R_1, \quad m_{2,1} := 2$$
$$\leftarrow R_3 - (8/2)R_1, \quad m_{3,1} := 4$$

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

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Example: Start from square matrix A and an identity matrix

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \quad \leftarrow R_2 - (4/2)R_1, \quad m_{2,1} := 2$$
$$\leftarrow R_3 - (8/2)R_1, \quad m_{3,1} := 4$$
$$\leftarrow R_4 - (6/2)R_1, \quad m_{4,1} := 3$$

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 1 \\ 3 & 5 & 5 \\ 4 & 6 & 8 \end{bmatrix}$$

$$\begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & & 1 & \\ 3 & & & 1 \end{bmatrix}$$

Example: Start from square matrix A and an identity matrix

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \quad \leftarrow R_2 - (4/2)R_1, \quad m_{2,1} := 2$$
$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \quad \leftarrow R_3 - (8/2)R_1, \quad m_{3,1} := 4$$
$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \quad \leftarrow R_4 - (6/2)R_1, \quad m_{4,1} := 3$$

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 3 & 5 & 5 & 0 \\ 4 & 6 & 8 & 0 \end{bmatrix} \quad \leftarrow R_3 - (3/1)R_2, \quad m_{3,2} := 3$$

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

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 2 & 2 & 0 & 0 \end{bmatrix}$$

Example: Start from square matrix A and an identity matrix

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \quad \leftarrow R_2 - (4/2)R_1, \quad m_{2,1} := 2$$
$$\leftarrow R_3 - (8/2)R_1, \quad m_{3,1} := 4$$
$$\leftarrow R_4 - (6/2)R_1, \quad m_{4,1} := 3$$

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 3 & 5 & 5 & 0 \\ 4 & 6 & 8 & 0 \end{bmatrix} \quad \leftarrow R_3 - (3/1)R_2, \quad m_{3,2} := 3$$
$$\leftarrow R_4 - (4/1)R_2, \quad m_{4,2} := 4$$

$$\begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & 3 & 1 & \\ 3 & 4 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 2 & 2 & & \\ 2 & 4 & & \end{bmatrix}$$

Example: Start from square matrix A and an identity matrix

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \quad \leftarrow R_2 - (4/2)R_1, \quad m_{2,1} := 2$$
$$\leftarrow R_3 - (8/2)R_1, \quad m_{3,1} := 4$$
$$\leftarrow R_4 - (6/2)R_1, \quad m_{4,1} := 3$$

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 3 & 5 & 5 & 0 \\ 4 & 6 & 8 & 0 \end{bmatrix} \quad \leftarrow R_3 - (3/1)R_2, \quad m_{3,2} := 3$$
$$\leftarrow R_4 - (4/1)R_2, \quad m_{4,2} := 4$$

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 2 & 2 & 2 & 0 \\ 2 & 4 & 4 & 0 \end{bmatrix} \quad \leftarrow R_4 - (2/2)R_3, \quad m_{4,3} := 1$$

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

Example: Start from square matrix A and an identity matrix

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \quad \leftarrow R_2 - (4/2)R_1, \quad m_{2,1} := 2$$
$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \quad \leftarrow R_3 - (8/2)R_1, \quad m_{3,1} := 4$$
$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \quad \leftarrow R_4 - (6/2)R_1, \quad m_{4,1} := 3$$

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 3 & 5 & 5 & 0 \\ 4 & 6 & 8 & 0 \end{bmatrix} \quad \leftarrow R_3 - (3/1)R_2, \quad m_{3,2} := 3$$
$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 3 & 5 & 5 & 0 \\ 4 & 6 & 8 & 0 \end{bmatrix} \quad \leftarrow R_4 - (4/1)R_2, \quad m_{4,2} := 4$$

$$L := \begin{bmatrix} 1 & & & \\ 2 & 1 & & \\ 4 & 3 & 1 & \\ 3 & 4 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 2 & 2 & & 0 \\ 2 & 4 & & 0 \end{bmatrix} \quad \leftarrow R_4 - (2/2)R_3, \quad m_{4,3} := 1$$

$$U := \begin{bmatrix} 2 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 2 & 2 & 0 \\ 0 & 0 & 2 & 2 \end{bmatrix}$$

We now have triangular factors L and U such that $LU = A$

Pseudo-code for LU decomposition:

LU decomposition without pivoting

Input: $A \in \mathbb{R}^{n \times n}$

$U \leftarrow A, L \leftarrow I$

for $j = 1 : n - 1$

for $i = j + 1 : n$

$L_{i,j} \leftarrow U_{i,j} / U_{j,j}$

$U_{i,j:n} \leftarrow U_{i,j:n} - L_{i,j} U_{j,j:n}$

(initialise matrices)

(loop through pivot columns)

(store multiplier in L matrix)

(update row i of U matrix)

end for

end for

Output: Matrices L and U

Existence of LU decomposition $A = LU$.

Proposition

For a given nonsingular matrix $A \in \mathbb{R}^{n \times n}$, the LU decomposition $A = LU$ exists and is unique iff all the leading principal submatrices of A are nonsingular.

Note: a *leading submatrix* is obtained from a matrix A by extracting its first k rows and columns: $A(1 : k, 1 : k)$.

- ▶ LU decomposition $A = LU$ has L unit lower triangular and U upper triangular
- ▶ Not always possible to find $A = LU$ for A nonsingular
- ▶ When A nonsingular, **always** possible to find permutation P such that $PA = LU$, i.e., so that PA has a Gauss (LU) factorisation

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. Central questions

2. LU decomposition with partial pivoting

3. Using LU decompositions to solve linear systems

Central questions:

- ▶ For any square $A \in \mathbb{R}^{n \times n}$, does a decomposition $A = LU$ exist?
- ▶ What is (partial) pivoting?
- ▶ What is a permutation matrix? How can it be represented?
- ▶ How is an LU decomposition $PA = LU$ computed?
- ▶ How do LU decompositions $A = LU$ or $PA = LU$ relate to solving linear systems of equations?
- ▶ How can LU decompositions be computed and used in PYTHON?

Existence of LU decomposition $A = LU$.

Proposition

For a given nonsingular matrix $A \in \mathbb{R}^{n \times n}$, the LU decomposition $A = LU$ exists and is unique iff (if and only if) all the leading principal submatrices of A are nonsingular.

Notes:

- ▶ a *leading submatrix* is obtained from a matrix A by extracting its first k rows and columns: $A(1 : k, 1 : k)$.
- ▶ LU decomposition $A = LU$ has L unit lower triangular and U upper triangular
- ▶ Not always possible to find $A = LU$ for A nonsingular
- ▶ When A nonsingular, **always** possible to find permutation P such that $PA = LU$, i.e., so that PA has an LU decomposition, also called a Gauss factorisation

A *leading submatrix* is obtained from a matrix A by extracting its first k rows and columns: $A(1 : k, 1 : k)$.

$$A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$$

has submatrices:

$$A(1, 1) = [2], \quad A(1 : 2, 1 : 2) = \begin{bmatrix} 2 & 1 \\ 4 & 3 \end{bmatrix},$$

$$A(1 : 3, 1 : 3) = \begin{bmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{bmatrix}, \text{ and } A \text{ itself}$$

Permutation matrices:

Definition (Permutation matrix)

A **permutation matrix** is any matrix obtained from interchanging the rows or columns of the identity matrix.

► e.g., $P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, $P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$, $P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$

- For any permutation matrix, $P^{-1} = P^T$
- Multiplication by permutation matrix can be used to permute rows or columns of a matrix A :
 - PA permutes **rows** of matrix A
 - AP^T permutes **columns** of matrix A
- Permutation matrices can be stored as single vector

Example: left multiplication by permutation matrix permutes rows

$$PA = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \end{bmatrix}$$

Example: right multiplication by transpose of permutation matrix permutes columns

$$AP^T = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 \\ 5 & 6 & 4 \\ 8 & 9 & 7 \end{bmatrix}$$

Recall:

In Gaussian elimination, **pivot element** on diagonal used to zero out entries

$$\text{pivot} = A_{k,k} \quad (k = 1 : n - 1)$$

Definition: Pivoting

Using permutations of the rows and/or columns of a matrix to change a pivot element

An example of the need for pivoting:

- ▶ Not every invertible matrix A has LU decomposition

$$A = LU$$

An example of the need for pivoting:

- ▶ Not every invertible matrix A has LU decomposition
- $$A = LU$$

Example (Pivoting mandatory)

- ▶ $A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \Rightarrow a_{1,1} = 0$ prevents direct elimination: $m_{2,1} = ?$

An example of the need for pivoting:

- ▶ Not every invertible matrix A has LU decomposition
- $$A = LU$$

Example (Pivoting mandatory)

- ▶ $A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \Rightarrow a_{1,1} = 0$ prevents direct elimination: $m_{2,1} = ?$
- ▶ Instead, use permutation matrix to swap rows 1 & 2

$$PA = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad \text{if} \quad P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

An example of the need for pivoting:

- ▶ Not every invertible matrix A has LU decomposition
$$A = LU$$

Example (Pivoting mandatory)

- ▶ $A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \Rightarrow a_{1,1} = 0$ prevents direct elimination: $m_{2,1} = ?$
- ▶ Instead, use permutation matrix to swap rows 1 & 2

$$PA = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad \text{if} \quad P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

- ▶ PA is upper triangular, so LU decomposition is $PA = LU$
with $P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, $L = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $U = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$

Further need for pivoting:

Example (Pivoting numerically useful)

Further need for pivoting:

Example (Pivoting numerically useful)

- ▶ Let $A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}$, so exact LU factors are

$$A = LU = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix} \begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{bmatrix}$$

Further need for pivoting:

Example (Pivoting numerically useful)

- ▶ Let $A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}$, so exact LU factors are

$$A = LU = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix} \begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{bmatrix}$$

- ▶ In finite precision arithmetic, $1 - 10^{20} \simeq -10^{20}$, so

$$\tilde{L} = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix} \text{ and } \tilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{bmatrix}$$

Further need for pivoting:

Example (Pivoting numerically useful)

- ▶ Let $A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}$, so exact LU factors are

$$A = LU = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix} \begin{bmatrix} 10^{-20} & 1 \\ 0 & 1 - 10^{20} \end{bmatrix}$$

- ▶ In finite precision arithmetic, $1 - 10^{20} \simeq -10^{20}$, so

$$\tilde{L} = \begin{bmatrix} 1 & 0 \\ 10^{20} & 1 \end{bmatrix} \text{ and } \tilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{bmatrix}$$

- ▶ Notice $\tilde{L} = L$ and $\tilde{U} \simeq U$ but $\tilde{L}\tilde{U} = \begin{bmatrix} 10^{-20} & 1 \\ 1 & \textcolor{red}{0} \end{bmatrix} \neq A$

We want to find P, L, U , such that $PA = LU$.

→ Use LU with partial pivoting.

Partial pivoting refers to swapping rows, but not columns:

- ▶ Initialise $U = A$, $L = I$, and $P = I$
- ▶ Loop through pivot columns ($j = 1 : n - 1$):
 - ▶ Find **largest entry** in pivot column j , say $U_{k,j}$, $k \geq j$
 - ▶ Interchange rows $j \leftrightarrow k$ in P and U
 - ▶ Interchange rows $j \leftrightarrow k$ in L **up to (not including) pivot column**, i.e. just the first $j - 1$ columns
 - ▶ Eliminate rows below row j as usual (Gaussian elimination)
- ▶ End result is three matrices P , L , and U such that $PA = LU$

Pseudo-code for LU decomposition with partial pivoting

Input: $A \in \mathbb{R}^{n \times n}$

$U \leftarrow A, L \leftarrow I, P \leftarrow I$

for $j = 1 : n - 1$

 Select $k \geq j$ to maximise $|U_{k,j}|$

$U_{j,j:n} \leftrightarrow U_{k,j:n}$

$L_{j,1:j-1} \leftrightarrow L_{k,1:j-1}$

$P_{j,:} \leftrightarrow P_{k,:}$

for $i = j + 1 : n$

$L_{i,j} \leftarrow U_{i,j}/U_{j,j}$

$U_{i,j:n} \leftarrow U_{i,j:n} - L_{i,j}U_{j,j:n}$

end for

end for

Output: Matrices L , U and P

(initialise matrices)

(loop through pivot columns)

(choose pivot element)

(swap rows of U)

(swap rows of L up to pivot)

(swap rows of P)

(loop through rows under pivot)

(store multiplier in L matrix)

(update row j of U matrix)

Example:

Given $A = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix}$, construct the LU decomposition with partial pivoting, i.e. find $PA = LU$.

Example:

Given $A = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix}$, construct the LU decomposition with partial pivoting, i.e. find $PA = LU$.

$$U = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix} \quad P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Initialise matrices; pivot column $k = 1$

Example:

Given $A = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix}$, construct the LU decomposition with partial pivoting, i.e. find $PA = LU$.

$$U = \begin{bmatrix} 3 & 6 & 4 \\ 1 & 1 & 3 \\ 2 & 2 & 2 \end{bmatrix} \quad P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$R_1 \leftrightarrow R_2$ $R_1 \leftrightarrow R_2$

Swap rows 1 and 2 in U and P

Example:

Given $A = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix}$, construct the LU decomposition with partial pivoting, i.e. find $PA = LU$.

$$U = \begin{bmatrix} 3 & 6 & 4 \\ -1 & \frac{5}{3} & \\ -2 & -\frac{2}{3} & \end{bmatrix} \quad P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{3} & 1 & 0 \\ \frac{2}{3} & 0 & 1 \end{bmatrix}$$

$$R_2 \leftarrow R_2 - \frac{1}{3}R_1$$

$$R_3 \leftarrow R_3 - \frac{2}{3}R_1$$

$$L_{2,1} \leftarrow \frac{1}{3}$$

$$L_{3,1} \leftarrow \frac{2}{3}$$

Eliminate beneath pivot column $k = 1$ in U , store multipliers in L

Example:

Given $A = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix}$, construct the LU decomposition with partial pivoting, i.e. find $PA = LU$.

$$U = \begin{bmatrix} 3 & 6 & 4 \\ -2 & -\frac{2}{3} & \\ -1 & \frac{5}{3} & \end{bmatrix}$$

$$R_2 \leftrightarrow R_3$$

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

$$R_2 \leftrightarrow R_3$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{2}{3} & 1 & 0 \\ \frac{1}{3} & 0 & 1 \end{bmatrix}$$

$$R_2 \leftrightarrow R_3$$

Swap rows 2 and 3 in U and P ; swap up to pivot column $k = 2$ in L

Example:

Given $A = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix}$, construct the LU decomposition with partial pivoting, i.e. find $PA = LU$.

$$U = \begin{bmatrix} 3 & 6 & 4 \\ -2 & -\frac{2}{3} & \textcolor{red}{2} \\ \textcolor{red}{2} & 0 & 0 \end{bmatrix} \quad P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{2}{3} & 1 & 0 \\ \frac{1}{3} & \frac{1}{2} & 1 \end{bmatrix}$$
$$R_3 \leftarrow R_3 - \frac{1}{2}R_2 \quad L_{3,2} \leftarrow \frac{1}{2}$$

Eliminate beneath pivot column $k = 2$ in U , store multiplier in L

Example:

Given $A = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix}$, construct the LU decomposition with partial pivoting, i.e. find $PA = LU$.

$$U = \begin{bmatrix} 3 & 6 & 4 \\ -2 & -\frac{2}{3} & 2 \end{bmatrix} \quad P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{2}{3} & 1 & 0 \\ \frac{1}{3} & \frac{1}{2} & 1 \end{bmatrix}$$

We arrive at the decomposition $PA = LU$.

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \quad P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Initialise matrices; pivot column $k = 1$

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 0 \\ 6 & 7 & 9 & 8 \end{bmatrix} \quad P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$R_1 \leftrightarrow R_3$ $R_1 \leftrightarrow R_3$

Swap rows 1 and 3 in U and P

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} & -\frac{3}{2} \\ -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} & -\frac{5}{4} \\ \frac{7}{4} & \frac{9}{4} & \frac{17}{4} & \end{bmatrix} \quad P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & 1 & 0 & 0 \\ \frac{1}{4} & 0 & 1 & 0 \\ \frac{3}{4} & 0 & 0 & 1 \end{bmatrix}$$

$$R_2 \leftarrow R_2 - \frac{1}{2}R_1$$

$$L_{2,1} \leftarrow \frac{1}{2}$$

$$R_3 \leftarrow R_3 - \frac{1}{4}R_1$$

$$L_{3,1} \leftarrow \frac{1}{4}$$

$$R_4 \leftarrow R_4 - \frac{3}{4}R_1$$

$$L_{4,1} \leftarrow \frac{3}{4}$$

Eliminate beneath pivot column $k = 1$ in U , store multipliers in L

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ \frac{7}{4} & \frac{9}{4} & \frac{17}{4} & \\ -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} & \\ -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} & \end{bmatrix} \quad P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{4} & 0 & 1 & 0 \\ \frac{1}{2} & 0 & 0 & 1 \end{bmatrix}$$

$R_2 \leftrightarrow R_4$ $R_2 \leftrightarrow R_4$ $R_2 \leftrightarrow R_4$

Swap rows 2 and 4 in U and P ; swap up to pivot column $k = 2$ in L

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ \frac{7}{4} & \frac{9}{4} & \frac{17}{4} & \frac{4}{7} \\ -\frac{2}{7} & -\frac{4}{7} & -\frac{6}{7} & -\frac{2}{7} \end{bmatrix} \quad P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{4} & -\frac{3}{7} & 1 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 0 & 1 \end{bmatrix}$$

$$R_3 \leftarrow R_3 + \frac{3}{7}R_2$$

$$R_4 \leftarrow R_4 + \frac{2}{7}R_2$$

$$L_{3,2} \leftarrow -\frac{3}{7}$$

$$L_{4,2} \leftarrow -\frac{2}{7}$$

Eliminate beneath pivot column $k = 2$ in U , store multipliers in L

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ \frac{7}{4} & \frac{9}{4} & \frac{17}{4} & -\frac{2}{7} \\ -\frac{6}{7} & -\frac{2}{7} & \frac{4}{7} & \end{bmatrix} \quad P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 1 & 0 \\ \frac{1}{4} & -\frac{3}{7} & 0 & 1 \end{bmatrix}$$

$R_3 \leftrightarrow R_4$ $R_3 \leftrightarrow R_4$ $R_3 \leftrightarrow R_4$

Swap rows 3 and 4 in U and P ; swap up to pivot column $k = 3$ in L

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ \frac{7}{4} & \frac{9}{4} & \frac{17}{4} & -\frac{2}{7} \\ -\frac{6}{7} & \frac{2}{3} & \frac{2}{3} & \frac{2}{3} \end{bmatrix} \quad P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 1 & 0 \\ \frac{1}{4} & -\frac{3}{7} & \frac{1}{3} & 1 \end{bmatrix}$$
$$R_4 \leftarrow R_4 - \frac{1}{3}R_3 \qquad \qquad \qquad L_{4,3} \leftarrow \frac{1}{3}$$

Eliminate beneath pivot column $k = 3$ in U , store multiplier in L

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ \frac{7}{4} & \frac{9}{4} & \frac{17}{4} & -\frac{2}{7} \\ -\frac{6}{7} & \frac{2}{3} & \end{bmatrix} \quad P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 1 & 0 \\ \frac{1}{4} & -\frac{3}{7} & \frac{1}{3} & 1 \end{bmatrix}$$

We arrive at the decomposition $PA = LU$

Solving $Ax = b$ from $PA = LU$

- ▶ Multiply by P :

$$PAx = Pb$$

- ▶ Replace PA by LU :

$$LUx = Pb$$

- ▶ Define $y = Ux$ and solve

$$Ly = Pb$$

for y by forward substitution.

- ▶ Solve

$$Ux = y$$

for x by backward substitution.

And we're done.

Using `scipy.linalg`:

```
import numpy as np
import scipy.linalg

A = np.array([[0.2, 1.4, -0.4, 12.3],
              [-2.3, 4.2, 1.1, -0.9],
              [9.2, -2.3, -0.1, 2.2],
              [3.4, 3.3, -10.1, 4.0]])
b = np.array([[0.2], [-1.2], [3.3], [0.2]])
Pt, L, U = scipy.linalg.lu(A)
Pb = np.matmul(Pt.T, b)
y = scipy.linalg.solve_triangular(L, Pb, lower=True)
x = scipy.linalg.solve_triangular(U, y, lower=False)
```

- ▶ `scipy.linalg.solve` computes solution directly.
- ▶ Better to compute LU decomposition first if you have many systems to solve with different right hand sides.

Remarks

- ▶ **Partial pivoting**: interchanging rows to use small multipliers.
- ▶ Partial pivoting: multipliers $L_{k,\ell}$ satisfy $|L_{k,\ell}| \leq 1$.
- ▶ Pivoting stabilises differences in elimination.
- ▶ Complete pivoting: swapping rows **and** columns.
- ▶ $PA = LU$ decomposition is the default way to solve linear systems.

Representing permutation matrices by a vector:

- ▶ Unnecessary to store entire matrix P ; single vector suffices
- ▶ Represent permutation matrix P by column vector \mathbf{p}
- ▶ Initialise $\mathbf{p} = (1 : n)^T$
- ▶ Swap rows of \mathbf{p} at each pivot step of Gaussian elimination
- ▶ End result: vector \mathbf{p} with order of rows of identity matrix to give P

$$P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \rightarrow \quad \mathbf{p} = \begin{bmatrix} 3 \\ 4 \\ 2 \\ 1 \end{bmatrix}$$

Exercise:

Given $A = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix}$, construct the LU decomposition

$$PA = LU.$$

Exercise:

Given $A = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix} \quad P = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Initialise matrices; pivot column $k = 1$

Exercise:

Given $A = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix}$, construct the LU decomposition

$$PA = LU.$$

$$U = \begin{bmatrix} 3 & 6 & 4 \\ 1 & 1 & 3 \\ 2 & 2 & 2 \end{bmatrix}$$

$$R_1 \leftrightarrow R_2$$

$$\mathbf{p} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}$$

$$R_1 \leftrightarrow R_2$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Swap rows 1 and 2 in U and \mathbf{p}

Exercise:

Given $A = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix}$, construct the LU decomposition

$$PA = LU.$$

$$U = \begin{bmatrix} 3 & 6 & 4 \\ -1 & \frac{5}{3} & \\ -2 & -\frac{2}{3} & \end{bmatrix}$$

$$R_2 \leftarrow R_2 - \frac{1}{3}R_1$$
$$R_3 \leftarrow R_3 - \frac{2}{3}R_1$$

$$\mathbf{p} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{3} & 1 & 0 \\ \frac{2}{3} & 0 & 1 \end{bmatrix}$$

$$L_{2,1} \leftarrow \frac{1}{3}$$
$$L_{3,1} \leftarrow \frac{2}{3}$$

Eliminate beneath pivot column $k = 1$ in U , store multipliers in L

Exercise:

Given $A = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix}$, construct the LU decomposition

$$PA = LU.$$

$$U = \begin{bmatrix} 3 & 6 & 4 \\ -2 & -\frac{2}{3} & \\ -1 & \frac{5}{3} & \end{bmatrix}$$

$$R_2 \leftrightarrow R_3$$

$$\mathbf{p} = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}$$

$$R_2 \leftrightarrow R_3$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{2}{3} & 1 & 0 \\ \frac{1}{3} & 0 & 1 \end{bmatrix}$$

$$R_2 \leftrightarrow R_3$$

Swap rows 2 and 3 in U and \mathbf{p} ; swap up to pivot column $k = 2$ in L

Exercise:

Given $A = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 3 & 6 & 4 \\ -2 & -\frac{2}{3} & \textcolor{red}{2} \\ \textcolor{red}{R}_3 \leftarrow R_3 - \frac{1}{2}R_2 \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{2}{3} & 1 & 0 \\ \frac{1}{3} & \frac{1}{2} & 1 \end{bmatrix}$$

Eliminate beneath pivot column $k = 2$ in U , store multiplier in L

Exercise:

Given $A = \begin{bmatrix} 1 & 1 & 3 \\ 3 & 6 & 4 \\ 2 & 2 & 2 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 3 & 6 & 4 \\ -2 & -\frac{2}{3} & 2 \end{bmatrix} \quad P = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{2}{3} & 1 & 0 \\ \frac{1}{3} & \frac{1}{2} & 1 \end{bmatrix}$$

We arrive at the decomposition $PA = LU$.

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix} \quad P = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Initialise matrices; pivot column $k = 1$

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 0 \\ 6 & 7 & 9 & 8 \end{bmatrix} \quad P = \begin{bmatrix} 3 \\ 2 \\ 1 \\ 4 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$R_1 \leftrightarrow R_3$ $R_1 \leftrightarrow R_3$

Swap rows 1 and 3 in U and P

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} & \\ -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} & \\ \frac{7}{4} & \frac{9}{4} & \frac{17}{4} & \end{bmatrix} \quad p = \begin{bmatrix} 3 \\ 2 \\ 1 \\ 4 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & 1 & 0 & 0 \\ \frac{1}{4} & 0 & 1 & 0 \\ \frac{3}{4} & 0 & 0 & 1 \end{bmatrix}$$

$$R_2 \leftarrow R_2 - \frac{1}{2}R_1$$

$$R_3 \leftarrow R_3 - \frac{1}{4}R_1$$

$$R_4 \leftarrow R_4 - \frac{3}{4}R_1$$

$$L_{2,1} \leftarrow \frac{1}{2}$$

$$L_{3,1} \leftarrow \frac{1}{4}$$

$$L_{4,1} \leftarrow \frac{3}{4}$$

Eliminate beneath pivot column $k = 1$ in U , store multipliers in L

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ \frac{7}{4} & \frac{9}{4} & \frac{17}{4} & \\ -\frac{3}{4} & -\frac{5}{4} & -\frac{5}{4} & \\ -\frac{1}{2} & -\frac{3}{2} & -\frac{3}{2} & \end{bmatrix} \quad P = \begin{bmatrix} 3 \\ 4 \\ 1 \\ 2 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{4} & 0 & 1 & 0 \\ \frac{1}{2} & 0 & 0 & 1 \end{bmatrix}$$

$R_2 \leftrightarrow R_4$ $R_2 \leftrightarrow R_4$ $R_2 \leftrightarrow R_4$

Swap rows 2 and 4 in U and P ; swap up to pivot column $k = 2$ in L

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ \frac{7}{4} & \frac{9}{4} & \frac{17}{4} & \frac{4}{7} \\ -\frac{2}{7} & -\frac{4}{7} & -\frac{6}{7} & -\frac{2}{7} \\ -\frac{6}{7} & -\frac{2}{7} & -\frac{6}{7} & -\frac{2}{7} \end{bmatrix} \quad P = \begin{bmatrix} 3 \\ 4 \\ 1 \\ 2 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{4} & -\frac{3}{7} & 1 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} R_3 &\leftarrow R_3 + \frac{3}{7}R_2 \\ R_4 &\leftarrow R_4 + \frac{2}{7}R_2 \end{aligned}$$

$$\begin{aligned} L_{3,2} &\leftarrow -\frac{3}{7} \\ L_{4,2} &\leftarrow -\frac{2}{7} \end{aligned}$$

Eliminate beneath pivot column $k = 2$ in U , store multipliers in L

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ \frac{7}{4} & \frac{9}{4} & \frac{17}{4} & -\frac{2}{7} \\ -\frac{6}{7} & -\frac{2}{7} & \frac{4}{7} & \end{bmatrix} \quad \mathbf{p} = \begin{bmatrix} 3 \\ 4 \\ 2 \\ 1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 1 & 0 \\ \frac{1}{4} & -\frac{3}{7} & 0 & 1 \end{bmatrix}$$

$R_3 \leftrightarrow R_4$ $R_3 \leftrightarrow R_4$ $R_3 \leftrightarrow R_4$

Swap rows 3 and 4 in U and \mathbf{p} ; swap up to pivot column $k = 3$ in L

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ \frac{7}{4} & \frac{9}{4} & \frac{17}{4} & \frac{2}{3} \\ -\frac{6}{7} & -\frac{2}{7} & \frac{2}{3} \end{bmatrix} \quad P = \begin{bmatrix} 3 \\ 4 \\ 2 \\ 1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 1 & 0 \\ \frac{1}{4} & -\frac{3}{7} & \frac{1}{3} & 1 \end{bmatrix}$$
$$R_4 \leftarrow R_4 - \frac{1}{3}R_3 \qquad \qquad \qquad L_{4,3} \leftarrow \frac{1}{3}$$

Eliminate beneath pivot column $k = 3$ in U , store multiplier in L

Exercise:

Given $A = \begin{bmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{bmatrix}$, construct the LU decomposition
 $PA = LU$.

$$U = \begin{bmatrix} 8 & 7 & 9 & 5 \\ \frac{7}{4} & \frac{9}{4} & \frac{17}{4} & -\frac{2}{7} \\ -\frac{6}{7} & -\frac{2}{7} & \frac{2}{3} \end{bmatrix} \quad P = \begin{bmatrix} 3 \\ 4 \\ 2 \\ 1 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{2} & -\frac{2}{7} & 1 & 0 \\ \frac{1}{4} & -\frac{3}{7} & \frac{1}{3} & 1 \end{bmatrix}$$

We arrive at the decomposition $PA = LU$

Summary

- ▶ Diagonal & triangular systems are simple to solve
- ▶ Gaussian elimination: converting square matrix to triangular form
- ▶ Gauss factorisation: finding L, U such that $A = LU$
- ▶ Permutation matrices (representation as matrices and vectors)
- ▶ Gaussian elimination with partial pivoting always possible
- ▶ Pivoting reduces magnitude of multipliers, stabilising elimination
- ▶ Decomposition $PA = LU$ useful for repeated solves

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. The three questions...
2. Vector norms
3. Quantifying errors using norms
4. Conditioning of linear equations

Central questions:

- ▶ What is a **vector norm**?
- ▶ What are some examples of norms?
- ▶ How are vector norms computed in NumPy / SciPy?
- ▶ How are norms useful in quantifying errors in solving linear systems?
- ▶ What are the **singular values** of a matrix?
- ▶ What is the condition number of a matrix? Computing it with **NumPy**.
- ▶ What does conditioning mean intuitively?

The *three questions* for approximation solutions to linear systems:

1. When does my computation work?
2. How accurate is the result?
3. How fast does my computation work?

Answer:

The *three questions* for approximation solutions to linear systems:

1. When does my computation work?
2. How accurate is the result?
3. How fast does my computation work?

Answer:

- ▶ The $A = LU$ decomposition works if and only if all leading principal submatrices of A (i.e. $A(1 : k, 1 : k)$ for $k \leq n$) are nonsingular. **Not recommended for linear solving!**

The *three questions* for approximation solutions to linear systems:

1. When does my computation work?
2. How accurate is the result?
3. How fast does my computation work?

Answer:

- ▶ The $A = LU$ decomposition works if and only if all leading principal submatrices of A (i.e. $A(1 : k, 1 : k)$ for $k \leq n$) are nonsingular. **Not recommended for linear solving!**
- ▶ The $PA = LU$ decomposition works if A is nonsingular. This is the default method for linear solving:

step 1: solve $Ly = Pb$ using forward substitution

step 2: solve $Ux = y$ using backward substitution

Next, we turn to the second question...

Motivation for vector norms

If solutions to linear systems are vectors, how can you tell how big the error is?

- ▶ Real numbers are **ordered**:

given $a, b \in \mathbb{R}$, either $a < b$, $a > b$, or $a = b$

- ▶ Vectors in \mathbb{R}^n are **not** ordered, e.g., expressions like

$$\begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} > \begin{pmatrix} -1 \\ 1 \\ -1 \end{pmatrix} \text{ or } \begin{pmatrix} -1 \\ 2 \\ -1 \end{pmatrix} < \begin{pmatrix} 1 \\ -1 \\ 3 \end{pmatrix}$$

do not make sense.

- ▶ **Norms** provide a way to order vectors, measure distance.

Definition (Vector norm)

Given a vector space V , a **norm** is a function $\|\cdot\| : V \rightarrow [0, \infty)$ satisfying three postulates:

1. $\|\mathbf{v}\| > 0$ if $\mathbf{v} \neq \mathbf{0}$ for every $\mathbf{v} \in V$
2. $\|\lambda\mathbf{v}\| = |\lambda|\|\mathbf{v}\|$ for every $\lambda \in \mathbb{R}$, $\mathbf{v} \in V$
3. $\|\mathbf{u} + \mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\|$ for every $\mathbf{u}, \mathbf{v} \in V$
(triangle inequality)

Definition (Vector norm)

Given a vector space V , a **norm** is a function $\|\cdot\| : V \rightarrow [0, \infty)$ satisfying three postulates:

1. $\|\mathbf{v}\| > 0$ if $\mathbf{v} \neq \mathbf{0}$ for every $\mathbf{v} \in V$
2. $\|\lambda\mathbf{v}\| = |\lambda|\|\mathbf{v}\|$ for every $\lambda \in \mathbb{R}$, $\mathbf{v} \in V$
3. $\|\mathbf{u} + \mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\|$ for every $\mathbf{u}, \mathbf{v} \in V$
(triangle inequality)

- ▶ $\|\mathbf{x}\|$ provides notion of **length** or **size** of vector \mathbf{x} .

Definition (Vector norm)

Given a vector space V , a **norm** is a function $\|\cdot\| : V \rightarrow [0, \infty)$ satisfying three postulates:

1. $\|\mathbf{v}\| > 0$ if $\mathbf{v} \neq \mathbf{0}$ for every $\mathbf{v} \in V$
2. $\|\lambda\mathbf{v}\| = |\lambda|\|\mathbf{v}\|$ for every $\lambda \in \mathbb{R}$, $\mathbf{v} \in V$
3. $\|\mathbf{u} + \mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\|$ for every $\mathbf{u}, \mathbf{v} \in V$
(triangle inequality)

- ▶ $\|\mathbf{x}\|$ provides notion of **length** or **size** of vector \mathbf{x} .
- ▶ $\|\mathbf{x} - \mathbf{y}\|$ provides notion of **distance** between vectors \mathbf{x}, \mathbf{y} .

The ℓ_2 -norm

$$\underline{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

$$\|\mathbf{x}\|_2 := \left[\sum_{k=1}^n |x_k|^2 \right]^{\frac{1}{2}} \quad \forall \mathbf{x} \in \mathbb{R}^n$$

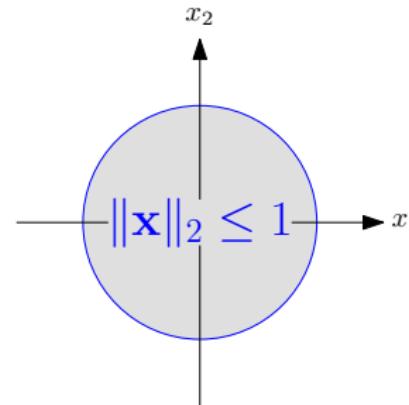
- ▶ Also called **Euclidean norm** or **2-norm**.
- ▶ Compute by `scipy.linalg.norm(x, 2)`.

$$\left\| \left[3, -4, 0, \frac{3}{2} \right]^T \right\|_2 = \sqrt{(3)^2 + (-4)^2 + (0)^2 + \left(\frac{3}{2}\right)^2} = \boxed{\frac{1}{2}\sqrt{109}}$$

$$\left\| [2, 1, -3, 4]^T \right\|_2 = \sqrt{(2)^2 + (1)^2 + (-3)^2 + (4)^2} = \boxed{\sqrt{30}}$$

ℓ_2 -norm in \mathbb{R}^2 :

$$\|\mathbf{x}\|_2 = \left(\sum_{k=1}^n |x_k|^2 \right)^{\frac{1}{2}}$$



Unit ball in ℓ_2 -norm = set of all vectors $\mathbf{x} \in \mathbb{R}^2$ with $\|\mathbf{x}\|_2 \leq 1$

$$= \{\mathbf{x} \in \mathbb{R}^2 \mid \|\mathbf{x}\|_2 \leq 1\}$$

$$= \{(x_1, x_2) \in \mathbb{R}^2 \mid \sqrt{|x_1|^2 + |x_2|^2} \leq 1\}$$

$$= \{(x_1, x_2) \in \mathbb{R}^2 \mid x_1^2 + x_2^2 \leq 1\}$$

= circle of radius 1 centred at origin

The ℓ_1 -norm

$$\|\mathbf{x}\|_1 := \sum_{k=1}^n |x_k| \quad \forall \mathbf{x} \in \mathbb{R}^n$$

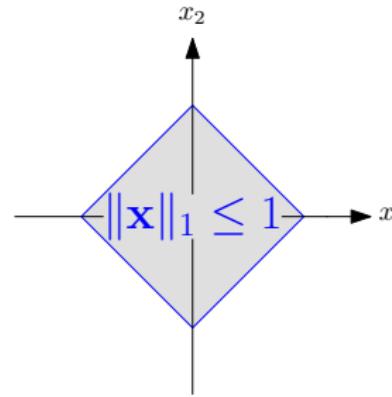
- ▶ Also called **Manhattan norm** or **1-norm**.
- ▶ Compute by `scipy.linalg.norm(x, 1)`.

$$\left\| \begin{bmatrix} 3, -4, 0, \frac{3}{2} \end{bmatrix}^T \right\|_1 = |3| + |-4| + |0| + \left| \frac{3}{2} \right| = \boxed{\frac{17}{2}}$$

$$\left\| [2, 1, -3, 4]^T \right\|_1 = |2| + |1| + |-3| + |4| = \boxed{10}$$

ℓ_1 -norm in \mathbb{R}^2 :

$$\|\mathbf{x}\|_1 = \sum_{k=1}^n |x_k|$$



Unit ball in ℓ_1 -norm = set of all vectors $\mathbf{x} \in \mathbb{R}^2$ with $\|\mathbf{x}\|_1 \leq 1$

$$= \{\mathbf{x} \in \mathbb{R}^2 \mid \|\mathbf{x}\|_1 \leq 1\}$$

$$= \{(x_1, x_2) \in \mathbb{R}^2 \mid |x_1| + |x_2| \leq 1\}$$

$$= \{(x_1, x_2) \in \mathbb{R}^2 \mid (\pm x_1) + (\pm x_2) \leq 1\}$$

= square with vertices $(\pm 1, 0)$, $(0, \pm 1)$

The ℓ_∞ -norm

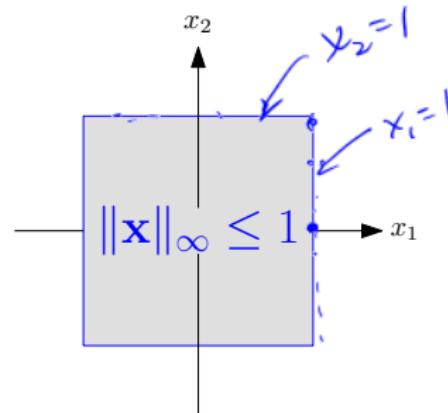
$$\|\mathbf{x}\|_\infty := \max_{\textcolor{blue}{n}} (|x_1|, |x_2|, \dots, |x_n|) \quad \forall \mathbf{x} \in \mathbb{R}^n$$

- ▶ Also called max/infinity/Chebyschev norm.
- ▶ Compute by `scipy.linalg.norm(x, scipy.inf)`.

$$\left\| \begin{bmatrix} 3, -4, 0, \frac{3}{2} \end{bmatrix}^T \right\|_\infty = \max \left(|3|, |-4|, |0|, \left| \frac{3}{2} \right| \right) = \boxed{4}$$
$$\left\| [2, 1, -3, 4]^T \right\|_\infty = \max (|2|, |1|, |-3|, |4|) = \boxed{4}$$

ℓ_∞ -norm in \mathbb{R}^2 :

$$\|\mathbf{x}\|_\infty = \max_{1 \leq k \leq n} |x_k|$$



Unit ball in ℓ_∞ -norm = set of all vectors $\mathbf{x} \in \mathbb{R}^2$ with $\|\mathbf{x}\|_\infty \leq 1$

$$= \{\mathbf{x} \in \mathbb{R}^2 \mid \|\mathbf{x}\|_\infty \leq 1\}$$

$$= \{(x_1, x_2) \in \mathbb{R}^2 \mid \max\{|x_1|, |x_2|\} \leq 1\}$$

$$= \{(x_1, x_2) \in \mathbb{R}^2 \mid |x_1| \leq 1 \text{ and } |x_2| \leq 1\}$$

= square with vertices $(\pm 1, \pm 1)$

The ℓ_p -norm ($p \geq 1$)

$$\|\mathbf{x}\|_p := \left[\sum_{k=1}^n |x_k|^p \right]^{\frac{1}{p}} \quad \forall \mathbf{x} \in \mathbb{R}^n$$

- ▶ Generalises norms observed so far.
- ▶ Compute by `scipy.linalg.norm(x, p)`.

$$\left\| \begin{bmatrix} 3, -4, 0, \frac{3}{2} \end{bmatrix}^T \right\|_4 = \sqrt[4]{|3|^4 + |-4|^4 + |0|^4 + \left| \frac{3}{2} \right|^4} = \boxed{\frac{1}{2} \sqrt[4]{5473}}$$
$$\left\| [2, 1, -3, 4]^T \right\|_3 = \sqrt[3]{|2|^3 + |1|^3 + |-3|^3 + |4|^2} = \boxed{\sqrt[3]{100}}$$

Norms and relative errors

- ▶ Help quantify the second of the three question:
 1. When does my computation work?
 - 2. **How accurate is the result?**
 3. How fast does my computation work?
- ▶ $\|\mathbf{x} - \mathbf{x}_*\|$ small means $\mathbf{x}_* \in \mathbb{R}^n$ approximates $\mathbf{x} \in \mathbb{R}^n$ well.

Norms and relative errors

- ▶ Help quantify the second of the three question:
 1. When does my computation work?
 2. How accurate is the result?
 3. How fast does my computation work?
- ▶ $\|\mathbf{x} - \mathbf{x}_*\|$ small means $\mathbf{x}_* \in \mathbb{R}^n$ approximates $\mathbf{x} \in \mathbb{R}^n$ well.
- ▶ Define **relative error of \mathbf{x}_* as an approximation of \mathbf{x} :**

$$\text{Relative error of } \mathbf{x}_* \text{ in norm } \|\cdot\| := \frac{\|\mathbf{x} - \mathbf{x}_*\|}{\|\mathbf{x}\|} \quad (\text{assuming } \mathbf{x} \neq \mathbf{0})$$

Norms and relative errors

- ▶ Help quantify the second of the three question:
 1. When does my computation work?
 2. How accurate is the result?
 3. How fast does my computation work?
- ▶ $\|\mathbf{x} - \mathbf{x}_*\|$ small means $\mathbf{x}_* \in \mathbb{R}^n$ approximates $\mathbf{x} \in \mathbb{R}^n$ well.
- ▶ Define **relative error of \mathbf{x}_* as an approximation of \mathbf{x} :**

$$\text{Relative error of } \mathbf{x}_* := \frac{\|\mathbf{x} - \mathbf{x}_*\|}{\|\mathbf{x}\|} \quad (\text{assuming } \mathbf{x} \neq \mathbf{0})$$

in norm $\|\cdot\|$

- ▶ Computing (relative) error requires choosing a norm.

Norms and relative errors

- ▶ Help quantify the second of the three question:
 1. When does my computation work?
 2. How accurate is the result?
 3. How fast does my computation work?
- ▶ $\|\mathbf{x} - \mathbf{x}_*\|$ small means $\mathbf{x}_* \in \mathbb{R}^n$ approximates $\mathbf{x} \in \mathbb{R}^n$ well.
- ▶ Define **relative error of \mathbf{x}_* as an approximation of \mathbf{x} :**

$$\text{Relative error of } \mathbf{x}_* := \frac{\|\mathbf{x} - \mathbf{x}_*\|}{\|\mathbf{x}\|} \quad (\text{assuming } \mathbf{x} \neq \mathbf{0})$$

in norm $\|\cdot\|$

- ▶ Computing (relative) error requires choosing a norm.
- ▶ Norm-wise errors can hide component-wise errors in vectors.

Example

Compute relative errors in the ∞ -norm, 1-norm, and 2-norm norms of \mathbf{x}_* as an approximation of \mathbf{x} if

$$\mathbf{x} = \begin{pmatrix} 1.0000 \\ 0.0100 \\ 0.0001 \end{pmatrix} \text{ and } \mathbf{x}_* = \begin{pmatrix} 1.0002 \\ 0.0103 \\ 0.0002 \end{pmatrix}$$

Example

Compute relative errors in the ∞ -norm, 1-norm, and 2-norm norms of \mathbf{x}_* as an approximation of \mathbf{x} if

$$\mathbf{x} = \begin{pmatrix} 1.0000 \\ 0.0100 \\ 0.0001 \end{pmatrix} \text{ and } \mathbf{x}_* = \begin{pmatrix} 1.0002 \\ 0.0103 \\ 0.0002 \end{pmatrix}$$

$$\Rightarrow \mathbf{e} := \mathbf{x} - \mathbf{x}_* = \begin{pmatrix} -0.0002 \\ -0.0003 \\ -0.0001 \end{pmatrix} = -10^{-4} \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}$$

Example

Compute relative errors in the ∞ -norm, 1-norm, and 2-norm norms of \mathbf{x}_* as an approximation of \mathbf{x} if

$$\mathbf{x} = \begin{pmatrix} 1.0000 \\ 0.0100 \\ 0.0001 \end{pmatrix} \text{ and } \mathbf{x}_* = \begin{pmatrix} 1.0002 \\ 0.0103 \\ 0.0002 \end{pmatrix}$$

$$\Rightarrow \mathbf{e} := \mathbf{x} - \mathbf{x}_* = \begin{pmatrix} -0.0002 \\ -0.0003 \\ -0.0001 \end{pmatrix} = -10^{-4} \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}$$

Absolute/relative error measured in all three norms $\simeq 10^{-4}$

Example

Compute relative errors in the ∞ -norm, 1-norm, and 2-norm norms of \mathbf{x}_* as an approximation of \mathbf{x} if

$$\mathbf{x} = \begin{pmatrix} 1.0000 \\ 0.0100 \\ 0.0001 \end{pmatrix} \text{ and } \mathbf{x}_* = \begin{pmatrix} 1.0002 \\ 0.0103 \\ 0.0002 \end{pmatrix}$$

$$\Rightarrow \mathbf{e} := \mathbf{x} - \mathbf{x}_* = \begin{pmatrix} -0.0002 \\ -0.0003 \\ -0.0001 \end{pmatrix} = -10^{-4} \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}$$

Absolute/relative error measured in all three norms $\simeq 10^{-4}$

However, relative error in last component is 100% !

Linear equations, errors, and residuals

- ▶ Data $A \in \mathbb{R}^{n \times n}$, $\mathbf{b} \in \mathbb{R}^{n \times 1}$ prescribed: solve $A\mathbf{x} = \mathbf{b}$ for \mathbf{x}

\mathbf{x} = true solution of $A\mathbf{x} = \mathbf{b}$

\mathbf{x}_* = computed solution of $A\mathbf{x} = \mathbf{b}$

Definition: error & residual

$$\mathbf{e} := \mathbf{x} - \mathbf{x}_* = \text{error vector} \qquad \|\mathbf{e}\| = \text{error}$$

$$\mathbf{r} := \mathbf{b} - A\mathbf{x}_* = \text{residual vector} \qquad \|\mathbf{r}\| = \text{residual}$$

Linear equations, errors, and residuals

- ▶ Data $A \in \mathbb{R}^{n \times n}$, $\mathbf{b} \in \mathbb{R}^{n \times 1}$ prescribed: solve $A\mathbf{x} = \mathbf{b}$ for \mathbf{x}

\mathbf{x} = true solution of $A\mathbf{x} = \mathbf{b}$

\mathbf{x}_* = computed solution of $A\mathbf{x} = \mathbf{b}$

Definition: error & residual

$$\mathbf{e} := \mathbf{x} - \mathbf{x}_* = \text{error vector} \qquad \|\mathbf{e}\| = \text{error}$$

$$\mathbf{r} := \mathbf{b} - A\mathbf{x}_* = \text{residual vector} \qquad \|\mathbf{r}\| = \text{residual}$$

- ▶ If $\mathbf{x}_* = \mathbf{x}$, $\|\mathbf{e}\| = \|\mathbf{r}\| = 0$.

Linear equations, errors, and residuals

- ▶ Data $A \in \mathbb{R}^{n \times n}$, $\mathbf{b} \in \mathbb{R}^{n \times 1}$ prescribed: solve $A\mathbf{x} = \mathbf{b}$ for \mathbf{x}

\mathbf{x} = true solution of $A\mathbf{x} = \mathbf{b}$

\mathbf{x}_* = computed solution of $A\mathbf{x} = \mathbf{b}$

Definition: error & residual

$$\mathbf{e} := \mathbf{x} - \mathbf{x}_* = \text{error vector} \qquad \|\mathbf{e}\| = \text{error}$$

$$\mathbf{r} := \mathbf{b} - A\mathbf{x}_* = \text{residual vector} \qquad \|\mathbf{r}\| = \text{residual}$$

- ▶ If $\mathbf{x}_* = \mathbf{x}$, $\|\mathbf{e}\| = \|\mathbf{r}\| = 0$.
- ▶ Generally, \mathbf{x} unknown, so \mathbf{e} not computable.

Linear equations, errors, and residuals

- ▶ Data $A \in \mathbb{R}^{n \times n}$, $\mathbf{b} \in \mathbb{R}^{n \times 1}$ prescribed: solve $A\mathbf{x} = \mathbf{b}$ for \mathbf{x}

\mathbf{x} = true solution of $A\mathbf{x} = \mathbf{b}$

\mathbf{x}_* = computed solution of $A\mathbf{x} = \mathbf{b}$

Definition: error & residual

$$\mathbf{e} := \mathbf{x} - \mathbf{x}_* = \text{error vector} \qquad \|\mathbf{e}\| = \text{error}$$

$$\mathbf{r} := \mathbf{b} - A\mathbf{x}_* = \text{residual vector} \qquad \|\mathbf{r}\| = \text{residual}$$

- ▶ If $\mathbf{x}_* = \mathbf{x}$, $\|\mathbf{e}\| = \|\mathbf{r}\| = 0$.
- ▶ Generally, \mathbf{x} unknown, so \mathbf{e} not computable.
- ▶ We know A , \mathbf{b} , \mathbf{x}_* , so \mathbf{r} computable.

“Good” linear system of equations:

- ▶ Consider linear system of equations

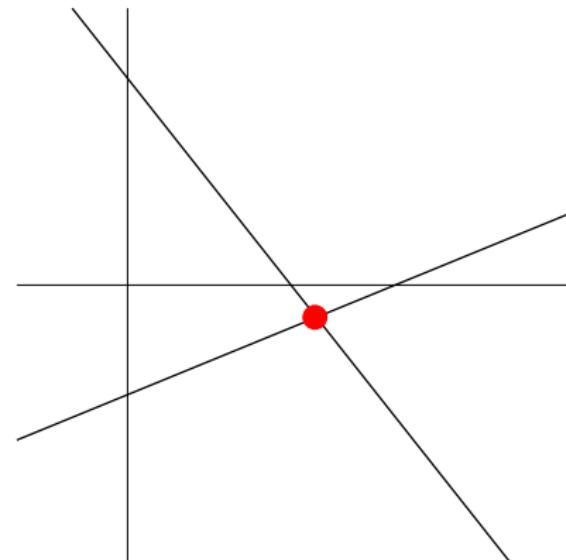
$$\begin{array}{rcl} x_1 + x_2 & = & 2 \\ x_1 - 3x_2 & = & 3 \end{array}$$

- ▶ In matrix form, $A\mathbf{x} = \mathbf{b}$ with

$$A = \begin{pmatrix} 1 & 1 \\ 1 & -3 \end{pmatrix},$$

$$\mathbf{b} = \begin{pmatrix} 2 \\ 3 \end{pmatrix},$$

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$



“Good” linear system of equations:

- ▶ Consider linear system of equations

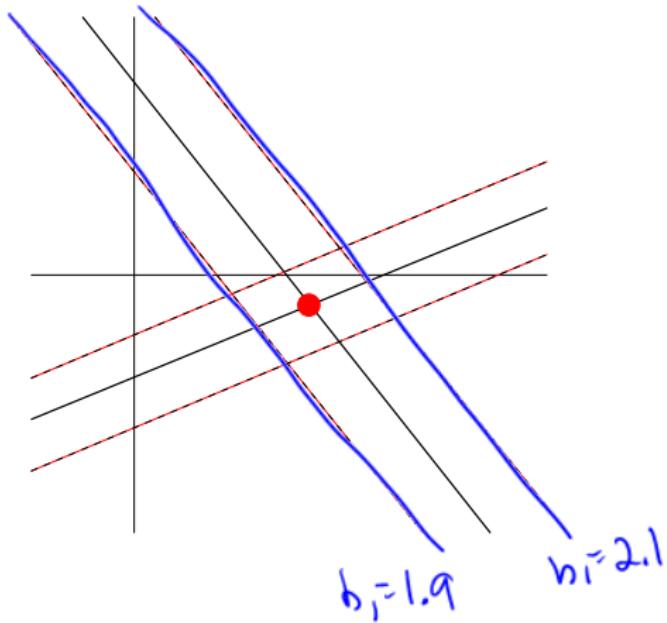
$$\begin{aligned}x_1 + x_2 &= 2 \\x_1 - 3x_2 &= 3\end{aligned}$$

- ▶ In matrix form, $A\mathbf{x} = \mathbf{b}$ with

$$A = \begin{pmatrix} 1 & 1 \\ 1 & -3 \end{pmatrix},$$

$$\mathbf{b}_* = \begin{pmatrix} 2 \pm 0.1 \\ 3 \pm 0.1 \end{pmatrix},$$

$$\mathbf{x}_* = \begin{pmatrix} x_{1*} \\ x_{2*} \end{pmatrix}$$



true value between
1.9 and 2.1

“Good” linear system of equations:

- ▶ Consider linear system of equations

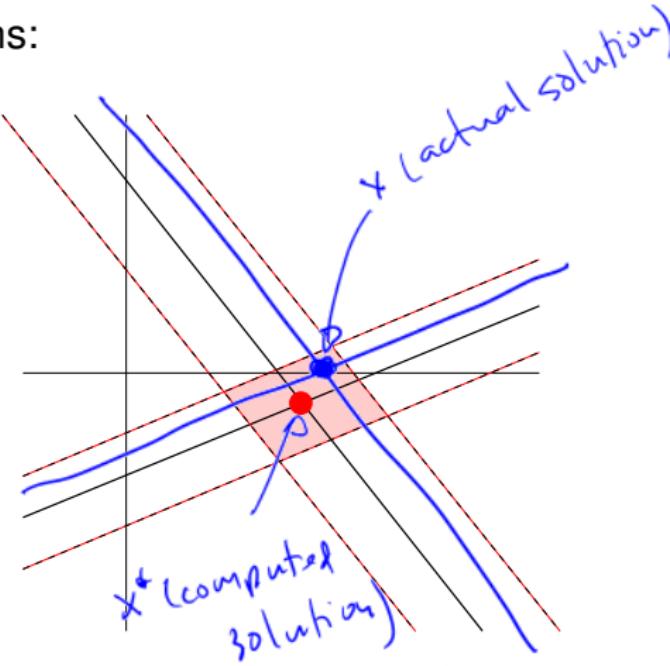
$$\begin{array}{rcl} x_1 + x_2 & = & 2 \\ x_1 - 3x_2 & = & 3 \end{array}$$

- ▶ In matrix form, $A\mathbf{x} = \mathbf{b}$ with

$$A = \begin{pmatrix} 1 & 1 \\ 1 & -3 \end{pmatrix},$$

$$\mathbf{b}_* = \begin{pmatrix} 2 \pm 0.1 \\ 3 \pm 0.1 \end{pmatrix},$$

$$\mathbf{x}_* = \begin{pmatrix} x_{1*} \\ x_{2*} \end{pmatrix}$$



- ▶ Small change in b leads to small change in x_* .

“Bad” linear system of equations:

- ▶ Consider linear system of equations

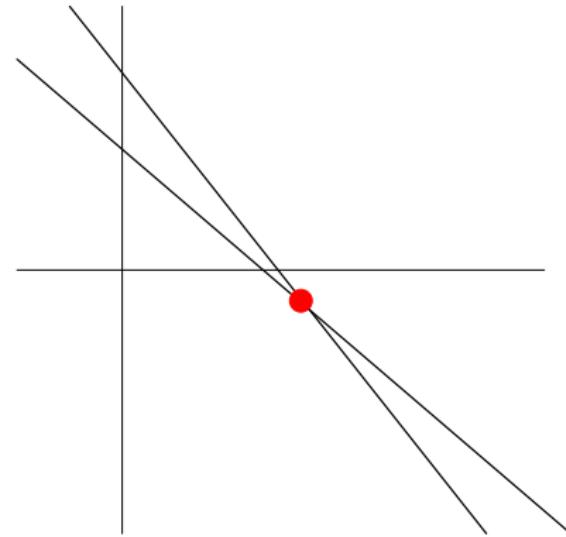
$$\begin{array}{rcl} x_1 + x_2 & = & 2 \\ x_1 + 0.9x_2 & = & 1.9 \end{array}$$

- ▶ In matrix form, $B\mathbf{x} = \mathbf{b}$ with

$$B = \begin{pmatrix} 1 & 1 \\ 1 & 0.9 \end{pmatrix},$$

$$\mathbf{b} = \begin{pmatrix} 2 \\ 1.9 \end{pmatrix},$$

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$



“Bad” linear system of equations:

- ▶ Consider linear system of equations

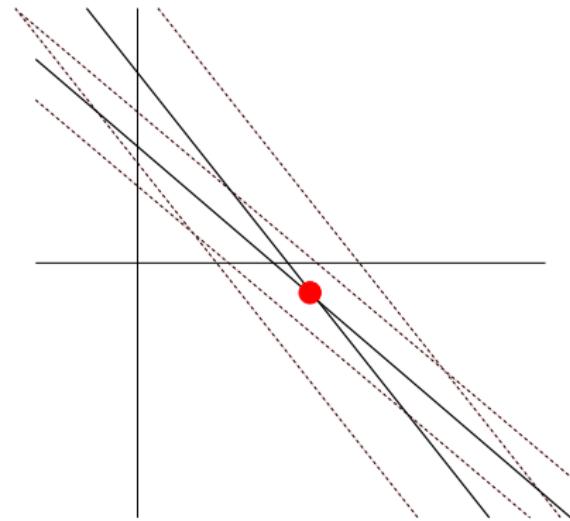
$$\begin{aligned}x_1 + x_2 &= 2 \\x_1 + 0.9x_2 &= 1.9\end{aligned}$$

- ▶ In matrix form, $B\mathbf{x} = \mathbf{b}$ with

$$B = \begin{pmatrix} 1 & 1 \\ 1 & 0.9 \end{pmatrix},$$

$$\mathbf{b}_* = \begin{pmatrix} 2 \pm 0.1 \\ 1.9 \pm 0.1 \end{pmatrix},$$

$$\mathbf{x}_* = \begin{pmatrix} x_{1*} \\ x_{2*} \end{pmatrix}$$



“Bad” linear system of equations:

- ▶ Consider linear system of equations

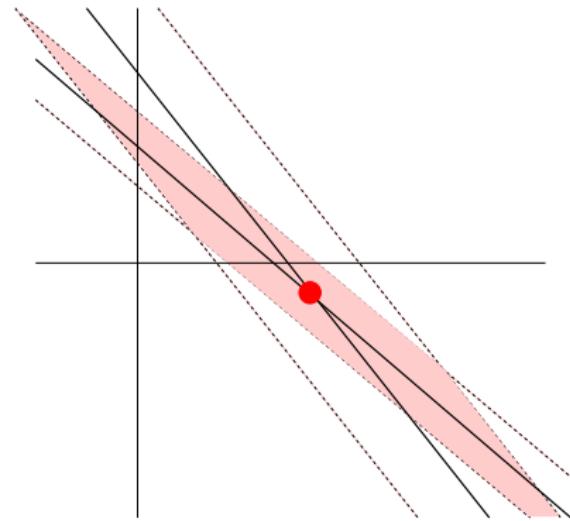
$$\begin{array}{rcl} x_1 + x_2 & = & 2 \\ x_1 + 0.9x_2 & = & 1.9 \end{array}$$

- ▶ In matrix form, $B\mathbf{x} = \mathbf{b}$ with

$$B = \begin{pmatrix} 1 & 1 \\ 1 & 0.9 \end{pmatrix},$$

$$\mathbf{b}_* = \begin{pmatrix} 2 \pm 0.1 \\ 1.9 \pm 0.1 \end{pmatrix},$$

$$\mathbf{x}_* = \begin{pmatrix} x_{1*} \\ x_{2*} \end{pmatrix}$$



- ▶ Small change in b leads to big change in x_* .

Condition numbers

- ▶ $K(A)$: Condition number of matrix A with $1 \leq K(A) < \infty$
- ▶ Key property: $\frac{\|e\|}{\|x\|} \leq K(A) \frac{\|r\|}{\|b\|}$, i.e.,

$$\text{relative error of } x_* \leq (\text{condition number}) (\text{relative residual of } x_*)$$

- ▶ Compute by `numpy.linalg.cond`

```
>>> import numpy
>>> import numpy.linalg
>>> A=numpy.array([[1.0,1.0],[1.0,-3.0]])
>>> numpy.linalg.cond(A,2)
2.6180339887498949
```

good →

```
>>> B=numpy.array([[1.0,1.0],[1.0,0.9]])
>>> numpy.linalg.cond(B,2)
```

bad → 38.073735174775756

Background:

- ▶ The condition number of a matrix is defined as the quotient of its largest to its smallest *singular values*.

$$K(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$$

Background:

- ▶ The condition number of a matrix is defined as the quotient of its largest to its smallest *singular values*.

$$K(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$$

- ▶ A singular value of A is the square root of an eigenvalue of $A^T A$, i.e.

$$A^T A \mathbf{w} = \sigma^2 \mathbf{w}$$

Background:

- ▶ The condition number of a matrix is defined as the quotient of its largest to its smallest *singular values*.

$$K(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$$

- ▶ A singular value of A is the square root of an eigenvalue of $A^T A$, i.e.

$$A^T A \mathbf{w} = \sigma^2 \mathbf{w}$$

- ▶ If A satisfies $A^T A = A A^T$ then its singular values equal the modulus of the eigenvalues ($\sigma = |\lambda|$).

Key property:

$$\frac{\|\mathbf{e}\|}{\|\mathbf{x}\|} \leq K(A) \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}$$

- ▶ The condition number $K(A)$ is an indicator of whether a system of linear equations $A\mathbf{x} = \mathbf{b}$ is “good” or “bad”

Key property:

$$\frac{\|\mathbf{e}\|}{\|\mathbf{x}\|} \leq K(A) \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}$$

- ▶ The condition number $K(A)$ is an indicator of whether a system of linear equations $A\mathbf{x} = \mathbf{b}$ is “good” or “bad”
- ▶ If $K(A)$ is small, it’s “good”: we call it **well-conditioned**

Key property:

$$\frac{\|\mathbf{e}\|}{\|\mathbf{x}\|} \leq K(A) \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}$$

- ▶ The condition number $K(A)$ is an indicator of whether a system of linear equations $A\mathbf{x} = \mathbf{b}$ is “good” or “bad”
- ▶ If $K(A)$ is small, it’s “good”: we call it **well-conditioned**
- ▶ If $K(A)$ is large, it’s “bad”: we call it **ill-conditioned**

Key property:

$$\frac{\|\mathbf{e}\|}{\|\mathbf{x}\|} \leq K(A) \frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}$$

- ▶ The condition number $K(A)$ is an indicator of whether a system of linear equations $\mathbf{Ax} = \mathbf{b}$ is “good” or “bad”
- ▶ If $K(A)$ is small, it’s “good”: we call it **well-conditioned**
- ▶ If $K(A)$ is large, it’s “bad”: we call it **ill-conditioned**
- ▶ Example of ill-conditioned:

$$A = \begin{pmatrix} 1 & 100 \\ 0 & 2 \end{pmatrix} \quad A^T A = \begin{pmatrix} 1 & 100 \\ 100 & 10004 \end{pmatrix}$$

with eigenvalues $\lambda_1 = 2$, $\lambda_2 = 1$ and singular values $\sigma_{\max} \approx 100$, $\sigma_{\min} \approx 0.02$ so $K(A) \approx 5002$.

For the earlier examples:



$$A = \begin{pmatrix} 1 & 1 \\ 1 & -3 \end{pmatrix} \text{ has } K(A) \approx 2.6$$

so the relative error in \mathbf{x}_* when solving $A\mathbf{x} = \mathbf{b}$ is at most 2.6 times larger than the relative residual.



$$B = \begin{pmatrix} 1 & 1 \\ 1 & .9 \end{pmatrix} \text{ has } K(A) \approx 38$$

so the relative error in \mathbf{x}_* when solving $B\mathbf{x} = \mathbf{b}$ can be as big as 38 times the relative residual.

► As a rule of thumb, if $K(A) \approx 10^q$, you can compute q digits less for \mathbf{x}_* than you know for \mathbf{b} .

A good example: the Vandermonde matrix (see weeks 7-8):

The *Vandermonde matrix* is defined as

$$V_{ij} = x_{i-1}^{n-j+1} \text{ for } 1 \leq i \leq n+1 \text{ and } 1 \leq j \leq n+1$$

The *Vandermonde matrix* for $n = 4$ is: $V =$

$$\begin{bmatrix} x_0^4 & x_0^3 & x_0^2 & x_0 & 1 \\ x_1^4 & x_1^3 & x_1^2 & x_1 & 1 \\ x_2^4 & x_2^3 & x_2^2 & x_2 & 1 \\ x_3^4 & x_3^3 & x_3^2 & x_3 & 1 \\ x_4^4 & x_4^3 & x_4^2 & x_4 & 1 \end{bmatrix}$$

Let $x_i = -1 + i\Delta$ for $i = 0, \dots, n$ and $\Delta = 2/n$
(gives equally spaced points between -1 and 1).

For $n = 20$, $K(V) \approx 8 \times 10^8$.

Let $b_i = x_{i-1} - x_{i-1}^2$. for $1 < i < n + 1$, then



$$Vx = b$$

has the *exact* solution

$$x = e_{20} - e_{19}$$

$$x = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ -1 \\ 0 \end{pmatrix}$$

Numerically solving (see accuracy.py in the code repository):

```
>>> import numpy as np
>>> import scipy.linalg
>>> xs=np.linspace(-1,1,21)
>>> V=np.vander(xs)
>>> def f(x):
...     return x-x**x
>>> r=f(xs)
>>> s=scipy.linalg.solve(V,r)
```



Accuracy_LinSys.py

$$\mathbf{x}_* = \begin{pmatrix} 0.0000000000244668 \\ 0.000000000004811 \\ -0.00000000000929254 \\ -0.0000000000023777 \\ 0.00000000001457681 \\ 0.000000000045536 \\ -0.00000000001227802 \\ -0.0000000000043483 \\ 0.00000000000604410 \\ 0.0000000000021749 \\ -0.00000000000177176 \\ -0.0000000000005221 \\ 0.0000000000030097 \\ 0.000000000000000312 \\ -0.0000000000000002738 \\ 0.000000000000000085 \\ 0.000000000000000115 \\ -0.000000000000000012 \\ -1.000000000000000002 \\ 1.000000000000000000 \\ 0.000000000000000000 \end{pmatrix}$$

→ supposed to
be 0

relative residual

relative error.

$$\frac{\|\mathbf{b} - V\mathbf{x}\|_2}{\|\mathbf{b}\|_2} \approx 10^{-15}; \quad \frac{\|\mathbf{x} - \mathbf{x}_*\|_2}{\|\mathbf{x}\|_2} \approx 10^{-9}$$



Summary

- ▶ Norms: quantify lengths of / distances between vectors.
- ▶ Definitions of ℓ_1 -, ℓ_2 -, and ℓ_∞ -norms.
- ▶ Linear equations: **errors** and **residuals** of computed solutions.
- ▶ Condition number $K(A)$: measure of the accuracy in solving $A\mathbf{x} = \mathbf{b}$.
 - ▶ Well-conditioned $K(A) \simeq 1$; ill-conditioned $K(A) \gg 1$.
 - ▶ $K(A)$ large \Rightarrow limited accuracy in solving $A\mathbf{x} = \mathbf{b}$ numerically.
 - ▶ Computation through `numpy.linalg.cond`.

2072U Computational Science I

Winter 2022

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. Complexity of algorithms

2. Some standard sums

3. Standard algorithms

Key questions:

- ▶ What is *computational complexity*?
- ▶ What is the computational complexity of standard algorithms?
 - ▶ factorials and summation of sequences?
 - ▶ recursive algorithms?
 - ▶ matrix-vector multiplication?
 - ▶ matrix-matrix multiplication?

The concept of computational complexity will help us answer question 3:

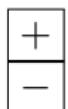
1. When does my computation work?
2. How accurate is the result?
3. **How fast does my computation work?**

Floating-point operations (Flops):

- ▶ Computational complexity of an algorithm: amount of work required to execute/carry out algorithm from start to finish.
- ▶ Traditional unit of complexity for numerical algorithms: the flop.

Floating-point operations (Flops):

- ▶ Computational complexity of an algorithm: amount of work required to execute/carry out algorithm from start to finish.
- ▶ Traditional unit of complexity for numerical algorithms: the flop.
- ▶ One flop = one floating-point operation.



(addition)



(subtraction)



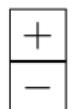
(multiplication)



(division)

Floating-point operations (Flops):

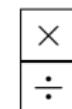
- ▶ Computational complexity of an algorithm: amount of work required to execute/carry out algorithm from start to finish.
- ▶ Traditional unit of complexity for numerical algorithms: the flop.
- ▶ One flop = one floating-point operation.



(addition)



(subtraction)



(multiplication)



(division)

How to count flops

1. Write pseudocode of algorithm clearly.
2. In each line, count number of flops ($+$, $-$, \times , \div).
3. Count number of times each line executes (e.g., in a **for** loop).
4. Multiply cost of each line by number of times it executes.

Summation identities and tricks:

- ▶ Use summation identities to count # times each line executes

$$\sum_{k=1}^n 1 = n,$$

$$\sum_{k=1}^n k = \frac{n(n+1)}{2},$$

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6},$$

$$\sum_{k=1}^n k^3 = \left[\frac{n(n+1)}{2} \right]^2$$

 (Σ)

Summation identities and tricks:

- ▶ Use summation identities to count # times each line executes

$$\sum_{k=1}^n 1 = n,$$

$$\sum_{k=1}^n k = \frac{n(n+1)}{2},$$

 (Σ)

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6},$$

$$\sum_{k=1}^n k^3 = \left[\frac{n(n+1)}{2} \right]^2$$

- ▶ Summation limits can be transformed if necessary:

$$\sum_{k=\alpha}^{\beta} a_k = \sum_{\ell=1}^{\beta-\alpha+1} a_{\ell+\alpha-1} \quad (\text{substitute } \ell = k - \alpha + 1)$$

K = $\ell + \alpha - 1$

Goal: change lower index of sum to 1 and apply identities
 (Σ)

Summation identities and tricks:

- ▶ Use summation identities to count # times each line executes

$$\sum_{k=1}^n 1 = n,$$

$$\sum_{k=1}^n k = \frac{n(n+1)}{2},$$

 (Σ)

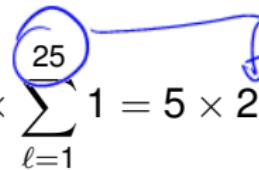
$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6},$$

$$\sum_{k=1}^n k^3 = \left[\frac{n(n+1)}{2} \right]^2$$

- ▶ Summation limits can be transformed if necessary:

$$\sum_{k=\alpha}^{\beta} a_k = \sum_{\ell=1}^{\beta-\alpha+1} a_{\ell+\alpha-1} \quad (\text{substitute } \ell = k - \alpha + 1)$$

e.g., $\sum_{k=3}^{27} 5 = 5 \times \sum_{\ell=1}^{25} 1 = 5 \times 25 = 125$



Goal: change lower index of sum to 1 and apply identities
 (Σ)

Computing a sum:

Input: vector $\mathbf{x} \in \mathbb{R}^n$

1: $S \leftarrow x_1$

2: **for** $k = 2 : n$ 

3: $S \leftarrow S + x_k$

4: **end for** 

Output: $S = \sum_{k=1}^n x_k$

$$\underline{\mathbf{x}} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

$$S = \sum_{k=1}^n x_k \quad \text{given } \mathbf{x} \in \mathbb{R}^n$$

Computing a sum:

Input: vector $\mathbf{x} \in \mathbb{R}^n$

1: $S \leftarrow x_1$

2: **for** $k = 2 : n$

3: $S \leftarrow S + x_k$

4: **end for**

Output: $S = \sum_{k=1}^n x_k$

$$S = \sum_{k=1}^n x_k \quad \text{given } \mathbf{x} \in \mathbb{R}^n$$

- ▶ One $[+]$ in Line 3: 1 flop

Computing a sum:

Input: vector $\mathbf{x} \in \mathbb{R}^n$

1: $S \leftarrow x_1$

2: **for** $k = 2 : n$

3: $S \leftarrow S + x_k$ $\rightarrow 1 \text{ flop}$

4: **end for**

Output: $S = \sum_{k=1}^n x_k$

$$S = \sum_{k=1}^n x_k \quad \text{given } \mathbf{x} \in \mathbb{R}^n$$

► One $[+]$ in Line 3: 1 flop

► Line 3 executes once for each $k = 2 : n$

Total cost of computing $\sum_{k=1}^n x_k$ is $\sum_{k=2}^n 1$

$$\sum_{l=1}^{n-1} 1 = n-1$$

Computing a sum:

Input: vector $\mathbf{x} \in \mathbb{R}^n$

- 1: $S \leftarrow x_1$
- 2: **for** $k = 2 : n$
- 3: $S \leftarrow S + x_k$
- 4: **end for**

Output: $S = \sum_{k=1}^n x_k$

$$S = \sum_{k=1}^n x_k \quad \text{given } \mathbf{x} \in \mathbb{R}^n$$

- ▶ One $[+]$ in Line 3: 1 flop
- ▶ Line 3 executes once for each $k = 2 : n$

Total cost of computing $\sum_{k=1}^n x_k$ is $\sum_{k=2}^n 1 =$ n - 1 flops

Computing a factorial:

Input: $n \in \mathbb{N}$ (assume $n > 2$)1: $P \leftarrow 2$ 2: **for** $k = 3 : n$ 3: $P \leftarrow P \times k$ 4: **end for****Output:** $P = n!$

$$0! = 1,$$

$$1! = 1,$$

$$2! = 2,$$

$$n! = n(n-1)(n-2) \cdots (2)(1)$$

$$= \prod_{k=1}^n k \quad \text{given } n \geq 2$$

Computing a factorial:

Input: $n \in \mathbb{N}$ (assume $n > 2$)1: $P \leftarrow 2$ 2: **for** $k = 3 : n$ 3: $P \leftarrow P \times k$ 4: **end for****Output:** $P = n!$

$$0! = 1,$$

$$1! = 1,$$

$$2! = 2,$$

$$n! = n(n-1)(n-2) \cdots (2)(1)$$

$$= \prod_{k=1}^n k \quad \text{given } n \geq 2$$

- ▶ One \times in Line 3: 1 flop

Computing a factorial:

Input: $n \in \mathbb{N}$ (assume $n > 2$)

1: $P \leftarrow 2$

2: **for** $k = 3 : n$

3: $P \leftarrow P \times k$ 

4: **end for**

Output: $P = n!$

$$0! = 1,$$

$$1! = 1,$$

$$2! = 2,$$

$$n! = n(n-1)(n-2) \cdots (2)(1)$$

$$= \prod_{k=1}^n k \quad \text{given } n \geq 2$$

- ▶ One \times in Line 3: 1 flop
- ▶ Line 3 executes once for each $k = 3 : n$

Total cost of computing $n! = \prod_{k=1}^n k$ is $\sum_{k=3}^n (1)$

Computing a factorial:

Input: $n \in \mathbb{N}$ (assume $n > 2$)1: $P \leftarrow 2$ 2: **for** $k = 3 : n$ 3: $P \leftarrow P \times k$ 4: **end for****Output:** $P = n!$

$$0! = 1,$$

$$1! = 1,$$

$$2! = 2,$$

$$n! = n(n-1)(n-2) \cdots (2)(1)$$

$$= \prod_{k=1}^n k \quad \text{given } n \geq 2$$

- ▶ One \times in Line 3: 1 flop
- ▶ Line 3 executes once for each $k = 3 : n$

Total cost of computing $n! = \prod_{k=1}^n k$ is $\sum_{k=3}^n (1) = \boxed{n-2 \text{ flops}}$

$$l \sim k-2 \Rightarrow \sum_{l=1}^{n-2} 1 = n-2$$

Computing an inner product:

Input: vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

1: $s \leftarrow x_1 \times y_1$ 1

2: **for** $k = 2 : n$ →

3: $s \leftarrow s + x_k \times y_k$ 2

4: **end for**

Output: $s = \mathbf{x}^T \mathbf{y}$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = x_1 y_1 + x_2 y_2 + \dots + x_n y_n$$

$\sum_{k=1}^n x_k y_k \quad \text{given } \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

Computing an inner product:

Input: vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

1: $s \leftarrow x_1 \times y_1$

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T \mathbf{y}$$

2: **for** $k = 2 : n$

$$= \sum_{k=1}^n x_k y_k \quad \text{given } \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$$

3: $s \leftarrow s + x_k \times y_k$

4: **end for**

Output: $s = \mathbf{x}^T \mathbf{y}$

- ▶ One \times in Line 1: 1 flop
- ▶ One $+$, one \times in Line 3: 2 flops

Computing an inner product:

Input: vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

```
1:  $s \leftarrow x_1 \times y_1$  (1)  
2: for  $k = 2 : n$    
3:    $s \leftarrow s + x_k \times y_k$    
4: end for
```

Output: $s = \mathbf{x}^T \mathbf{y}$

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T \mathbf{y}$$

$$= \sum_{k=1}^n x_k y_k \quad \text{given } \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$$

- ▶ One \times in Line 1: 1 flop
- ▶ One $+$, one \times in Line 3: 2 flops
- ▶ Line 1 executes exactly once
- ▶ Line 3 executes once for each $k = 2 : n$

Total cost of computing $\mathbf{x}^T \mathbf{y}$ is $1 + \sum_{k=2}^n 2$

Computing an inner product:

Input: vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ 1: $s \leftarrow x_1 \times y_1$

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T \mathbf{y}$$

2: **for** $k = 2 : n$

$$= \sum_{k=1}^n x_k y_k \quad \text{given } \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$$

3: $s \leftarrow s + x_k \times y_k$ 4: **end for****Output:** $s = \mathbf{x}^T \mathbf{y}$

- ▶ One \times in Line 1: 1 flop
- ▶ One $+$, one \times in Line 3: 2 flops
- ▶ Line 1 executes exactly once
- ▶ Line 3 executes once for each $k = 2 : n$

Total cost of computing $\mathbf{x}^T \mathbf{y}$ is $1 + \sum_{k=2}^n 2 = \boxed{2n - 1 \text{ flops}}$

Computing a matrix-vector product (matvec):

Input: matrix $A \in \mathbb{R}^{n \times n}$, vector $\mathbf{x} \in \mathbb{R}^{n \times 1}$

1: **for** $j = 1 : n$

2: $c_j \leftarrow A_{j1} \times x_1$

$$\mathbf{c} = A\mathbf{x} \in \mathbb{R}^{n \times 1}$$

3: **for** $k = 2 : n$

4: $c_j \leftarrow c_j + A_{jk} \times x_k$

$$c_j = \sum_{k=1}^n A_{jk} b_k \quad (j = 1 : n)$$

5: **end for**

6: **end for**

Output: $\mathbf{c} = Ax$

Computing a matrix-vector product (matvec):

Input: matrix $A \in \mathbb{R}^{n \times n}$, vector $\mathbf{x} \in \mathbb{R}^{n \times 1}$

1: **for** $j = 1 : n$ ↗

2: $c_j \leftarrow A_{j1} \times x_1$

$$\mathbf{c} = A\mathbf{x} \in \mathbb{R}^{n \times 1}$$

3: **for** $k = 2 : n$ ↗

4: $c_j \leftarrow c_j + A_{jk} \times x_k$ ↙

$$c_j = \sum_{k=1}^n A_{jk} b_k \quad (j = 1 : n)$$

5: **end for**

6: **end for**

Output: $\mathbf{c} = A\mathbf{x}$

- ▶ Line 2: one \times ; Line 4: one $+$, one \times

Computing a matrix-vector product (matvec):

Input: matrix $A \in \mathbb{R}^{n \times n}$, vector $\mathbf{x} \in \mathbb{R}^{n \times 1}$

```
1: for  $j = 1 : n$ 
2:    $c_j \leftarrow A_{j1} \times x_1$  ①
3:   for  $k = 2 : n$ 
4:      $c_j \leftarrow c_j + A_{jk} \times x_k$ 
5:   end for
```

6: **end for**

Output: $\mathbf{c} = A\mathbf{x}$

- ▶ Line 2: one \times ; Line 4: one $+$, one \times
- ▶ Line 2 executes once for each $j = 1 : n$
- ▶ Line 4 executes once for each $k = 2 : n$ and $j = 1 : n$

Total cost of computing $A\mathbf{x}$ is $\sum_{j=1}^n \left(1 + \sum_{k=2}^n 2 \right)$
 $(2n - 1) \cdot n$

$$\mathbf{c} = A\mathbf{x} \in \mathbb{R}^{n \times 1}$$

$$c_j = \sum_{k=1}^n A_{jk} b_k \quad (j = 1 : n)$$

Computing a matrix-vector product (matvec):

Input: matrix $A \in \mathbb{R}^{n \times n}$, vector $\mathbf{x} \in \mathbb{R}^{n \times 1}$

1: **for** $j = 1 : n$

2: $c_j \leftarrow A_{j1} \times x_1$

$$\mathbf{c} = A\mathbf{x} \in \mathbb{R}^{n \times 1}$$

3: **for** $k = 2 : n$

4: $c_j \leftarrow c_j + A_{jk} \times x_k$

$$c_j = \sum_{k=1}^n A_{jk} b_k \quad (j = 1 : n)$$

5: **end for**

6: **end for**

Output: $\mathbf{c} = A\mathbf{x}$

- ▶ Line 2: one \times ; Line 4: one $+$, one \times
- ▶ Line 2 executes once for each $j = 1 : n$
- ▶ Line 4 executes once for each $k = 2 : n$ and $j = 1 : n$

Total cost of computing $A\mathbf{x}$ is $\sum_{j=1}^n \left(1 + \sum_{k=2}^n 2 \right) = \boxed{2n^2 - n \text{ flops}}$

Computing a matrix-matrix product:

$$\begin{pmatrix} & \text{red circle} \\ & \text{green circle} \\ & \text{blue circle} \end{pmatrix} \begin{pmatrix} & \text{red circle} \\ & \text{green circle} \\ & \text{blue circle} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

Input: matrices $A, B \in \mathbb{R}^{n \times n}$

1: **for** $j = 1 : n$ *outer-loop*
2: **for** $\ell = 1 : n$ *inner-loop*
3: $C_{j\ell} \leftarrow A_{j1} \times B_{1\ell}$
4: **for** $k = 2 : n$
5: $C_{j,\ell} \leftarrow C_{j\ell} + A_{jk} \times B_{k\ell}$
6: **end for**
7: **end for**
8: **end for**
Output: $C = AB \in \mathbb{R}^{n \times n}$

Computing a matrix-matrix product:

Input: matrices $A, B \in \mathbb{R}^{n \times n}$

```
1: for  $j = 1 : n$ 
2:   for  $\ell = 1 : n$ 
3:      $C_{j\ell} \leftarrow A_{j1} \times B_{1\ell}$             $\Leftarrow 1 \text{ flop } (j, \ell = 1 : n)$ 
4:     for  $k = 2 : n$ 
5:        $C_{j,\ell} \leftarrow C_{j\ell} + A_{jk} \times B_{k\ell}$   $\Leftarrow 2 \text{ flops } (j, \ell = 1 : n; k = 2 : n)$ 
6:     end for
7:   end for
8: end for
```

Output: $C = AB \in \mathbb{R}^{n \times n}$

Computing a matrix-matrix product:

Input: matrices $A, B \in \mathbb{R}^{n \times n}$

1: **for** $j = 1 : n$
2: **for** $\ell = 1 : n$
3: $C_{j\ell} \leftarrow A_{j1} \times B_{1\ell}$ $\Leftarrow 1 \text{ flop } (j, \ell = 1 : n)$
4: **for** $k = 2 : n$ -
5: $C_{j,\ell} \leftarrow C_{j\ell} + A_{jk} \times B_{k\ell}$ $\Leftarrow 2 \text{ flops } (j, \ell = 1 : n; k = 2 : n)$
6: **end for**
7: **end for**
8: **end for**

Output: $C = AB \in \mathbb{R}^{n \times n}$

Total cost of computing AB is

$$\sum_{j=1}^n \sum_{\ell=1}^n \left(1 + \sum_{k=2}^n 2 \right) = \boxed{2n^3 - n^2 \text{ flops}}$$

$2n-1$

Summarised:

- ▶ Sum $S = \sum_{k=1}^n x_k$: cost is $n - 1$ $\boxed{+}$ or $\boxed{n - 1 \text{ flops}}$

Summarised:

- ▶ Sum $S = \sum_{k=1}^n x_k$: cost is $n - 1$ $\boxed{+}$ or $n - 1$ flops
- ▶ Inner product $\mathbf{x}^T \mathbf{y} = \sum_{k=1}^n x_k y_k$: $n \boxed{\times}$ & $n - 1 \boxed{+}$ or $2n - 1$ flops

Summarised:

- ▶ Sum $S = \sum_{k=1}^n x_k$: cost is $n - 1$ $\boxed{+}$ or $n - 1$ flops
- ▶ Inner product $\mathbf{x}^T \mathbf{y} = \sum_{k=1}^n x_k y_k$: $n \boxed{\times}$ & $n - 1 \boxed{+}$ or $2n - 1$ flops
- ▶ Matvec (matrix-vector product) $\mathbf{c} = A\mathbf{x} \in \mathbb{R}^{n \times 1}$: for $j = 1 : n$,

$$c_j = \sum_{k=1}^n A_{jk} b_k = A_{j:} \mathbf{b} \quad \Rightarrow \text{ } n \text{ inner products: } \boxed{2n^2 - n \text{ flops}}$$

Summarised:

- ▶ Sum $S = \sum_{k=1}^n x_k$: cost is $n - 1$ $\boxed{+}$ or $n - 1$ flops
- ▶ Inner product $\mathbf{x}^T \mathbf{y} = \sum_{k=1}^n x_k y_k$: $n \boxed{\times}$ & $n - 1 \boxed{+}$ or $2n - 1$ flops
- ▶ Matvec (matrix-vector product) $\mathbf{c} = A\mathbf{x} \in \mathbb{R}^{n \times 1}$: for $j = 1 : n$,
 $c_j = \sum_{k=1}^n A_{jk} b_k = A_{j:} \cdot \mathbf{b} \quad \Rightarrow \text{ } n \text{ inner products: } \boxed{2n^2 - n \text{ flops}}$
- ▶ Matrix-matrix product $C = AB$: for $j = 1 : n$ and $\ell = 1 : n$,
 $C_{j\ell} = \sum_{k=1}^n A_{jk} B_{k\ell} = A_{j:} \cdot B_{:\ell} \quad \Rightarrow n^2 \text{ inner products: } \boxed{2n^3 - n^2 \text{ flops}}$

Remarks

- ▶ Assume all floating-point operations have equal cost
- ▶ Ignore memory access or overwriting in computing cost
- ▶ Precise definitions of flops vary in distinct texts/papers
- ▶ Count special function evaluations (e.g., `sqrt`, etc.) as needed
- ▶ Branching statements (`if` or `case`) can require extra care
- ▶ Complexity analysis possible for memory/storage, etc.
- ▶ Complexity analysis of recursively defined functions yields recurrence relations to solve

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. Complexity of algorithms

2. Big-Oh notation

Key questions:

- ▶ What is the computational complexity of
 - ▶ Gaußion elimination / LU-decomposition?
 - ▶ polynomial evaluation?
- ▶ What does Landau notation (“Big-Oh”) mean?
- ▶ How can the implications of Landau notation be visualised?
- ▶ How does complexity relate to **actual performance of programs**?

Summary of what we learned so far:

Summary of what we learned so far:

- ▶ *Computational complexity* is counted in the number of *floating-point operations* (FLOPS).

Summary of what we learned so far:

- ▶ *Computational complexity* is counted in the number of *floating-point operations* (FLOPS).
- ▶ Usually, the (maximal) number of flops in a code can be found as
 - ▶ a sum for (nested) loops or
 - ▶ the solution to a recursion relation for recursive codes.

Summary of what we learned so far:

- ▶ *Computational complexity* is counted in the number of *floating-point operations* (FLOPS).
- ▶ Usually, the (maximal) number of flops in a code can be found as
 - ▶ a sum for (nested) loops or
 - ▶ the solution to a recursion relation for recursive codes.
- ▶ Under a number of simplifying assumptions, the number of flops determines the time it takes to run a code.

Summary of what we learned so far:

- ▶ *Computational complexity* is counted in the number of *floating-point operations* (FLOPS).
- ▶ Usually, the (maximal) number of flops in a code can be found as
 - ▶ a sum for (nested) loops or
 - ▶ the solution to a recursion relation for recursive codes.
- ▶ Under a number of simplifying assumptions, the number of flops determines the time it takes to run a code.
- ▶ The number of flops taken for some simple computations

sum of n terms	$n - 1$	<i>O(n)</i>
product of n factors	$n - 1$	<i>O(n)</i>
dot product of n -vectors	$2n - 1$	<i>O(n)</i>
$n \times n$ matrix–vector product	$2n^2 - n$	<i>O(n^2)</i>
$n \times n$ matrix–matrix product	$2n^3 - n^2$	<i>O(n^3)</i>

Counting flops for Gaussian elimination:

Input: augmented matrix $A \in \mathbb{R}^{n \times (n+1)}$

for $i = 1 : n - 1$

for $j = i + 1 : n$

$$m \leftarrow A_{j,i} / A_{i,i}$$

$\rightarrow A(j, i) \leftarrow 0$

for $k = i + 1 : n + 1$

$$A(j, k) \leftarrow A(j, k) - m A(i, k)$$

end

end

end

So the number of flops is:

flops =

Counting flops for Gaussian elimination:

Input: augmented matrix $A \in \mathbb{R}^{n \times (n+1)}$

for $i = 1 : n - 1$

for $j = i + 1 : n$

$\rightarrow m \leftarrow A_{j,i}/A_{i,i}$ ← 1 flop

$A(j, i) \leftarrow 0$

for $k = i + 1 : n + 1$

$\rightarrow A(j, k) \leftarrow A(j, k) - m A(i, k)$ ← 2 flops

end

end

end

So the number of flops is:

flops =

Counting flops for Gaussian elimination:

Input: augmented matrix $A \in \mathbb{R}^{n \times (n+1)}$

for $i = 1 : n - 1$

for $j = i + 1 : n$

$m \leftarrow A_{j,i}/A_{i,i}$ $\leftarrow 1$ flop

$A(j, i) \leftarrow 0$

for $k = i + 1 : n + 1$

$A(j, k) \leftarrow A(j, k) - m A(i, k)$ $\leftarrow 2$ flops

end

end

end

So the number of flops is:

$$\# \text{ flops} = \sum_{i=1}^{n-1}$$

Counting flops for Gaussian elimination:

Input: augmented matrix $A \in \mathbb{R}^{n \times (n+1)}$

for $i = 1 : n - 1$

for $j = i + 1 : n$

$m \leftarrow A_{j,i}/A_{i,i}$ ← 1 flop

$A(j, i) \leftarrow 0$

for $k = i + 1 : n + 1$

$A(j, k) \leftarrow A(j, k) - m A(i, k)$ ← 2 flops

end

end

end

So the number of flops is:

$$\# \text{ flops} = \sum_{i=1}^{n-1} \sum_{j=i+1}^n$$

Counting flops for Gaussian elimination:

Input: augmented matrix $A \in \mathbb{R}^{n \times (n+1)}$

for $i = 1 : n - 1$

for $j = i + 1 : n$

$m \leftarrow A_{j,i}/A_{i,i}$ $\leftarrow 1$ flop

$A(j, i) \leftarrow 0$

for $k = i + 1 : n + 1$

$A(j, k) \leftarrow A(j, k) - m A(i, k)$ $\leftarrow 2$ flops

end

end

end

So the number of flops is:

$$\# \text{ flops} = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \left(1 + \right)$$

Counting flops for Gaussian elimination:

Input: augmented matrix $A \in \mathbb{R}^{n \times (n+1)}$

for $i = 1 : n - 1$

for $j = i + 1 : n$

$m \leftarrow A_{j,i}/A_{i,i}$ $\leftarrow 1$ flop

$A(j, i) \leftarrow 0$

for $k = i + 1 : n + 1$

$A(j, k) \leftarrow A(j, k) - m A(i, k)$ $\leftarrow 2$ flops

end

end

end

So the number of flops is:

$$\# \text{ flops} = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \left(1 + \sum_{k=i+1}^{n+1} \right)$$

Counting flops for Gaussian elimination:

Input: augmented matrix $A \in \mathbb{R}^{n \times (n+1)}$

for $i = 1 : n - 1$

for $j = i + 1 : n$

$m \leftarrow A_{j,i}/A_{i,i}$

$A(j,i) \leftarrow 0$

for $k = i + 1 : n + 1$

$A(j,k) \leftarrow A(j,k) - m A(i,k)$

end

end

end

So the number of flops is:

$$\# \text{ flops} = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \left(1 + \sum_{k=i+1}^{n+1} 2 \right)$$

$$d = k - i$$

Counting flops for Gaussian elimination:

Input: augmented matrix $A \in \mathbb{R}^{n \times (n+1)}$

for $i = 1 : n - 1$

for $j = i + 1 : n$

$m \leftarrow A_{j,i}/A_{i,i}$ ← 1 flop

$A(j, i) \leftarrow 0$

for $k = i + 1 : n + 1$

$A(j, k) \leftarrow A(j, k) - m A(i, k)$ ← 2 flops

end

end

end

So the number of flops is:

$$\# \text{ flops} = \sum_{i=1}^{n-1} \left[\sum_{j=i+1}^n (2n - 2i + 3) \right]$$

$\downarrow l = j - i$

Counting flops for Gaussian elimination:

Input: augmented matrix $A \in \mathbb{R}^{n \times (n+1)}$

for $i = 1 : n - 1$

for $j = i + 1 : n$

$m \leftarrow A_{j,i}/A_{i,i}$

$\leftarrow 1 \text{ flop}$

$A(j, i) \leftarrow 0$

for $k = i + 1 : n + 1$

$A(j, k) \leftarrow A(j, k) - m A(i, k)$

$\leftarrow 2 \text{ flops}$

end

end

end

So the number of flops is:

$$\# \text{ flops} = \sum_{i=1}^{n-1} (2n - 2i + 3)(n - i)$$

Counting flops for Gaussian elimination:

Input: augmented matrix $A \in \mathbb{R}^{n \times (n+1)}$

for $i = 1 : n - 1$

for $j = i + 1 : n$

$m \leftarrow A_{j,i}/A_{i,i}$ $\leftarrow 1$ flop

$A(j, i) \leftarrow 0$

for $k = i + 1 : n + 1$

$A(j, k) \leftarrow A(j, k) - m A(i, k)$ $\leftarrow 2$ flops

end

end

end

So the number of flops is:

$$\# \text{ flops} = \boxed{\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n}$$

Computing LU decomposition (without pivoting):

Input: $A \in \mathbb{R}^{n \times n}$

- 1: $U \leftarrow A, L \leftarrow I$ (initialise matrices)
- 2: **for** $j = 1 : n - 1$ (loop through pivot columns)
- 3: **for** $i = j + 1 : n$
- 4: $L_{ij} \leftarrow U_{ij} / U_{jj}$ (store multiplier in L matrix)
- 5: **for** $k = j : n$
- 6: $U_{ik} \leftarrow U_{ik} - L_{ij} U_{jk}$ (update row i of U matrix)
- 7: **end for**
- 8: **end for**
- 9: **end for**

Output: Matrices L and U

Computing LU decomposition (without pivoting):

Input: $A \in \mathbb{R}^{n \times n}$

- 1: $U \leftarrow A, L \leftarrow I$ (initialise matrices)
- 2: **for** $j = 1 : n - 1$ (loop through pivot columns)
- 3: **for** $i = j + 1 : n$
- 4: $L_{ij} \leftarrow U_{ij} / U_{jj}$ (store multiplier in L matrix)
- 5: **for** $k = j : n$
- 6: $U_{ik} \leftarrow U_{ik} - L_{ij} U_{jk}$ (update row i of U matrix)
- 7: **end for**
- 8: **end for**
- 9: **end for**

Output: Matrices L and U

- ▶ Line 4: one \div ; Line 6: one $+$, one \times

Computing LU decomposition (without pivoting):

$$\text{\# flops} = \sum_{j=1}^{n-1} \left[\sum_{i=j+1}^n \left(1 + \sum_{k=j}^n 2 \right) \right]$$

Computing LU decomposition (without pivoting):

$$\begin{aligned}\# \text{ flops} &= \sum_{j=1}^{n-1} \left[\sum_{i=j+1}^n \left(1 + \sum_{k=j}^n 2 \right) \right] \\ &= \sum_{j=1}^{n-1} \left[\sum_{i=1}^{n-j} \left(1 + 2 \sum_{k=1}^{n-j+1} 1 \right) \right]\end{aligned}$$

Computing LU decomposition (without pivoting):

$$\begin{aligned}\# \text{ flops} &= \sum_{j=1}^{n-1} \left[\sum_{i=j+1}^n \left(1 + \sum_{k=j}^{\textcolor{red}{n}} 2 \right) \right] \\ &= \sum_{j=1}^{n-1} \left[\sum_{i=1}^{n-j} \left(1 + 2 \sum_{k=1}^{\textcolor{red}{n-j+1}} 1 \right) \right]\end{aligned}$$

Computing LU decomposition (without pivoting):

$$\begin{aligned}\#\text{ flops} &= \sum_{j=1}^{n-1} \left[\sum_{i=j+1}^n \left(1 + \sum_{k=j}^n 2 \right) \right] \\ &= \sum_{j=1}^{n-1} \left[\sum_{i=1}^{n-j} \left(1 + 2 \sum_{k=1}^{n-j+1} 1 \right) \right] \\ &= \sum_{j=1}^{n-1} \left[\sum_{i=1}^{n-j} (1 + 2(n - j + 1)) \right]\end{aligned}$$

Computing LU decomposition (without pivoting):

$$\begin{aligned}\#\text{ flops} &= \sum_{j=1}^{n-1} \left[\sum_{i=j+1}^n \left(1 + \sum_{k=j}^n 2 \right) \right] \\ &= \sum_{j=1}^{n-1} \left[\sum_{i=1}^{n-j} \left(1 + 2 \sum_{k=1}^{n-j+1} 1 \right) \right] \\ &= \sum_{j=1}^{n-1} \left[\sum_{i=1}^{n-j} (1 + 2(n - j + 1)) \right] = \sum_{j=1}^{n-1} \left[(2n - 2j + 3) \sum_{i=1}^{n-j} 1 \right]\end{aligned}$$


Computing LU decomposition (without pivoting):

$$\begin{aligned}\#\text{ flops} &= \sum_{j=1}^{n-1} \left[\sum_{i=j+1}^n \left(1 + \sum_{k=j}^n 2 \right) \right] \\ &= \sum_{j=1}^{n-1} \left[\sum_{i=1}^{n-j} \left(1 + 2 \sum_{k=1}^{n-j+1} 1 \right) \right] \\ &= \sum_{j=1}^{n-1} \left[\sum_{i=1}^{n-j} (1 + 2(n - j + 1)) \right] = \sum_{j=1}^{n-1} \left[(2n - 2j + 3) \sum_{i=1}^{n-j} 1 \right] \\ &= \sum_{j=1}^{n-1} [(2n - 2j + 3)(n - j)]\end{aligned}$$

Computing LU decomposition (without pivoting):

$$\begin{aligned}\#\text{ flops} &= \sum_{j=1}^{n-1} \left[\sum_{i=j+1}^n \left(1 + \sum_{k=j}^n 2 \right) \right] \\ &= \sum_{j=1}^{n-1} \left[\sum_{i=1}^{n-j} \left(1 + 2 \sum_{k=1}^{n-j+1} 1 \right) \right] \\ &= \sum_{j=1}^{n-1} \left[\sum_{i=1}^{n-j} (1 + 2(n-j+1)) \right] = \sum_{j=1}^{n-1} \left[(2n - 2j + 3) \sum_{i=1}^{n-j} 1 \right] \\ &= \sum_{j=1}^{n-1} [(2n - 2j + 3)(n - j)] \\ &= 2 \sum_{j=1}^{n-1} j^2 - (4n + 3) \sum_{j=1}^{n-1} j + (2n^2 + 3n) \sum_{j=1}^{n-1} 1\end{aligned}$$

$\sum_{j=1}^{n-1} j$

Computing LU decomposition (without pivoting):

$$\begin{aligned}\text{\# flops} &= \sum_{j=1}^{n-1} \left[\sum_{i=j+1}^n \left(1 + \sum_{k=j}^n 2 \right) \right] \\ &= \sum_{j=1}^{n-1} \left[\sum_{i=1}^{n-j} \left(1 + 2 \sum_{k=1}^{n-j+1} 1 \right) \right] \\ &= \sum_{j=1}^{n-1} \left[\sum_{i=1}^{n-j} (1 + 2(n-j+1)) \right] = \sum_{j=1}^{n-1} \left[(2n - 2j + 3) \sum_{i=1}^{n-j} 1 \right] \\ &= \sum_{j=1}^{n-1} [(2n - 2j + 3)(n - j)] \\ &= 2 \sum_{j=1}^{n-1} j^2 - (4n + 3) \sum_{j=1}^{n-1} j + (2n^2 + 3n) \sum_{j=1}^{n-1} 1 = \boxed{\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n}\end{aligned}$$

So the complexity of Gaussian elimination and LU-decomposition are the same.

In either case, to obtain the solution of the linear system, we also need forward / backward substitution:

$$\mathbf{Ax} = \mathbf{b} \rightarrow \text{Gaussian elimination} \rightarrow (\mathbf{LA})\mathbf{x} = (\mathbf{L}\mathbf{b})$$

with (\mathbf{LA}) upper triangular

$$\mathbf{Ax} = \mathbf{b} \rightarrow \text{LU-decomposition} \rightarrow (\mathbf{LU})\mathbf{x} = \mathbf{b} \rightarrow$$

first solve $\mathbf{L}\mathbf{z} = \mathbf{b}$ (lower triangular)

then solve $\mathbf{U}\mathbf{x} = \mathbf{z}$ (upper triangular)

So the complexity of solving the linear system is that of Gaussian elimination / LU-decomposition **plus** that of forward / backward substitution.

Computing solution of $U\mathbf{x} = \mathbf{c}$ by back substitution:

Input: $U \in \mathbb{R}^{n \times n}$, $\mathbf{c} \in \mathbb{R}^{n \times 1}$

- 1: **for** $k = n$ **to** 1 **step** -1
- 2: $x_k \leftarrow c_k$
- 3: **for** $\ell = k + 1 : n$
- 4: $x_k \leftarrow x_k - U_{k\ell}x_\ell \rightarrow 2$
- 5: **end for**
- 6: $x_k \leftarrow x_k / U_{kk} \rightarrow 1$
- 7: **end for**

(loop from last row)
 (initialise vector)
 (loop through rows beneath k)
 (use x_ℓ already computed)
 (divide by diagonal element)

Output: Vector $\mathbf{x} \in \mathbb{R}^{n \times 1}$ such that $U\mathbf{x} = \mathbf{c}$ (i.e., $\mathbf{x} = U^{-1}\mathbf{c}$)

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

Computing solution of $U\mathbf{x} = \mathbf{c}$ by back substitution:

Input: $U \in \mathbb{R}^{n \times n}$, $\mathbf{c} \in \mathbb{R}^{n \times 1}$

- 1: **for** $k = n$ **to** 1 **step** -1 (loop from last row)
- 2: $x_k \leftarrow c_k$ (initialise vector)
- 3: **for** $\ell = k + 1 : n$ (loop through rows beneath k)
- 4: $x_k \leftarrow x_k - U_{k\ell}x_\ell$ (use x_ℓ already computed)
- 5: **end for**
- 6: $x_k \leftarrow x_k / U_{kk}$ (divide by diagonal element)
- 7: **end for**

Output: Vector $\mathbf{x} \in \mathbb{R}^{n \times 1}$ such that $U\mathbf{x} = \mathbf{c}$ (i.e., $\mathbf{x} = U^{-1}\mathbf{c}$)

Algorithm concisely summarised by single formula:

OR

$$x_k = \frac{1}{U_{kk}} \left(c_k - \sum_{\ell=k+1}^n U_{k\ell}x_\ell \right) \quad (k = 1 : n)$$

Computing solution of $U\mathbf{x} = \mathbf{c}$ by back substitution:

$$x_k = \frac{1}{U_{kk}} \left(c_k - \sum_{\ell=k+1}^n U_{k\ell} x_\ell \right) \quad (k = n: (-1): 1)$$

Computing solution of $U\mathbf{x} = \mathbf{c}$ by back substitution:

$$x_k = \frac{1}{U_{kk}} \left(c_k - \sum_{\ell=k+1}^n U_{k\ell} x_\ell \right) \quad (k = n: (-1): 1)$$

- ▶ Computing x_k : $n - k$ \times , $n - k - 1$ $+$, 1 $-$, and 1 \div ,

Computing solution of $U\mathbf{x} = \mathbf{c}$ by back substitution:

$$x_k = \frac{1}{U_{kk}} \left(c_k - \sum_{\ell=k+1}^n U_{k\ell} x_\ell \right) \quad (k = n: (-1): 1)$$

- ▶ Computing x_k : $n - k$ \times , $n - k - 1$ $+$, 1 $-$, and 1 \div ,
- ⇒ to compute x_k requires $2n - 2k + 1$ flops

Computing solution of $U\mathbf{x} = \mathbf{c}$ by back substitution:

$$x_k = \frac{1}{U_{kk}} \left(c_k - \sum_{\ell=k+1}^n U_{k\ell} x_\ell \right) \quad (k = n: (-1): 1)$$

- ▶ Computing x_k : $n - k$ \times , $n - k - 1$ $+$, 1 $-$, and 1 \div ,
- ⇒ to compute x_k requires $2n - 2k + 1$ flops
- ▶ Computing **all** x_k for $k = 1 : n$ requires

$$\text{Cost} = \sum_{k=1}^n (2n - 2k + 1)$$

Computing solution of $U\mathbf{x} = \mathbf{c}$ by back substitution:

$$x_k = \frac{1}{U_{kk}} \left(c_k - \sum_{\ell=k+1}^n U_{k\ell} x_\ell \right) \quad (k = n: (-1): 1)$$

- ▶ Computing x_k : $n - k$ \times , $n - k - 1$ $+$, 1 $-$, and 1 \div ,
- ⇒ to compute x_k requires $2n - 2k + 1$ flops
- ▶ Computing **all** x_k for $k = 1:n$ requires

$$\begin{aligned} \text{Cost} &= \sum_{k=1}^n (2n - 2k + 1) \\ &= (2n + 1) \sum_{k=1}^n 1 - 2 \sum_{k=1}^n k \end{aligned}$$

Computing solution of $U\mathbf{x} = \mathbf{c}$ by back substitution:

$$x_k = \frac{1}{U_{kk}} \left(c_k - \sum_{\ell=k+1}^n U_{k\ell}x_\ell \right) \quad (k = n:(-1):1)$$

- ▶ Computing x_k : $n - k$ \times , $n - k - 1$ $+$, 1 $-$, and 1 \div ,
- ⇒ to compute x_k requires $2n - 2k + 1$ flops
- ▶ Computing all x_k for $k = 1:n$ requires

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

Computing solution of $U\mathbf{x} = \mathbf{c}$ by back substitution:

$$x_k = \frac{1}{U_{kk}} \left(c_k - \sum_{\ell=k+1}^n U_{k\ell} x_\ell \right) \quad (k = n: (-1): 1)$$

- ▶ Computing x_k : $n - k$ \times , $n - k - 1$ $+$, 1 $-$, and 1 \div ,
- \Rightarrow to compute x_k requires $2n - 2k + 1$ flops
- ▶ Computing all x_k for $k = 1:n$ requires

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

Computing solution of $U\mathbf{x} = \mathbf{c}$ by back substitution:

$$x_k = \frac{1}{U_{kk}} \left(c_k - \sum_{\ell=k+1}^n U_{k\ell}x_\ell \right) \quad (k = n: (-1): 1)$$

- ▶ Computing x_k : $n - k$ \times , $n - k - 1$ $+$, 1 $-$, and 1 \div ,
- ⇒ to compute x_k requires $2n - 2k + 1$ flops
- ▶ Computing all x_k for $k = 1:n$ requires

$$\begin{aligned}\text{Cost} &= \sum_{k=1}^n (2n - 2k + 1) \\ &= (2n + 1) \sum_{k=1}^n 1 - 2 \sum_{k=1}^n k \\ &= (2n + 1)(n) - 2 \left(\frac{n(n + 1)}{2} \right) = \boxed{n^2 \text{ flops}}\end{aligned}$$

Naive polynomial evaluation:

- ▶ Consider the polynomial $f_1(x)$ defined by

$$f_1(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$$

Naive polynomial evaluation:

- ▶ Consider the polynomial $f_1(x)$ defined by

$$\begin{aligned}f_1(x) &= \cancel{x^7} - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \\&= (1) \times \cancel{x} +\end{aligned}$$

Naive polynomial evaluation:

- ▶ Consider the polynomial $f_1(x)$ defined by

$$\begin{aligned}f_1(x) &= x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \\&= (1) \times x + \\&\quad (-7) \times x +\end{aligned}$$

Naive polynomial evaluation:

- ▶ Consider the polynomial $f_1(x)$ defined by

$$\begin{aligned}f_1(x) &= x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \\&= (1) \times x + \\&\quad (-7) \times x + \\&\quad (21) \times x \times x \times x \times x \times x \times x\end{aligned}$$

Naive polynomial evaluation:

- ▶ Consider the polynomial $f_1(x)$ defined by

$$\begin{aligned}f_1(x) &= x^7 - 7x^6 + 21x^5 - \color{red}{35x^4} + 35x^3 - 21x^2 + 7x - 1 \\&= (1) \times x + \\&\quad (-7) \times x + \\&\quad (21) \times x \times x \times x \times x \times x \times x + \\&\quad \color{red}{(-35)} \times x \times x \times x \times x \times x +\end{aligned}$$

Naive polynomial evaluation:

- ▶ Consider the polynomial $f_1(x)$ defined by

$$\begin{aligned}f_1(x) &= x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \\&= (1) \times x + \\&\quad (-7) \times x + \\&\quad (21) \times x \times x \times x \times x \times x \times x + \\&\quad (-35) \times x \times x \times x \times x \times x + \\&\quad (35) \times x \times x \times x +\end{aligned}$$

Naive polynomial evaluation:

- ▶ Consider the polynomial $f_1(x)$ defined by

$$\begin{aligned}f_1(x) &= x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - \color{red}{21x^2} + 7x - 1 \\&= (1) \times x + \\&\quad (-7) \times x + \\&\quad (21) \times x \times x \times x \times x \times x \times x + \\&\quad (-35) \times x \times x \times x \times x \times x + \\&\quad (35) \times x \times x \times x + \\&\quad \color{red}{(-21)} \times x \times x +\end{aligned}$$

Naive polynomial evaluation:

- ▶ Consider the polynomial $f_1(x)$ defined by

$$\begin{aligned}f_1(x) &= x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \\&= (1) \times x + \\&\quad (-7) \times x + \\&\quad (21) \times x \times x \times x \times x \times x \times x + \\&\quad (-35) \times x \times x \times x \times x \times x + \\&\quad (35) \times x \times x \times x + \\&\quad (-21) \times x \times x + \\&\quad (7) \times x +\end{aligned}$$

Naive polynomial evaluation:

- ▶ Consider the polynomial $f_1(x)$ defined by

$$\begin{aligned}f_1(x) &= x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \\&= (1) \times x + \\&\quad (-7) \times x + \\&\quad (21) \times x \times x \times x \times x \times x \times x + \\&\quad (-35) \times x \times x \times x \times x \times x + \\&\quad (35) \times x \times x \times x + \\&\quad (-21) \times x \times x + \\&\quad (7) \times x + \\&\quad (-1)\end{aligned}$$

Naive polynomial evaluation:

- ▶ Consider the polynomial $f_1(x)$ defined by

$$\begin{aligned}f_1(x) &= x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1 \\&= (1) \times x + \cancel{7} \\&\quad (-7) \times x + \cancel{6} \\&\quad (21) \times x \times x \times x \times x \times x \times x + \cancel{5} \\&\quad (-35) \times x \times x \times x \times x \times x + \cancel{4} \\&\quad (35) \times x \times x \times x + \cancel{3} \\&\quad (-21) \times x \times x + \cancel{2} \\&\quad (7) \times x + \cancel{1} \\&\quad (-1) \quad \cancel{1}\end{aligned}$$

- ▶ 28 $\boxed{\times}$ & 7 $\boxed{+}$ to compute $f_1(x)$

$$x(x^6 - 7x^5) + 21x^5 \equiv x(21x^4 + x(x^5 - 7x^4))$$

Naive polynomial evaluation vs. nested evaluation:

- $(x^7 - 7x^6) + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$ in **nested form** is

$$f_2(x) = -1 + x(7 + x(-21 + x(35 + x(-35 + x(21 + x(-7 + x)))))))$$

x^7

Naive polynomial evaluation vs. nested evaluation:

- ▶ $x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$ in **nested form** is

$$f_2(x) = -1 + x(7 + x(-21 + x(35 + x(-35 + x(21 + x(-7 + x)))))))$$

- ▶ 6 \times & 7 $+$ in $f_2(x)$ $= 13$

Naive polynomial evaluation vs. nested evaluation:

- ▶ $x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$ in **nested form** is

$$f_2(x) = -1 + x(7 + x(-21 + x(35 + x(-35 + x(21 + x(-7 + x)))))))$$

- ▶ 6 \times & 7 $+$ in $f_2(x)$
- ▶ $f_1(x) \equiv f_2(x)$ algebraically
- ▶ However, $f_2(x)$ has dramatically lower operation count

Naive algorithm for polynomial evaluation:

Input: $\{a_k\}_{k=0}^n, x \in \mathbb{R}$

$$p(x) = a_0 + a_1x^1 + \cdots + a_nx^n$$

1: $y \leftarrow a_0$ 2: **for** $k = 1 : n$ 3: term $\leftarrow a_k$ 4: **for** $\ell = 1 : k$ 5: term \leftarrow term $\times x$ 6: **end for**7: $y \leftarrow y + \text{term}$ 8: **end for****Output:** $y = p(x) = \sum_{k=0}^n a_k x^k$

Naive algorithm for polynomial evaluation:

Input: $\{a_k\}_{k=0}^n, x \in \mathbb{R}$

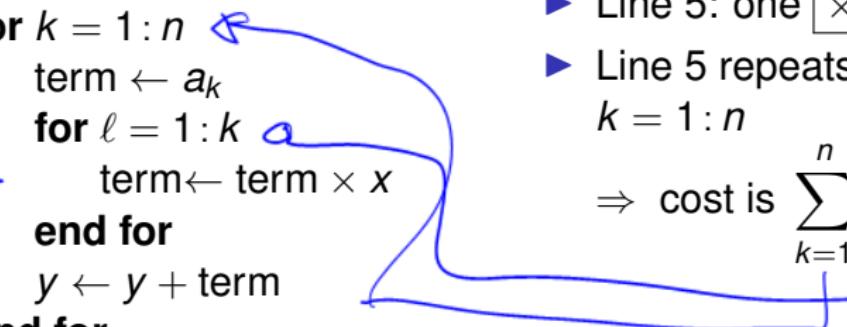
$$p(x) = a_0 + a_1x^1 + \cdots + a_nx^n$$

1: $y \leftarrow a_0$ 2: **for** $k = 1 : n$ 3: term $\leftarrow a_k$ 4: **for** $\ell = 1 : k$ → 5: term \leftarrow term $\times x$ 6: **end for**→ 7: $y \leftarrow y + \text{term}$ 8: **end for****Output:** $y = p(x) = \sum_{k=0}^n a_k x^k$ ▶ Line 5: one \times ; Line 7: one $+$

Naive algorithm for polynomial evaluation:

Input: $\{a_k\}_{k=0}^n, x \in \mathbb{R}$

```
1:  $y \leftarrow a_0$ 
2: for  $k = 1 : n$  do
3:   term  $\leftarrow a_k$ 
4:   for  $\ell = 1 : k$  do
5:     term  $\leftarrow$  term  $\times x$ 
6:   end for
7:    $y \leftarrow y + \text{term}$ 
8: end for
```



$$p(x) = a_0 + a_1 x^1 + \cdots + a_n x^n$$

- ▶ Line 5: one \times ; Line 7: one $+$
- ▶ Line 5 repeats for $\ell = 1 : k$,
 $k = 1 : n$

$$\Rightarrow \text{cost is } \sum_{k=1}^n \sum_{\ell=1}^k 1 = \frac{n(n+1)}{2}$$

Output: $y = p(x) = \sum_{k=0}^n a_k x^k$

Naive algorithm for polynomial evaluation:

Input: $\{a_k\}_{k=0}^n, x \in \mathbb{R}$ 1: $y \leftarrow a_0$ 2: **for** $k = 1 : n$ 3: term $\leftarrow a_k$ 4: **for** $\ell = 1 : k$ 5: term \leftarrow term $\times x$ 6: **end for**7: $y \leftarrow y + \text{term}$ 8: **end for****Output:** $y = p(x) = \sum_{k=0}^n a_k x^k$

$$p(x) = a_0 + a_1 x^1 + \cdots + a_n x^n$$

- Line 5: one \times ; Line 7: one $+$

- Line 5 repeats for $\ell = 1 : k$, $k = 1 : n$

\Rightarrow cost is $\sum_{k=1}^n \sum_{\ell=1}^k 1 = \frac{n(n+1)}{2}$

- Line 7 repeats for $k = 1 : n$

\Rightarrow cost is $\sum_{k=1}^n 1 = n$ flops

$$\sum_{k=1}^n \left(1 + \sum_{\ell=1}^k 1 \right) = \left(\sum_{k=1}^n 1 \right)$$

$$\begin{aligned} \sum_{k=1}^n \left(\sum_{\ell=1}^k 1 \right) &= n + \sum_{k=1}^n k \\ &\approx n + \frac{n(n+1)}{2} \end{aligned}$$

Naive algorithm for polynomial evaluation:

Input: $\{a_k\}_{k=0}^n, x \in \mathbb{R}$

```
1:  $y \leftarrow a_0$ 
2: for  $k = 1 : n$ 
3:   term  $\leftarrow a_k$ 
4:   for  $\ell = 1 : k$ 
5:     term  $\leftarrow$  term  $\times x$ 
6:   end for
7:    $y \leftarrow y + \text{term}$ 
8: end for
```

Output: $y = p(x) = \sum_{k=0}^n a_k x^k$

$$p(x) = a_0 + a_1 x^1 + \cdots + a_n x^n$$

- ▶ Line 5: one \times ; Line 7: one $+$
- ▶ Line 5 repeats for $\ell = 1 : k$,
 $k = 1 : n$
 \Rightarrow cost is $\sum_{k=1}^n \sum_{\ell=1}^k 1 = \frac{n(n+1)}{2}$
- ▶ Line 7 repeats for $k = 1 : n$
 \Rightarrow cost is $\sum_{k=1}^n 1 = n$ flops
- ▶ Total cost is $\frac{n^2 + 3n}{2}$ flops

Horner algorithm for nested polynomial evaluation:

$$\begin{aligned} p(x) &= \sum_{k=0}^n a_k x^k \\ &= a_0 + a_1 x^1 + \cdots + a_n x^n \\ &\equiv \underbrace{a_0 + (a_1 + (a_2 + \cdots (a_{n-1} + a_n x) x \cdots) x)}_{p(x) \text{ rewritten in nested form}} x \end{aligned}$$

Input: $\{a_k\}_{k=0}^n, x \in \mathbb{R}$

1: $y \leftarrow a_n$
2: **for** $k = (n - 1)$ **to** 0 **step** -1
3: $y \leftarrow x \times y + a_k$
4: **end for**

Output: $y = p(x)$

Horner algorithm for nested polynomial evaluation:

$$\begin{aligned} p(x) &= \sum_{k=0}^n a_k x^k \\ &= a_0 + a_1 x^1 + \cdots + a_n x^n \\ &\equiv \underbrace{a_0 + (a_1 + (a_2 + \cdots (a_{n-1} + a_n x) x \cdots) x)}_{p(x) \text{ rewritten in nested form}} x \end{aligned}$$

Input: $\{a_k\}_{k=0}^n, x \in \mathbb{R}$

- 1: $y \leftarrow a_n$
- 2: **for** $k = (n - 1)$ **to** 0 **step** -1
- 3: $y \leftarrow x \times y + a_k$
- 4: **end for**

Output: $y = p(x)$

► Observe loop counter
decreasing

Horner algorithm for nested polynomial evaluation:

$$\begin{aligned} p(x) &= \sum_{k=0}^n a_k x^k \\ &= a_0 + a_1 x^1 + \cdots + a_n x^n \\ &\equiv \underbrace{a_0 + (a_1 + (a_2 + \cdots (a_{n-1} + a_n x) \cdots)x)x}_{p(x) \text{ rewritten in nested form}} \end{aligned}$$

Input: $\{a_k\}_{k=0}^n, x \in \mathbb{R}$

1: $y \leftarrow a_n$

~~2:~~ **for** $k = (n - 1)$ **to** 0 **step** -1

3: $y \leftarrow x \times y + a_k$ ~~5~~

4: **end for**

Output: $y = p(x)$

- ▶ Observe loop counter **decreasing**
- ▶ Line 3: one $+$ & one \times
- ▶ Line 3 executes once for $k = 0 : n - 1$
- ▶ Total cost is $\sum_{k=0}^{n-1} 2$

Horner algorithm for nested polynomial evaluation:

$$\begin{aligned} p(x) &= \sum_{k=0}^n a_k x^k \\ &= a_0 + a_1 x^1 + \cdots + a_n x^n \\ &\equiv \underbrace{a_0 + (a_1 + (a_2 + \cdots (a_{n-1} + a_n x) x \cdots) x)}_{p(x) \text{ rewritten in nested form}} x \end{aligned}$$

Input: $\{a_k\}_{k=0}^n, x \in \mathbb{R}$

1: $y \leftarrow a_n$
2: **for** $k = (n - 1)$ **to** 0 **step** -1
3: $y \leftarrow x \times y + a_k$
4: **end for**

Output: $y = p(x)$

► Observe loop counter
decreasing

► Line 3: one $+$ & one \times

► Line 3 executes once for
 $k = 0 : n - 1$

► Total cost is $\sum_{k=0}^{n-1} 2 =$ **$2n$ flops**

Some conclusions:

- ▶ The total cost of solving a linear system with Gaussian elimination is
 - elimination flops + backward substitution flops =

$$\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n + n^2 = \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n$$

Some conclusions:

- ▶ The total cost of solving a linear system with Gaussian elimination is

$$\text{elimination flops} + \text{backward substitution flops} =$$

$$\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n + n^2 = \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n$$

=

- ▶ And with LU-decomposition:

$$\text{decomposition flops} + \text{backward flops} + \text{forward flops} =$$

$$\frac{2}{3}n^3 + \frac{5}{2}n^2 - \frac{7}{6}n$$

=

Some conclusions:

- ▶ The total cost of solving a linear system with Gaussian elimination is

$$\text{elimination flops} + \text{backward substitution flops} =$$

$$\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n + n^2 = \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n$$

- ▶ And with LU-decomposition:

$$\text{decomposition flops} + \text{backward flops} + \text{forward flops} =$$

$$\frac{2}{3}n^3 + \frac{5}{2}n^2 - \frac{7}{6}n$$

- ▶ The cost of evaluating a polynomial of order n is $2n$ – when done in the right way.

Some conclusions:

- ▶ The total cost of solving a linear system with Gaussian elimination is

elimination flops + backward substitution flops =

$$\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n + n^2 = \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n$$

- ▶ And with LU-decomposition:

decomposition flops + backward flops + forward flops =

$$\frac{2}{3}n^3 + \frac{5}{2}n^2 - \frac{7}{6}n$$

- ▶ The cost of evaluating a polynomial of order n is $2n$ – when done in the right way.
- ▶ A simple re-ordering can reduce the complexity drastically!

Some conclusions:

- ▶ The total cost of solving a linear system with Gaussian elimination is

$$\text{elimination flops} + \text{backward substitution flops} =$$

$$\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n + n^2 = \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n$$

- ▶ And with LU-decomposition:

$$\text{decomposition flops} + \text{backward flops} + \text{forward flops} =$$

$$\frac{2}{3}n^3 + \frac{5}{2}n^2 - \frac{7}{6}n$$

- ▶ The cost of evaluating a polynomial of order n is $2n$ – when done in the right way.
- ▶ A simple re-ordering can reduce the complexity drastically!

If n is large only the *leading term* matters...

The “Big-Oh” symbol is useful to make this observation formal.



Definition (“Big-Oh”)

Let $\{\underline{x^{(n)}}\}$ and $\{\underline{y^{(n)}}\}$ be two sequences. Then $\underline{x^{(n)}} = O(\underline{y^{(n)}})$ (pronounced $x^{(n)}$ is “big-Oh” of $y^{(n)}$) iff there exist constants C and N such that $|x^{(n)}| \leq C|y^{(n)}|$ whenever $n \geq N$.

- ▶ $x^{(n)} = O(y^{(n)})$ means $\{x^{(n)}\}$ asymptotically dominated by $\{y^{(n)}\}$
- ▶ If $x^{(n)} = O(y^{(n)})$ then $\lim_{n \rightarrow \infty} |\frac{x^{(n)}}{y^{(n)}}| \leq C$ for some finite $C \geq 0$
- ▶ “Infinite asymptotics”: behaviour of sequences as $n \rightarrow \infty$

$$\lim_{x \rightarrow \infty} \frac{2x^3 + x^2}{x^3} = 2$$

Definition (“Big-Oh”)

Let $\{x^{(n)}\}$ and $\{y^{(n)}\}$ be two sequences. Then $x^{(n)} = O(y^{(n)})$ (pronounced $x^{(n)}$ is “big-Oh” of $y^{(n)}$) iff there exist constants C and N such that $|x^{(n)}| \leq C|y^{(n)}|$ whenever $n \geq N$.

- ▶ $x^{(n)} = O(y^{(n)})$ means $\{x^{(n)}\}$ asymptotically dominated by $\{y^{(n)}\}$
- ▶ If $x^{(n)} = O(y^{(n)})$ then $\lim_{n \rightarrow \infty} \left| \frac{x^{(n)}}{y^{(n)}} \right| \leq C$ for some finite $C \geq 0$
- ▶ “Infinite asymptotics”: behaviour of sequences as $n \rightarrow \infty$
- ▶ LU-decomposition uses $\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n$ flops and so is $O(n^3)$. (order n^3)

Definition (“Big-Oh”)

Let $\{x^{(n)}\}$ and $\{y^{(n)}\}$ be two sequences. Then $x^{(n)} = O(y^{(n)})$ (pronounced $x^{(n)}$ is “big-Oh” of $y^{(n)}$) iff there exist constants C and N such that $|x^{(n)}| \leq C|y^{(n)}|$ whenever $n \geq N$.

- ▶ $x^{(n)} = O(y^{(n)})$ means $\{x^{(n)}\}$ asymptotically dominated by $\{y^{(n)}\}$
- ▶ If $x^{(n)} = O(y^{(n)})$ then $\lim_{n \rightarrow \infty} \left| \frac{x^{(n)}}{y^{(n)}} \right| \leq C$ for some finite $C \geq 0$
- ▶ “Infinite asymptotics”: behaviour of sequences as $n \rightarrow \infty$
- ▶ LU-decomposition uses $\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n$ flops and so is $O(n^3)$.
- ▶ Forward / backward substitution uses n^2 flops and so is $O(n^2)$.

Definition (“Big-Oh”)

Let $\{x^{(n)}\}$ and $\{y^{(n)}\}$ be two sequences. Then $x^{(n)} = O(y^{(n)})$ (pronounced $x^{(n)}$ is “big-Oh” of $y^{(n)}$) iff there exist constants C and N such that $|x^{(n)}| \leq C|y^{(n)}|$ whenever $n \geq N$.

- ▶ $x^{(n)} = O(y^{(n)})$ means $\{x^{(n)}\}$ asymptotically dominated by $\{y^{(n)}\}$
- ▶ If $x^{(n)} = O(y^{(n)})$ then $\lim_{n \rightarrow \infty} \left| \frac{x^{(n)}}{y^{(n)}} \right| \leq C$ for some finite $C \geq 0$
- ▶ “Infinite asymptotics”: behaviour of sequences as $n \rightarrow \infty$
- ▶ LU-decomposition uses $\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n$ flops and so is $O(n^3)$. ←
- ▶ Forward / backward substitution uses n^2 flops and so is $O(n^2)$. ←
- ▶ “Smart” polynomial evaluation uses $2n$ flops and so is $O(n)$. ←

How to visualize this?

$2n^3 + 2n^2$; as n get larger this looks like $2n^3$

If the number of flops f grows as $O(n^p)$, then for large n

$$f \approx \alpha n^p \text{ for some positive alpha}$$

so that

$$\ln(f) \approx (\ln(\alpha) + p \ln(n))$$

so that $\ln(f)$ depends linearly on $\ln(n)$. Plot on a logarithmic scale and find the slope...

$$y = b + mx$$

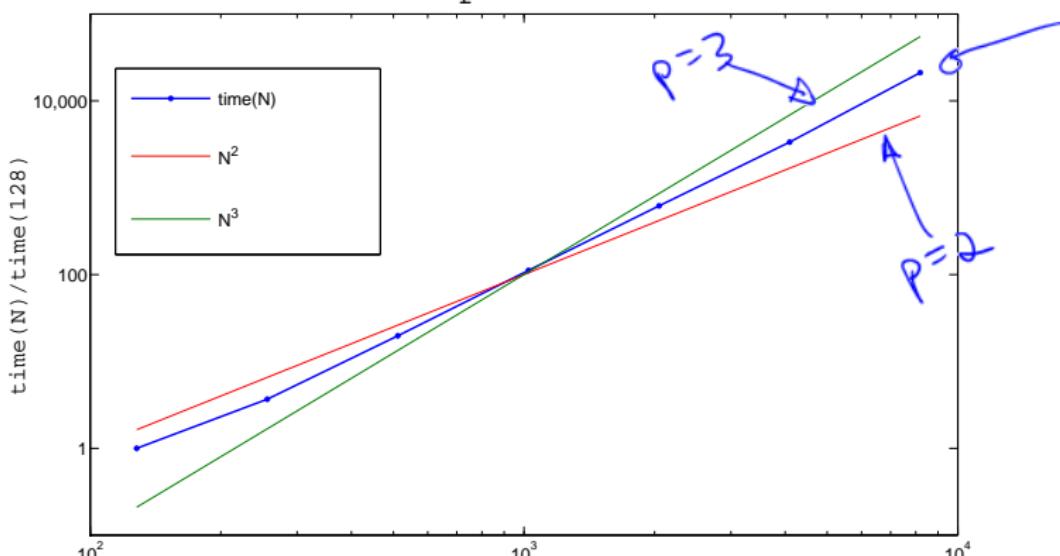
Example: LU-decomposition.

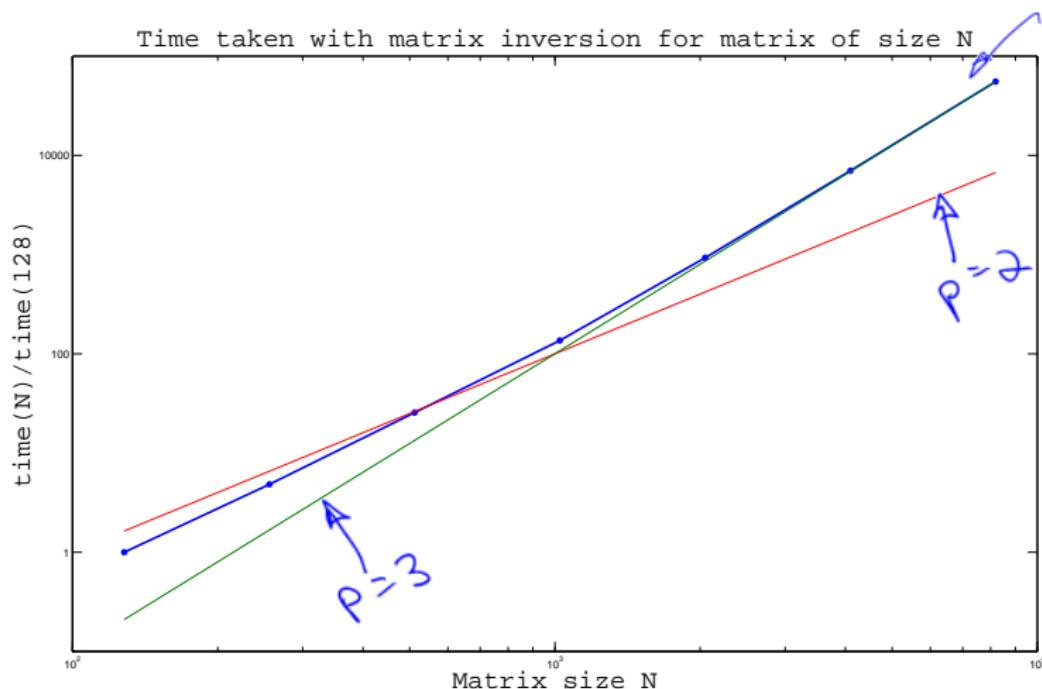
$$f = \frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n \approx \frac{2}{3}n^3$$

so we expect a straight line with slope 3.

In practice, the slope is in between 2 and 3. Highly optimized linear algebra routines can achieve a scaling close to $p = 2.4$.

Time taken for LU-decomposition of a matrix a size \mathbf{l}





The *Vandermonde matrix* arises in interpolation problems, which we will be covering soon in class. One form of the Vandermonde matrix has elements:

$$V_{ij} = x_{i-1}^{j-1} \text{ for } 1 \leq i \leq N+1 \text{ and } 1 \leq j \leq N+1, \quad (1)$$

where $x_i = i/N$, for $i = 0, \dots, N$, (which are equally spaced points between 0 and 1). For $N = 4$, the matrix looks like:

$$V = \begin{bmatrix} 1 & x_0 & x_0^2 & x_0^3 & x_0^4 \\ 1 & x_1 & x_1^2 & x_1^3 & x_1^4 \\ 1 & x_2 & x_2^2 & x_2^3 & x_2^4 \\ 1 & x_3 & x_3^2 & x_3^3 & x_3^4 \\ 1 & x_4 & x_4^2 & x_4^3 & x_4^4 \end{bmatrix}$$

with $(x_0, x_1, x_2, x_3, x_4) = (0, 1/4, 1/2, 3/4, 1)$.

It turns out that as N gets large, this matrix becomes very poorly conditioned, and therefore the solutions of linear systems involving it become inaccurate. In this question we explore this further.

- (a) Write a psuedo-code for building the matrix V , for a given N and with elements given in equation (1), and compute the computational complexity of your algorithm, i.e. for a given N , determine the number of FLOPS it takes to build the matrix. In terms of “Big-Oh” notation, what is the asymptotic behaviour of your algorithm as N increases? Try to make your algorithm

Pseudo code for constructing Vandermonde matrix

Input. integer N

1. Initialize V as empty $N+1 \times N+1$ matrix

2. for $i=1, N+1$ do (rows)

3. for $j=1, N+1$ do (columns)

(a) $V_{ij} \leftarrow \left(\frac{(i-1)}{N} \right)^{j-1}$ \rightarrow

4. end j

5. end i

1 add, 1 divide, j^2 mult

Output: V an $N+1 \times N+1$ matrix

$$\text{flops: } \sum_{i=1}^{N+1} \sum_{j=1}^{N+1} j \quad (\text{from line } 3(a))$$

$$= \sum_{i=1}^{N+1} \left[\frac{(N+1)(N+1-1)}{2} \right]$$

$$\sum_{i=1}^n i = \frac{n(n-1)}{2}$$

$$= \sum_{i=1}^{N+1} \frac{N(N+1)}{2}$$

$$= \frac{N(N+1)^2}{2}$$

$$= \frac{1}{2} N^3 + N^2 + \frac{1}{2} N$$

$$= \mathcal{O}(N^3)$$

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. Newton iteration for systems

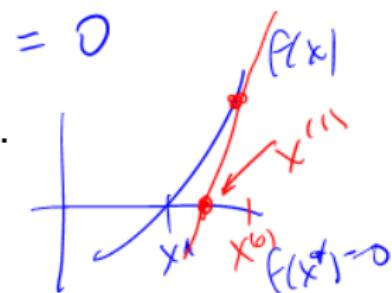
2. Pseudo-code

3. Trivial test

We want to find the solution to a 2D system, i.e. we want to find x_1 and x_2 such that

$$\left\{ \begin{array}{l} f_1(x_1, x_2) = 0 \\ f_2(x_1, x_2) = 0 \end{array} \right. \quad \begin{aligned} & x_1^2 + x_2 = 0 \\ & x_1 - x_2 + 1 = 0 \end{aligned}$$

If equations are linear, we can use LU-decomp.
What if they're nonlinear?



Newton-Raphson iteration can be generalized to n equations with n unknowns.

Alternative derivation in 1D:

$$f(x + \delta x) \approx f(x) + f'(x)\delta x = 0 \Rightarrow \delta x = -\frac{f(x)}{f'(x)}$$

Note that, in general, we need **the same number of equations and unknowns** to find (isolated) solutions...

$$f_1(x_1 + \delta x_1, x_2 + \delta x_2) \approx f_1(x_1, x_2) + \frac{\partial f_1}{\partial x_1}(x_1, x_2)\delta x_1 + \frac{\partial f_1}{\partial x_2}(x_1, x_2)\delta x_2 \quad \approx^0$$
$$f_2(x_1 + \delta x_1, x_2 + \delta x_2) \approx f_2(x_1, x_2) + \frac{\partial f_2}{\partial x_1}(x_1, x_2)\delta x_1 + \frac{\partial f_2}{\partial x_2}(x_1, x_2)\delta x_2 \quad \approx^0$$

which is a linear system of equations to solve for $(\delta x_1, \delta x_2)$.

In matrix form:

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x_1, x_2) & \frac{\partial f_1}{\partial x_2}(x_1, x_2) \\ \frac{\partial f_2}{\partial x_1}(x_1, x_2) & \frac{\partial f_2}{\partial x_2}(x_1, x_2) \end{pmatrix} \begin{pmatrix} \delta x_1 \\ \delta x_2 \end{pmatrix} = - \begin{pmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{pmatrix}$$

J

which we can
find

↑
update vector ↑
right-hand side

$$\underline{\delta x} = \underline{b}$$

Problem statement: given a function $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ and an initial guess $\mathbf{x}^{(0)} \in \mathbb{R}^n$, find an approximate solution to $\mathbf{f}(\mathbf{x}) = \mathbf{0}$.

The derivation of the iterative method is similar to that of Newton iteration:

$$\rightarrow f(x + \delta x) \approx f(x) + f'(x)\delta x = 0 \Rightarrow \delta x = -\frac{f(x)}{f'(x)} \text{ for } f(x), x, \delta x \in \mathbb{R}$$

becomes

$$\rightarrow \mathbf{f}(\mathbf{x} + \delta \mathbf{x}) \approx \mathbf{f}(\mathbf{x}) + \underline{D\mathbf{f}} \delta \mathbf{x} = 0 \Rightarrow D\mathbf{f} \delta \mathbf{x} = -\mathbf{f}(\mathbf{x}) \text{ for } \mathbf{f}(\mathbf{x}), \mathbf{x}, \delta \mathbf{x} \in \mathbb{R}^n$$

That is, the update $\delta \mathbf{x}$ is found from the linear system:

$$D\mathbf{f} \delta \mathbf{x} = \begin{pmatrix} \partial_{x_1} f_1(\mathbf{x}) & \partial_{x_2} f_1(\mathbf{x}) & \cdots & \partial_{x_n} f_1(\mathbf{x}) \\ \partial_{x_1} f_2(\mathbf{x}) & \partial_{x_2} f_2(\mathbf{x}) & \cdots & \partial_{x_n} f_2(\mathbf{x}) \\ \vdots & \vdots & & \vdots \\ \partial_{x_1} f_n(\mathbf{x}) & \partial_{x_2} f_n(\mathbf{x}) & \cdots & \partial_{x_n} f_n(\mathbf{x}) \end{pmatrix} \begin{pmatrix} \delta x_1 \\ \delta x_2 \\ \vdots \\ \delta x_n \end{pmatrix} = - \begin{pmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{x}) \end{pmatrix}$$

Jacobian

Reminder: Pseudo-code for Newton-Raphson iteration

Input: $f, f', x^{(0)}$, ϵ_{st}

for $k = 0, 1, 2, \dots$ **until** convergence

$\rightarrow r^{(k)} \leftarrow f(x^{(k)})$ (evaluate nonlinear residual)

$\rightarrow \delta x^{(k)} \leftarrow -[f'(x^{(k)})]^{-1} r^{(k)}$ (compute Newton-Raphson step)

$\rightarrow x^{(k+1)} \leftarrow x^{(k)} + \delta x^{(k)}$ (compute next iterate)

 Test for convergence

\rightarrow if $|\delta x| < \epsilon_{\text{st}}$ then break.
 ↑

end for

Output: $x^{(k)}$

- ▶ Terminology: $r^{(k)} := f(x^{(k)})$ = **residual**
- ▶ Terminology: $\delta x^{(k)} := -[f'(x^{(k)})]^{-1} r^{(k)}$ = **update**

Function NewtonSystem

Input: f , Df , $x^{(0)}$, epsx
 Initializations:

Output: $x^{(k+1)}$

1. $\text{conv} \leftarrow \text{False}$

2. For $k = 0, 1, 2 \dots$ convergence or max iterations

Evaluate the residual vector:

a. $r_vec \leftarrow f(x^{(k)})$

Evaluate the Jacobian:

b. $J \leftarrow Df(x^{(k)})$

Solve the linear system:

c. $\delta x \leftarrow \text{scipy.linalg.solve}(J, -r_vec)$

Update the solution:

d. $x^{(k+1)} \leftarrow x^{(k)} + \delta x$

Check for convergence:

(δx is a vector).

e. if $\text{norm}(\delta x) \leq \text{epsx}$, then $\text{conv} = \text{True}$

end for



e.g. $\|\delta x\|_2$

First test problem:

$$\left\{ \begin{array}{l} f_1(x_1, x_2) = 2 \exp(x_1 x_2) - 2x_1 + 2x_2 - 2 = 0 \\ f_2(x_1, x_2) = x_1^5 + x_1 x_2^5 - 2x_2 = 0 \end{array} \right.$$

with the exact solution $\mathbf{x}^* = (0, 0)^t$. The Jacobian is:

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{pmatrix}$$

$$J = \begin{bmatrix} 2 \exp(x_1 x_2) x_2 - 2 & 2 \exp(x_1 x_2) x_1 + 2 \\ 5x_1^4 + x_2^5 & 5x_1 x_2^4 - 2 \end{bmatrix}$$

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. Interpolation of data
2. Polynomial interpolation
3. Polynomial interpolation in a monomial basis
4. What conditions, how fast, how accurate?

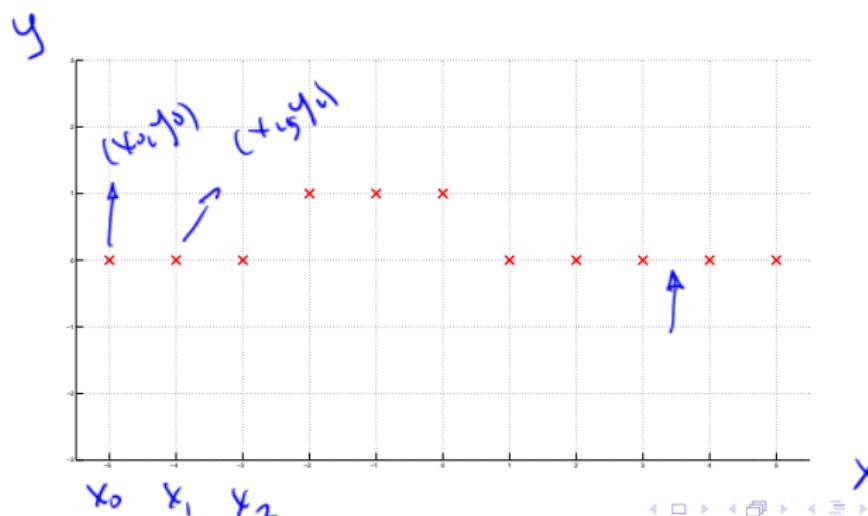
Key questions:

- ▶ What is an interpolant?
- ▶ How is the interpolation problem defined?
- ▶ What does polynomial interpolation mean?
- ▶ How is the solution of a polynomial interpolation problem found?
- ▶ What is a Vandermonde matrix?
- ▶ Our 3 questions: What conditions, how fast, how accurate?

Interpolation problem

Given $n + 1$ data points $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$ with x_k distinct ($k = 0 : n$), determine a function \tilde{f} that satisfies

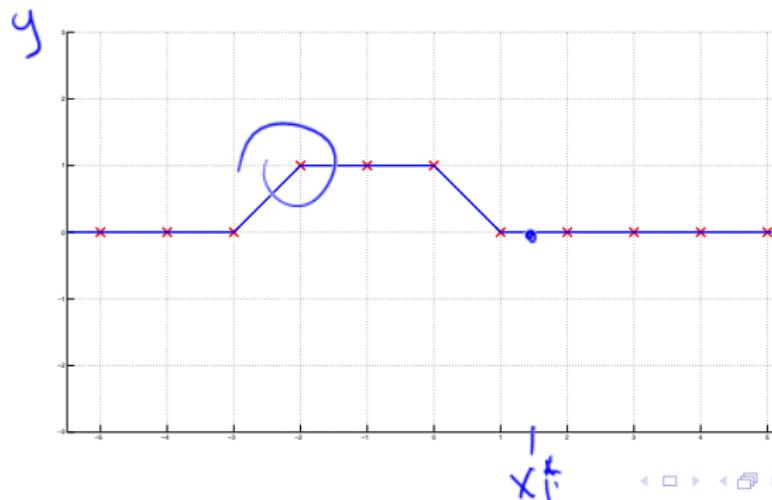
$$\tilde{f}(x_k) = y_k \quad (k = 0 : n).$$



Interpolation problem

Given $n + 1$ data points $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$ with x_k distinct ($k = 0 : n$), determine a function \tilde{f} that satisfies

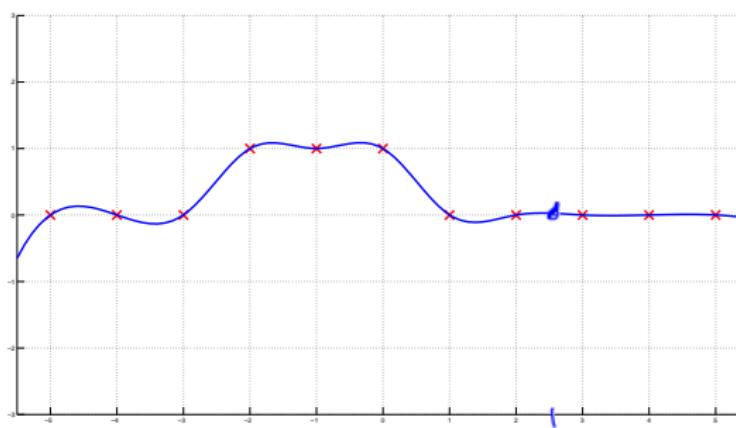
$$\tilde{f}(x_k) = y_k \quad (k = 0 : n).$$



Interpolation problem

Given $n + 1$ data points $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$ with x_k distinct ($k = 0 : n$), determine a function \tilde{f} that satisfies

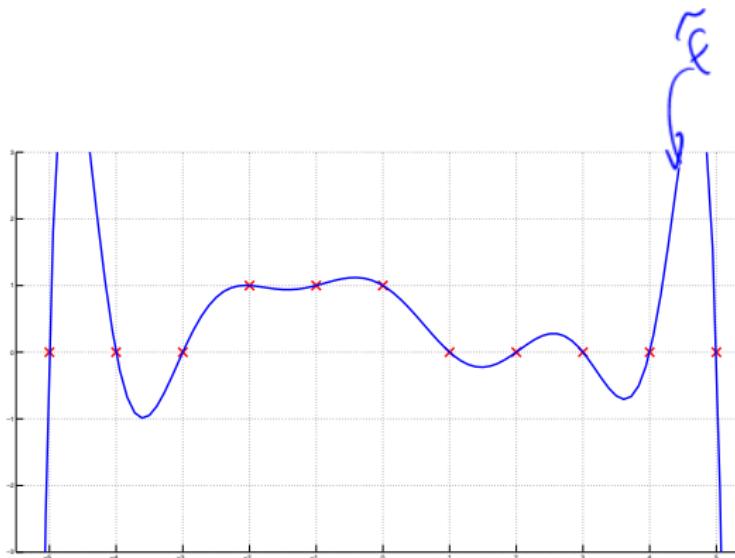
$$\tilde{f}(x_k) = y_k \quad (k = 0 : n).$$



Interpolation problem

Given $n + 1$ data points $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$ with x_k distinct ($k = 0 : n$), determine a function \tilde{f} that satisfies

$$\tilde{f}(x_k) = y_k \quad (k = 0 : n).$$



Remarks:

- ▶ \tilde{f} is called an **interpolating function** or **interpolant**.
- ▶ x_k are **interpolation points** or **nodes** or **abscissa**.
- ▶ \tilde{f} provides values for points in between the x_k
- ▶ Desirable to have \tilde{f} smooth, differentiable, easy to compute.

Remarks:

- ▶ \tilde{f} is called an **interpolating function** or **interpolant**.
- ▶ x_k are **interpolation points** or **nodes** or **abscissa**.
- ▶ \tilde{f} provides values for points in between the x_k
- ▶ Desirable to have \tilde{f} smooth, differentiable, easy to compute.
- ▶ Particular sources of interpolation problems:
 - ▶ Curve fitting (e.g. for graphics).
 - ▶ Data from experiments (see also least squares).
 - ▶ Approximate solution of differential equations

Remarks:

- ▶ \tilde{f} is called an **interpolating function** or **interpolant**.
- ▶ x_k are **interpolation points** or **nodes** or **abscissa**.
- ▶ \tilde{f} provides values for points in between the x_k
- ▶ Desirable to have \tilde{f} smooth, differentiable, easy to compute.
- ▶ Particular sources of interpolation problems:
 - ▶ Curve fitting (e.g. for graphics).
 - ▶ Data from experiments (see also least squares).
 - ▶ Approximate solution of differential equations
- ▶ Example: linear interpolation using SciPy's `interp1d`

Other classes of interpolating functions

→ Polynomial interpolant:

$$\begin{aligned}\tilde{f}(x) &= \sum_{k=0}^n a_k x^k \\ &= a_0 + a_1 x + \cdots + a_n x^n\end{aligned}$$

Trigonometric interpolant: $\tilde{f}(x) = \sum_{k=-M}^M c_k e^{ikx} \quad (M = \lfloor n/2 \rfloor)$

$$= c_{-M} e^{-iMx} + \cdots + c_0 + \cdots + c_n e^{iMx}$$

Rational interpolant: $\tilde{f}(x) = \frac{\sum_{j=0}^k a_j x^j}{\sum_{\ell=0}^{n-k-1} a_{k+\ell+1} x^\ell} \quad (k < n)$

$$= \frac{a_0 + a_1 x^1 + \cdots + a_k x^k}{a_{k+1} + a_{k+2} x^1 + \cdots + a_n x^{n-k-1}}$$

Interpolation

- ▶ As stated, problem does not specify \tilde{f} uniquely.
- ▶ Need to restrict class of functions.

Interpolation

- ▶ As stated, problem does not specify \tilde{f} uniquely.
- ▶ Need to restrict class of functions.
- ▶ Choose \tilde{f} from a vector space of dimension $n + 1$

$$\underline{\tilde{f}(x)} = \sum_{k=0}^n a_k \phi_k(x) = a_0 \phi_0(x) + a_1 \phi_1(x) + \cdots + a_n \phi_n(x)$$

- ▶ ϕ_k are **basis functions**, a_k are coefficients.

Interpolation

- ▶ As stated, problem does not specify \tilde{f} uniquely.
- ▶ Need to restrict class of functions.
- ▶ Choose \tilde{f} from a vector space of dimension $n + 1$

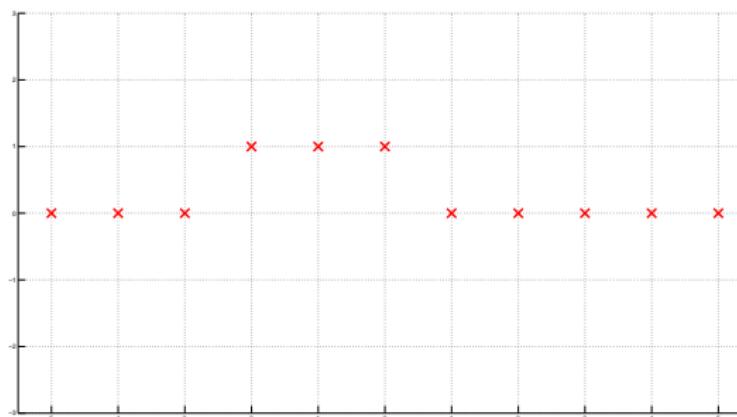
$$\tilde{f}(x) = \sum_{k=0}^n a_k \phi_k(x) = a_0 \phi_0(x) + a_1 \phi_1(x) + \cdots + a_n \phi_n(x)$$

- ▶ ϕ_k are **basis functions**, a_k are coefficients.
- ▶ For $\tilde{f} = \sum_{k=0}^n a_k \phi_k$, a_k depend linearly on data y_k ($k = 0 : n$).
- ▶ For rational interpolation, different linear equations result.

Recall: Interpolation problem

Given $n + 1$ data points $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$ with x_k distinct ($k = 0 : n$), determine a function \tilde{f} that satisfies

$$\tilde{f}(x_k) = y_k \quad (k = 0 : n).$$



Polynomial interpolation problem

Given $n + 1$ data points $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$ with x_k distinct ($k = 0 : n$), determine a polynomial function Π_n of degree at most n that satisfies

goes through all data

$$\Pi_n(x_k) = y_k \quad (k = 0 : n).$$

In the case where the value y_k represents the value of a continuous function f sampled at $x = x_k$ ($k = 0 : n$), the **interpolating polynomial** (or **interpolant**) is denoted $\Pi_n f$.

- ▶ n is (maximum) degree of interpolant.
- ▶ $n + 1$ is number of data points.

Polynomial interpolation

- ▶ Polynomials easy to evaluate, differentiate, etc.
- ▶ Π_n lies in vector space of polynomials of degree at most n .
- ▶ $n + 1$ coefficients to determine as $\deg(\Pi_n) \leq n$.

$$\Pi_n(x) = \sum_{k=0}^n a_k \phi_k(x) = a_0 \phi_0(x) + a_1 \phi_1(x) + \cdots + a_n \phi_n(x)$$

Polynomial interpolation

- ▶ Polynomials easy to evaluate, differentiate, etc.
- ▶ Π_n lies in vector space of polynomials of degree at most n .
- ▶ $n + 1$ coefficients to determine as $\deg(\Pi_n) \leq n$.

$$\Pi_n(x) = \sum_{k=0}^n a_k \phi_k(x) = a_0 \phi_0(x) + a_1 \phi_1(x) + \cdots + a_n \phi_n(x)$$

- ▶ **Practical problem:** Given $n + 1$ data points $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$ with x_k distinct ($k = 0 : n$), find the a_k such that

$$\Pi_n(x_k) = y_k \quad (k = 0 : n).$$

Polynomial interpolation

- ▶ Polynomials easy to evaluate, differentiate, etc.
- ▶ Π_n lies in vector space of polynomials of degree at most n .
- ▶ $n + 1$ coefficients to determine as $\deg(\Pi_n) \leq n$.

$$\Pi_n(x) = \sum_{k=0}^n a_k \phi_k(x) = a_0 \phi_0(x) + a_1 \phi_1(x) + \cdots + a_n \phi_n(x)$$

- ▶ **Practical problem:** Given $n + 1$ data points $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$ with x_k distinct ($k = 0 : n$), find the a_k such that

$$\Pi_n(x_k) = y_k \quad (k = 0 : n).$$

- ▶ Gives $n + 1$ equations in the $n + 1$ unknowns a_k
- ▶ Reduces to linear algebra: solution of linear system of equations.

Polynomial interpolation

Theorem (Existence/Uniqueness of polynomial interpolation:)

*If all interpolation nodes are distinct the interpolant exists.
If we select the polynomial interpolant of the lowest possible order, it is unique.*

Interpolation in power (monomial) basis:

- ▶ Choose $\phi_k(x) = x^k$, and thus write polynomial Π_n as

$$\begin{aligned}\Pi_n(x) &= \sum_{k=0}^n a_k x^k \\ &= a_0 x^0 + a_1 x^1 + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \\ &= a_0 + x^1 a_1 + x^2 a_2 + \cdots + x^{n-1} a_{n-1} + x^n a_n\end{aligned}$$

Interpolation in power (monomial) basis:

- ▶ Choose $\phi_k(x) = x^k$, and thus write polynomial Π_n as

$$\begin{aligned} P_n(x) &= \sum_{k=0}^n a_k x^k \\ &= a_0 x^0 + a_1 x^1 + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \\ &= a_0 + x^1 a_1 + x^2 a_2 + \cdots + x^{n-1} a_{n-1} + x^n a_n \end{aligned}$$

- Write out interpolation conditions $\Pi_n(x_k) = y_k$ ($k = 0:n$)

$$\begin{aligned} 1 \cdot a_0 + x_0^1 \cdot a_1 + \cdots + x_0^{n-1} \cdot a_{n-1} + x_0^n \cdot a_n &= y_0 \\ 1 \cdot a_0 + x_1^1 \cdot a_1 + \cdots + x_1^{n-1} \cdot a_{n-1} + x_1^n \cdot a_n &= y_1 \\ 1 \cdot a_0 + x_2^1 \cdot a_1 + \cdots + x_2^{n-1} \cdot a_{n-1} + x_2^n \cdot a_n &= y_2 \\ \vdots &\quad \vdots & \vdots & \vdots & = & \vdots \\ 1 \cdot a_0 + x_n^1 \cdot a_1 + \cdots + x_n^{n-1} \cdot a_{n-1} + x_n^n \cdot a_n &= y_n \end{aligned}$$

Vandermonde system

- In matrix form, $V\mathbf{a} = \mathbf{y}$ where

$$\underbrace{\begin{bmatrix} 1 & x_0^1 & \cdots & x_0^{n-1} & x_0^n \\ 1 & x_1^1 & \cdots & x_1^{n-1} & x_1^n \\ 1 & x_2^1 & \cdots & x_2^{n-1} & x_2^n \\ \vdots & \vdots & & \vdots & \vdots \\ 1 & x_n^1 & \cdots & x_n^{n-1} & x_n^n \end{bmatrix}}_V \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}}_a = \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}_y$$

- Matrix $V \in \mathbb{R}^{(n+1) \times (n+1)}$ is **Vandermonde matrix**
- $(n+1) \times (n+1)$ linear system of equations to solve for \mathbf{a}

Example of polynomial interpolation

- ▶ Consider data $(1.0, 2.0)$, $(1.1, 2.5)$, and $(1.2, 1.5)$
- ▶ In this case, $n = 2$ and data $\{(x_k, y_k)\}_{k=0}^n$ are

x_k	0	1	2
x_k	1.0	1.1	1.2
y_k	2.0	2.5	1.5

Example of polynomial interpolation

- ▶ Consider data $(1.0, 2.0)$, $(1.1, 2.5)$, and $(1.2, 1.5)$
- ▶ In this case, $n = 2$ and data $\{(x_k, y_k)\}_{k=0}^n$ are

k	0	1	2
x_k	1.0	1.1	1.2
y_k	2.0	2.5	1.5

- ▶ Seek polynomial Π_2 of degree at most 2 of the form

$$\Pi_2(x) = \sum_{k=0}^2 a_k x^k = a_0 + a_1 x + a_2 x^2$$

Example of polynomial interpolation

- ▶ Consider data $(1.0, 2.0)$, $(1.1, 2.5)$, and $(1.2, 1.5)$
- ▶ In this case, $n = 2$ and data $\{(x_k, y_k)\}_{k=0}^n$ are

k	0	1	2
x_k	1.0	1.1	1.2
y_k	2.0	2.5	1.5

- ▶ Seek polynomial Π_2 of degree at most 2 of the form

$$\Pi_2(x) = \sum_{k=0}^2 a_k x^k = a_0 + a_1 x + a_2 x^2$$

- ▶ Interpolation conditions are

$$\Pi_2(x_0) = 1 \cdot a_0 + 1.0 \cdot a_1 + 1.00 \cdot a_2 = 2.$$

$$\Pi_2(x_1) = 1 \cdot a_0 + 1.1 \cdot a_1 + 1.21 \cdot a_2 = 2.5.$$

$$\Pi_2(x_2) = 1 \cdot a_0 + 1.2 \cdot a_1 + 1.44 \cdot a_2 = 1.5$$

Example of polynomial interpolation (cont.)

- ▶ Write system as $V\mathbf{a} = \mathbf{y}$

$$V\mathbf{a} = \begin{bmatrix} 1 & 1.0 & 1.00 \\ 1 & 1.1 & 1.21 \\ 1 & 1.2 & 1.44 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 2.5 \\ 1.5 \end{bmatrix} = \mathbf{y}$$

Example of polynomial interpolation (cont.)

- ▶ Write system as $V\mathbf{a} = \mathbf{y}$

$$V\mathbf{a} = \begin{bmatrix} 1 & 1.0 & 1.00 \\ 1 & 1.1 & 1.21 \\ 1 & 1.2 & 1.44 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 2.5 \\ 1.5 \end{bmatrix} = \mathbf{y}$$

- ▶ Solving the 3×3 linear system yields

$$\mathbf{a} = [a_0, a_1, a_2]^T = [-85.5, 162.5, -75]^T$$

- ▶ Resulting polynomial interpolant is

$$\begin{aligned} I_2(x) &= \sum_{k=0}^2 a_k x^k \\ &= -85.5 + 162.5x - 75x^2 \end{aligned}$$

Theorem (Existence/Uniqueness of polynomial interpolation:)

*If all interpolation nodes are distinct the interpolant exists.
If we select the polynomial interpolant of the lowest possible order, it is unique.*

Proof of the theorem on existence and uniqueness:

Proof of the theorem on existence and uniqueness:

- ▶ Obviously, the solution exists if (and only if) the Vandermonder matrix is invertible, in which case the polynomial coefficients are $\mathbf{a} = V^{-1}\mathbf{y}$.

Proof of the theorem on existence and uniqueness:

- ▶ Obviously, the solution exists if (and only if) the Vandermonder matrix is invertible, in which case the polynomial coefficients are $\mathbf{a} = V^{-1}\mathbf{y}$.
- ▶ Now assume that this matrix *is* singular, that means that there are numbers α_k such that:

→ $\alpha_0 + \alpha_1 x_k + \alpha_2 x_k^2 + \dots + \alpha_n x_k^n = 0$ for $k = 0, \dots, n$

Proof of the theorem on existence and uniqueness:

- ▶ Obviously, the solution exists if (and only if) the Vandermonder matrix is invertible, in which case the polynomial coefficients are $\mathbf{a} = V^{-1}\mathbf{y}$.
- ▶ Now assume that this matrix *is* singular, that means that there are numbers α_k such that:

$$\alpha_0 + \alpha_1 x_k + \alpha_2 x_k^2 + \dots + \alpha_n x_k^n = 0 \text{ for } k = 0, \dots, n$$

- ▶ ... that would mean that the polynomial $\alpha_0 + \alpha_1 x + \alpha_2 x^2 + \dots + \alpha_n x^n$ has $(n+1)$ zeros.

Proof of the theorem on existence and uniqueness:

- ▶ Obviously, the solution exists if (and only if) the Vandermonder matrix is invertible, in which case the polynomial coefficients are $\mathbf{a} = V^{-1}\mathbf{y}$.
- ▶ Now assume that this matrix *is* singular, that means that there are numbers α_k such that:

$$\alpha_0 + \alpha_1 x_k + \alpha_2 x_k^2 + \dots + \alpha_n x_k^n = 0 \text{ for } k = 0, \dots, n$$

- ▶ ... that would mean that the polynomial $\alpha_0 + \alpha_1 x + \alpha_2 x^2 + \dots + \alpha_n x^n$ has $(n+1)$ zeros.
- ▶ However, this polynomial is of order n , so it can have no more than n zeros.

Proof of the theorem on existence and uniqueness:

- ▶ Obviously, the solution exists if (and only if) the Vandermonde matrix is invertible, in which case the polynomial coefficients are $\mathbf{a} = V^{-1}\mathbf{y}$.
- ▶ Now assume that this matrix *is* singular, that means that there are numbers α_k such that:

$$\alpha_0 + \alpha_1 x_k + \alpha_2 x_k^2 + \dots + \alpha_n x_k^n = 0 \text{ for } k = 0, \dots, n$$

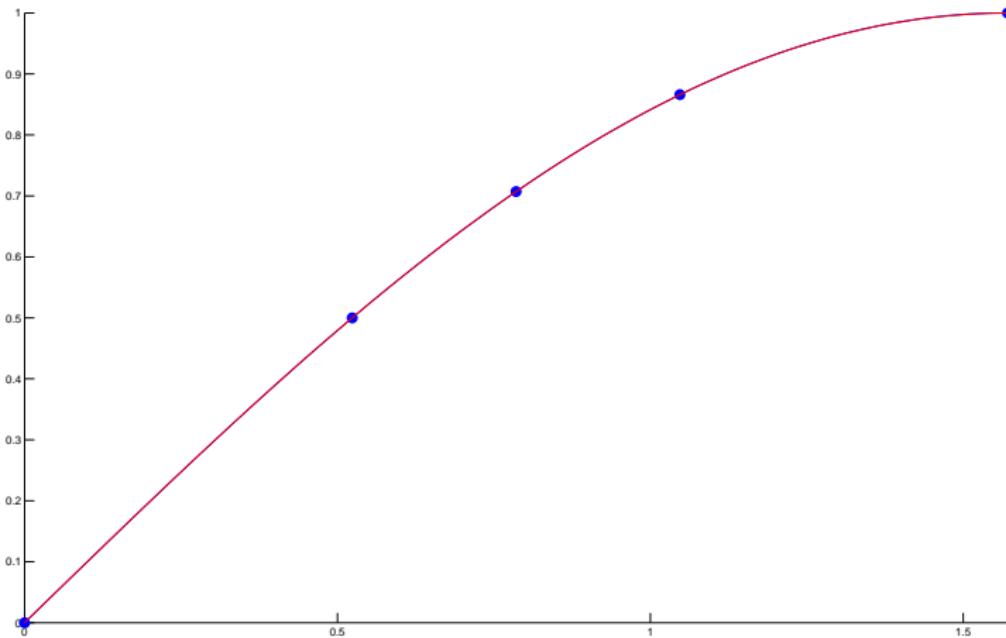
- ▶ ... that would mean that the polynomial $\alpha_0 + \alpha_1 x + \alpha_2 x^2 + \dots + \alpha_n x^n$ has $(n+1)$ zeros.
- ▶ However, this polynomial is of order n , so it can have no more than n zeros.
- ▶ This contradicts the assumption that the Vandermonde matrix is singular. Therefore, it is non-singular.

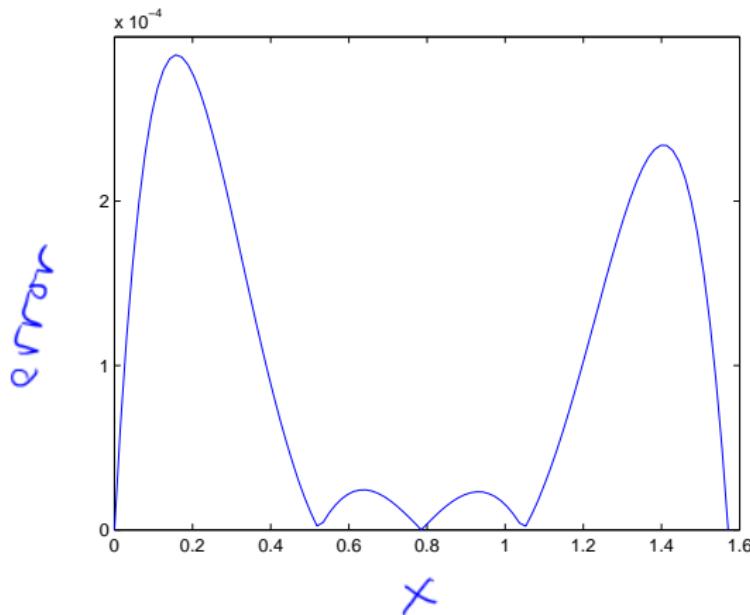
Another example:

x	0	$\pi/6$	$\pi/4$	$\pi/3$	$\pi/2$
$\sin(x)$	0	$1/2$	$1/\sqrt{2}$	$\sqrt{3}/2$	1

these exact values can be used to approximate $\sin(x)$ by a polynomial:

$$\begin{aligned}\sin(x) \approx & 0.9956261 x + 0.021372984 x^2 - 0.2043407 x^3 \\ & + 0.02879711 x^4 \quad (\text{for } 0 < x < \pi/2)\end{aligned}$$





The difference between $\sin(x)$ and $\Pi_4(x)$ is less than 3×10^{-4} on $[0, \pi/2]$.

The three questions:

1. When does it work?
2. How fast does it work?
3. How accurate is the result?

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. What conditions, how fast, how accurate?
2. Interpolation error
3. Computational complexity
4. Taylor polynomials

Key questions:

- ▶ Polynomial interpolation: what conditions, how fast, how accurate?
- ▶ What are Taylor polynomials?
- ▶ What are the differences between Taylor and interpolating polynomials?

The three questions:

1. When does it work?

2. How fast does it work?

3. How accurate is the result?

Theorem (Polynomial interpolation error)

Let $\{x_k\}_{k=0}^n$ be distinct interpolation nodes in $I \subset \mathbb{R}$ and let f be $n+1$ times continuously differentiable in I . Then, $\forall x \in I$, $\exists \xi \in I$ such that

$$E_n(x) := f(x) - \Pi_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{k=0}^n (x - x_k),$$

where Π_n is the unique polynomial interpolant of degree at most n that interpolates the data $\{(x_k, f(x_k))\}_{k=0}^n$.

Theorem (Polynomial interpolation error)

Let $\{x_k\}_{k=0}^n$ be distinct interpolation nodes in $I \subset \mathbb{R}$ and let f be $n + 1$ times continuously differentiable in I . Then, $\forall x \in I$, $\exists \xi \in I$ such that

$$E_n(x) := f(x) - \Pi_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{k=0}^n (x - x_k),$$

where Π_n is the unique polynomial interpolant of degree at most n that interpolates the data $\{(x_k, f(x_k))\}_{k=0}^n$.

- ▶ $|E_n(x)| = |f(x) - \Pi_n(x)|$ is the **error of polynomial interpolation**

Theorem (Polynomial interpolation error)

Let $\{x_k\}_{k=0}^n$ be distinct interpolation nodes in $I \subset \mathbb{R}$ and let f be $n+1$ times continuously differentiable in I . Then, $\forall x \in I$, $\exists \xi \in I$ such that

$$E_n(x) := f(x) - \Pi_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{k=0}^n (x - x_k),$$

where Π_n is the unique polynomial interpolant of degree at most n that interpolates the data $\{(x_k, f(x_k))\}_{k=0}^n$.

- ▶ $|E_n(x)| = |f(x) - \Pi_n(x)|$ is the **error of polynomial interpolation**
- ▶ What theorem says: size of error controlled by
 - ▶ size of $f^{(n+1)}(\xi)$
 - ▶ size of $\prod_{k=0}^n (x - x_k)$

Theorem (Polynomial interpolation error)

Let $\{x_k\}_{k=0}^n$ be distinct interpolation nodes in $I \subset \mathbb{R}$ and let f be $n+1$ times continuously differentiable in I . Then, $\forall x \in I$, $\exists \xi \in I$ such that

$$E_n(x) := f(x) - \Pi_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{k=0}^n (x - x_k),$$

where Π_n is the unique polynomial interpolant of degree at most n that interpolates the data $\{(x_k, f(x_k))\}_{k=0}^n$.

- ▶ $|E_n(x)| = |f(x) - \Pi_n(x)|$ is the **error of polynomial interpolation**
- ▶ What theorem says: size of error controlled by
 - ▶ size of $f^{(n+1)}(\xi)$
 - ▶ size of $\prod_{k=0}^n (x - x_k)$
- ▶ This error comes on top of the error of linear solving.

Because we can't know ξ unless we know the error, for practical application we use the theorem to find an upper bound on the error. In particular, the theorem implies:

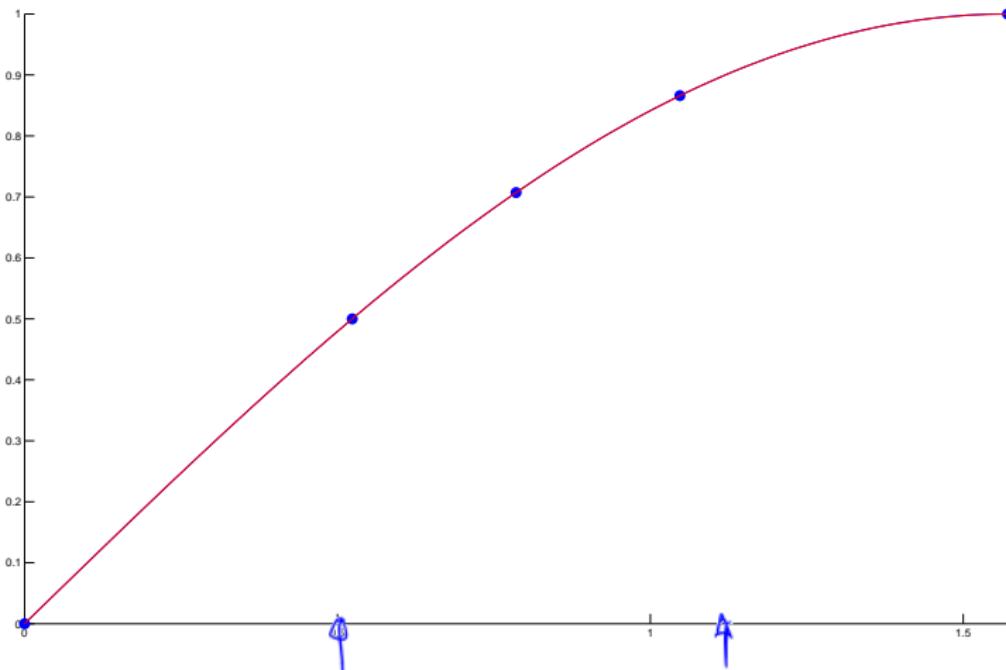
$$|E_n(x)| := |f(x) - \Pi_n(x)| \leq \max_{x \in I} \left[\frac{|f^{(n+1)}(\xi)|}{(n+1)!} \prod_{k=0}^n |x - x_k|, \right]$$

Back to the last example:

x	0	$\pi/6$	$\pi/4$	$\pi/3$	$\pi/2$
$\sin(x)$	0	$1/2$	$1/\sqrt{2}$	$\sqrt{3}/2$	1

these exact values can be used to approximate $\sin(x)$ by a polynomial:

$$\begin{aligned}\sin(x) \approx & 0.9956261 x + 0.021372984 x^2 - 0.2043407 x^3 \\ & + 0.02879711 x^4 \quad (\text{for } 0 < x < \pi/2)\end{aligned}$$



$$I \in [0, \frac{\pi}{2}], \quad f(x) = \sin x, \quad f'''(x) = -\cos x$$

$$f'(x) = \cos x$$

$$f''(x) = -\sin x$$

$$f^{(4)}(x) = \sin x$$

$$f^{(5)}(x) = \cos x$$

Maximum Error

$n=4$

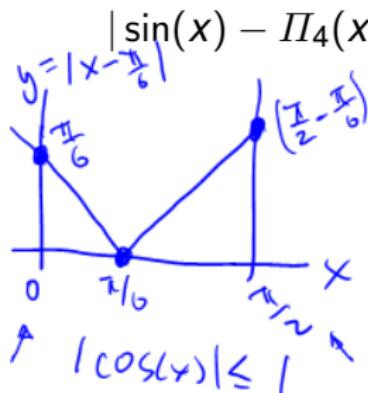
$$|f(x) - P_n(x)| \leq \max_{x \in I} \frac{|f^{(n+1)}(x)|}{(n+1)!} \prod_{k=0}^n |x - x_k|, \quad \hookrightarrow |x-x_0| |x-x_1| |x-x_2|$$

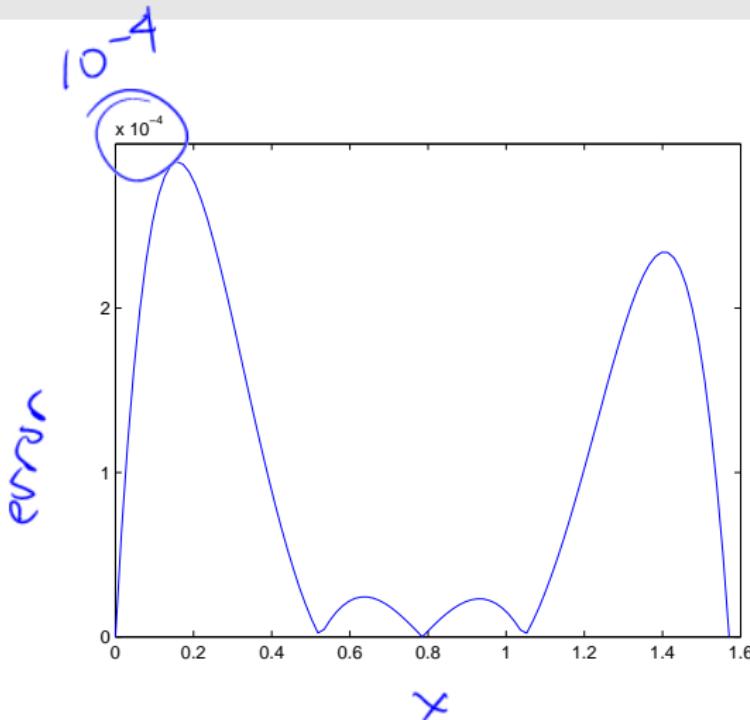
$$|\sin(x) - P_4(x)| \leq \max_{x \in I} \frac{|f^{(5)}(x)|}{5!} |x| \left|x - \frac{\pi}{6}\right| \left|x - \frac{\pi}{4}\right| \left|x - \frac{\pi}{3}\right| \left|x - \frac{\pi}{2}\right|$$

$$\leq \max_{x \in I} \frac{|\cos(x)|}{5!} |x| \left|x - \frac{\pi}{6}\right| \left|x - \frac{\pi}{4}\right| \left|x - \frac{\pi}{3}\right| \left|x - \frac{\pi}{2}\right|$$

$$\leq \frac{1}{5!} \left(\frac{\pi}{2}\right) \left(\frac{\pi}{2} - \frac{\pi}{6}\right) \left(\frac{\pi}{2} - \frac{\pi}{4}\right) \left(\frac{\pi}{2} - \frac{\pi}{3}\right) \left(\frac{\pi}{3}\right) \left(\frac{\pi}{2}\right)$$

$$= 0.0177 \quad (\text{our max error})$$





The difference between $\sin(x)$ and $\Pi_4(x)$ is less than 3×10^{-4} on $[0, \pi/2]$.

0.0177

?

Three conclusions we can draw from the error formula:

1. Functions with *small higher derivatives* are well-approximated by interpolating polynomials.
Such functions are *smooth*. The smoothest functions are the polynomials.
2. We can choose the location of the interpolation nodes to minimize the error (see CSII).

In fact, it turns out that *equidistant* nodes are bad. Good sets of interpolation nodes have more nodes near the boundaries.

3. **Extrapolation** is far more dangerous than *interpolation*.
The upper bound for the error of extrapolation grows without bound – as x^{n+1} .

Recall: To compute the polynomial interpolant, we can solve
 $V\mathbf{a} = \mathbf{y}$:

$$\underbrace{\begin{bmatrix} 1 & x_0^1 & \cdots & x_0^{n-1} & x_0^n \\ 1 & x_1^1 & \cdots & x_1^{n-1} & x_1^n \\ 1 & x_2^1 & \cdots & x_2^{n-1} & x_2^n \\ \vdots & \vdots & & \vdots & \vdots \\ 1 & x_n^1 & \cdots & x_n^{n-1} & x_n^n \end{bmatrix}}_V \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}}_a = \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}_y$$

$\rightarrow O(n^2)$

To compute FLOPS, we need to consider

- ▶ cost of building the matrix (see annotated Lecture 9 Slides)
- ▶ cost of solving the system $\rightarrow O(n^3)$

Is there a more efficient approach?

Is there another approach that doesn't lead to badly conditioned systems?

Recall: The interpolant is written as:

$$\Pi_n(x) = \sum_{k=0}^n a_k \phi_k(x) = a_0 \underbrace{\phi_0(x)} + a_1 \underbrace{\phi_1(x)} + \cdots + a_n \underbrace{\phi_n(x)}$$

Recall: The interpolant is written as:

→ $\Pi_n(x) = \sum_{k=0}^n a_k \phi_k(x) = a_0 \phi_0(x) + a_1 \phi_1(x) + \cdots + a_n \phi_n(x)$

Instead of using monomial basis $\phi_k(x) = x^k$, use:

Newton polynomial basis

$$\phi_0(x) = 1, \quad \phi_1(x) = x - x_0, \quad \phi_2(x) = (x - x_0)(x - x_1), \dots$$

$$\phi_n(x) = (x - x_0)(x - x_1) \dots (x - x_{n-1})$$

Recall: The interpolant is written as:

$$\Pi_n(x) = \sum_{k=0}^n a_k \phi_k(x) = a_0 \phi_0(x) + a_1 \phi_1(x) + \dots + a_n \phi_n(x)$$

Instead of using monomial basis $\phi_k(x) = x^k$, use:

Newton polynomial basis

$$\phi_0(x) = 1, \quad \phi_1(x) = x - x_0, \quad \phi_2(x) = (x - x_0)(x - x_1), \dots$$

$$\phi_n(x) = (x - x_0)(x - x_1)\dots(x - x_{n-1})$$

With this basis, the resulting system of linear equations for the coefficients $\{a_k\}_{k=0}^n$ is $M\mathbf{a} = \mathbf{y}$, where

- ▶ the matrix M is now triangular
- ▶ the matrix M depends on the interpolation points x_k

$$\Pi_n(x) = \sum_{k=0}^n a_k \phi_k(x) = a_0 \phi_0(x) + a_1 \phi_1(x) + \cdots + a_n \phi_n(x)$$

Write out interpolation conditions $\boxed{\Pi_n(x_k) = y_k}$ for new basis

$$\Pi_n(x) = \sum_{k=0}^n a_k \phi_k(x) = a_0 \phi_0(x) + a_1 \phi_1(x) + \cdots + a_n \phi_n(x)$$

Write out interpolation conditions $\Pi_n(x_k) = y_k$ for new basis

- ▶ $\Pi_n(x_0) = y_0 \rightarrow a_0 = y_0$

$$\Pi_n(x) = \sum_{k=0}^n a_k \phi_k(x) = a_0 \phi_0(x) + a_1 \phi_1(x) + \cdots + a_n \phi_n(x)$$

Write out interpolation conditions $\Pi_n(x_k) = y_k$ for new basis

- ▶ $\Pi_n(x_0) = y_0 \rightarrow a_0 = y_0$ x_0, x_1
- ▶ $\Pi_n(\underline{x_1}) = y_1 \rightarrow a_0 + a_1(x_1 - x_0) = y_1$

$$\Pi_n(x) = \sum_{k=0}^n a_k \phi_k(x) = a_0 \phi_0(x) + a_1 \phi_1(x) + \cdots + a_n \phi_n(x)$$

Write out interpolation conditions $\Pi_n(x_k) = y_k$ for new basis

- ▶ $\Pi_n(x_0) = y_0 \rightarrow a_0 = y_0$
- ▶ $\Pi_n(x_1) = y_1 \rightarrow a_0 + a_1(x_1 - x_0) = y_1$
- ▶ $\Pi_n(x_2) = y_2 \rightarrow a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1) = y_2$

$$\Pi_n(x) = \sum_{k=0}^n a_k \phi_k(x) = a_0 \phi_0(x) + a_1 \phi_1(x) + \cdots + a_n \phi_n(x)$$

Write out interpolation conditions $\Pi_n(x_k) = y_k$ for new basis

- ▶ $\Pi_n(x_0) = y_0 \rightarrow a_0 = y_0$
- ▶ $\Pi_n(x_1) = y_1 \rightarrow a_0 + a_1(x_1 - x_0) = y_1$
- ▶ $\Pi_n(x_2) = y_2 \rightarrow a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1) = y_2$
- ▶ \vdots

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & \dots \\ 1 & (x_1 - x_0) & 0 & 0 & \dots \\ 1 & (x_2 - x_0) & (x_2 - x_0)(x_2 - x_1) & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}}_M \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}}_a = \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}_y$$

Computational Complexity

- ▶ Using **monomial basis** (Vandermonde system)
 - ▶ cost of building the matrix is $O(N^2)$ (see annotated Lecture 9 Slides)
 - ▶ cost of solving the system: $O(N^3)$ (see Lecture 9)

Computational Complexity

- ▶ Using **monomial basis** (Vandermonde system)
 - ▶ cost of building the matrix is $O(N^2)$ (see annotated Lecture 9 Slides)
 - ▶ cost of solving the system: $O(N^3)$ (see Lecture 9)
- ▶ Using **Newton polynomial basis**
 - ▶ cost of building the matrix: takes fewer, but still $O(N^2)$
 - ▶ cost of solving triangular system: $O(N^2)$ (just requires forward substitution)



Computational Complexity

- ▶ Using **monomial basis** (Vandermonde system)
 - ▶ cost of building the matrix is $O(N^2)$ (see annotated Lecture 9 Slides)
 - ▶ cost of solving the system: $O(N^3)$ (see Lecture 9)
- ▶ Using **Newton polynomial basis**
 - ▶ cost of building the matrix: takes fewer, but still $O(N^2)$
 - ▶ cost of solving triangular system: $O(N^2)$ (just requires forward substitution)

Condition number

Computational Complexity

- ▶ Using **monomial basis** (Vandermonde system)
 - ▶ cost of building the matrix is $O(N^2)$ (see annotated Lecture 9 Slides)
 - ▶ cost of solving the system: $O(N^3)$ (see Lecture 9)
- ▶ Using **Newton polynomial basis**
 - ▶ cost of building the matrix: takes fewer, but still $O(N^2)$
 - ▶ cost of solving triangular system: $O(N^2)$ (just requires forward substitution)

Condition number

- ▶ The condition number of triangular matrix resulting from using Newton polynomial basis is much smaller than the condition number of the Vandermonde matrix
- ▶ Newton polynomial basis leads to more accurate results

Theorem (Taylor's Theorem with Lagrange Remainder)

If $f^{(n)}$ is continuous in the interval $[a, b]$ and if $f^{(n+1)}$ exists on the open interval (a, b) , then for any points c and x in $[a, b]$ there exists a ξ between c and x such that

$$f(x) = T_n(x) + R_n(x), \quad \text{with}$$

$T_n(x)$ = *nth Taylor polynomial*

$$= \sum_{k=0}^n \frac{f^{(k)}(c)}{k!} (x - c)^k \quad \text{and}$$

$R_n(x)$ = *remainder or truncation error*

$$= \frac{f^{(n+1)}(\xi)}{(n+1)!} (x - c)^{n+1}$$

about 4 term

Example of a Taylor polynomial: e.g., $f(x) = e^x$, $c = 0$, $n = 3$

Example of a Taylor polynomial: e.g., $f(x) = e^x$, $c = 0$, $n = 3$

- ▶ $f(0) = f'(0) = f''(0) = f^{(3)}(0) = e^0 = 1$, $f^{(4)}(\xi) = e^\xi$

Example of a Taylor polynomial: e.g., $f(x) = e^x$, $c = 0$, $n = 3$

- ▶ $f(0) = f'(0) = f''(0) = f^{(3)}(0) = e^0 = 1$, $f^{(4)}(\xi) = e^\xi$
- ▶ $T_3(x) = \sum_{k=0}^3 \frac{f^{(k)}(0)}{k!} (x - 0)^k = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}$

Example of a Taylor polynomial: e.g., $f(x) = e^x$, $c = 0$, $n = 3$

- ▶ $f(0) = f'(0) = f''(0) = f^{(3)}(0) = e^0 = 1$, $f^{(4)}(\xi) = e^\xi$
- ▶ $T_3(x) = \sum_{k=0}^3 \frac{f^{(k)}(0)}{k!} (x - 0)^k = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}$
- ▶ $R_3(x) = \frac{f^{(4)}(\xi)}{4!} x^4 = \frac{e^\xi}{24} x^4$

Example of a Taylor polynomial: e.g., $f(x) = e^x$, $c = 0$, $n = 3$

► $f(0) = f'(0) = f''(0) = f^{(3)}(0) = e^0 = 1$, $f^{(4)}(\xi) = e^\xi$

► $T_3(x) = \sum_{k=0}^3 \frac{f^{(k)}(0)}{k!} (x - 0)^k = 1 + x + \frac{x^2}{2} + \frac{x^3}{6}$

► $R_3(x) = \frac{f^{(4)}(\xi)}{4!} x^4 = \frac{e^\xi}{24} x^4$

For this function, we can consider the *series*

$$\exp(x) = \sum_{k=0}^{\infty} \underbrace{\frac{1}{k!}}_{\text{ }} x^k$$

This series converges for any $x \in \mathbb{R}$.

We say it has an *infinite radius of convergence*.

Second example: Taylor polynomials for $\sin x$:

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} \quad (x \in \mathbb{R})$$

again with an infinite radius of convergence.

Second example: Taylor polynomials for $\sin x$:

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} \quad (x \in \mathbb{R})$$

again with an infinite radius of convergence.

$$S_1(x) = x$$

Second example: Taylor polynomials for $\sin x$:

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} \quad (x \in \mathbb{R})$$

again with an infinite radius of convergence.

$$S_1(x) = x$$

$$S_3(x) = x - \frac{x^3}{3!}$$

Nested form of polynomials simplifies evaluation.

Second example: Taylor polynomials for $\sin x$:

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} \quad (x \in \mathbb{R})$$

again with an infinite radius of convergence.

$$S_1(x) = x$$

$$S_3(x) = x - \frac{x^3}{3!} = x \left(1 - \frac{x^2}{2 \cdot 3}\right)$$

Nested form of polynomials simplifies evaluation.

Second example: Taylor polynomials for $\sin x$:

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} \quad (x \in \mathbb{R})$$

again with an infinite radius of convergence.

$$S_1(x) = x$$

$$S_3(x) = x - \frac{x^3}{3!} = x \left(1 - \frac{x^2}{2 \cdot 3}\right)$$

$$S_5(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!}$$

Nested form of polynomials simplifies evaluation.

Second example: Taylor polynomials for $\sin x$:

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} \quad (x \in \mathbb{R})$$

again with an infinite radius of convergence.

$$S_1(x) = x$$

$$S_3(x) = x - \frac{x^3}{3!} = x \left(1 - \frac{x^2}{2 \cdot 3} \right)$$

$$S_5(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} = x \left(1 + \frac{x^2}{2 \cdot 3} \left(-1 + \frac{x^2}{4 \cdot 5} \right) \right)$$

Nested form of polynomials simplifies evaluation.

Second example: Taylor polynomials for $\sin x$:

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} \quad (x \in \mathbb{R})$$

again with an infinite radius of convergence.

$$S_1(x) = x$$

$$S_3(x) = x - \frac{x^3}{3!} = x \left(1 - \frac{x^2}{2 \cdot 3} \right)$$

$$S_5(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} = x \left(1 + \frac{x^2}{2 \cdot 3} \left(-1 + \frac{x^2}{4 \cdot 5} \right) \right)$$

$$S_7(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

Nested form of polynomials simplifies evaluation.

Second example: Taylor polynomials for $\sin x$:

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} \quad (x \in \mathbb{R})$$

again with an infinite radius of convergence.

$$S_1(x) = x$$

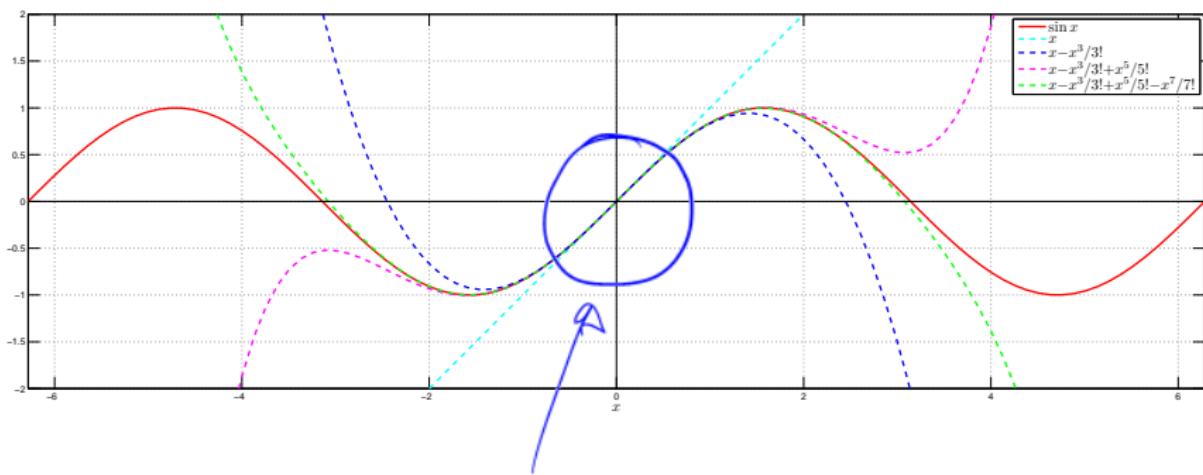
$$S_3(x) = x - \frac{x^3}{3!} = x \left(1 - \frac{x^2}{2 \cdot 3} \right)$$

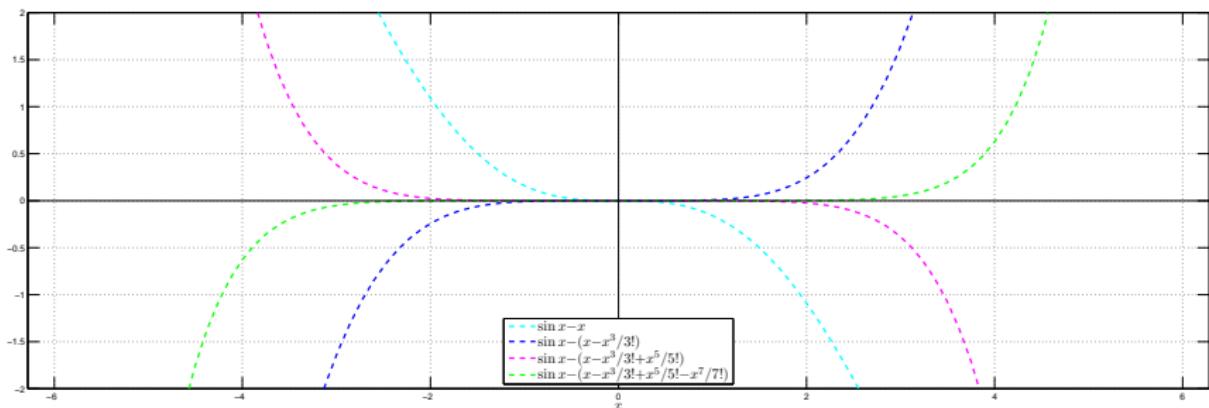
$$S_5(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} = x \left(1 + \frac{x^2}{2 \cdot 3} \left(-1 + \frac{x^2}{4 \cdot 5} \right) \right)$$

$$S_7(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} = x \left(1 + \frac{x^2}{2 \cdot 3} \left(-1 + \frac{x^2}{4 \cdot 5} \left(1 - \frac{x^2}{6 \cdot 7} \right) \right) \right)$$

Nested form of polynomials simplifies evaluation.

$\sin x$ and Taylor polynomials



Taylor polynomial errors for $\sin x$ 

Third example: $f(x) = \ln(1 + x)$.

$$f^{(1)}(x) = \frac{1}{1+x}$$

$$f^{(3)}(x) = \frac{2}{(1+x)^3}$$

$$f^{(2)}(x) = \frac{-1}{(1+x)^2}$$

$$f^{(k)}(x) = \frac{(-1)^{k+1}(k-1)!}{(1+x)^k}$$

so $f^{(1)}(0) = 1$, $f^{(2)}(0) = -1$, $f^{(3)}(0) = 2$, ...,
 $f^{(k)}(0) = (-1)^{k+1}(k-1)!$ and

$$\ln(1+x) = \sum_{k=1}^n \frac{(-1)^{k+1}}{k} x^k + R_n(x), \quad R_n(x) = \frac{(-1)^n x^{n+1}}{(n+1)(1+\xi)^{n+1}}$$

This series converges for $-1 < x < 1$, we say the radius of convergence is 1.

Summary for Taylor polynomials:

- ▶ The Taylor polynomial exists if f is sufficiently smooth.
- ▶ We have an expression for the remainder.
- ▶ The computation of Taylor polynomials requires the computation of derivatives. **Not many functions have such simple derivatives as in our examples!**
- ▶ Properties of f at $x = c$ completely determine $T_n(x)$, so all information comes from $x = c$. **The Taylor polynomial is useful as a local approximation only.**
- ▶ For $|x - c|$ large, many terms needed for convergence.

In contrast, polynomial interpolation is useful on a whole domain.

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. Piecewise polynomial approximation

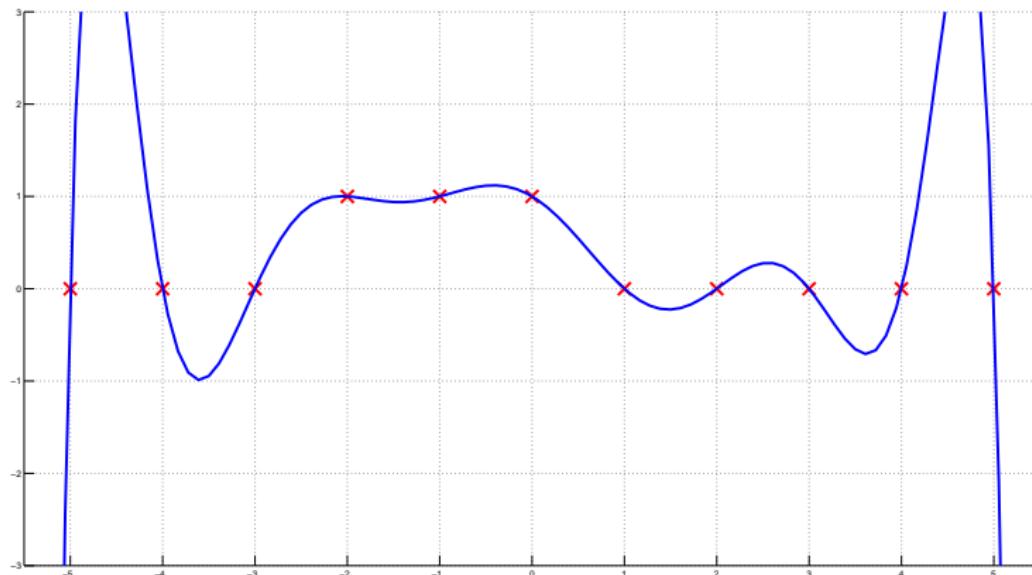
2. Piecewise linear interpolation

3. Piecewise cubic interpolation

We have seen two kinds of polynomial approximation of a function $f(x)$:

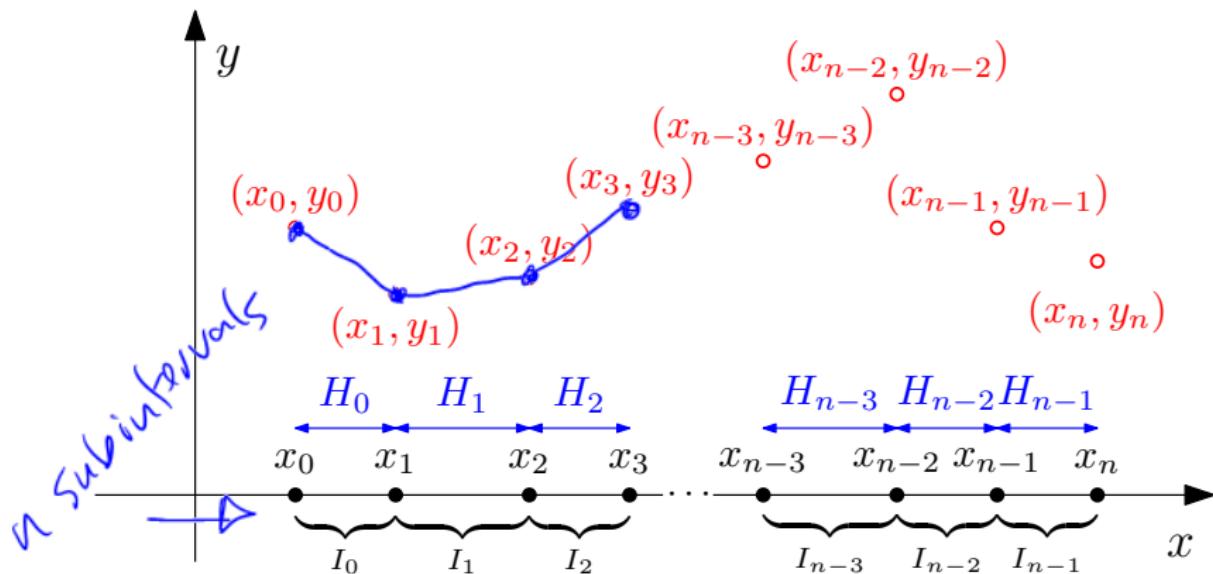
- ▶ interpolation using a single polynomial:
 - ▶ n^{th} order polynomial through $(n + 1)$ points takes solving an $(n + 1) \times (n + 1)$ linear system;
 - ▶ the interpolant exists if all interpolation nodes are distinct;
 - ▶ we have an upper bound for the interpolation error in terms of the nodes, the $(n + 1)^{\text{st}}$ derivative of f and the order, n ;
 - ▶ the error is bounded on the whole domain of interpolation.
- ▶ Taylor polynomials:
 - ▶ n^{th} order approximation takes computation of n derivatives;
 - ▶ we have an upper bound for the approximation error in terms of the $(n + 1)^{\text{st}}$ derivative of the function, the distance to the base point, $(x - c)$, and the the order, n ;
 - ▶ even if the approximation is good locally, for larger $|x - c|$ the error can be large or even infinite.

Polynomial interpolation: errors can grow with the $(n + 1)^{\text{st}}$ derivative of f



Piecewise polynomial interpolation provides a compromise between **global** and **local** approximation.

The idea is simple: compute a low-order interpolant on each domain $[x_k, x_{k+1}]$, then “glue” the pieces together.



- ▶ Assume data to interpolate is $\{(x_k, y_k)\}_{k=0}^n$ with

$$a = x_0 < x_1 < \cdots < x_n = b$$

- ▶ Interval $[a, b]$ subdivided into n subintervals

$$I_0 := [x_0, x_1], I_1 := [x_1, x_2], \dots, I_{n-1} := [x_{n-1}, x_n]$$

- ▶ Distinct values x_k are called **breakpoints/knots**
- ▶ Each subinterval has width
$$H_k := x_{k+1} - x_k \quad (k = 0 : n - 1)$$
- ▶ $H := \max_{0 \leq k \leq n-1} H_k$ characterises maximum width
- ▶ We will compute a **low-order** polynomial on each I_k . We can keep the interpolation error small:
 - ▶ low order of derivative $f^{(n+1)}(\xi)$
 - ▶ smaller maximum of $\prod_{k=0}^n |x - x_k|$

- ▶ Consider interpolating with two points (x_0, y_0) and (x_1, y_1) .
- ▶ Polynomial interpolant of degree at most one is

→
$$\Pi_1(x) = y_0 + \left(\frac{y_1 - y_0}{x_1 - x_0} \right) (x - x_0)$$

- ▶ Coefficients are y_0 and $\delta_0 := \frac{y_1 - y_0}{x_1 - x_0}$ (first divided difference).
- ▶ Coefficients determined without solving linear system.
- ▶ Generalises to linear interpolant on each I_k ($k = 0 : n - 1$).

- Generalising gives piecewise-defined interpolant Π_1^H :

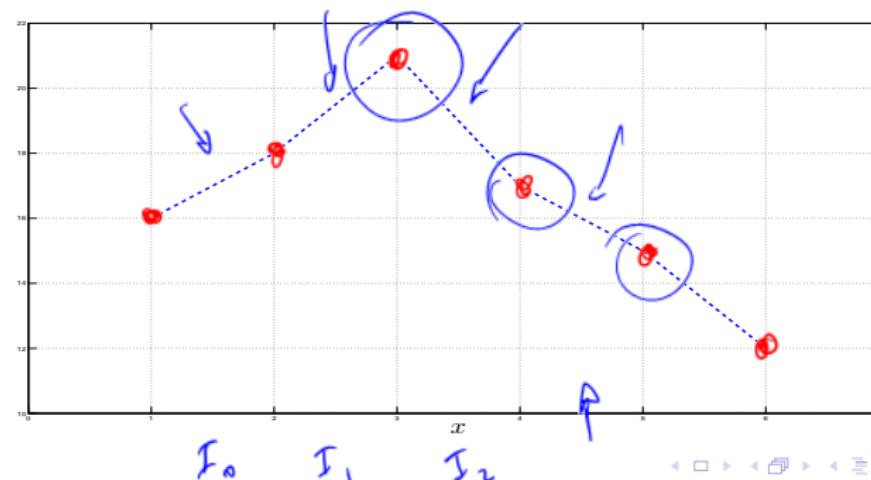
$$\Pi_1^H(x) = \begin{cases} y_0 + \delta_0(x - x_0), & x_0 \leq x < x_1 \\ y_1 + \delta_1(x - x_1), & x_1 \leq x < x_2 \\ \vdots & \vdots \\ y_{n-2} + \delta_{n-2}(x - x_{n-2}), & x_{n-2} \leq x < x_{n-1} \\ y_{n-1} + \delta_{n-1}(x - x_{n-1}), & x_{n-1} \leq x \leq x_n \end{cases}$$

with first divided differences δ_k defined by

$$\delta_k := \frac{y_{k+1} - y_k}{x_{k+1} - x_k} \quad (k = 0 : n-1)$$

- $\Pi_1^H(x)$ implicitly truncated outside interval $[x_0, x_n]$

```
x=[range(1,7)]  
y=[16,18,21,17,15,12]  
xx=scipy.linspace(1,6)  
f = scipy.interpolate.interp1d(x,y,'linear')  
yy = f(xx)  
plt.plot(xx,yy,'--')  
plt.plot(x,y,'o')  
plt.xlim(0,7)  
plt.show()
```



The piecewise linear interpolant is **not differentiable**. Often we need the interpolant to be smoother.

Solution: take higher order polynomials on each interval I_k and *match their derivatives at the knots*.

If we choose third-order polynomials, they are called *cubic splines*.

Let the spline on I_k be called $S_k(x)$. Then

- 1. $S_{k-1}(x_k) = S_k(x_k) = y_k$ for $k = 1, \dots, n-1$, $S_0(x_0) = y_0$,
 $S_{n-1}(x_n) = y_n$;
- 2. at every knot (except x_0 and x_n) $S'_{k-1}(x_k) = S'_k(x_k)$;
- 3. at every knot (except x_0 and x_n) $S''_{k-1}(x_k) = S''_k(x_k)$;
- 4. either $S''_0(x_0) = S''_{n-1}(x_n) = 0$ (*free splines*) or
 $S'_0(x_0) = f'(x_0)$, $S'_{n-1}(x_n) = f'(x_n)$ (*clamped splines*). }

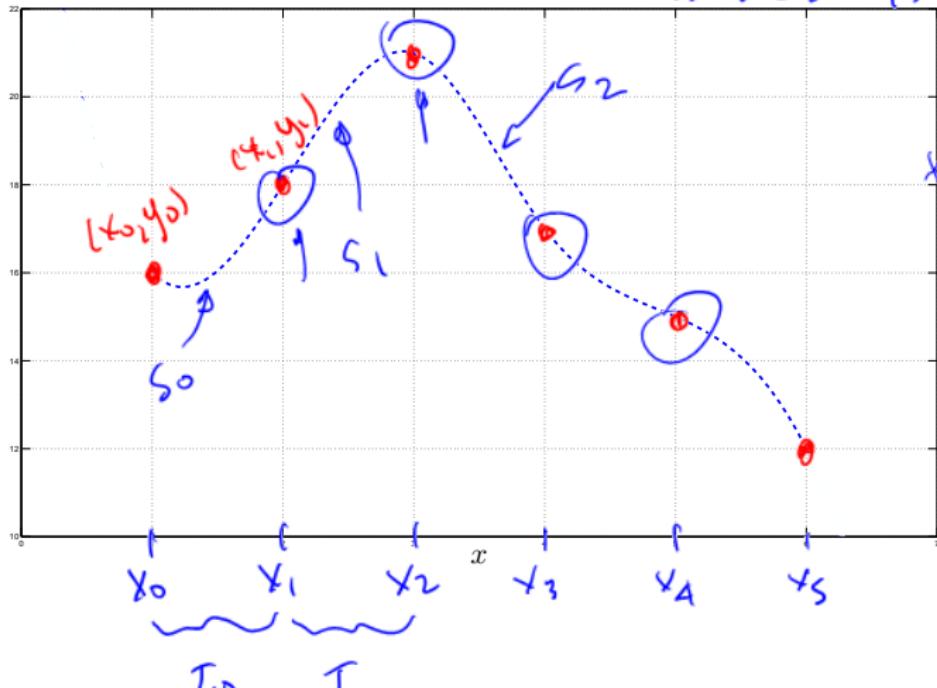
$$S_k(x) = a_k + b_k x + c_k x^2 + d_k x^3$$

5 x

$$S_k(x) = a_k + b_k x + c_k x^2 + d_k x^3 \quad (\text{4 coeffs for each})$$

 $\Rightarrow \text{need } 4 \times 5 = 20$

conditions
to specify
the 8 coeffs.



1. S_k goes through data
 $\Rightarrow 2 \times \# \text{ subintervals} \Rightarrow 2 \times 5 = 10$

2. $S'_{k-1}(x) = S'_k(x)$ at return knots
(e.g. want $S'_0(x_1) = S'_1(x_1)$)
 $\Rightarrow \# \text{ of conditions} = \# \text{ of return knots} = 4$

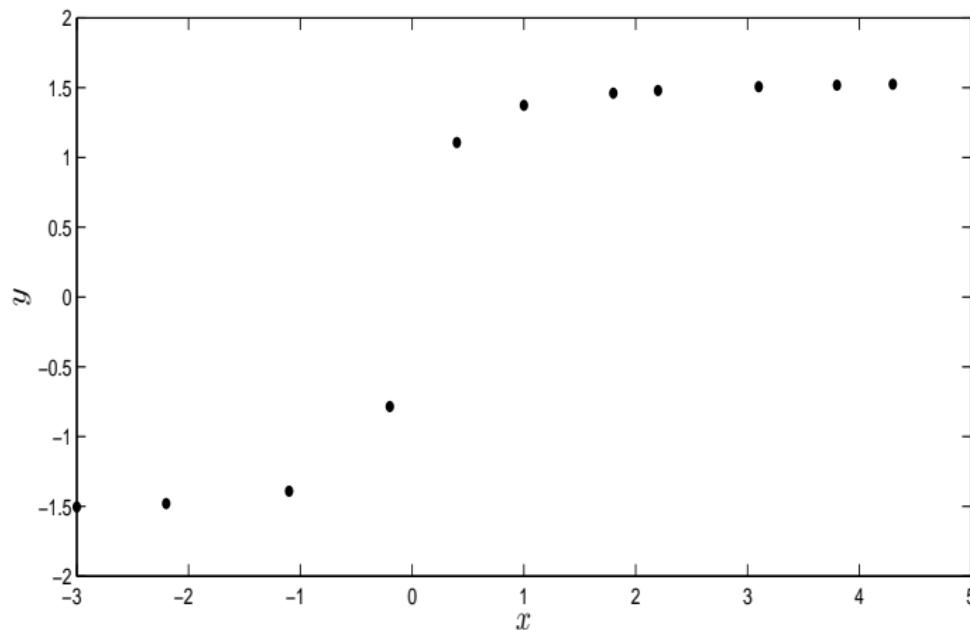
3. $S''_{k-1}(x_k) = S'_k(x_k)$
 $\Rightarrow \# \text{ of conditions} = 4$

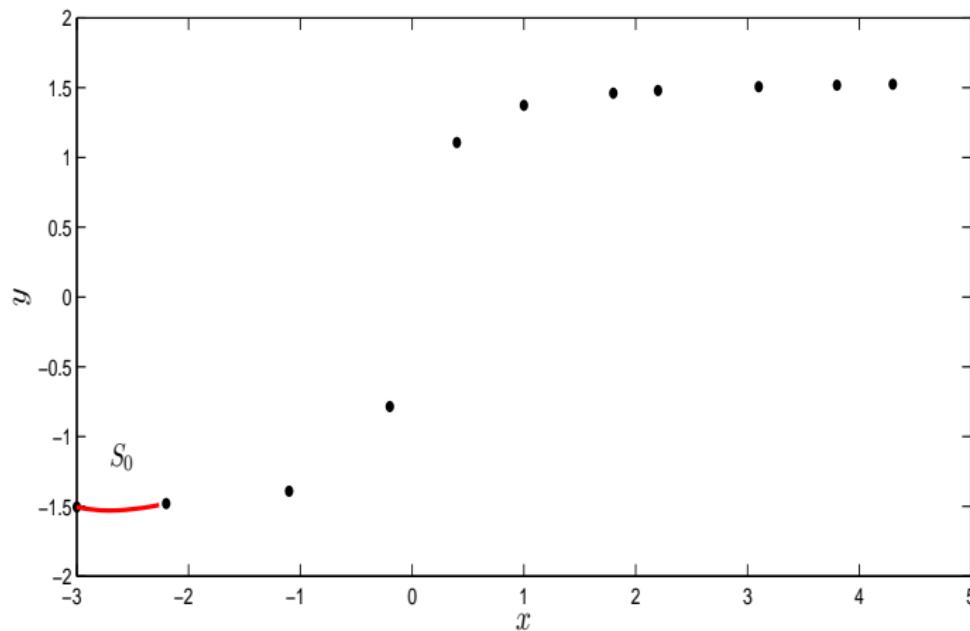
just 18 \Rightarrow need 2 more
 \Rightarrow use condition 4.

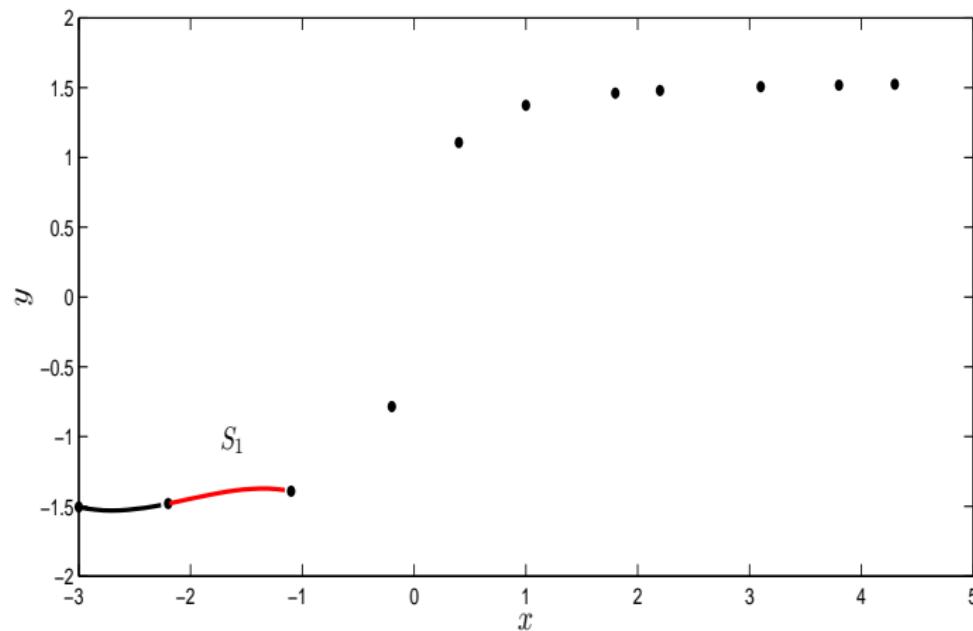
- ▶ The interpolant is now defined as

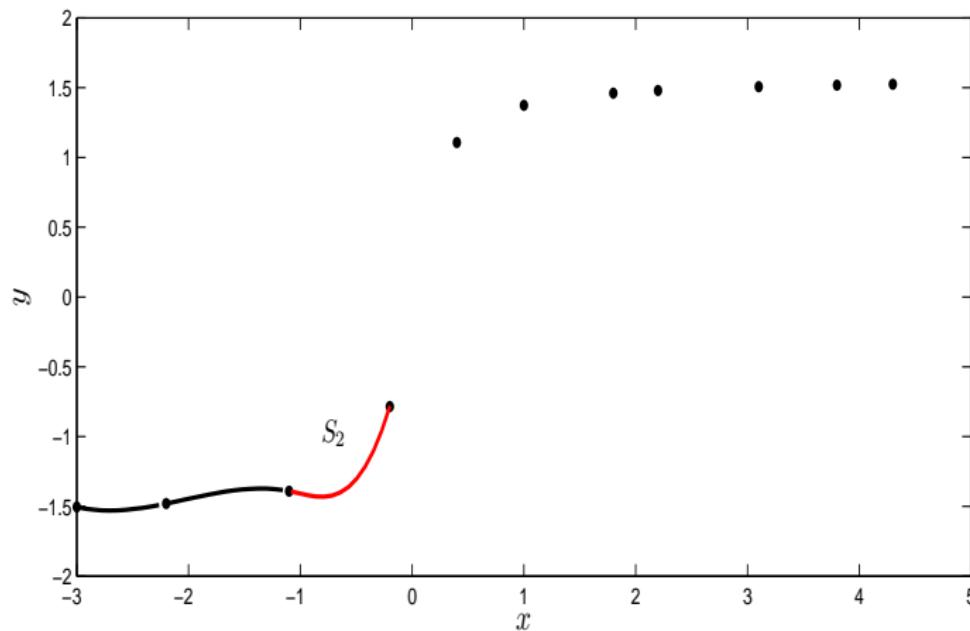
$$S(x) = \begin{cases} S_0(x), & x_0 \leq x < x_1 \\ S_1(x), & x_1 \leq x < x_2 \\ \vdots & \vdots \\ S_{n-2}(x), & x_{n-2} \leq x < x_{n-1} \\ S_{n-1}(x), & x_{n-1} \leq x \leq x_n \end{cases}$$

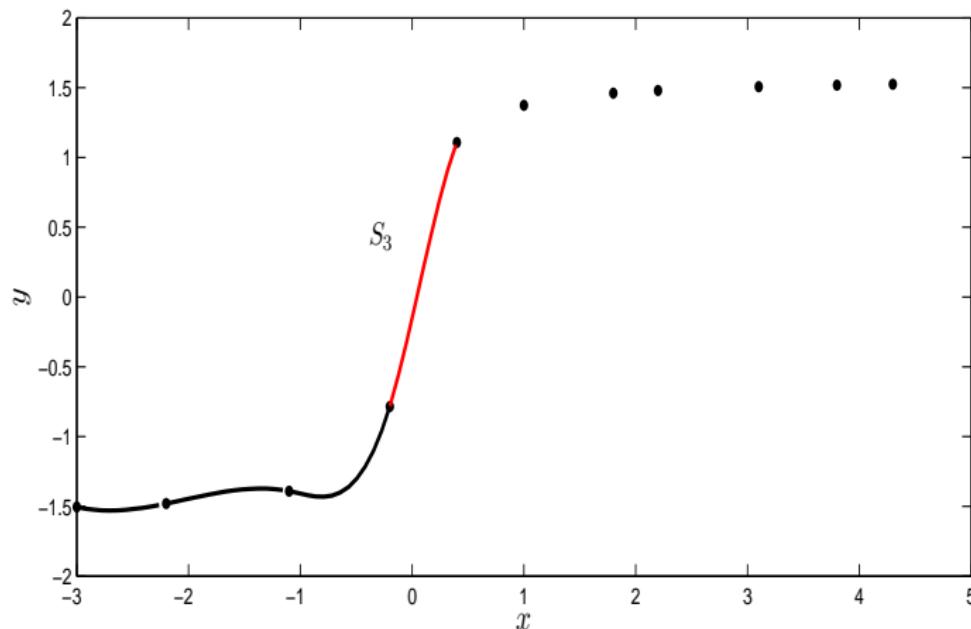
- ▶ $S(x)$ is twice continuously differentiable.
- ▶ Its third derivative is piecewise constant.
- ▶ $S(x)$ is found by solving a sparse linear system.

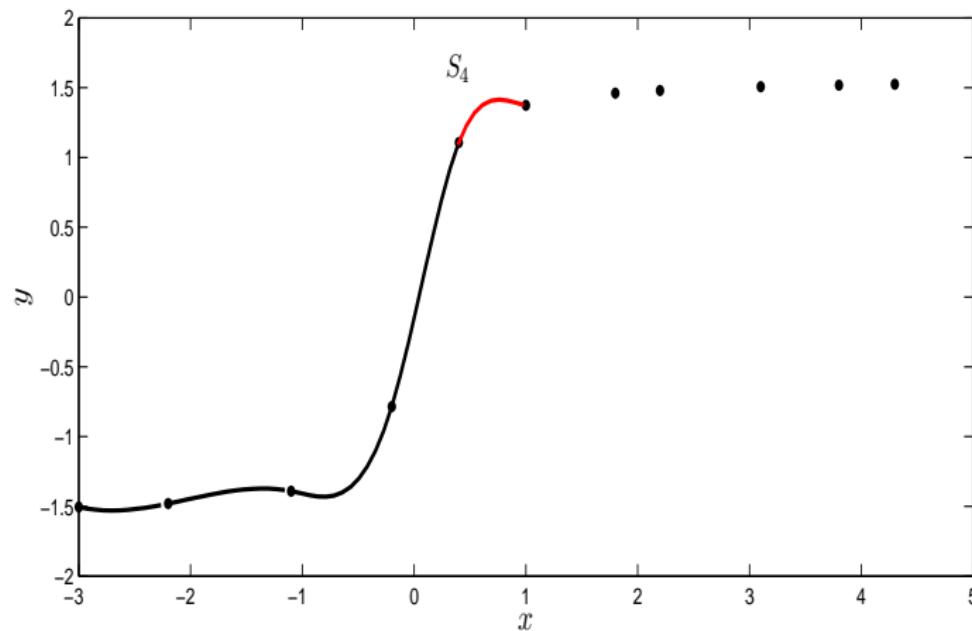


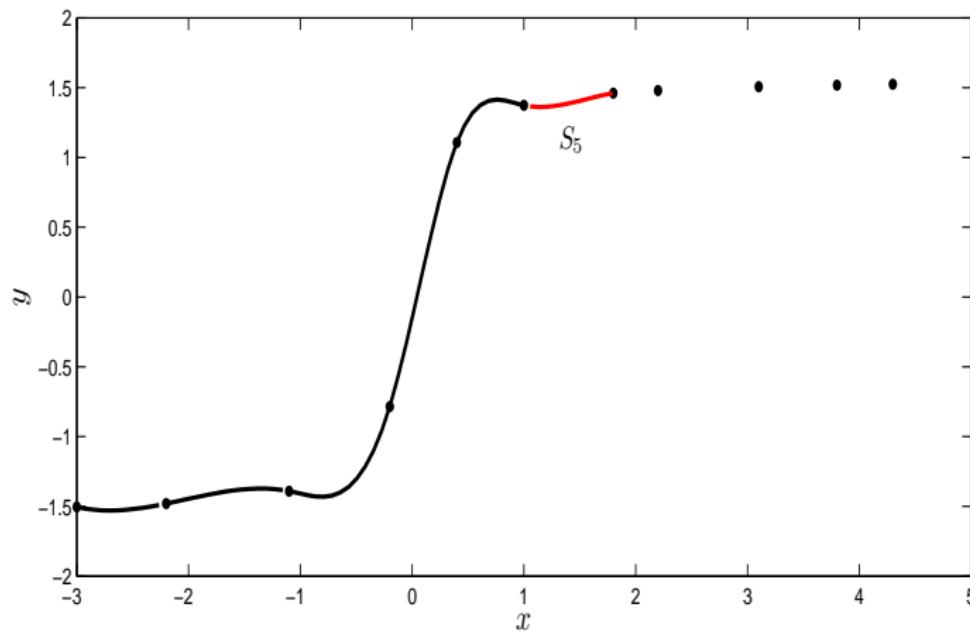


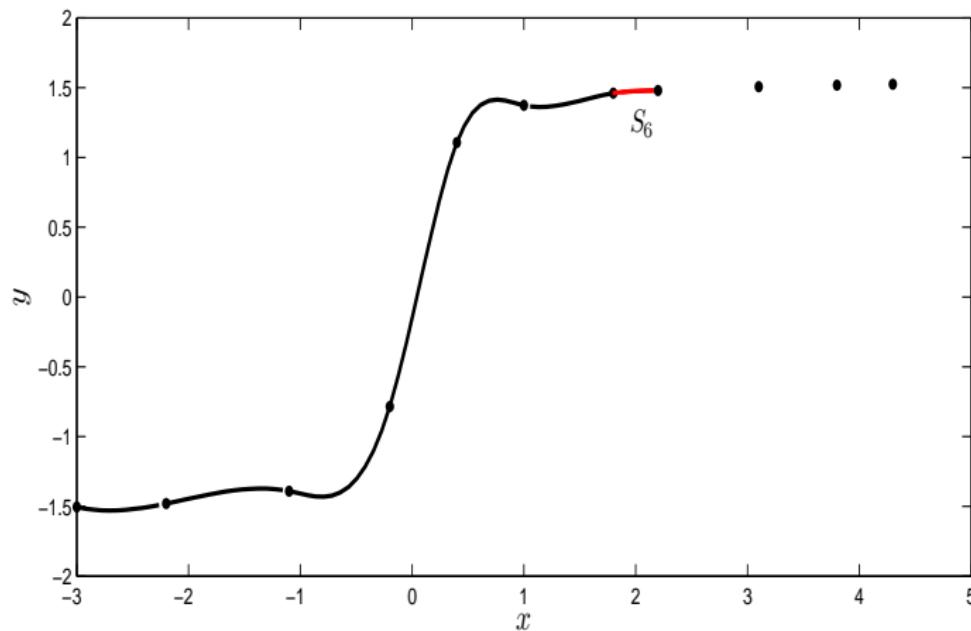


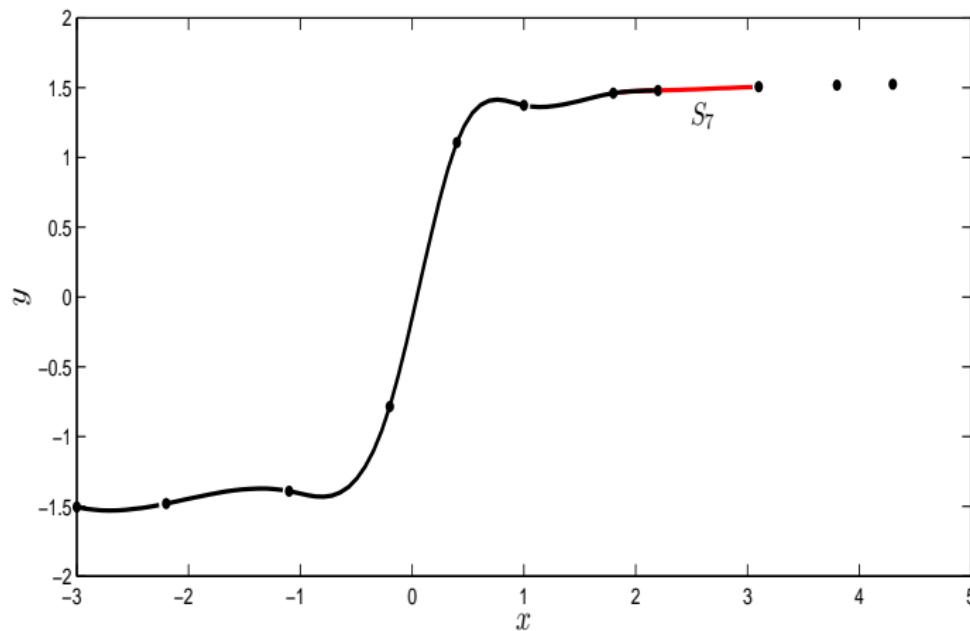


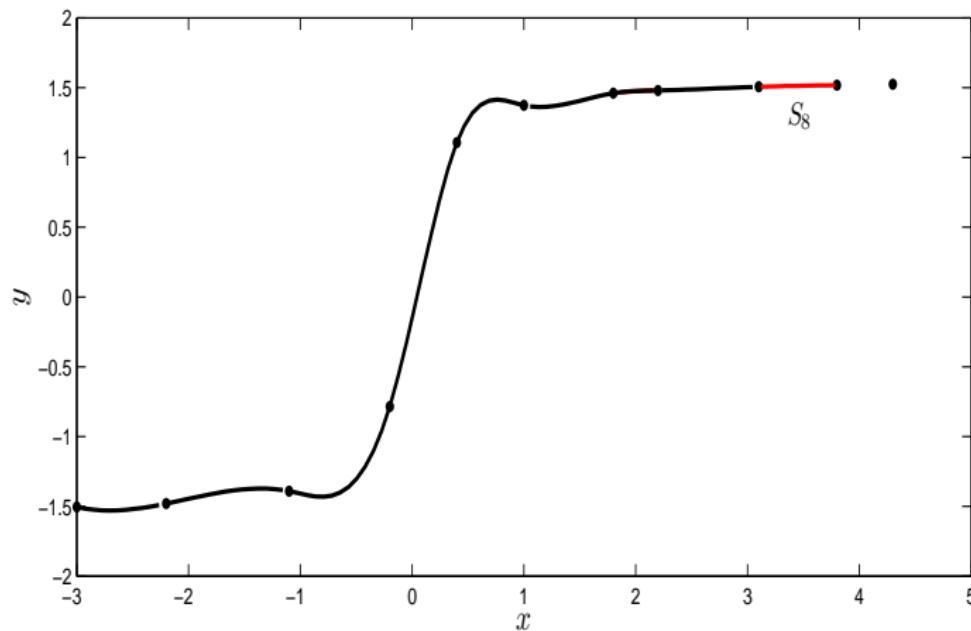


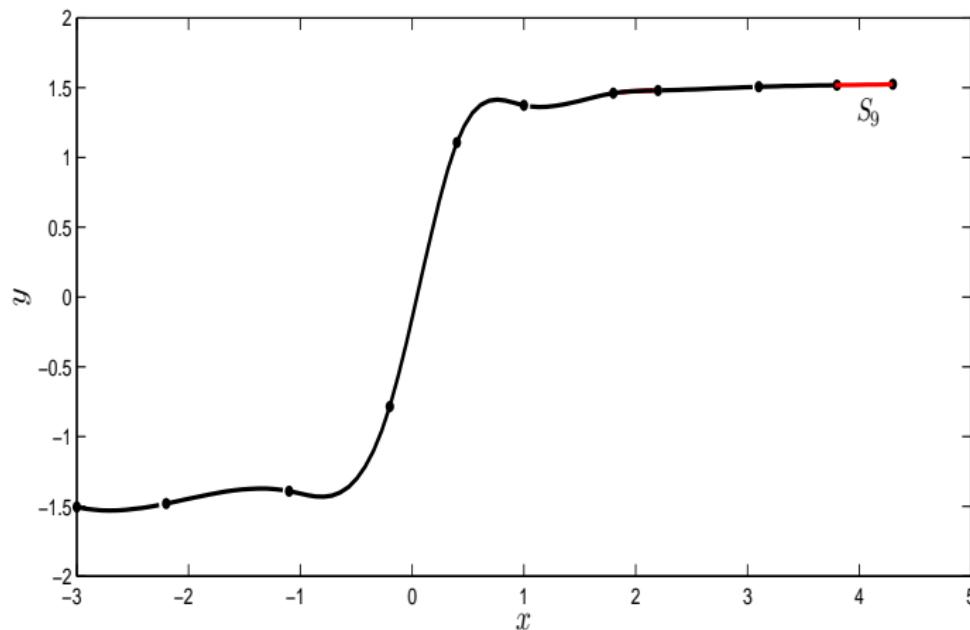


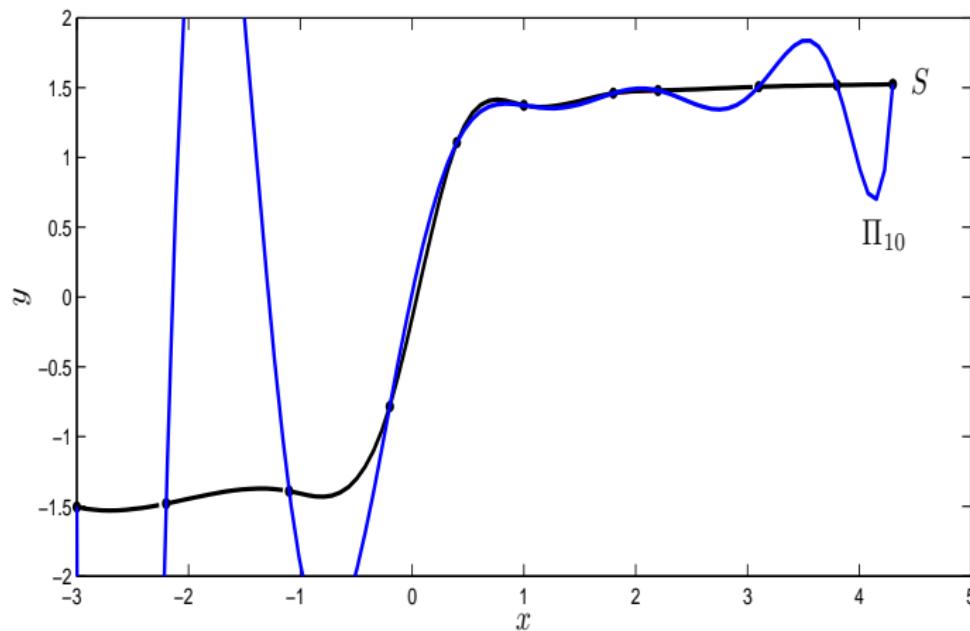












2072U Computational Science I

Winter 2023

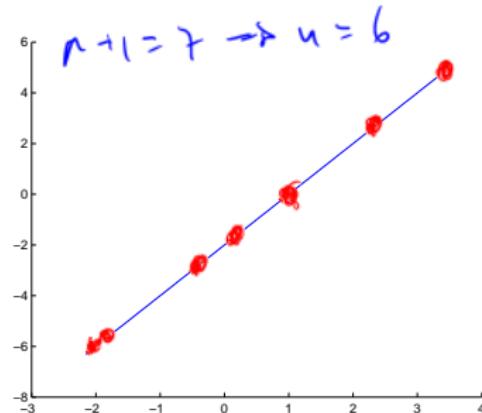
Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. Least Squares: introduction
2. Overdetermined systems of linear equations
3. Overdetermined systems with SCIPY
4. A first-order least-squares fit

Today's questions:

- ▶ Why use least squares instead of interpolation?
- ▶ What type of equations do we need to solve?
- ▶ What is “least”? What do we minimise?
- ▶ How can we do this with SCIPY?
- ▶ What equations do we get for a linear approximation?

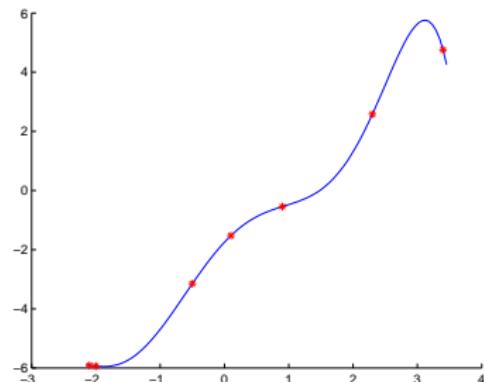
- ▶ Interpolation is useful for approximating smooth functions, e.g.
 - ▶ hard-to-evaluate functions (\sin , \cos , $\operatorname{arctanh}$, ...)
 - ▶ solutions to differential or functional equations.
- ▶ Existence of the interpolant is guaranteed.
- ▶ Efficient methods exist (see Lecture 16)
- ▶ We have an explicit upper bound for the interpolation error.



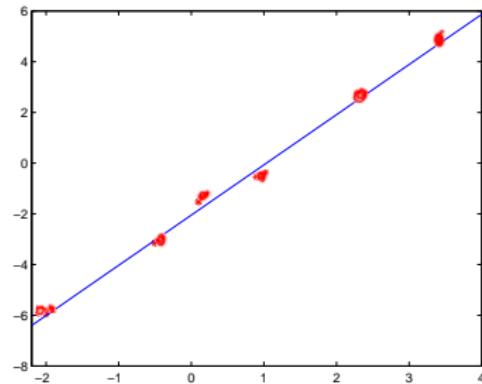
$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \dots$$

$\uparrow \quad \uparrow$
 $= 0 \quad \neq 0$

- ▶ Interpolation does not make sense for data with *noise*. In real-world applications, noise stems from
 - ▶ uncertainty in measurements,
 - ▶ cumulating numerical error,
 - ▶ stochastic processes,
 - ▶ ...
- ▶ Forcing the interpolant to go through noisy data gives strange results.



- ▶ Instead we could try to find a *low-order polynomial* approximant that interpolates the data.
- ▶ The set of equations for this approximant will be *over-determined*; generically, the approximant cannot satisfy all conditions.
- ▶ We can only find an *approximate* solution to these equations.
- ▶ We want to do this in a unique, optimal way.
- ▶ This leads to the *least-squares* solution.



$$P_1(x) = a_0 + a_1 x$$

There are more conditions of the form $P(x_i) = y_i$ than there are coefficients in P . This leads to an **overdetermined** system of linear equations:

$$\begin{bmatrix} A \end{bmatrix} \begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} b \end{bmatrix}$$

A system of linear equations $Ax = b$ where,

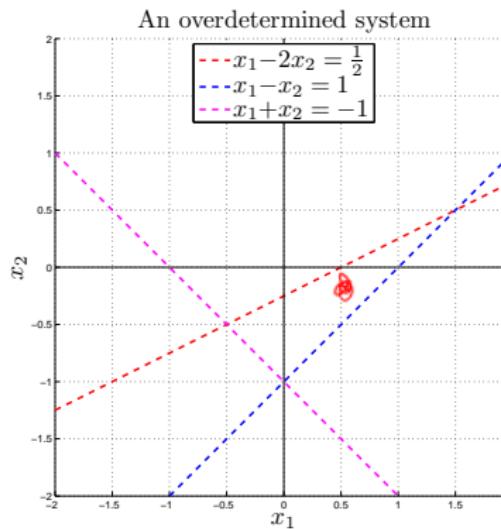
- ▶ $A \in \mathbb{R}^{N \times M}$ (i.e., A is “tall and thin”)
- ▶ $b \in \mathbb{R}^{N \times 1}$ (RHS vector)
- ▶ $x \in \mathbb{R}^{M \times 1}$ (vector of unknowns)
- ▶ $N > M$ (more equations than unknowns)

An overdetermined linear system in \mathbb{R}^2

$$\left\{ \begin{array}{l} x_1 - 2x_2 = 1/2 \\ x_1 - x_2 = 1 \\ x_1 + x_2 = -1 \end{array} \right.$$

- ▶ $N = 3$ equations
- ▶ $M = 2$ unknowns
- ▶ Overdetermined
($N > M$)
- ▶ No solution exists
- ▶ In matrix form,

$$\begin{bmatrix} 1 & -2 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1/2 \\ 1 \\ -1 \end{bmatrix}$$



Reminder: Residuals and norms on \mathbb{R}^N .

When solving

$$A\mathbf{y} = \mathbf{b}; \quad \mathbf{y} \in \mathbb{R}^M, \quad A \in \mathbb{R}^{N \times M}, \quad \mathbf{b} \in \mathbb{R}^N$$

the **residual** of \mathbf{y} is

$$\textcolor{red}{\rightarrow} \quad r(\mathbf{y}) = \mathbf{b} - A\mathbf{y}$$

“the amount by which \mathbf{y} *fails to satisfy* system $A\mathbf{x} = \mathbf{b}$ ” Recall:
to quantify **size** of vectors, introduce **norms**:

Reminder: Residuals and norms on \mathbb{R}^N .

When solving

$$A\mathbf{y} = \mathbf{b}; \quad \mathbf{y} \in \mathbb{R}^M, \quad A \in \mathbb{R}^{N \times M}, \quad \mathbf{b} \in \mathbb{R}^N$$

the **residual** of \mathbf{y} is

$$\mathbf{r}(\mathbf{y}) = \mathbf{b} - A\mathbf{y}$$

“the amount by which \mathbf{y} *fails to satisfy* system $A\mathbf{x} = \mathbf{b}$ ” Recall:
to quantify **size** of vectors, introduce **norms**:

- ▶ ℓ_1 -norm: $\|\mathbf{x}\|_1 := \sum_{k=1}^N |x_k|$ for any $\mathbf{x} \in \mathbb{R}^N$
- ▶ ℓ_2 -norm: $\|\mathbf{x}\|_2 := \left[\sum_{k=1}^N |x_k|^2 \right]^{\frac{1}{2}}$ for any $\mathbf{x} \in \mathbb{R}^N$
- ▶ ℓ_∞ -norm: $\|\mathbf{x}\|_\infty := \max_{1 \leq k \leq N} |x_k|$ for any $\mathbf{x} \in \mathbb{R}^N$

How to “solve” an overdetermined system?

- ▶ $\mathbf{Ax} = \mathbf{b}$ has solution $\mathbf{x} \in \mathbb{R}^M$ iff $\mathbf{b} \in \text{range}(A)$.
- ▶ Generically, **no solution exists** for overdetermined systems.

How to “solve” an overdetermined system?

- ▶ $\mathbf{Ax} = \mathbf{b}$ has solution $\mathbf{x} \in \mathbb{R}^M$ iff $\mathbf{b} \in \text{range}(A)$.
- ▶ Generically, **no solution exists** for overdetermined systems.
- ⇒ Goal: find vector $\mathbf{x}^* \in \mathbb{R}^M$ that **minimises size of residual $\mathbf{r}(\mathbf{x}^*)$** .

How to “solve” an overdetermined system?

- ▶ $\mathbf{Ax} = \mathbf{b}$ has solution $\mathbf{x} \in \mathbb{R}^M$ iff $\mathbf{b} \in \text{range}(A)$.
- ▶ Generically, **no solution exists** for overdetermined systems.
- ⇒ Goal: find vector $\mathbf{x}^* \in \mathbb{R}^M$ that **minimises size of residual $\mathbf{r}(\mathbf{x}^*)$** .
- ▶ To minimise $\mathbf{r}(\mathbf{x}^*)$, measure size with some norm:

$$\min_{\mathbf{x} \in \mathbb{R}^M} \|\mathbf{r}(\mathbf{x})\| = \min_{\mathbf{x} \in \mathbb{R}^M} \|\mathbf{b} - \mathbf{Ax}\|$$

- ▶ \mathbf{x}^* is a **minimiser** of $\|\mathbf{r}(\mathbf{x})\|$ in that norm, i.e.,

$$\|\mathbf{r}(\mathbf{x}^*)\| \leq \|\mathbf{r}(\mathbf{y})\| \text{ for all } \mathbf{y} \in \mathbb{R}^M$$

Conceivable objective functions to minimise:

- ▶ Using ℓ_∞ -norm, define

$$\Phi_\infty(\mathbf{y}) := \|\mathbf{b} - A\mathbf{y}\|_\infty = \max_{1 \leq k \leq N} |b_k - \sum_{\ell=1}^M A_{k\ell} y_\ell|$$

= maximum abs. value of components of residual.

- ▶ Using ℓ_1 -norm, define

$$\Phi_1(\mathbf{y}) := \frac{1}{N} \|\mathbf{b} - A\mathbf{y}\|_1 = \frac{1}{N} \sum_{k=1}^N |b_k - \sum_{\ell=1}^M A_{k\ell} y_\ell|$$

= average of abs. value of residual components.

- ▶ Both these norms are nonsmooth (nondifferentiable).

Least-squares approximation of $A\mathbf{x} = \mathbf{b}$

$$\begin{aligned}\text{Define } \Phi(\mathbf{y}) &:= \frac{1}{N} \|\mathbf{b} - A\mathbf{y}\|_2^2 \\ &= \frac{1}{N} \sum_{k=1}^N \left(b_k - \sum_{\ell=1}^M A_{k\ell} y_\ell \right)^2 \\ &= \text{Mean-square residual of } \mathbf{y}.\end{aligned}$$

Least-squares approximation of $\mathbf{Ax} = \mathbf{b}$

$$\begin{aligned}\text{Define } \Phi(\mathbf{y}) &:= \frac{1}{N} \|\mathbf{b} - \mathbf{A}\mathbf{y}\|_2^2 \\ &= \frac{1}{N} \sum_{k=1}^N \left(b_k - \sum_{\ell=1}^M A_{k\ell} y_\ell \right)^2 \\ &= \text{Mean-square residual of } \mathbf{y}.\end{aligned}$$

A least-squares approximation of a solution of $\mathbf{Ax} = \mathbf{b}$ is a vector \mathbf{x}^* that minimises Φ over \mathbb{R}^M , i.e., so that $\Phi(\mathbf{x}^*) \leq \Phi(\mathbf{y})$ for all $\mathbf{y} \in \mathbb{R}^M$.

How to minimise this objective function?

- ▶ Φ for least-squares approximation is differentiable
- ▶ Condition for \mathbf{x}^* minimising Φ : **critical point**



→ $\frac{\partial}{\partial x_k^*} [\Phi(\mathbf{x}^*)] = 0 \quad (k = 1 : M)$

- ▶ For $N > M$ with A of full rank, critical point \mathbf{x}^* of Φ is

$$\mathbf{x}^* = (A^T A)^{-1} (A^T \mathbf{b}),$$

i.e., \mathbf{x}^* is determined by solving the **normal equations**

A is $N \times M$
 A^T is $M \times N$

$$A^T A \mathbf{x}^* = A^T \mathbf{b}$$

\mathbf{x} is $M \times 1$
 $A^T \mathbf{b} \rightarrow M \times 1$

- ▶ Coefficient matrix $A^T A$ is square ($M \times M$), symmetric

$A^T A$ is $(M \times N)(N \times M)$ is $M \times M$

M eqns
M unknowns

Previous example: preceding system $A\mathbf{x} = \mathbf{b}$ is

$$\begin{bmatrix} 1 & -2 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1/2 \\ 1 \\ -1 \end{bmatrix}$$

$A \in \mathbb{R}^{3 \times 2}$



Normal equations $A^T A \mathbf{x}^* = A^T \mathbf{b}$ are

$$\begin{bmatrix} 1 & 1 & 1 \\ -2 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & -2 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1^* \\ x_2^* \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ -2 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1/2 \\ 1 \\ -1 \end{bmatrix} \quad \text{or}$$

$$A^T A \rightarrow \begin{bmatrix} 3 & -2 \\ -2 & 6 \end{bmatrix} \begin{bmatrix} x_1^* \\ x_2^* \end{bmatrix} = \begin{bmatrix} 1/2 \\ -3 \end{bmatrix} \quad \text{with } A^T \mathbf{b}$$

$$\Rightarrow \mathbf{x}^* = \begin{bmatrix} x_1^* \\ x_2^* \end{bmatrix} = \begin{bmatrix} -3/14 \\ -4/7 \end{bmatrix}$$

Least-squares approximation & normal equations

- ▶ `scipy.linalg.lstsq(A, b)` computes least-squares approximations of overdetermined systems

```
A = numpy.array([[1.0,-2.0],[1.0,-1.0],[1.0,1.0]])
b = numpy.array([0.5,1.0,-1.0])
xstar, res, rnk, s = scipy.linalg.lstsq(A,b)
```

What you need to understand:

- ▶ Use `scipy.linalg.lstsq(A, b)` for tall/thin systems
- ▶ In addition to the least-squares solution (`xstar`), it also computes the mean-square residual (`res`), the rank of `A` (`rnk`) and the singular values of `A` (`s`)

Least-squares fit with straight line:

- ▶ Let $\Pi_1(x)$ be a straight line, i.e., $\Pi_1(x) = a_0 + a_1 x$
- ▶ Given $\{(x_k, y_k)\}_{k=0}^n$, interpolation conditions are

$$\Pi_1(x_k) = a_0 + a_1 x_k = y_k \quad (k = 0 : n)$$

- ▶ Write system of equations to fit in matrix form

$$\left. \begin{array}{l} 1 \cdot a_0 + x_0 \cdot a_1 = y_0 \\ 1 \cdot a_0 + x_1 \cdot a_1 = y_1 \\ \vdots \\ 1 \cdot a_0 + x_n \cdot a_1 = y_n \end{array} \right\}$$

 \Rightarrow

$$\begin{bmatrix} 1 & x_0 \\ 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \underbrace{\begin{bmatrix} a_0 \\ a_1 \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}}_{\mathbf{y}}$$

$$V \in \mathbb{R}^{(n+1) \times 2}$$

$$\mathbf{a} \in \mathbb{R}^{2 \times 1}$$

$$\mathbf{y} \in \mathbb{R}^{(n+1) \times 1}$$

- ▶ The least-squares approximant is found by solving the 2×2 problem $V^T V \mathbf{a} = V^T \mathbf{y}$.

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. Finite differences

2. Big O

3. Finite differences on a grid

Today's questions:

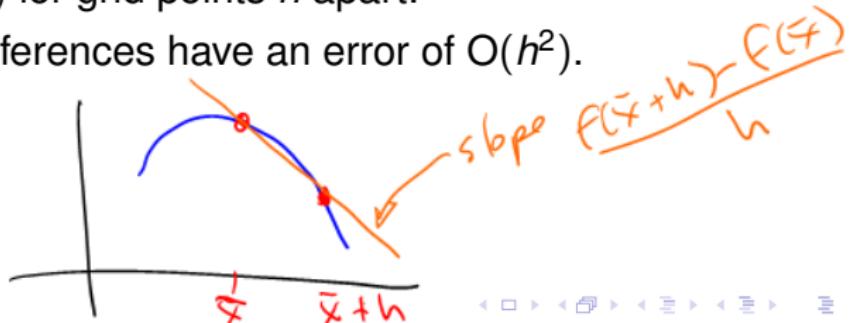
- ▶ What is numerical differentiation?
- ▶ What did $O(h^k)$ mean again?
- ▶ How to approximate derivatives of functions on a grid?

Finite differences in a nutshell:

- ▶ Finite-difference approximation of derivatives stems from

$$f'(\bar{x}) = \lim_{h \rightarrow 0} \frac{f(\bar{x} + h) - f(\bar{x})}{h}$$

- ▶ Discard the limit and take h to be small but finite.
- ▶ Error measured as function of h : how fast does it go to zero?
- ▶ *Forward* differences and *backward* differences have an error of $O(h)$ for grid points h apart.
- ▶ *Centered* differences have an error of $O(h^2)$.



Definition (“Big-Oh”)

Let $f(h)$ and $g(h)$ be two functions. Then $f(h) = O(g(h))$ as $h \rightarrow h^*$ if there exist constants $C > 0$ and $\delta > 0$ such that $|f(h)| \leq C|g(h)|$ whenever $0 < |h - h^*| < \delta$.



Definition (“Big-Oh”)

Let $f(h)$ and $g(h)$ be two functions. Then $f(h) = O(g(h))$ as $h \rightarrow h^*$ iff there exist constants $C > 0$ and $\delta > 0$ such that $|f(h)| \leq C|g(h)|$ whenever $0 < |h - h^*| < \delta$.

- ▶ “Infinitesimal asymptotics”: concerned with $h \rightarrow 0$
- ▶ Pronounced $f(h)$ is “big-oh” of $g(h)$ as h approaches h^*

$$f(h) = O(g(h)) \text{ means } \lim_{h \rightarrow h^*} \left| \frac{f(h)}{g(h)} \right| \leq C \text{ for some finite } C$$

$$\begin{aligned} h^* &= 0 \\ g(h) &= h^p \end{aligned}$$

Definition (“Big-Oh”)

Let $f(h)$ and $g(h)$ be two functions. Then $f(h) = O(g(h))$ as $h \rightarrow h^*$ iff there exist constants $C > 0$ and $\delta > 0$ such that $|f(h)| \leq C|g(h)|$ whenever $0 < |h - h^*| < \delta$.

- ▶ “Infinitesimal asymptotics”: concerned with $h \rightarrow 0$
- ▶ Pronounced $f(h)$ is “big-oh” of $g(h)$ as h approaches h^*

$$f(h) = O(g(h)) \text{ means } \lim_{h \rightarrow h^*} \left| \frac{f(h)}{g(h)} \right| \leq C \text{ for some finite } C$$

- ▶ $f(h) = O(g(h))$ means that as $h \rightarrow h^*, f(h) \rightarrow Cg(h)$

asymptotic behaviour.

$f(h)$ $g(h)$

Examples

 $\frac{f(h)}{g(h)}$

- ▶ $\sin(3h^5)/h^5 = O(1)$ as $h \rightarrow 0$: $\lim_{h \rightarrow 0} \frac{\sin(3h^5)/h^5}{1} = 3$

Examples

- ▶ $\sin(3h^5)/h^5 = O(1)$ as $h \rightarrow 0$: $\lim_{h \rightarrow 0} \frac{\sin(3h^5)/h^5}{1} = 3$
- ▶ i.e. as $h \rightarrow 0$, $\sin(3h^5)/h^5 \rightarrow 3(1)$

Examples

- ▶ $\sin(3h^5)/h^5 = O(1)$ as $h \rightarrow 0$: $\lim_{h \rightarrow 0} \frac{\sin(3h^5)/h^5}{1} = 3$
 - ▶ i.e. as $h \rightarrow 0$, $\sin(3h^5)/h^5 \rightarrow 3(1)$
 - ▶ $(e^{2h} - 1)^2 = O(h^2)$ as $h \rightarrow 0$: $\lim_{h \rightarrow 0} \frac{(e^{2h} - 1)^2}{h^2} = 4$ ↗
use l'Hopital's rule.
- $f(h)$ $g(h)$ $\frac{f(h)}{g(h)}$

Examples

- ▶ $\sin(3h^5)/h^5 = O(1)$ as $h \rightarrow 0$: $\lim_{h \rightarrow 0} \frac{\sin(3h^5)/h^5}{1} = 3$
- ▶ i.e. as $h \rightarrow 0$, $\sin(3h^5)/h^5 \rightarrow 3(1)$
- ▶ $(e^{2h} - 1)^2 = O(h^2)$ as $h \rightarrow 0$: $\lim_{h \rightarrow 0} \frac{(e^{2h} - 1)^2}{h^2} = 4$
- ▶ i.e. as $h \rightarrow 0$, $(e^{2h} - 1)^2 \rightarrow 4(h^2)$

$$f(h) \rightarrow 4g(h)$$

Examples

- ▶ $\sin(3h^5)/h^5 = O(1)$ as $h \rightarrow 0$: $\lim_{h \rightarrow 0} \frac{\sin(3h^5)/h^5}{1} = 3$
- ▶ i.e. as $h \rightarrow 0$, $\sin(3h^5)/h^5 \rightarrow 3(1)$
- ▶ $(e^{2h} - 1)^2 = O(h^2)$ as $h \rightarrow 0$: $\lim_{h \rightarrow 0} \frac{(e^{2h} - 1)^2}{h^2} = 4$
- ▶ i.e. as $h \rightarrow 0$, $(e^{2h} - 1)^2 \rightarrow 4(h^2)$
- ▶ $\sin(2h) - 2h = O(h^3)$ as $h \rightarrow 0$: $\lim_{h \rightarrow 0} \frac{\sin(2h) - 2h}{h^3} = 4$

Examples

- ▶ $\sin(3h^5)/h^5 = O(1)$ as $h \rightarrow 0$: $\lim_{h \rightarrow 0} \frac{\sin(3h^5)/h^5}{1} = 3$
- ▶ i.e. as $h \rightarrow 0$, $\sin(3h^5)/h^5 \rightarrow 3(1)$
- ▶ $(e^{2h} - 1)^2 = O(h^2)$ as $h \rightarrow 0$: $\lim_{h \rightarrow 0} \frac{(e^{2h} - 1)^2}{h^2} = 4$
- ▶ i.e. as $h \rightarrow 0$, $(e^{2h} - 1)^2 \rightarrow 4(h^2)$
- ▶ $\sin(2h) - 2h = O(h^3)$ as $h \rightarrow 0$: $\lim_{h \rightarrow 0} \frac{\sin(2h) - 2h}{h^3} = 4$
- ▶ i.e. as $h \rightarrow 0$, $\sin(2h) - 2h \rightarrow 4(h^3)$

Big O and errors

- ▶ If the error E of an approximation is $O(h)$

Big O and errors

- ▶ If the error E of an approximation is $O(h)$
- ▶ then as $h \rightarrow 0$, $E \rightarrow Ch$

Big O and errors

- ▶ If the error E of an approximation is $O(h)$
- ▶ then as $h \rightarrow 0$, $E \rightarrow Ch$ $\xrightarrow{h \rightarrow h/2} E \rightarrow E/2$
- ▶ which means cutting h by 1/2 reduces E by 1/2

Big O and errors

- ▶ If the error E of an approximation is $O(h)$
- ▶ then as $h \rightarrow 0$, $E \rightarrow Ch$
- ▶ which means cutting h by 1/2 reduces E by 1/2
- ▶ If the error E of an approximation is $O(h^2)$

Big O and errors

- ▶ If the error E of an approximation is $O(h)$
- ▶ then as $h \rightarrow 0$, $E \rightarrow Ch$
- ▶ which means cutting h by 1/2 reduces E by 1/2
- ▶ If the error E of an approximation is $O(h^2)$
- ▶ then as $h \rightarrow 0$, $E \rightarrow Ch^2$

Big O and errors

- ▶ If the error E of an approximation is $O(h)$

- ▶ then as $h \rightarrow 0$, $E \rightarrow Ch$

- ▶ which means cutting h by 1/2 reduces E by 1/2

- ▶ If the error E of an approximation is $O(h^2)$

- ▶ then as $h \rightarrow 0$, $E \rightarrow Ch^2$

- ▶ which means cutting h by 1/2 reduces E by 1/4

$$\begin{aligned} h &\rightarrow h/2 & E &\rightarrow (Ch^2) \cdot \left(\frac{h}{2}\right)^2 = \frac{Ch^2}{4} \\ &&&= \frac{E}{4} \end{aligned}$$

$O(h^2)$

Finite-difference approximations of $f'(\bar{x})$

Finite difference.

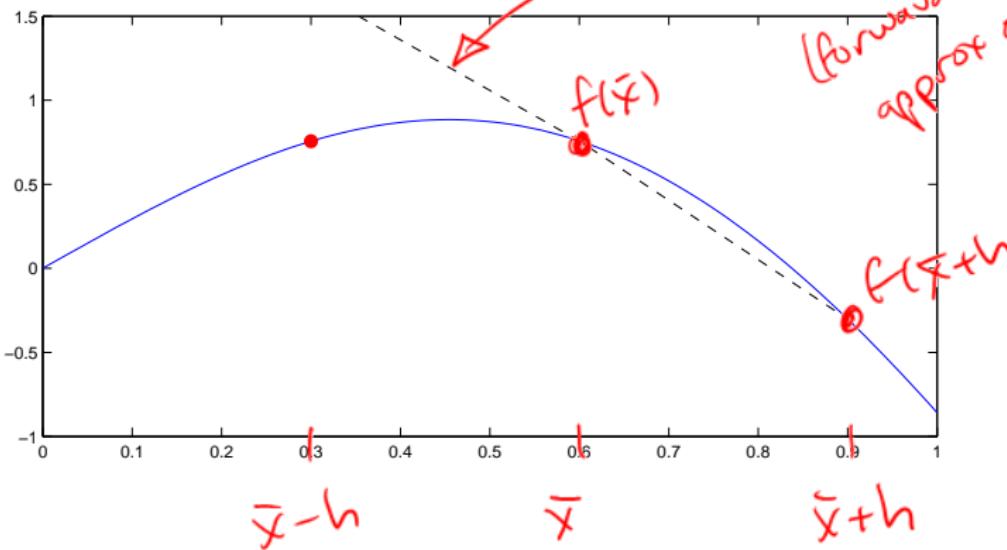
$$(\delta_+ f)(\bar{x}) := \frac{f(\bar{x} + h) - f(\bar{x})}{h} = f'(\bar{x}) + \tilde{O}(h)$$

$$(\delta_- f)(\bar{x}) := \frac{f(\bar{x}) - f(\bar{x} - h)}{h} = f'(\bar{x}) + \tilde{O}(h)$$

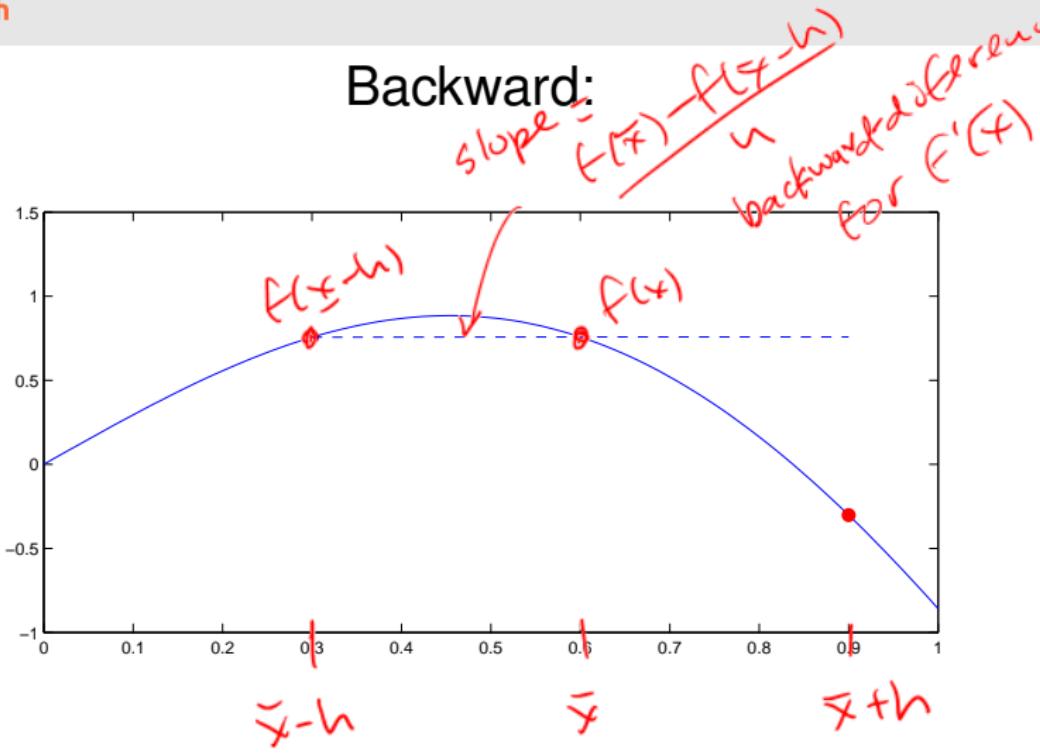
$$(\delta f)(\bar{x}) := \frac{f(\bar{x} + h) - f(\bar{x} - h)}{2h} = f'(\bar{x}) + \tilde{O}(h^2)$$

- ► $(\delta_+ f)(\bar{x})$ is **forward-difference approximation** for $f'(\bar{x})$
- ► $(\delta_- f)(\bar{x})$ is **backward-difference approximation** for $f'(\bar{x})$
- ► $(\delta f)(\bar{x})$ is **centred-difference approximation** for $f'(\bar{x})$

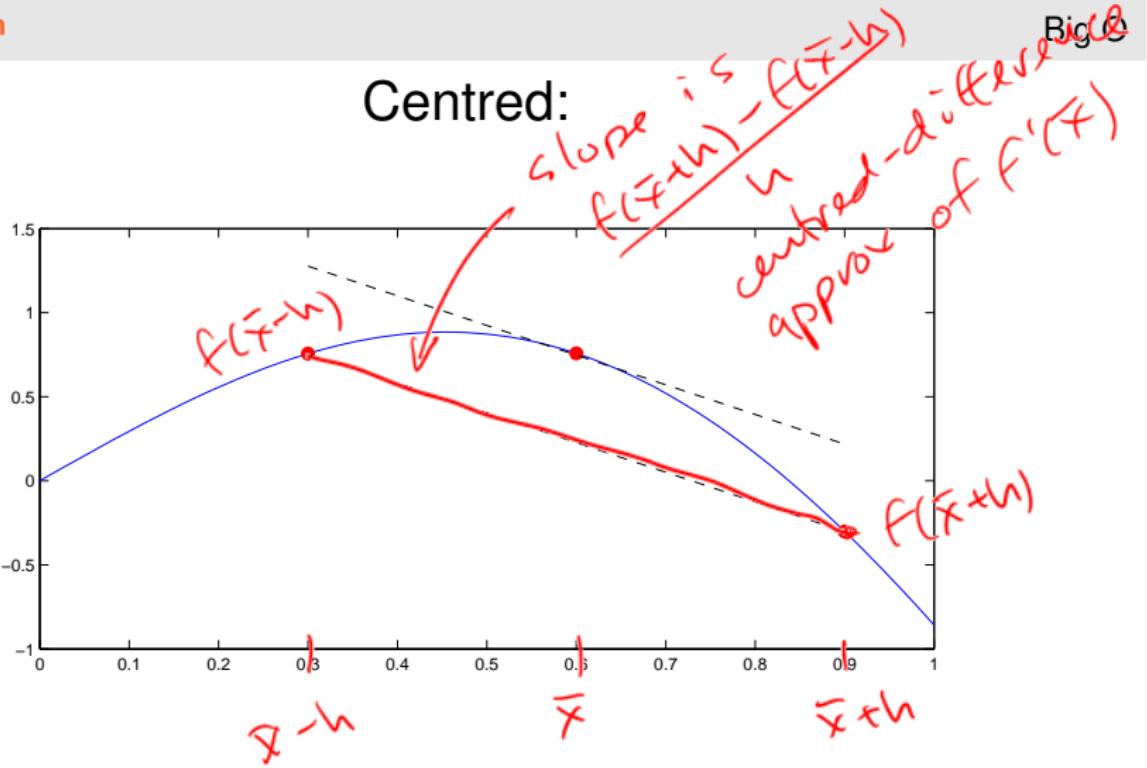
Forward:



Backward:



Centred:



Example

Let $f(x) = x^2$. Compute the forward difference approximation of $f'(2)$ with $h = 1/2$ and $h = 1/4$.

F A

$$\blacktriangleright f'(\bar{x}) \approx (\delta_+ f)(\bar{x}) := \frac{f(\bar{x} + h) - f(\bar{x})}{h}$$

Example

Let $f(x) = x^2$. Compute the forward difference approximation of $f'(2)$ with $h = 1/2$ and $h = 1/4$.

- ▶ $f'(\bar{x}) \approx (\delta_+ f)(\bar{x}) := \frac{f(\bar{x} + h) - f(\bar{x})}{h}$
- ▶ First take $\bar{x} = 2$ and $h = 1/2$, so

Example

Let $f(x) = x^2$. Compute the forward difference approximation of $f'(2)$ with $h = 1/2$ and $h = 1/4$.

- ▶ $f'(\bar{x}) \approx (\delta_+ f)(\bar{x}) := \frac{f(\bar{x} + h) - f(\bar{x})}{h}$
- ▶ First take $\bar{x} = 2$ and $h = 1/2$, so
- ▶ $f'(2) \approx \frac{f(5/2) - f(2)}{1/2} = \frac{25/4 - 4}{1/2} = 9/2$

Example

Let $f(x) = x^2$. Compute the forward difference approximation of $f'(2)$ with $h = 1/2$ and $h = 1/4$.

- ▶ $f'(\bar{x}) \approx (\delta_+ f)(\bar{x}) := \frac{f(\bar{x} + h) - f(\bar{x})}{h}$
- ▶ First take $\bar{x} = 2$ and $h = 1/2$, so
- ▶ $f'(2) \approx \frac{f(5/2) - f(2)}{1/2} = \frac{25/4 - 4}{1/2} = 9/2$
- ▶ Actual value: $f'(2) = 4$, so error is $1/2$

Example

Let $f(x) = x^2$. Compute the forward difference approximation of $f'(2)$ with $h = 1/2$ and $h = 1/4$.

- ▶ $f'(\bar{x}) \approx (\delta_+ f)(\bar{x}) := \frac{f(\bar{x} + h) - f(\bar{x})}{h}$
- ▶ First take $\bar{x} = 2$ and $h = 1/2$, so
- ▶ $f'(2) \approx \frac{f(5/2) - f(2)}{1/2} = \frac{25/4 - 4}{1/2} = 9/2$
- ▶ Actual value: $f'(2) = 4$, so error is $1/2$
- ▶ Now take $\bar{x} = 2$ and $h = 1/4$, so

Example

Let $f(x) = x^2$. Compute the forward difference approximation of $f'(2)$ with $h = 1/2$ and $h = 1/4$.

- ▶ $f'(\bar{x}) \approx (\delta_+ f)(\bar{x}) := \frac{f(\bar{x} + h) - f(\bar{x})}{h}$
- ▶ First take $\bar{x} = 2$ and $h = 1/2$, so
- ▶ $f'(2) \approx \frac{f(5/2) - f(2)}{1/2} = \frac{25/4 - 4}{1/2} = 9/2$
- ▶ Actual value: $f'(2) = 4$, so error is $1/2$
- ▶ Now take $\bar{x} = 2$ and $h = 1/4$, so
- ▶ $f'(2) \approx \frac{f(9/4) - f(2)}{1/4} = \frac{81/16 - 4}{1/4} = 17/4$

Example

Let $f(x) = x^2$. Compute the forward difference approximation of $f'(2)$ with $h = 1/2$ and $h = 1/4$.

- ▶ $f'(\bar{x}) \approx (\delta_+ f)(\bar{x}) := \frac{f(\bar{x} + h) - f(\bar{x})}{h}$
- ▶ First take $\bar{x} = 2$ and $h = 1/2$, so
- ▶ $f'(2) \approx \frac{f(5/2) - f(2)}{1/2} = \frac{25/4 - 4}{1/2} = 9/2$
- ▶ Actual value: $f'(2) = 4$, so error is $1/2$ for $h = \frac{1}{2}$
- ▶ Now take $\bar{x} = 2$ and $h = 1/4$, so
- ▶ $f'(2) \approx \frac{f(9/4) - f(2)}{1/4} = \frac{81/16 - 4}{1/4} = 17/4$
- ▶ Error is $1/4$ $h = \frac{1}{4}$


Example

Let $f(x) = x^2$. Compute the forward difference approximation of $f'(2)$ with $h = 1/2$ and $h = 1/4$.

- ▶ $f'(\bar{x}) \approx (\delta_+ f)(\bar{x}) := \frac{f(\bar{x} + h) - f(\bar{x})}{h}$
- ▶ First take $\bar{x} = 2$ and $h = 1/2$, so
- ▶ $f'(2) \approx \frac{f(5/2) - f(2)}{1/2} = \frac{25/4 - 4}{1/2} = 9/2$
- ▶ Actual value: $f'(2) = 4$, so error is $1/2$
- ▶ Now take $\bar{x} = 2$ and $h = 1/4$, so
- ▶ $f'(2) \approx \frac{f(9/4) - f(2)}{1/4} = \frac{81/16 - 4}{1/4} = 17/4$
- ▶ Error is $1/4$
- ▶ i.e. we cut h in half and the error was cut in half as expected for an $O(h)$ approximation

Centred-difference approximation for $f''(\bar{x})$:

- ▶ Recalling Taylor's theorem,

$$f(\bar{x} + h) = f(\bar{x}) + hf'(\bar{x}) + \frac{h^2}{2}f''(\bar{x}) + \frac{h^3}{6}f'''(\bar{x}) + O(h^4)$$

$$f(\bar{x} - h) = f(\bar{x}) - hf'(\bar{x}) + \frac{h^2}{2}f''(\bar{x}) - \frac{h^3}{6}f'''(\bar{x}) + O(h^4)$$

Centred-difference approximation for $f''(\bar{x})$:

- ▶ Recalling Taylor's theorem,

$$f(\bar{x} + h) = f(\bar{x}) + \cancel{hf'(\bar{x})} + \frac{h^2}{2} f''(\bar{x}) + \cancel{\frac{h^3}{6} f'''(\bar{x})} + O(h^4)$$

$$f(\bar{x} - h) = f(\bar{x}) - \cancel{hf'(\bar{x})} + \frac{h^2}{2} f''(\bar{x}) - \cancel{\frac{h^3}{6} f'''(\bar{x})} + O(h^4)$$

- ▶ Adding gives

$$f(\bar{x} - h) + f(\bar{x} + h) = 2f(\bar{x}) + h^2 f''(\bar{x}) + O(h^4)$$

Centred-difference approximation for $f''(\bar{x})$:

- ▶ Recalling Taylor's theorem,

$$f(\bar{x} + h) = f(\bar{x}) + hf'(\bar{x}) + \frac{h^2}{2}f''(\bar{x}) + \frac{h^3}{6}f'''(\bar{x}) + O(h^4)$$

$$f(\bar{x} - h) = f(\bar{x}) - hf'(\bar{x}) + \frac{h^2}{2}f''(\bar{x}) - \frac{h^3}{6}f'''(\bar{x}) + O(h^4)$$

- ▶ Adding gives

$$f(\bar{x} - h) + f(\bar{x} + h) = 2f(\bar{x}) + h^2f''(\bar{x}) + O(h^4)$$

- ▶ We arrive at the *centred-difference formula*

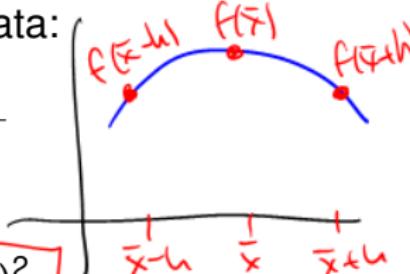
$$f''(\bar{x}) = (\delta^2 f)(\bar{x}) + \Theta(h^2)$$

$$(\delta^2 f)(\bar{x}) := \frac{f(\bar{x} + h) - 2f(\bar{x}) + f(\bar{x} - h)}{h^2}$$

Alternative derivation:

Compute the interpolating polynomial for the data:

x_k	$\bar{x} - h$	\bar{x}	$\bar{x} + h$
y_k	$f(\bar{x} - h)$	$f(\bar{x})$	$f(\bar{x} + h)$



Use a convenient form for the interpolant:

$$\rightarrow P_2(x) = a + b(x - \bar{x}) + c(x - \bar{x})^2$$

Equations:

$$P_2(\bar{x} - h) = a - bh + ch^2 = f(\bar{x} - h)$$

$$P_2(\bar{x}) = a = f(\bar{x})$$

$$P_2(\bar{x} + h) = a + bh + ch^2 = f(\bar{x} + h)$$

so that

$$P_2(x) = [f(\bar{x})] + \left[\frac{1}{2h} [f(\bar{x} + h) - f(\bar{x} - h)] \right] (x - \bar{x}) \\ + \left[\frac{1}{2h^2} [f(\bar{x} - h) - 2f(\bar{x}) + f(\bar{x} + h)] \right] (x - \bar{x})^2$$

And finally

$$P_2'(x) = b + 2c(x - \bar{x})$$

$$P_2''(x) = 2c$$

$$f'(\bar{x}) \approx P_2'(\bar{x}) = \frac{1}{2h} [f(\bar{x} + h) - f(\bar{x} - h)]$$

$$f''(\bar{x}) \approx P_2''(\bar{x}) = \frac{1}{h^2} [f(\bar{x} - h) - 2f(\bar{x}) + f(\bar{x} + h)]$$

The error also follows:

$$\rightarrow f(x) - P_2(x) = \frac{f^{(3)}(\xi(x))}{3!} (x - \bar{x} + h)(x - \bar{x})(x - \bar{x} - h) \Rightarrow$$

$$\rightarrow f'(\bar{x}) - P_2'(\bar{x}) = \frac{f^{(3)}(\xi(x))}{3!} h^2 = O(h^2)$$

In general: to find the finite difference formula for the k^{th} derivative,

- ▶ place $k + 1$ interpolation nodes symmetrically around \bar{x} and h apart;
- ▶ compute the interpolating polynomial P_k on these nodes;
- ▶ approximate $f^k(\bar{x})$ by $P^k(\bar{x})$;
- ▶ derive the error from the interpolation error bound.

Assume function values specified at equispaced nodes

$$x_k = x_0 + kh \quad (k = 0 : n)$$

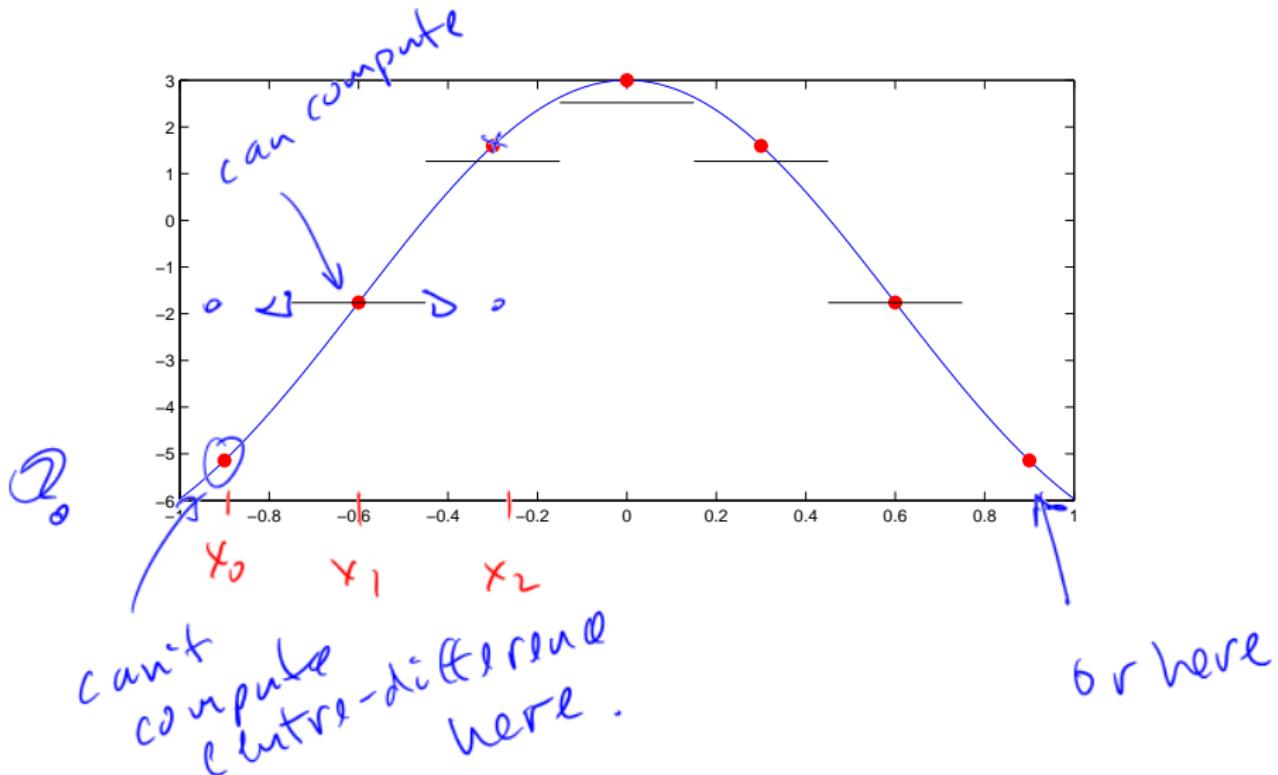
Finite-difference formulas apply on different range of nodes

Forward: $f'(x_k) \simeq (\delta_+ f)(x_k) = \frac{f(x_{k+1}) - f(x_k)}{h} \quad (k = 0 : n - 1)$

Backward: $f'(x_k) \simeq (\delta_- f)(x_k) = \frac{f(x_k) - f(x_{k-1})}{h} \quad (k = 1 : n)$

Centred: $f'(x_k) \simeq (\delta f)(x_k) = \frac{f(x_{k+1}) - f(x_{k-1})}{2h} \quad (k = 1 : n - 1)$

Centred differences on a grid:



NUMPY's `diff` command for vectors:

$$y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-2} \\ y_{N-1} \\ y_N \end{bmatrix} \Rightarrow \text{numpy.diff}(y) = \begin{bmatrix} y_2 - y_1 \\ y_3 - y_2 \\ \vdots \\ y_{N-1} - y_{N-2} \\ y_N - y_{N-1} \end{bmatrix}$$

- ▶ For vectors, `numpy.diff` subtracts successive entries
 - ▶ length of `diff(y)` = (length of `y`) – 1

Higher-order differences with `diff`:

- ▶ Higher-order differences: repeated application of `diff`

$$y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-2} \\ y_{N-1} \\ y_N \end{bmatrix} \Rightarrow \text{diff}(y, 2) = \begin{bmatrix} y_3 - 2y_2 + y_1 \\ y_4 - 2y_3 + y_2 \\ \vdots \\ y_N - 2y_{N-1} + y_{N-2} \end{bmatrix}$$

- ▶ `numpy.diff(y, 2)` equivalent to
`numpy.diff(numpy.diff(y))`

Numerical derivatives in with `diff`:

- ▶ `diff` computes differences of successive elements in a vector.
- ▶ `diff` operates on arrays of numerical data.
- ▶ For uniform grid (i.e. each element of an array f corresponds to values of a function at equally spaced points), $\frac{1}{h} * \text{diff}(f)$ gives first-order derivative (either forward or backward)

Numerical derivatives in with `diff`:

- ▶ `diff` computes differences of successive elements in a vector.
- ▶ `diff` operates on arrays of numerical data.
- ▶ For uniform grid (i.e. each element of an array f corresponds to values of a function at equally spaced points), $\frac{1}{h} * \text{diff}(f)$ gives first-order derivative (either forward or backward)
- ▶ `diff` does not do symbolic differentiation (e.g., as in MAPLE).

Numerical derivatives in with `diff`:

- ▶ `diff` computes differences of successive elements in a vector.
- ▶ `diff` operates on arrays of numerical data.
- ▶ For uniform grid (i.e. each element of an array f corresponds to values of a function at equally spaced points), $\frac{1}{h} * \text{diff}(f)$ gives first-order derivative (either forward or backward)
- ▶ `diff` does not do symbolic differentiation (e.g., as in MAPLE).
- ▶ For nonuniform grids, need to do the differences explicitly; use in conjunction with “vectorised” division for convenience.

Definition (“Little-Oh”)

Let $f(h)$ and $g(h)$ be two functions. Then $f(h) = o(g(h))$ as $h \rightarrow h^*$ iff for every constant $C > 0$, there exists $\delta > 0$ such that $|f(h)| \leq C|g(h)|$ whenever $0 < |h - h^*| < \delta$.

Definition (“Little-Oh”)

Let $f(h)$ and $g(h)$ be two functions. Then $f(h) = o(g(h))$ as $h \rightarrow h^*$ iff for every constant $C > 0$, there exists $\delta > 0$ such that $|f(h)| \leq C|g(h)|$ whenever $0 < |h - h^*| < \delta$.

- ▶ Pronounced $f(h)$ is “little-oh” of $g(h)$ as h approaches h^*

$$f(h) = o(g(h)) \text{ means } \lim_{h \rightarrow h^*} \left| \frac{f(h)}{g(h)} \right| = 0$$

Definition (“Little-Oh”)

Let $f(h)$ and $g(h)$ be two functions. Then $f(h) = o(g(h))$ as $h \rightarrow h^*$ iff for every constant $C > 0$, there exists $\delta > 0$ such that $|f(h)| \leq C|g(h)|$ whenever $0 < |h - h^*| < \delta$.

- ▶ Pronounced $f(h)$ is “little-oh” of $g(h)$ as h approaches h^*

$$f(h) = o(g(h)) \text{ means } \lim_{h \rightarrow h^*} \left| \frac{f(h)}{g(h)} \right| = 0$$

- ▶ $f(h) = o(g(h))$ means $f(h) \rightarrow 0$ **faster** than $g(h)$

Examples

- ▶ $\sin^2 h = o(h)$ as $h \rightarrow 0$ because $\lim_{h \rightarrow 0} \frac{\sin^2 h}{h} = 0$

Examples

- ▶ $\sin^2 h = o(h)$ as $h \rightarrow 0$ because $\lim_{h \rightarrow 0} \frac{\sin^2 h}{h} = 0$
- ▶ $(e^{2h} - 1)^2 = o(h)$ as $h \rightarrow 0$ because $\lim_{h \rightarrow 0} \frac{(e^{2h} - 1)^2}{h} = 0$

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. Quadrature formulas

2. Midpoint formula

3. Trapezoidal formula

4. Simpson formula

5. Composite formula

6. Error of quadrature

Key questions

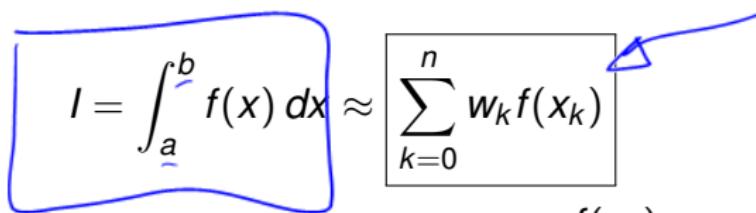
- ▶ What is a quadrature formula?
- ▶ What are the essential components of a quadrature formula?
- ▶ How do quadrature rules relate to polynomial interpolation?
- ▶ What is a composite quadrature formula?
- ▶ How can standard quadrature formulas be applied?

Quadrature formulas:

- ▶ **Quadrature**: numerical approximation of definite integrals.
- ▶ Quadrature formulas have basic form

$$\boxed{I = \int_a^b f(x) dx} \approx \boxed{\sum_{k=0}^n w_k f(x_k)}$$

= w_0 f(x_0) + w_1 f(x_1) + \cdots + w_n f(x_n)



- ▶ Replaces integration by weighted sum of function evaluations.

Quadrature formulas:

- ▶ **Quadrature**: numerical approximation of definite integrals.
- ▶ Quadrature formulas have basic form

$$\begin{aligned} I = \int_a^b f(x) dx &\approx \boxed{\sum_{k=0}^n w_k f(x_k)} \\ &= w_0 f(x_0) + w_1 f(x_1) + \cdots + w_n f(x_n) \end{aligned}$$

- ▶ Replaces integration by weighted sum of function evaluations.
- ▶ $\{x_k\}_{k=0}^n$ are **quadrature nodes/points**.
- ▶ $\{w_k\}_{k=0}^n$ are **quadrature weights**.
- ▶ Weights depend on $\{x_k\}_{k=0}^n$, width $b - a$ of interval.

Quadrature formulas and polynomial interpolation:

For quadrature formula I_{appr} to approximate $I = \int_a^b f(x) dx$:

Quadrature formulas and polynomial interpolation:

For quadrature formula I_{appr} to approximate $I = \int_a^b f(x) dx$:

- ▶ Choose quadrature points/nodes/abscissae $\{x_k\}_{k=0}^n$

$$a \leq x_0 < x_1 < \dots < x_{n-1} < x_n \leq b$$

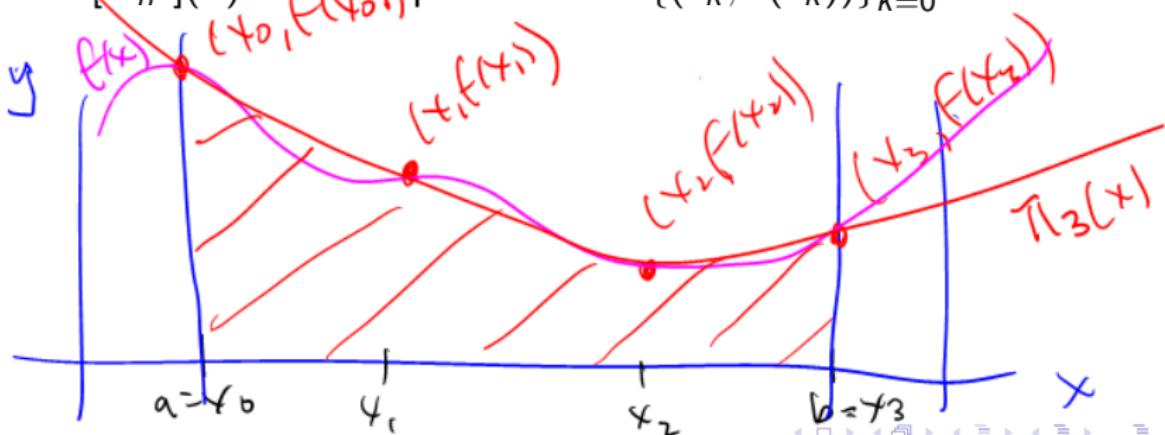
Quadrature formulas and polynomial interpolation:

For quadrature formula I_{appr} to approximate $I = \int_a^b f(x) dx$:

- ▶ Choose quadrature points/nodes/abscissae $\{x_k\}_{k=0}^n$

$$a \leq x_0 < x_1 < \dots < x_{n-1} < x_n \leq b$$

- ▶ Sample f at quadrature points and construct polynomial $[\Pi_n f](x)$ that interpolates data $\{(x_k, f(x_k))\}_{k=0}^n$



Quadrature formulas and polynomial interpolation:

For quadrature formula I_{appr} to approximate $I = \int_a^b f(x) dx$:

- ▶ Choose quadrature points/nodes/abscissae $\{x_k\}_{k=0}^n$

$$a \leq x_0 < x_1 < \dots < x_{n-1} < x_n \leq b$$

- ▶ Sample f at quadrature points and construct polynomial $[\Pi_n f](x)$ that interpolates data $\{(x_k, f(x_k))\}_{k=0}^n$
- ▶ From interpolant $\Pi_n f$, compute

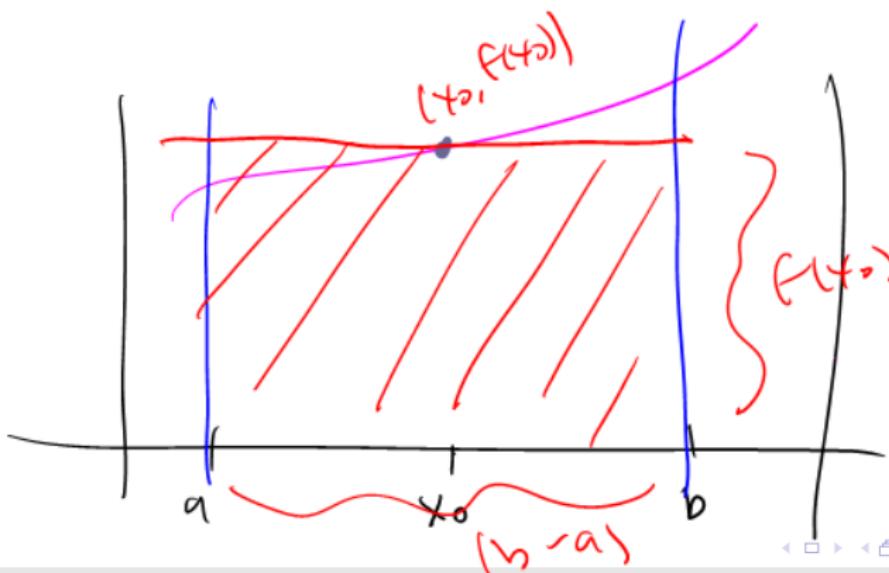
$$I_{\text{appr}} := \int_a^b [\Pi_n f](x) dx \simeq \int_a^b f(x) dx = I$$

Midpoint formula

fit one degree polynomial
(constant)

The midpoint formula I_{mp} is given by

$$I_{mp} := (b - a)f\left(\frac{a + b}{2}\right).$$



area = $f(t_0)(b-a)$
where
 $t_0 = \frac{(a+b)}{2}$
is the midpoint

Midpoint formula

The **midpoint formula** I_{mp} is given by

$$I_{mp} := (b - a)f\left(\frac{a + b}{2}\right).$$

$\downarrow \quad \downarrow$
 $x_0 \quad y_0$

- ▶ Sample f at midpoint $x = \frac{a + b}{2}$ of interval $[a, b]$

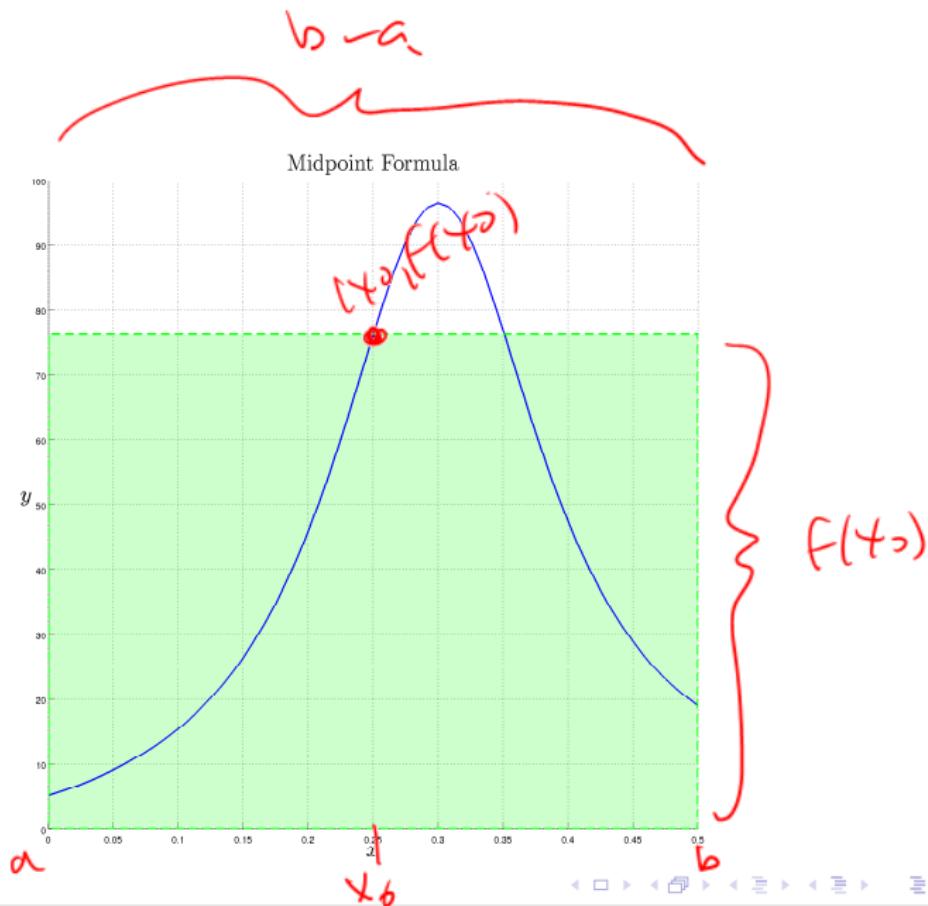
Midpoint formula

The midpoint formula I_{mp} is given by

$$I_{mp} := (b - a)f\left(\frac{a + b}{2}\right).$$

- ▶ Sample f at midpoint $x = \frac{a + b}{2}$ of interval $[a, b]$
- ▶ Approximate $\int_a^b f(x) dx$ by rectangle

$$\text{Area} = (\text{width})(\text{height}) = (b - a)f\left(\frac{a + b}{2}\right)$$



Use the midpoint formula I_{mp} to approximate

$$I = \int_0^1 \sqrt{1 + x^4} dx.$$

Use the midpoint formula I_{mp} to approximate

$$I = \int_0^1 \sqrt{1 + x^4} dx.$$

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$.

Use the midpoint formula I_{mp} to approximate

$$I = \int_0^1 \sqrt{1 + x^4} dx.$$

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$.
- ▶ Midpoint of $[a, b] = [0, 1]$ is at
 $\bar{x} = (a + b)/2 = (0 + 1)/2 = \frac{1}{2}$.

Use the midpoint formula I_{mp} to approximate

$$I = \int_0^1 \sqrt{1 + x^4} dx.$$

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$.
- ▶ Midpoint of $[a, b] = [0, 1]$ is at
 $\bar{x} = (a + b)/2 = (0 + 1)/2 = \frac{1}{2}$.
- ▶ Midpoint formula approximation is

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

Use the midpoint formula I_{mp} to approximate

$$I = \int_0^1 \sqrt{1 + x^4} dx.$$

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$.
- ▶ Midpoint of $[a, b] = [0, 1]$ is at
 $\bar{x} = (a + b)/2 = (0 + 1)/2 = \frac{1}{2}$.
- ▶ Midpoint formula approximation is

$$I_m = (b - a)f\left(\frac{a + b}{2}\right) = (1 - 0)f\left(\frac{1}{2}\right)$$

Use the midpoint formula I_{mp} to approximate

$$I = \int_0^1 \sqrt{1 + x^4} dx.$$

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$.
- ▶ Midpoint of $[a, b] = [0, 1]$ is at
 $\bar{x} = (a + b)/2 = (0 + 1)/2 = \frac{1}{2}$.
- ▶ Midpoint formula approximation is

$$\begin{aligned} I_m &= (b - a)f\left(\frac{a + b}{2}\right) = (1 - 0)f\left(\frac{1}{2}\right) \\ &= \sqrt{1 + \left(\frac{1}{2}\right)^4} \end{aligned}$$

Use the midpoint formula I_{mp} to approximate

$$I = \int_0^1 \sqrt{1 + x^4} dx.$$

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$.
- ▶ Midpoint of $[a, b] = [0, 1]$ is at
 $\bar{x} = (a + b)/2 = (0 + 1)/2 = \frac{1}{2}$.
- ▶ Midpoint formula approximation is

$$\begin{aligned} I_m &= (b - a)f\left(\frac{a + b}{2}\right) = (1 - 0)f\left(\frac{1}{2}\right) \\ &= \sqrt{1 + \left(\frac{1}{2}\right)^4} = \sqrt{1 + \frac{1}{16}} \end{aligned}$$

Use the midpoint formula I_{mp} to approximate

$$I = \int_0^1 \sqrt{1 + x^4} dx.$$

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$.
- ▶ Midpoint of $[a, b] = [0, 1]$ is at
 $\bar{x} = (a + b)/2 = (0 + 1)/2 = \frac{1}{2}$.
- ▶ Midpoint formula approximation is

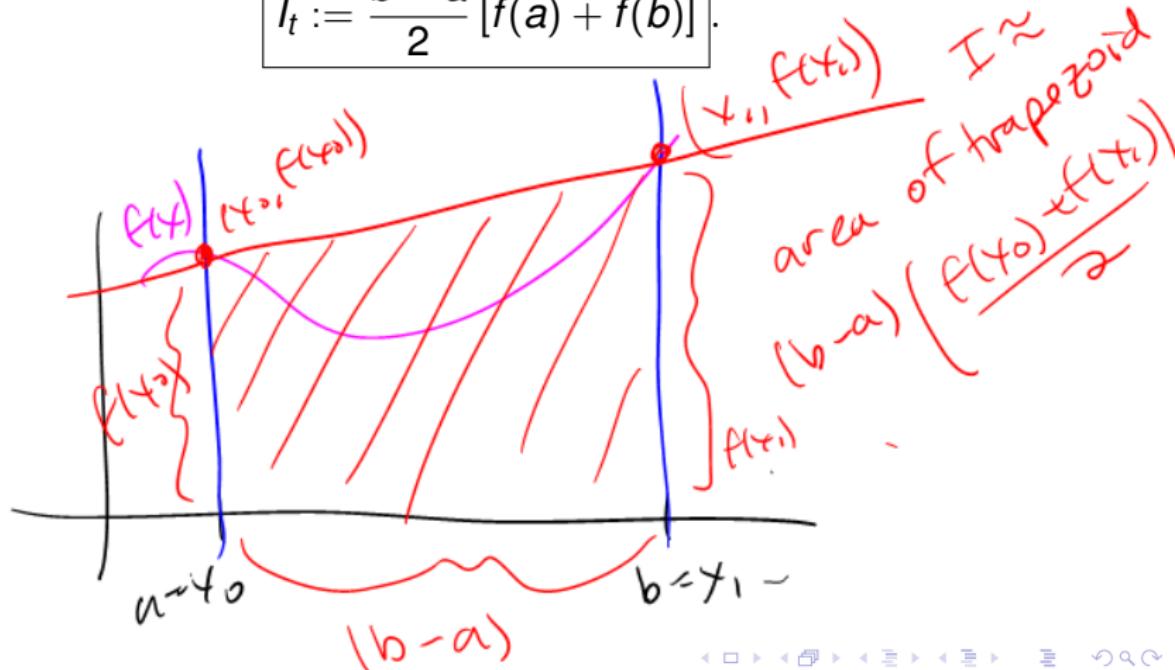
$$\begin{aligned} I_m &= (b - a)f\left(\frac{a + b}{2}\right) = (1 - 0)f\left(\frac{1}{2}\right) \\ &= \sqrt{1 + \left(\frac{1}{2}\right)^4} = \sqrt{1 + \frac{1}{16}} \\ &= \boxed{\frac{\sqrt{17}}{4} \simeq 1.030776406404415} \end{aligned}$$

fit 1st degree polynomial (line)

Trapezoidal formula

The **trapezoidal formula** I_t is given by

$$I_t := \frac{b-a}{2} [f(a) + f(b)].$$



Trapezoidal formula

The **trapezoidal formula** I_t is given by

$$I_t := \frac{b-a}{2} [f(a) + f(b)].$$

ψ_0, ψ_1 x_0 x_1

- ▶ Sample f at end points $x = a, b$ of interval $[a, b]$.

Trapezoidal formula

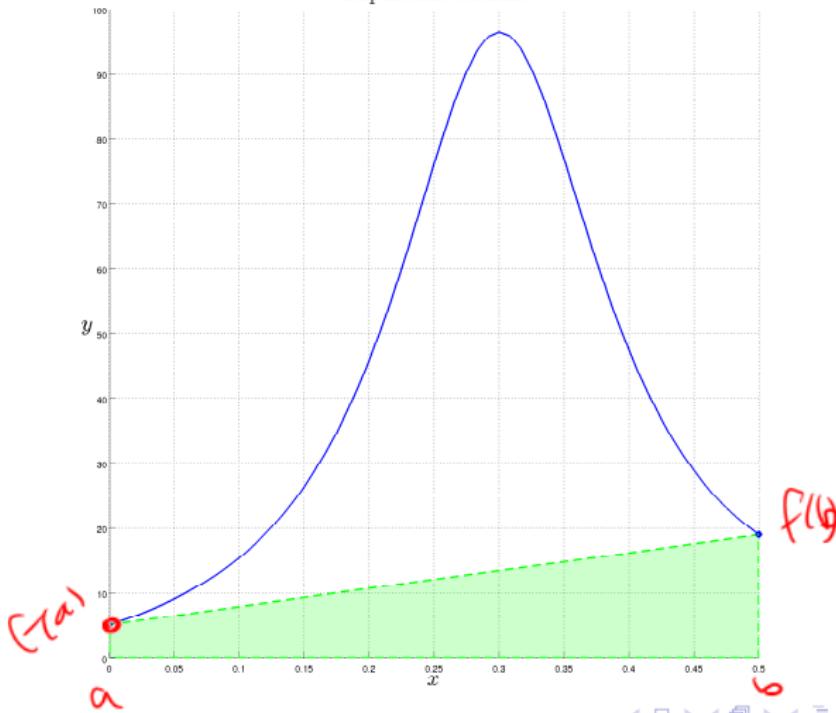
The **trapezoidal formula** I_t is given by

$$I_t := \frac{b-a}{2} [f(a) + f(b)].$$

- ▶ Sample f at end points $x = a, b$ of interval $[a, b]$.
- ▶ Approximate $\int_a^b f(x) dx$ by **trapezoid**

$$\text{Area} = (\text{width})(\text{average height}) = (b-a) \left(\frac{f(a) + f(b)}{2} \right)$$

Trapezoidal Formula



Or using interpolation:

x_k	a	b
y_k	$f(a)$	$f(b)$

data

Find the interpolant $P_1(x) = a_0 + a_1 x$ from

$$\begin{aligned} P_1(a) &= a_0 + a_1 \quad a = f(a) \\ P_1(b) &= a_0 + a_1 \quad b = f(b) \end{aligned} \Leftrightarrow \begin{pmatrix} 1 & a \\ 1 & b \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} f(a) \\ f(b) \end{pmatrix}$$

so that

$$\begin{aligned} f(x) &= P_1(x) + E(x) = \left[\frac{bf(a) - af(b)}{b-a} \right] + \left[\frac{f(b) - f(a)}{b-a} \right] x \\ &\quad + \frac{1}{2} f^{(2)}(\xi(x))(x-a)(x-b) \end{aligned}$$

so that

solve for

know

know

$$\begin{aligned} I &= \int_a^b f(x) dx = \int_a^b P_1(x) dx + \int_a^b E(x) dx \\ &= \boxed{\frac{1}{2}(b-a)(f(a) + f(b))} + \frac{1}{2} \int_a^b f^{(2)}(\xi(x))(x-a)(x-b)dx \\ &= \frac{1}{2}(b-a)(f(a) + f(b)) - \frac{1}{12} f^{(2)}(\xi')(b-a)^3 \end{aligned}$$

approx to I

error.

so that the error is

$$\left| \int_a^b f(x) dx - \frac{1}{2}(b-a)(f(a) + f(b)) \right| = O(h^3)$$

with $h = b - a$.

Approximate $I = \int_0^1 \sqrt{1 + x^4} dx$ using the trapezoidal formula
 I_t .

Approximate $I = \int_0^1 \sqrt{1 + x^4} dx$ using the trapezoidal formula I_t .

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$.

Approximate $I = \int_0^1 \sqrt{1 + x^4} dx$ using the trapezoidal formula I_t .

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$.
- ▶ Trapezoidal formula approximation is

$$I_t = \frac{b - a}{2} [f(a) + f(b)]$$

Approximate $I = \int_0^1 \sqrt{1 + x^4} dx$ using the trapezoidal formula I_t .

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$.
- ▶ Trapezoidal formula approximation is

$$\begin{aligned}I_t &= \frac{b-a}{2} [f(a) + f(b)] \\&= \frac{1-0}{2} \left[\sqrt{1+0^4} + \sqrt{1+1^4} \right]\end{aligned}$$

Approximate $I = \int_0^1 \sqrt{1 + x^4} dx$ using the trapezoidal formula I_t .

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$.
- ▶ Trapezoidal formula approximation is

$$\begin{aligned}I_t &= \frac{b-a}{2} [f(a) + f(b)] \\&= \frac{1-0}{2} [\sqrt{1+0^4} + \sqrt{1+1^4}] \\&= \frac{1}{2} [1 + \sqrt{2}] \\&\simeq 1.207106781186547\end{aligned}$$

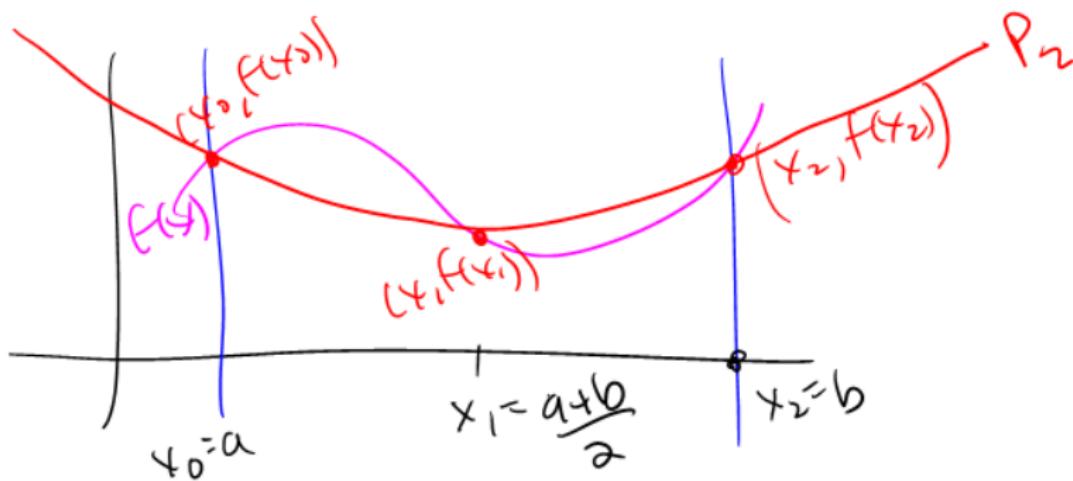
fit to 2nd degree poly (quadratic)

Simpson formula

The **Simpson formula** I_s is given by

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

$$I_s := \frac{b-a}{6} \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right].$$



Simpson formula

The **Simpson formula** I_s is given by

$$I_s := \frac{b-a}{6} \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right].$$

- ▶ Sample f at end points $x = a, b$ & midpoint $\bar{x} = \frac{a+b}{2}$ of $[a, b]$, use quadratic interpolation.

Simpson formula

The **Simpson formula** I_s is given by

$$I_s := \frac{b-a}{6} \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right].$$

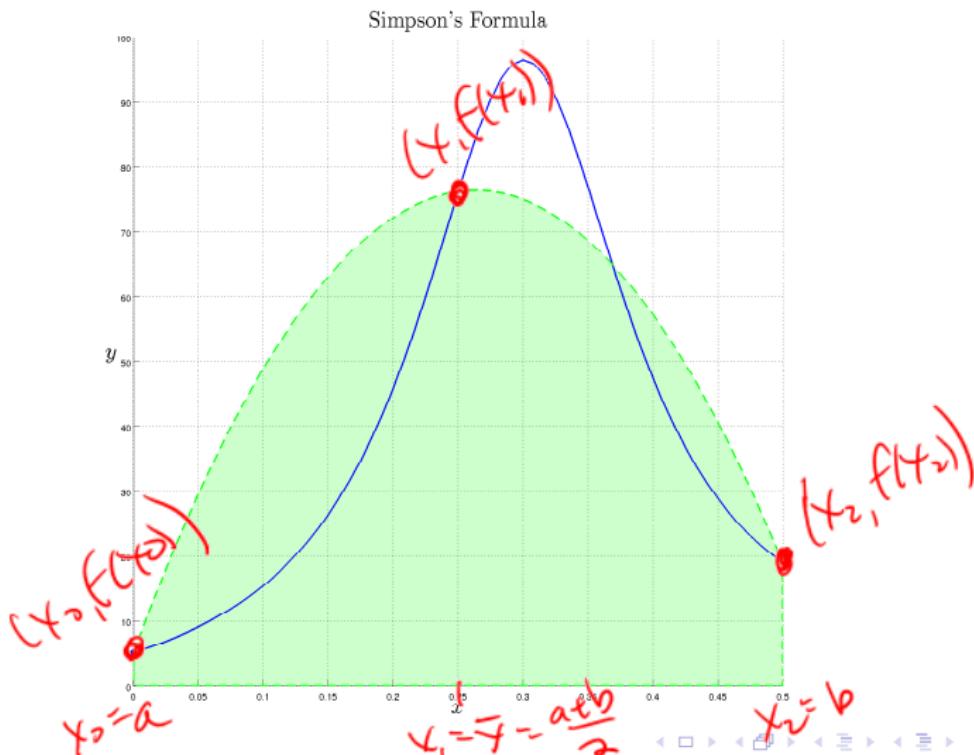
- ▶ Sample f at end points $x = a, b$ & midpoint $\bar{x} = \frac{a+b}{2}$ of $[a, b]$, use quadratic interpolation.
- ▶ Construct interpolant through $(a, f(a)), (\bar{x}, f(\bar{x})), (b, f(b))$.

Simpson formula

The **Simpson formula** I_s is given by

$$I_s := \frac{b-a}{6} \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right].$$

- ▶ Sample f at end points $x = a, b$ & midpoint $\bar{x} = \frac{a+b}{2}$ of $[a, b]$, use quadratic interpolation.
- ▶ Construct interpolant through $(a, f(a)), (\bar{x}, f(\bar{x})), (b, f(b))$.
- ▶ Approximate $\int_a^b f(x) dx$ by **area under parabola**.



Simpson formula: example

Use the Simpson formula I_s to approximate $I = \int_0^1 \sqrt{1 + x^4} dx$.

Simpson formula: example

Use the Simpson formula I_s to approximate $I = \int_0^1 \sqrt{1 + x^4} dx$.

- Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$

Simpson formula: example

Use the Simpson formula I_s to approximate $I = \int_0^1 \sqrt{1 + x^4} dx$.

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$
- ▶ Midpoint of $[a, b] = [0, 1]$ is $\bar{x} = (a + b)/2 = (0 + 1)/2 = \frac{1}{2}$

Simpson formula: example

Use the Simpson formula I_s to approximate $I = \int_0^1 \sqrt{1 + x^4} dx$.

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$
- ▶ Midpoint of $[a, b] = [0, 1]$ is $\bar{x} = (a + b)/2 = (0 + 1)/2 = \frac{1}{2}$
- ▶ Simpson formula approximation is

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

Simpson formula: example

Use the Simpson formula I_s to approximate $I = \int_0^1 \sqrt{1 + x^4} dx$.

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$
- ▶ Midpoint of $[a, b] = [0, 1]$ is $\bar{x} = (a + b)/2 = (0 + 1)/2 = \frac{1}{2}$
- ▶ Simpson formula approximation is

$$\begin{aligned} I_s &= \frac{b - a}{6} \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right] \\ &= \frac{(1 - 0)}{6} \left[f(0) + 4f\left(\frac{1}{2}\right) + f(1) \right] \end{aligned}$$

Simpson formula: example

Use the Simpson formula I_s to approximate $I = \int_0^1 \sqrt{1 + x^4} dx$.

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$
- ▶ Midpoint of $[a, b] = [0, 1]$ is $\bar{x} = (a + b)/2 = (0 + 1)/2 = \frac{1}{2}$
- ▶ Simpson formula approximation is

$$\begin{aligned} I_s &= \frac{b - a}{6} \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right] \\ &= \frac{(1 - 0)}{6} \left[f(0) + 4f\left(\frac{1}{2}\right) + f(1) \right] \\ &= \frac{1}{6} \left[1 + 4 \cdot \frac{\sqrt{17}}{4} + \sqrt{2} \right] \end{aligned}$$

Simpson formula: example

Use the Simpson formula I_s to approximate $I = \int_0^1 \sqrt{1 + x^4} dx$.

- ▶ Here, $a = 0$, $b = 1$, $f(x) = \sqrt{1 + x^4}$
- ▶ Midpoint of $[a, b] = [0, 1]$ is $\bar{x} = (a + b)/2 = (0 + 1)/2 = \frac{1}{2}$
- ▶ Simpson formula approximation is

$$\begin{aligned} I_s &= \frac{b - a}{6} \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right] \\ &= \frac{(1 - 0)}{6} \left[f(0) + 4f\left(\frac{1}{2}\right) + f(1) \right] \\ &= \frac{1}{6} \left[1 + 4 \cdot \frac{\sqrt{17}}{4} + \sqrt{2} \right] \\ &= \boxed{(1 + \sqrt{17} + \sqrt{2})/6 \simeq 1.089553197998459} \end{aligned}$$

- ▶ Recall: polynomial interpolation improved by piecewise polynomial interpolation (shorter intervals, lower degree).
- ▶ Partition $[a, b]$ into M subintervals $I_k = [x_{k-1}, x_k]$ ($k = 1 : M$)
- ▶ Partitioning $[a, b]$ as described yields

$$I = \int_a^b f(x) dx = \sum_{k=1}^M \left[\int_{I_k} f(x) dx \right] = \sum_{k=1}^M \left[\int_{x_{k-1}}^{x_k} f(x) dx \right]$$

- ▶ **Composite quadrature formulas:** apply to basic quadrature formulas to partitioned subintervals on $[a, b]$.
- ▶ Consider subintervals of equal size. In this case, we have

$$x_\ell := a + \ell h = a + \frac{\ell(b - a)}{M} \quad (\ell = 0 : M)$$

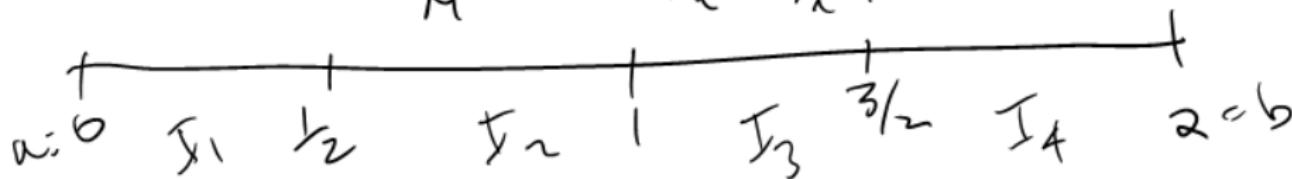
$$h = \frac{b-a}{M}$$

Partition $[0, 2]$ into 4 equal subintervals
 $(a=0, b=2, M=4) \rightarrow h = \frac{b-a}{M} = \frac{2-0}{4} = \frac{1}{2}$

nodes: $x_e = a + eh = 0 + e \frac{1}{2} = \frac{e}{2}$

$$\Rightarrow x_0 = 0, x_1 = \frac{1}{2}, x_2 = 1, x_3 = \frac{3}{2}, x_4 = 2$$

since $h = \frac{b-a}{M} \Rightarrow x_e - x_{e-1} = h$



$$I_k = [x_{k-1}, x_k] \Rightarrow$$

$$I_1 = [x_0, x_1] = [0, \frac{1}{2}], I_2 = [x_1, x_2] = [\frac{1}{2}, 1] \text{ etc}$$

$$\Rightarrow I = \int_0^2 f(x) dx = \sum_{k=1}^4 \int_{x_{k-1}}^{x_k} f(x) dx$$

$$= \int_0^{1/2} f(x) dx + \int_{1/2}^1 f(x) dx + \int_1^{3/2} f(x) dx + \int_{3/2}^2 f(x) dx$$

now approximate each of these w/ a quadrature formula.

e.g. trapezoidal formula

$$\int_{x_{k-1}}^{x_k} f(x) dx = \left(\frac{x_k - x_{k-1}}{2} \right) \overbrace{[f(x_k) + f(x_{k-1})]}^{\Rightarrow h} = \frac{h}{2} [f(x_k) + f(x_{k-1})]$$

$$I \approx h \left[\frac{1}{2} f(x_0) + \frac{1}{2} f(x_1) \right] + h \left[\frac{1}{2} f(x_1) + \frac{1}{2} f(x_2) \right] \\ + h \left[\frac{1}{2} f(x_2) + \frac{1}{2} f(x_3) \right] + h \left[\frac{1}{2} f(x_3) + \frac{1}{2} f(x_4) \right]$$

\Rightarrow

$$I \approx h \left[\frac{1}{2} f(0) + \frac{1}{2} f(x_1) + f\left(\frac{1}{2}\right) + f(1) + f\left(\frac{3}{2}\right) \right]$$

$$w_0 = \frac{h}{2}, w_1 = \frac{h}{2}, w_2 = w_3 = h$$

quadrature weights.

Composite midpoint formula

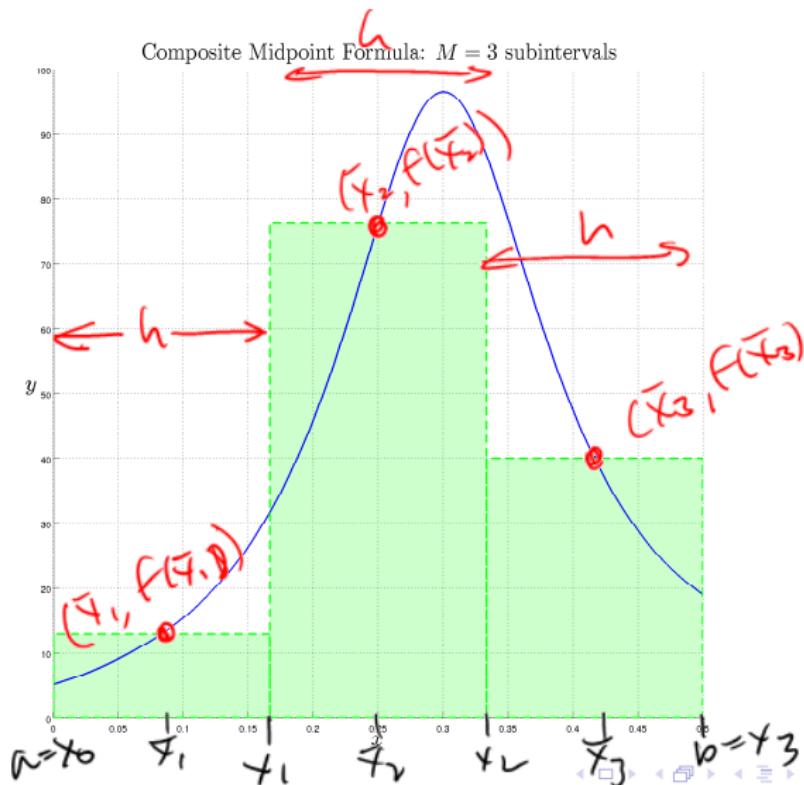
Let $[a, b]$ be partitioned into M equal subintervals of width $h = (b - a)/M$ with endpoints $x_\ell = a + \ell h$ ($\ell = 0 : M$). Define the midpoints \bar{x}_k of each subinterval by

$$\bar{x}_k := \frac{x_{k-1} + x_k}{2} \quad (k = 1 : M).$$

☞

Then, the **composite midpoint formula** is

$$I_{mp}^c := h \sum_{k=1}^M f(\bar{x}_k).$$



Approximate $I = \frac{2}{\sqrt{\pi}} \int_{\frac{1}{2}}^2 e^{-x^2} dx$ using the composite midpoint formula I_{mp}^c with $M = 3$ subintervals.

Approximate $I = \frac{2}{\sqrt{\pi}} \int_{\frac{1}{2}}^2 e^{-x^2} dx$ using the composite midpoint formula I_{mp}^c with $M = 3$ subintervals.

► Here, $f(x) = \frac{2}{\sqrt{\pi}} e^{-x^2}$, $a = \frac{1}{2}$, $b = 2$, $M = 3$, so

→ $h = \frac{b-a}{M} = \frac{1}{2}$

Approximate $I = \frac{2}{\sqrt{\pi}} \int_{\frac{1}{2}}^2 e^{-x^2} dx$ using the composite midpoint formula I_{mp}^c with $M = 3$ subintervals.

- ▶ Here, $f(x) = \frac{2}{\sqrt{\pi}} e^{-x^2}$, $a = \frac{1}{2}$, $b = 2$, $M = 3$, so

$$h = \frac{b-a}{M} = \frac{1}{2}$$

- ▶ Endpoints of subintervals are $x_\ell = a + \ell h$ ($\ell = 0 : M$), i.e.,

$$x_0 = \frac{1}{2}, \quad x_1 = 1, \quad x_2 = \frac{3}{2}, \quad x_3 = 2$$

Approximate $I = \frac{2}{\sqrt{\pi}} \int_{\frac{1}{2}}^2 e^{-x^2} dx$ using the composite midpoint formula I_{mp}^c with $M = 3$ subintervals.

- ▶ Here, $f(x) = \frac{2}{\sqrt{\pi}} e^{-x^2}$, $a = \frac{1}{2}$, $b = 2$, $M = 3$, so

$$h = \frac{b-a}{M} = \frac{1}{2}$$

- ▶ Endpoints of subintervals are $x_\ell = a + \ell h$ ($\ell = 0 : M$), i.e.,

$$x_0 = \frac{1}{2}, \quad x_1 = 1, \quad x_2 = \frac{3}{2}, \quad x_3 = 2$$

- ▶ Midpoints of subintervals are $\bar{x}_k := \frac{x_{k-1}+x_k}{2}$ ($k = 1 : M$), i.e.,

$$\bar{x}_1 = \frac{3}{4}, \quad \bar{x}_2 = \frac{5}{4}, \quad \bar{x}_3 = \frac{7}{4}$$

Approximate $I = \frac{2}{\sqrt{\pi}} \int_{\frac{1}{2}}^2 e^{-x^2} dx$ using the composite midpoint formula I_{mp}^c with $M = 3$ subintervals.

Approximate $I = \frac{2}{\sqrt{\pi}} \int_{\frac{1}{2}}^2 e^{-x^2} dx$ using the composite midpoint formula I_{mp}^c with $M = 3$ subintervals.

- ▶ Thus, the composite midpoint formula yields the approximation

$$I_{mp}^c = h \sum_{k=1}^M f(\bar{x}_k)$$

Approximate $I = \frac{2}{\sqrt{\pi}} \int_{\frac{1}{2}}^2 e^{-x^2} dx$ using the composite midpoint formula I_{mp}^c with $M = 3$ subintervals.

- ▶ Thus, the composite midpoint formula yields the approximation

$$I_{mp}^c = h \sum_{k=1}^M f(\bar{x}_k) = \frac{1}{2} [f(\bar{x}_1) + f(\bar{x}_2) + f(\bar{x}_3)]$$

Approximate $I = \frac{2}{\sqrt{\pi}} \int_{\frac{1}{2}}^2 e^{-x^2} dx$ using the composite midpoint formula I_{mp}^c with $M = 3$ subintervals.

- ▶ Thus, the composite midpoint formula yields the approximation

$$\begin{aligned} I_{mp}^c &= h \sum_{k=1}^M f(\bar{x}_k) = \frac{1}{2} [f(\bar{x}_1) + f(\bar{x}_2) + f(\bar{x}_3)] \\ &= \frac{1}{2} \frac{2}{\sqrt{\pi}} \left[e^{-(3/4)^2} + e^{-(5/4)^2} + e^{-(7/4)^2} \right] \end{aligned}$$

Approximate $I = \frac{2}{\sqrt{\pi}} \int_{\frac{1}{2}}^2 e^{-x^2} dx$ using the composite midpoint formula I_{mp}^c with $M = 3$ subintervals.

- ▶ Thus, the composite midpoint formula yields the approximation

$$\begin{aligned} I_{mp}^c &= h \sum_{k=1}^M f(\bar{x}_k) = \frac{1}{2} [f(\bar{x}_1) + f(\bar{x}_2) + f(\bar{x}_3)] \\ &= \frac{1}{2} \frac{2}{\sqrt{\pi}} \left[e^{-(3/4)^2} + e^{-(5/4)^2} + e^{-(7/4)^2} \right] \\ &= \frac{1}{\sqrt{\pi}} \left[e^{-9/16} + e^{-25/16} + e^{-49/16} \right] \end{aligned}$$

Approximate $I = \frac{2}{\sqrt{\pi}} \int_{\frac{1}{2}}^2 e^{-x^2} dx$ using the composite midpoint formula I_{mp}^c with $M = 3$ subintervals.

- ▶ Thus, the composite midpoint formula yields the approximation

$$\begin{aligned} I_{mp}^c &= h \sum_{k=1}^M f(\bar{x}_k) = \frac{1}{2} [f(\bar{x}_1) + f(\bar{x}_2) + f(\bar{x}_3)] \\ &= \frac{1}{2} \frac{2}{\sqrt{\pi}} \left[e^{-(3/4)^2} + e^{-(5/4)^2} + e^{-(7/4)^2} \right] \\ &= \frac{1}{\sqrt{\pi}} \left[e^{-9/16} + e^{-25/16} + e^{-49/16} \right] \\ &\simeq \boxed{0.466113593786324} \end{aligned}$$

Composite trapezoidal formula

Let $[a, b]$ be partitioned into M equal subintervals of width $h = (b - a)/M$ with endpoints $x_\ell = a + \ell h$ ($\ell = 0 : M$). Then, the **composite trapezoidal formula** is

$$I_t^c := h \left[\frac{f(a) + f(b)}{2} + \sum_{k=1}^{M-1} f(x_k) \right].$$

Composite trapezoidal formula

Let $[a, b]$ be partitioned into M equal subintervals of width $h = (b - a)/M$ with endpoints $x_\ell = a + \ell h$ ($\ell = 0 : M$). Then, the **composite trapezoidal formula** is

$$I_t^c := h \left[\frac{f(a) + f(b)}{2} + \sum_{k=1}^{M-1} f(x_k) \right].$$

Observe $I = \sum_{k=1}^M \int_{x_{k-1}}^{x_k} f(x) dx$

Composite trapezoidal formula

Let $[a, b]$ be partitioned into M equal subintervals of width $h = (b - a)/M$ with endpoints $x_\ell = a + \ell h$ ($\ell = 0 : M$). Then, the **composite trapezoidal formula** is

$$I_t^c := h \left[\frac{f(a) + f(b)}{2} + \sum_{k=1}^{M-1} f(x_k) \right].$$

Observe $I = \sum_{k=1}^M \int_{x_{k-1}}^{x_k} f(x) dx \simeq \sum_{k=1}^M I_t(f; x_{k-1}, x_k)$

Composite trapezoidal formula

Let $[a, b]$ be partitioned into M equal subintervals of width $h = (b - a)/M$ with endpoints $x_\ell = a + \ell h$ ($\ell = 0 : M$). Then, the **composite trapezoidal formula** is

$$I_t^c := h \left[\frac{f(a) + f(b)}{2} + \sum_{k=1}^{M-1} f(x_k) \right].$$

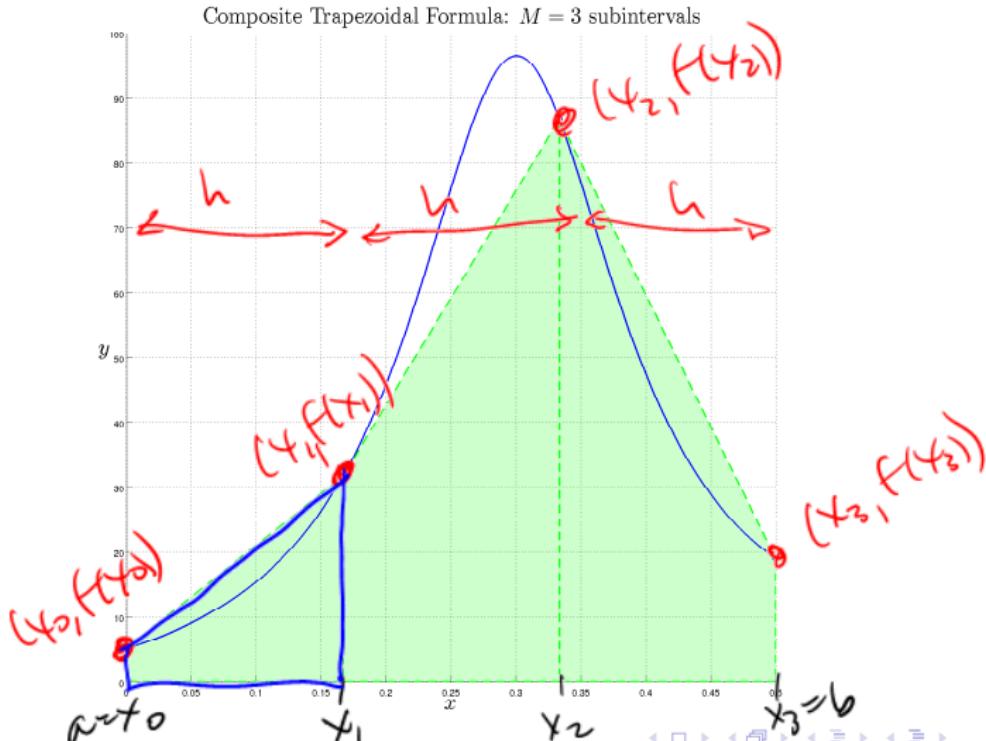
$$\begin{aligned} \text{Observe } I &= \sum_{k=1}^M \int_{x_{k-1}}^{x_k} f(x) dx \simeq \sum_{k=1}^M I_t(f; x_{k-1}, x_k) \\ &= \sum_{k=1}^M \frac{h}{2} [f(x_{k-1}) + f(x_k)] \end{aligned}$$

Composite trapezoidal formula

Let $[a, b]$ be partitioned into M equal subintervals of width $h = (b - a)/M$ with endpoints $x_\ell = a + \ell h$ ($\ell = 0 : M$). Then, the **composite trapezoidal formula** is

$$I_t^c := h \left[\frac{f(a) + f(b)}{2} + \sum_{k=1}^{M-1} f(x_k) \right].$$

$$\begin{aligned} \text{Observe } I &= \sum_{k=1}^M \int_{x_{k-1}}^{x_k} f(x) dx \simeq \sum_{k=1}^M I_t(f; x_{k-1}, x_k) \\ &= \sum_{k=1}^M \frac{h}{2} [f(x_{k-1}) + f(x_k)] = h \left[\frac{f(a) + f(b)}{2} + \sum_{k=1}^{M-1} f(x_k) \right] \end{aligned}$$



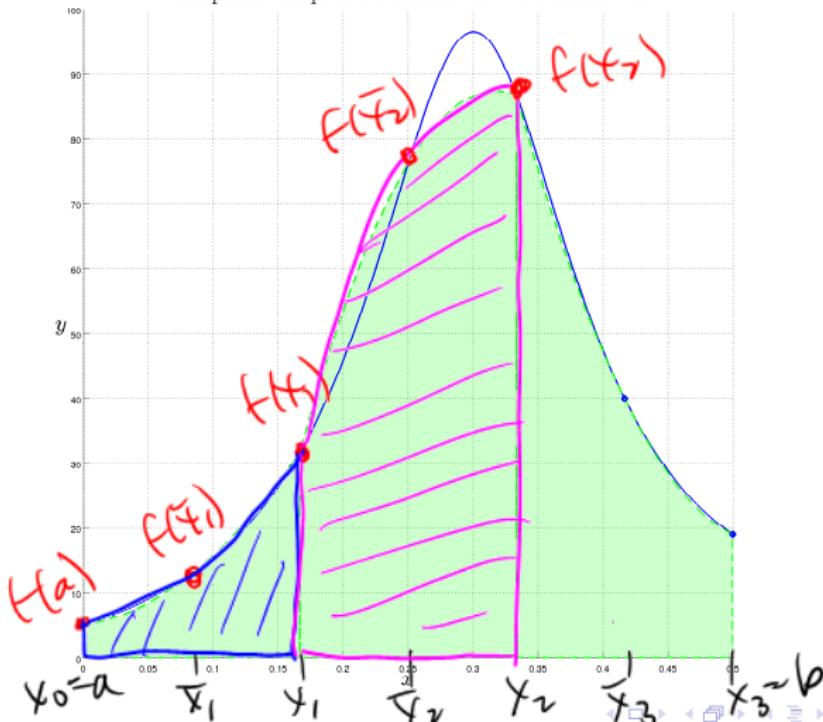
Composite Simpson formula

Let $[a, b]$ be partitioned into M equal subintervals of width $h = (b - a)/M$ with endpoints $x_\ell = a + \ell h$ ($\ell = 0 : M$). Define the midpoints \bar{x}_k of each subinterval by

$$\bar{x}_k := \frac{x_{k-1} + x_k}{2} \quad (k = 1 : M).$$

Then, the composite Simpson formula is

$$I_s^c := \frac{h}{6} \sum_{k=1}^M [f(x_{k-1}) + 4f(\bar{x}_k) + f(x_k)].$$

Composite Simpson's Formula: $M = 3$ subintervals

Caution!

- ▶ Different notations & terminology in books & software:
 - ▶ “panels” vs. “subintervals”;
 - ▶ indexing from 0 vs. indexing from 1;
 - ▶ width of “ h ” (panel or subinterval);

Caution!

- ▶ Different notations & terminology in books & software:
 - ▶ “panels” vs. “subintervals”;
 - ▶ indexing from 0 vs. indexing from 1;
 - ▶ width of “ h ” (panel or subinterval);
- ▶ Do not memorise or blindly apply formulas. **Think!**
- ▶ When in doubt, **check details carefully.**
- ▶ Construct examples you can verify by hand.
- ▶ Know syntax and interpretation of **your** quadrature software.

Error of midpoint formula

$$= \boxed{I - I_{mp} = \frac{(b-a)^3}{24} f''(\xi)} \quad \text{if } f \in C^2[a, b]$$

Error of trapezoidal formula

$$= \boxed{I - I_t = -\frac{(b-a)^3}{12} f''(\xi)} \quad \text{if } f \in C^2[a, b]$$

Error of Simpson formula

$$= \boxed{I - I_s = -\frac{(b-a)^5}{2880} f^{(4)}(\xi)} \quad \text{if } f \in C^4[a, b]$$

Recall $h := (b - a)/M = \text{width of subintervals}$

$\Theta(h^2)$

Error of composite midpoint formula

$$= I - I_{mp}^c = \frac{(b-a)}{24} h^2 f''(\xi) \quad \text{if } f \in C^2[a, b]$$

Error of composite trapezoidal formula

$$= I - I_t^c = -\frac{(b-a)}{12} h^2 f''(\xi) \quad \text{if } f \in C^2[a, b]$$

$\Theta(h^4)$

Error of composite Simpson formula

$$= I - I_s^c = -\frac{(b-a)}{2880} h^4 f^{(4)}(\xi) \quad \text{if } f \in C^4[a, b]$$

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. Differential Equations

2. Boundary value problems

3. Numerical solution

Today's questions:

- ▶ What is a differential equation?
- ▶ What is a boundary value problem?
- ▶ How do we approximate solutions to boundary value problems?

Main points so far:

- ▶ Finite difference formulas can be derived from polynomial interpolation:
 1. Sample the function f on grid points x_{k-j}, \dots, x_{k+j} .
 2. Compute the polynomial interpolant $P_{2j}(x)$.
 3. Approximate $f'(x_k)$ by $P'_{2j}(x_k)$.
- ▶ The error for polynomial interpolation gives us the error of numerical differentiation, usually as an order estimate, $O(h^q)$, where $h = \max_k |x_{k+1} - x_k|$.

A Differential Equation (DE) has derivatives of unknown function.

Example: Population growth with unlimited resources, where u is the population at time t



$$u'(t) = \beta u(t), \quad t \in (0, \infty)$$

A Differential Equation (DE) has derivatives of unknown function.

Example: Population growth with unlimited resources, where u is the population at time t

rate of change

$$u'(t) = \beta u(t), \quad t \in (0, \infty)$$

- ▶ The rate of change of the population (i.e $u'(t)$) at any given time is proportional to the population $u(t)$ at that time,

A Differential Equation (DE) has derivatives of unknown function.

Example: Population growth with unlimited resources, where u is the population at time t

$$u'(t) = \beta u(t), \quad t \in (0, \infty)$$

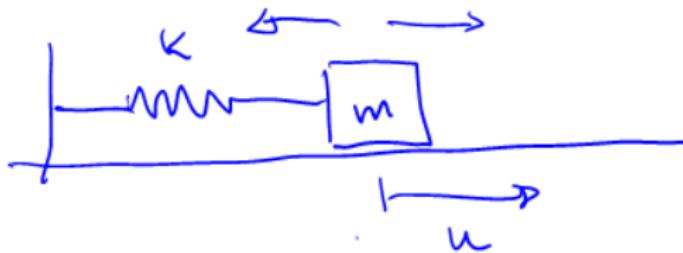
- ▶ The rate of change of the population (i.e $u'(t)$) at any given time is proportional to the population $u(t)$ at that time,
- ▶ β is the constant of proportionality, and is referred to as the birth rate

A Differential Equation (DE) has derivatives of unknown function.

Example: Newton's 2nd law for mass on a spring:

$$m u''(t) + k u(t) = 0, \quad t \in (0, \pi/2)$$

where $u(t)$ is the displacement of the object at time t , m is the mass of the object and k is the spring constant



A Differential Equation (DE) has derivatives of unknown function.

Example: Newton's 2nd law for mass on a spring:

$$m u''(t) + k u(t) = 0, \quad t \in (0, \pi/2)$$

where $u(t)$ is the displacement of the object at time t , m is the mass of the object and k is the spring constant

- ▶ Newton's 2nd law (force = mass times acceleration):
 $F = ma$

A Differential Equation (DE) has derivatives of unknown function.

Example: Newton's 2nd law for mass on a spring:

$$m u''(t) + k u(t) = 0, \quad t \in (0, \pi/2)$$

where $u(t)$ is the displacement of the object at time t , m is the mass of the object and k is the spring constant

- ▶ Newton's 2nd law (force = mass times acceleration):
 $F = ma$
- ▶ acceleration is the 2nd derivative of displacement (position): $a = u''(t)$

A Differential Equation (DE) has derivatives of unknown function.

Example: Newton's 2nd law for mass on a spring:

$$m u''(t) + k u(t) = 0, \quad t \in (0, \pi/2)$$

where $u(t)$ is the displacement of the object at time t , m is the mass of the object and k is the spring constant

- ▶ Newton's 2nd law (force = mass times acceleration):

$$F = ma$$

- ▶ acceleration is the 2nd derivative of displacement (position): $a = u''(t)$

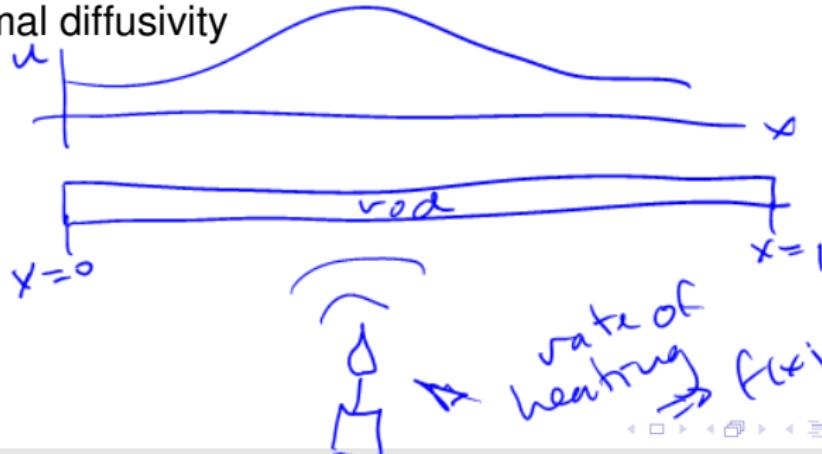
- ▶ Force F on object is given by Hooke's law: $F = -k u(t)$

A Differential Equation (DE) has derivatives of unknown function.

Example: Poisson Equation: steady state distribution of the temperature in a rod, being heated at a rate $f(x)$:

→ $\alpha u''(x) = -f(x), \quad x \in (0, 1)$

where $u(x)$ is the temperature of the rod at location x , and α is the thermal diffusivity



A Differential Equation (DE) has derivatives of unknown function.

Example:



$$u'(x) - u(x) = 0, \quad x \in (0, \infty)$$

A *solution* is a **function** that satisfies the equation:

solution : $u(x) = e^x$ (satisfies equation)

so does : $u(x) = 0$

and $u(x) = 2e^x$

and $u(x) = ce^x$ where c is a constant

A Differential Equation (DE) has derivatives of unknown function.

Example:

$$u''(x) + u(x) = 0, \quad x \in (0, \pi/2)$$

A solution:

$$u(x) = \cos x \quad \text{satisfies equation}$$

so does

$$u(x) = \sin x$$

also $u(x) = A\sin x + B\cos x$

where A and B are any constant

Remarks:

- ▶ If we want the solution of a DE to be unique, we need to specify extra condition(s)
 - ▶ one extra condition is needed for each derivative of $u(x)$

Remarks:

- ▶ If we want the solution of a DE to be unique, we need to specify extra condition(s)
 - ▶ one extra condition is needed for each derivative of $u(x)$
- ▶ These conditions are generally set by specifying the value of $u(x)$ and/or its derivatives at the boundaries of the domain (i.e at the ends of the interval over which we want to find $u(x)$)

Remarks:

- ▶ If we want the solution of a DE to be unique, we need to specify extra condition(s)
 - ▶ one extra condition is needed for each derivative of $u(x)$
- ▶ These conditions are generally set by specifying the value of $u(x)$ and/or its derivatives at the boundaries of the domain (i.e at the ends of the interval over which we want to find $u(x)$)
- ▶ **Example:**

$$u'(x) - u(x) = 0, \quad x \in (0, \infty)$$

extra condition : require $u(0) = 2$

$$u(x) = Ce^x$$

$$\Rightarrow u(0) = Ce^0 = C = 2 \Rightarrow u(x) = 2e^x$$

satisfies DE and $u(0) = 2$

Example:

$$u''(x) + u(x) = 0, \quad x \in (0, \pi/2)$$

$u(x) = A\sin x + B\cos x$ satisfies equation for any A, B

need 2 extra conditions

e.g. $\begin{matrix} u(0) = 1 \\ u'(0) = 0 \end{matrix}$ $\xrightarrow{\text{IVP}}$

$$\Rightarrow u(x) = \cos x \quad (A=0, B=1)$$

satisfies equation and both conditions

e.g. $\begin{matrix} u(0) = 0 \\ u(\pi/2) = 1 \end{matrix}$ $\xrightarrow{\text{BVP}}$

$$\Rightarrow u = \sin x \quad (A=0, B=1)$$

satisfies eqn and both conditions

- If we have a differential equation with all conditions specified at the left boundary, e.g. $u(0) = 0$, $u'(0) = -1$, we call the associated problem:



Initial Value Problem (IVP)

- ▶ If we have a differential equation with all conditions specified at the left boundary, e.g. $u(0) = 0$, $u'(0) = -1$, we call the associated problem:

Initial Value Problem (IVP)

- ▶ **Example:**

$$u''(t) + u(t) = 0, \quad u(0) = 2, \quad u'(0) = 0$$

- ▶ If we have a differential equation with all conditions specified at the left boundary, e.g. $u(0) = 0$, $u'(0) = -1$, we call the associated problem:

Initial Value Problem (IVP)

- ▶ **Example:**

$$u''(t) + u(t) = 0, \quad u(0) = 2, \quad u'(0) = 0$$

- ▶ If we have a differential equation with conditions specified at both boundaries, e.g. $u(0) = 0$, $u(1) = -1$, we call the associated problem:



Boundary Value Problem (BVP)

- ▶ If we have a differential equation with all conditions specified at the left boundary, e.g. $u(0) = 0$, $u'(0) = -1$, we call the associated problem:

Initial Value Problem (IVP)

- ▶ **Example:**

$$u''(t) + u(t) = 0, \quad u(0) = 2, \quad u'(0) = 0$$

- ▶ If we have a differential equation with conditions specified at both boundaries, e.g. $u(0) = 0$, $u(1) = -1$, we call the associated problem:

Boundary Value Problem (BVP)

- ▶ **Example:**

$$u''(x) + u(x) = 0, \quad u(0) = 0, \quad u(1) = 1$$

Example: Boundary Value Problem: Poisson equation (in 1d)

$$\begin{cases} -u''(x) = f(x), & x \in (0, 1) \\ u(0) = \alpha, & (\text{Dirichlet boundary condition at } x = 0) \\ u(1) = \beta & (\text{Dirichlet boundary condition at } x = 1) \end{cases}$$



- ▶ Example: $-u''(x) = \sin(\pi x)$, $x \in (0, 1)$, $u(0) = 0$, $u(1) = 0$
- ▶ That is, we are looking for a function $u(x)$ that satisfies the equation and some conditions on the boundary
- ▶ where we are given the problem data: RHS function $f(x)$ & values α, β .
- ▶ Analytic solution generally not available.

Look for solution on a grid (or mesh):

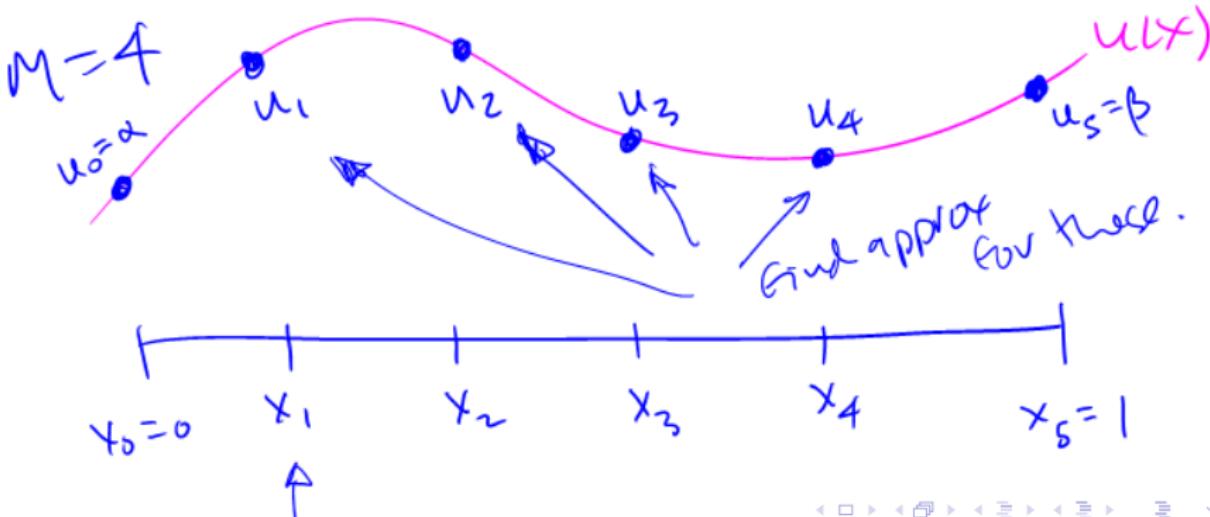
- Given interval $[0, 1]$, choose integer $N > 2$ and set

$$u_i = u(x_i)$$

$$h := \frac{1}{N+1}, \quad x_j = jh \quad (j = 0 : N+1)$$

unknown
func.
↓

- Define $\{u_j\}_{j=1}^N$ such that $u_j \equiv u(x_j)$ ($j = 0, \dots, N+1$).



Look for solution on a grid (or mesh):

- ▶ Given interval $[0, 1]$, choose integer $N > 2$ and set

$$h := \frac{1}{N+1}, \quad x_j = jh \quad (j = 0 : N+1)$$

- ▶ Define $\{u_j\}_{j=1}^N$ such that $u_j \equiv u(x_j)$ ($j = 0, \dots, N+1$).
- ▶ Unknowns to determine are N values u_1, u_2, \dots, u_N .

$$u_i = u(x_i)$$

Look for solution on a grid (or mesh):

- ▶ Given interval $[0, 1]$, choose integer $N > 2$ and set

$$h := \frac{1}{N+1}, \quad x_j = jh \quad (j = 0 : N+1)$$

- ▶ Define $\{u_j\}_{j=1}^N$ such that $u_j \equiv u(x_j)$ ($j = 0, \dots, N+1$).
- ▶ Unknowns to determine are N values u_1, u_2, \dots, u_N .
- ▶ Use boundary data to set $u_0 = \alpha$ and $u_{N+1} = \beta$.

Look for solution on a grid (or mesh):

- ▶ Given interval $[0, 1]$, choose integer $N > 2$ and set

$$h := \frac{1}{N+1}, \quad x_j = jh \quad (j = 0 : N+1)$$

- ▶ Define $\{u_j\}_{j=1}^N$ such that $u_j \equiv u(x_j)$ ($j = 0, \dots, N+1$).
- ▶ Unknowns to determine are N values u_1, u_2, \dots, u_N .
- ▶ Use boundary data to set $u_0 = \alpha$ and $u_{N+1} = \beta$.
- ▶ Determine the u_j by requiring the differential equation to be satisfied at the grid points x_j

Look for solution on a grid (or mesh):

- ▶ Given interval $[0, 1]$, choose integer $N > 2$ and set

$$h := \frac{1}{N+1}, \quad x_j = jh \quad (j = 0 : N+1)$$

- ▶ Define $\{u_j\}_{j=1}^N$ such that $u_j \equiv u(x_j)$ ($j = 0, \dots, N+1$).
- ▶ Unknowns to determine are N values u_1, u_2, \dots, u_N .
- ▶ Use boundary data to set $u_0 = \alpha$ and $u_{N+1} = \beta$.
- ▶ Determine the u_j by requiring the differential equation to be satisfied at the grid points x_j
- ▶ Replace the derivative u'' at x_j by a finite-difference approximation, for instance $(u_{j-1} - 2u_j + u_{j+1})/2h$.

Look for solution on a grid (or mesh):

- ▶ Given interval $[0, 1]$, choose integer $N > 2$ and set

$$h := \frac{1}{N+1}, \quad x_j = jh \quad (j = 0 : N+1)$$

- ▶ Define $\{u_j\}_{j=1}^N$ such that $u_j \equiv u(x_j)$ ($j = 0, \dots, N+1$).
- ▶ Unknowns to determine are N values u_1, u_2, \dots, u_N .
- ▶ Use boundary data to set $u_0 = \alpha$ and $u_{N+1} = \beta$.
- ▶ Determine the u_j by requiring the differential equation to be satisfied at the grid points x_j
- ▶ Replace the derivative u'' at x_j by a finite-difference approximation, for instance $(u_{j-1} - 2u_j + u_{j+1})/2h$.
- ▶ Numerical solution consists of mesh $\{x_j\}_{j=0}^{N+1}$ and values $\{u_j\}_{j=0}^{N+1}$.

Numerical approximation for 1d Poisson problem:

- ▶ Find approximate solution that satisfies equation at each interior mesh point x_j ($j = 1 : N$) and at the boundary.

At each x_j , we have

$$\rightarrow -u''(x_j) = f(x_j)$$

for all
 $j = 1 \text{ to } M$

- ▶ And so, we have the $N + 2$ equations:

$$j = 0 :$$

$$u(x_0) = \alpha$$

$$j = 1 :$$

$$-u''(x_1) = f(x_1)$$

4

$$j = 2 :$$

$$-u''(x_2) = f(x_2)$$

$$j = 3 :$$

$$-u''(x_3) = f(x_3)$$

⋮

⋮

$$j = N - 1 :$$

$$-u''(x_{N-1}) = f(x_{N-1})$$

$$j = N :$$

$$-u''(x_N) = f(x_N)$$

$$j = N + 1 :$$

$$u(x_{N+1}) = \beta$$

- Writing $u_j = u(x_j)$ and using finite-difference to approximate $u''(x)$, we have

$$-u''(x_j) = f(x_j) \rightarrow \frac{-u_{j-1} + 2u_j - u_{j+1}}{h^2} = f_j \equiv f(x_j)$$

- Multiplying by h^2 , we obtain, the $N + 2$ equations become

$$j = 0 :$$

finite approx of u''

$$u_0 = \alpha$$

$$\bullet u_0$$

$$\bullet u_1$$

$$\bullet u_2$$

$$j = 1 :$$

$$-u_0 + 2u_1 - u_2 = h^2 f_1$$

$$\bullet$$

$$\bullet$$

$$\bullet$$

$$j = 2 :$$

$$-u_1 + 2u_2 - u_3 = h^2 f_2$$

$$\bullet$$

$$\bullet$$

$$\bullet$$

$$j = 3 :$$

$$-u_2 + 2u_3 - u_4 = h^2 f_3$$

$$\bullet$$

$$\bullet$$

$$\bullet$$

$$\vdots$$

$$\vdots$$

$$j = N - 1 : \quad -u_{N-2} + 2u_{N-1} - u_N = h^2 f_{N-1}$$

$$j = N : \quad -u_{N-1} + 2u_N - u_{N+1} = h^2 f_N$$

$$j = N + 1 : \quad u_{N+1} = \beta$$

- Eliminating u_0 and u_{N+1} , we obtain N equations in the N unknowns $u_j, j = 1 : N$

$$j = 1 : \quad 2u_1 - u_2 = h^2 f_1 + \alpha$$

$$j = 2 : \quad -u_1 + 2u_2 - u_3 = h^2 f_2$$

$$j = 3 : \quad -u_2 + 2u_3 - u_4 = h^2 f_3$$

 \vdots \vdots

$$j = N - 1 : \quad -u_{N-2} + 2u_{N-1} - u_N = h^2 f_{N-1}$$

$$j = N : \quad -u_{N-1} + 2u_N = h^2 f_N + \beta$$

Reduce to a linear algebra problem:

- Define vectors $\vec{u}_h := \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{N-1} \\ u_N \end{bmatrix}$ and $\vec{f} := \begin{bmatrix} f_1 + \alpha/h^2 \\ f_2 \\ \vdots \\ f_{N-1} \\ f_N + \beta/h^2 \end{bmatrix}$
- Define tridiagonal matrix

$$A := \text{tridiag}(-1, 2, -1) = \begin{bmatrix} 2 & -1 & & & & \\ -1 & 2 & \ddots & & & \\ & \ddots & \ddots & \ddots & & \\ & & 0 & & & \\ & & & -1 & 2 & \\ & & & & & \end{bmatrix} \in \mathbb{R}^{N \times N}$$

∴ Finding unknown values $\{u_j\}_{j=1}^N$ means solving

$$A\vec{u}_h = h^2\vec{f}$$

Example computation: let

$$f(x) = xe^{-2x}, \quad u(0) = \alpha = -\frac{1}{4}, \quad u(1) = \beta = 0$$

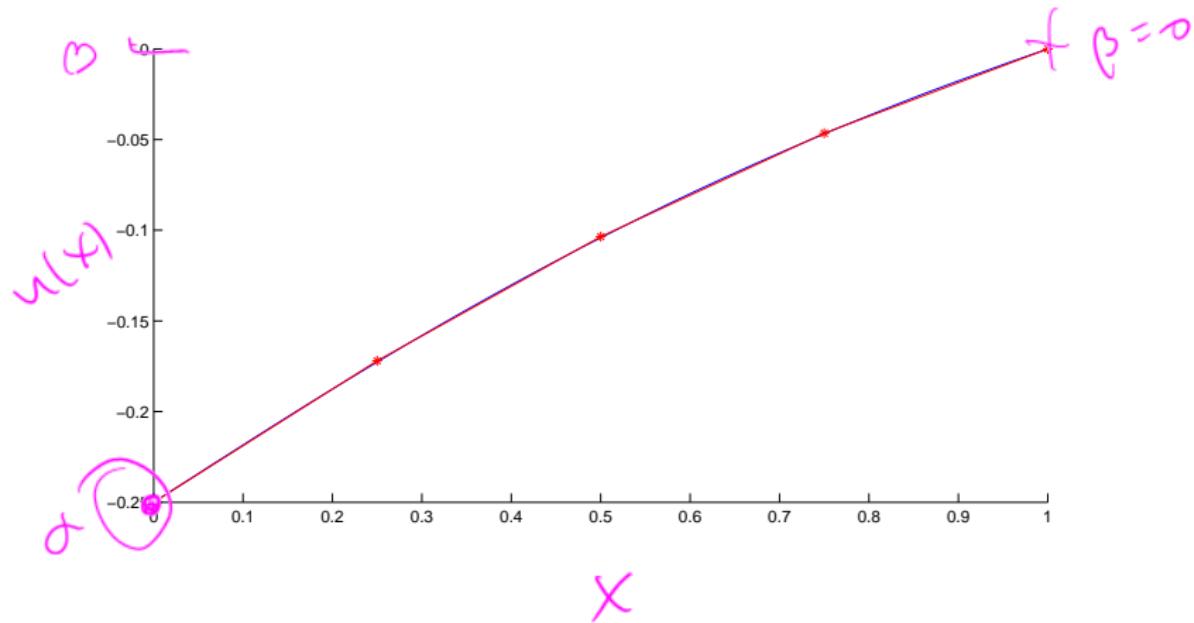
then the exact solution is given by

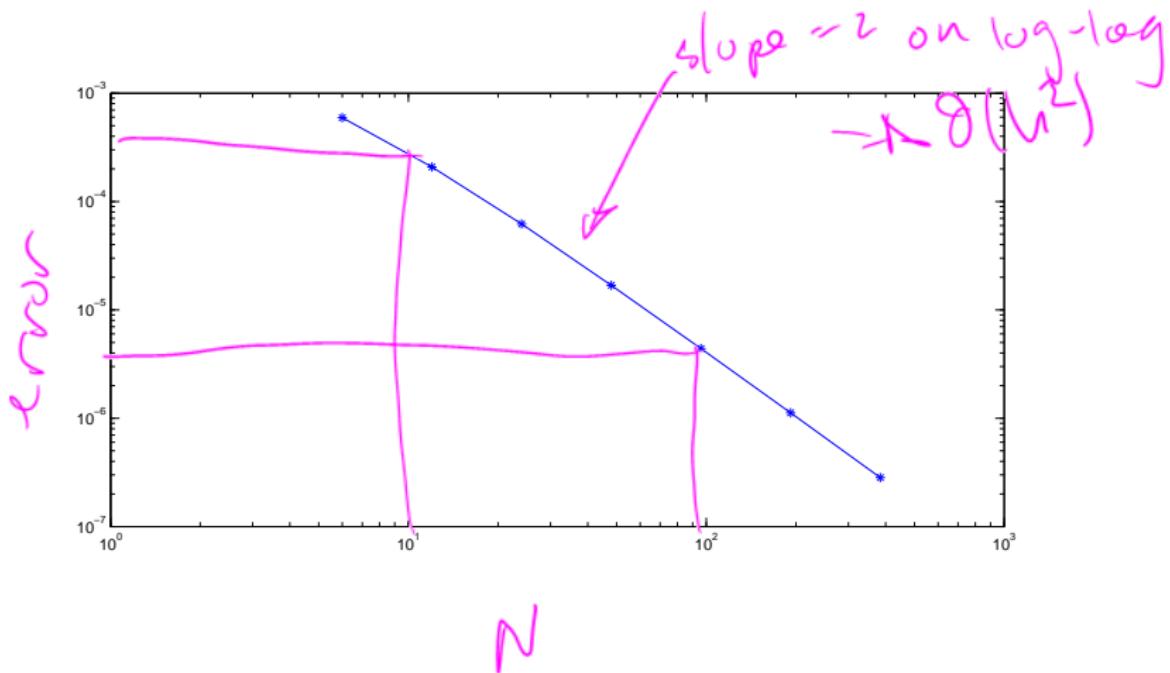
$$u(x) = -\frac{1}{4}e^{-2x} - \frac{1}{4}xe^{-2x} + \frac{1}{2}e^{-2x}$$

Verify:

Python code:

```
N = 4; h = 1.0/(1.0+N)
xs = np.linspace(0,1,N+2)
def f(x):
    return x*np.exp(-2.0*x)
alpha = -1.0/4.0; beta = 0.0
def U(x):
    return -np.exp(-2.0*x)/4.0 -
           x * np.exp(-2.0*x)/4.0 + np.exp(-2.0)*x/2.0
A = scipy.sparse.diags([[1]*(N-1), [-2]*N, [1]*(N-1)],
                       [-1, 0, 1])
b = h**2 * f(xs[1:N+1])
b[0] = b[0] + alpha; b[N-1] = b[N-1] + beta
u_i = scipy.sparse.linalg.spsolve(-A, b)
u = scipy.append([alpha], u_i);
u = scipy.append(u, [beta])
```





Remarks:

- ▶ Numerical solution of boundary value problems:
 - ▶ Identify the problem carefully (problem data, geometry, etc.)
 - ▶ Discretise it carefully (e.g., by finite-differences).
 - ▶ Identify and solve discrete equations as a linear system.

Remarks:

- ▶ Numerical solution of boundary value problems:
 - ▶ Identify the problem carefully (problem data, geometry, etc.)
 - ▶ Discretise it carefully (e.g., by finite-differences).
 - ▶ Identify and solve discrete equations as a linear system.
- ▶ Finite-difference methods extend to partial derivatives, PDEs.

Remarks:

- ▶ Numerical solution of boundary value problems:
 - ▶ Identify the problem carefully (problem data, geometry, etc.)
 - ▶ Discretise it carefully (e.g., by finite-differences).
 - ▶ Identify and solve discrete equations as a linear system.
- ▶ Finite-difference methods extend to partial derivatives, PDEs.
- ▶ Linear DE \Rightarrow linear algebraic equations for \vec{u}_h .
- ▶ Nonlinear DE \Rightarrow nonlinear equations for \vec{u}_h .

Remarks:

- ▶ Numerical solution of boundary value problems:
 - ▶ Identify the problem carefully (problem data, geometry, etc.)
 - ▶ Discretise it carefully (e.g., by finite-differences).
 - ▶ Identify and solve discrete equations as a linear system.
- ▶ Finite-difference methods extend to partial derivatives, PDEs.
- ▶ Linear DE \Rightarrow linear algebraic equations for \vec{u}_h .
- ▶ Nonlinear DE \Rightarrow nonlinear equations for \vec{u}_h .
- ▶ Accuracy of finite-difference formulas relates to accuracy of computed solutions.
- ▶ Finer grids/meshes \Rightarrow more accurate solutions, but larger matrices (although the matrices are sparse!).

2072U Computational Science I

Winter 2023

Week	Topic
1	Introduction
1–2	Solving nonlinear equations in one variable
3–4	Solving systems of (non)linear equations
5–6	Computational complexity
6–8	Interpolation and least squares
8–10	Integration & differentiation
10–12	Additional Topics

1. What is Data Science?
2. What is Machine Learning?
3. Principal Component Analysis (PCA)
4. Singular Value Decompositions (SVDs)
Aside: Low rank approximations

Data Science: Some reasonable definitions

- ▶ Data science is the study of the generalizable extraction of knowledge from data [Dhar, ACM communications, 56, 12].
- ▶ Data science is a broad field of study pertaining to data systems and processes, aimed at maintaining data sets and deriving meaning out of them. [mygreatlearning.com]

Examples:

Data Science: Some reasonable definitions

- ▶ Data science is the study of the generalizable extraction of knowledge from data [Dhar, ACM communications, 56, 12].
- ▶ Data science is a broad field of study pertaining to data systems and processes, aimed at maintaining data sets and deriving meaning out of them. [mygreatlearning.com]

Examples:

- ▶ lesion detection

Data Science: Some reasonable definitions

- ▶ Data science is the study of the generalizable extraction of knowledge from data [Dhar, ACM communications, 56, 12].
- ▶ Data science is a broad field of study pertaining to data systems and processes, aimed at maintaining data sets and deriving meaning out of them. [mygreatlearning.com]

Examples:

- ▶ lesion detection
- ▶ climate analysis

Data Science: Some reasonable definitions

- ▶ Data science is the study of the generalizable extraction of knowledge from data [Dhar, ACM communications, 56, 12].
- ▶ Data science is a broad field of study pertaining to data systems and processes, aimed at maintaining data sets and deriving meaning out of them. [mygreatlearning.com]

Examples:

- ▶ lesion detection
- ▶ climate analysis
- ▶ churn rate

Data Science: Some reasonable definitions

- ▶ Data science is the study of the generalizable extraction of knowledge from data [Dhar, ACM communications, 56, 12].
- ▶ Data science is a broad field of study pertaining to data systems and processes, aimed at maintaining data sets and deriving meaning out of them. [mygreatlearning.com]

Examples:

- ▶ lesion detection
- ▶ climate analysis
- ▶ churn rate
- ▶ housing price prediction

Data Science: Some reasonable definitions

- ▶ Data science is the study of the generalizable extraction of knowledge from data [Dhar, ACM communications, 56, 12].
- ▶ Data science is a broad field of study pertaining to data systems and processes, aimed at maintaining data sets and deriving meaning out of them. [mygreatlearning.com]

Examples:

- ▶ lesion detection
- ▶ climate analysis
- ▶ churn rate
- ▶ housing price prediction
- ▶ sentiment analysis

Lesion detection

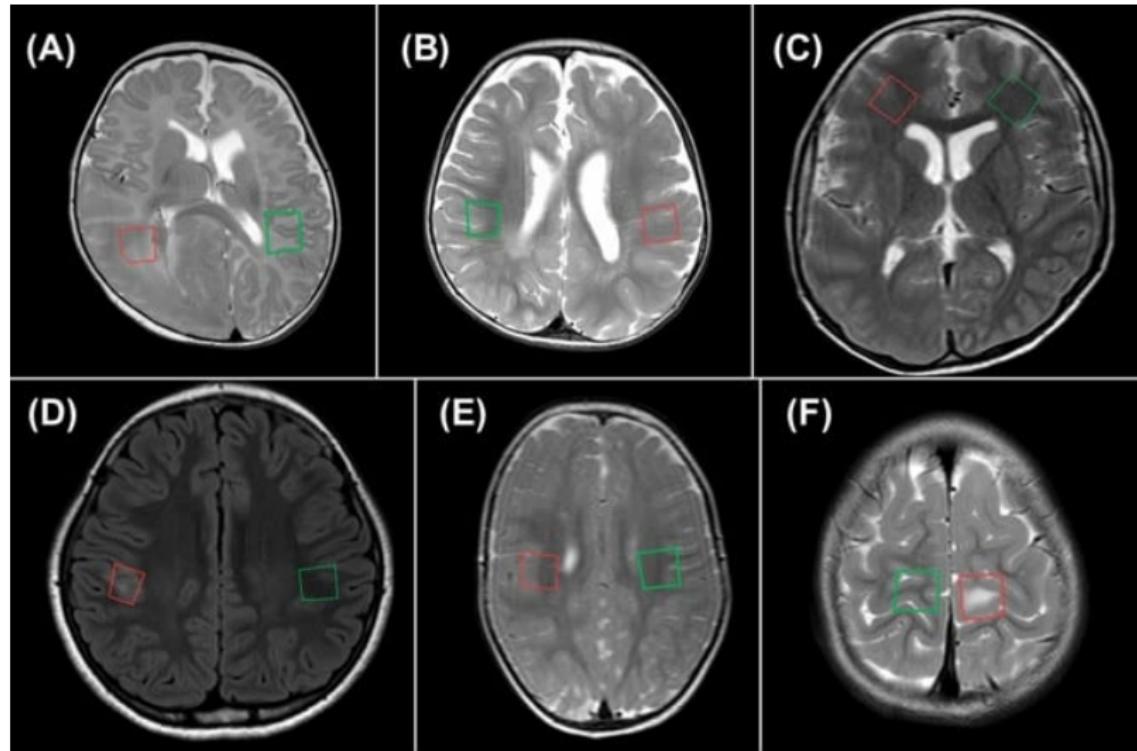


Figure: [analyticsindiamag.com/the-new-nih-dataset-and-ai-may-[▶](#)



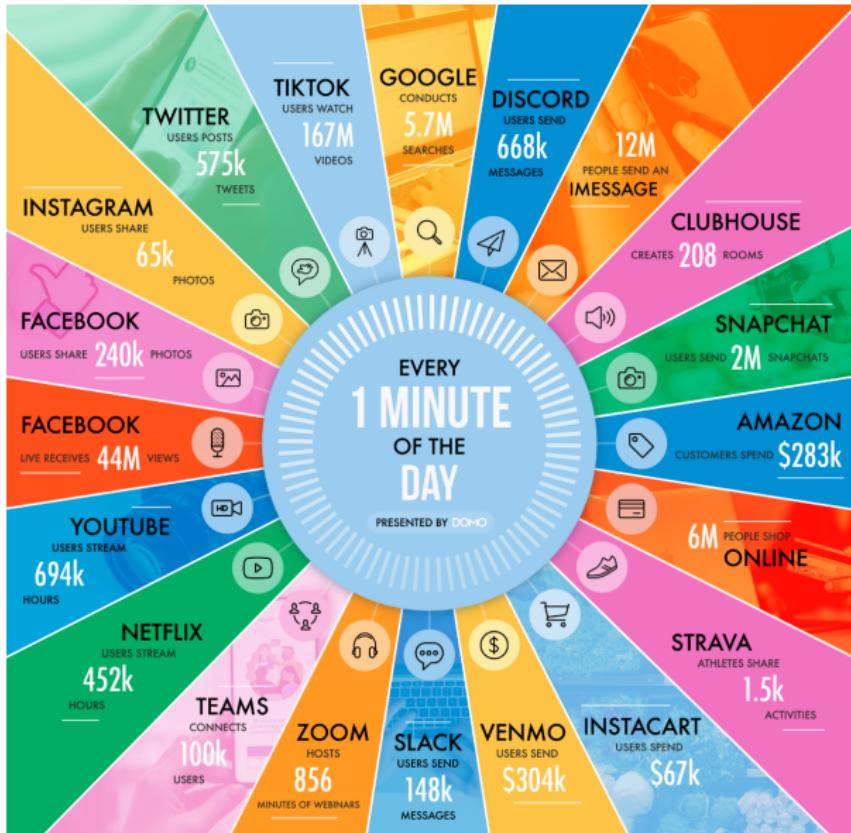
Data Science: Some reasonable definitions

- ▶ Data science is an interdisciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from noisy, structured and unstructured data, and to apply knowledge and actionable insights from data across a broad range of application domains. Data science is related to data mining, machine learning and big data. [Wikipedia]

Data Science: Some reasonable definitions

- ▶ Data science is an interdisciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from noisy, structured and unstructured data, and to apply knowledge and actionable insights from data across a broad range of application domains. Data science is related to data mining, machine learning and big data. [Wikipedia]
- ▶ Data science is the practice of mining large data sets of raw data, both structured and unstructured, to identify patterns and extract actionable insight from them. This is an interdisciplinary field, and the foundations of data science include statistics, inference, computer science, predictive analytics, machine learning algorithm development, and new technologies to gain insights from big data. [omnisci.com]

Data Science: It's about the data



Data Science: It's about the data

- ▶ it's not just internet / it's not just business
- ▶ storage capacities allow for accumulation of all kinds of data, such as:

Data Science: It's about the data

- ▶ it's not just internet / it's not just business
- ▶ storage capacities allow for accumulation of all kinds of data, such as:
 - ▶ medical data
 - ▶ diagnostic images
 - ▶ time series of physiological variables

Data Science: It's about the data

- ▶ it's not just internet / it's not just business
- ▶ storage capacities allow for accumulation of all kinds of data, such as:
 - ▶ medical data
 - ▶ diagnostic images
 - ▶ time series of physiological variables
 - ▶ environmental data
 - ▶ satellite images
 - ▶ time series of wind, temperature, humidity, ...

Data Science: It's about the data

- ▶ it's not just internet / it's not just business
- ▶ storage capacities allow for accumulation of all kinds of data, such as:
 - ▶ medical data
 - ▶ diagnostic images
 - ▶ time series of physiological variables
 - ▶ environmental data
 - ▶ satellite images
 - ▶ time series of wind, temperature, humidity, ...
 - ▶ and many other areas

Data Science: It's about the data

- ▶ Sooooo much data...
 - ▶ text,
 - ▶ video,
 - ▶ images,
 - ▶ user conversations,
 - ▶ click-streams,
 - ▶ social media platforms,
 - ▶ the internet of things (IOT devices),
 - ▶ legacy databases,
 - ▶ data sourced from data providers

Data Science: It's about the data

- ▶ Sooooo much data...
 - ▶ text,
 - ▶ video,
 - ▶ images,
 - ▶ user conversations,
 - ▶ click-streams,
 - ▶ social media platforms,
 - ▶ the internet of things (IOT devices),
 - ▶ legacy databases,
 - ▶ data sourced from data providers
- ▶ Structured vs. unstructured data
 - ▶ structured: e.g. spreadsheets
 - ▶ unstructured: can't put in a spreadsheet, e.g. text, videos, images

So much data → Big Data

- ▶ Big data is a field that treats ways to capture, manage and efficiently process, data sets that are too large or complex to be dealt with by traditional data-processing application software

So much data → Big Data

- ▶ Big data is a field that treats ways to capture, manage and efficiently process, data sets that are too large or complex to be dealt with by traditional data-processing application software
- ▶ Characteristics of big data include high volume, high velocity and high variety

So much data → Big Data

- ▶ Big data is a field that treats ways to capture, manage and efficiently process, data sets that are too large or complex to be dealt with by traditional data-processing application software
- ▶ Characteristics of big data include high volume, high velocity and high variety
- ▶ use tools such as Apache Hadoop or Apache Spark for data processing and storage of very large volume of data

So much data → Big Data

- ▶ Big data is a field that treats ways to capture, manage and efficiently process, data sets that are too large or complex to be dealt with by traditional data-processing application software
- ▶ Characteristics of big data include high volume, high velocity and high variety
- ▶ use tools such as Apache Hadoop or Apache Spark for data processing and storage of very large volume of data
- ▶ enables data to be stored and processed across a network of computers rather than in a big chunk on one machine

What do you do with all this data?

- ▶ first you have to put it into a usable form: ‘data wrangling’
 - ▶ collect
 - ▶ change formatting
 - ▶ validate
 - ▶ ‘clean’

What do you do with all this data?

- ▶ then you ‘mine’ it → Data Mining

What do you do with all this data?

- ▶ then you ‘mine’ it → Data Mining
- ▶ search for meaning in the vast amount of data
(extraction of knowledge or ‘actionable insight’ from data)
- ▶ look for patterns and trends in the data and make predictions

What do you do with all this data?

- ▶ then you ‘mine’ it → Data Mining
- ▶ search for meaning in the vast amount of data
(extraction of knowledge or ‘actionable insight’ from data)
- ▶ look for patterns and trends in the data and make predictions
 - ▶ profiling customer behaviors, needs and disposable income in order to offer targeted ads,

What do you do with all this data?

- ▶ then you ‘mine’ it → Data Mining
- ▶ search for meaning in the vast amount of data (extraction of knowledge or ‘actionable insight’ from data)
- ▶ look for patterns and trends in the data and make predictions
 - ▶ profiling customer behaviors, needs and disposable income in order to offer targeted ads,
 - ▶ in financial institutions, tracking customer transactions for unusual behaviors, and flagging fraudulent transactions,

What do you do with all this data?

- ▶ then you ‘mine’ it → Data Mining
- ▶ search for meaning in the vast amount of data (extraction of knowledge or ‘actionable insight’ from data)
- ▶ look for patterns and trends in the data and make predictions
 - ▶ profiling customer behaviors, needs and disposable income in order to offer targeted ads,
 - ▶ in financial institutions, tracking customer transactions for unusual behaviors, and flagging fraudulent transactions,
 - ▶ predicting a patient’s likelihood for specific health conditions and prioritizing treatment,

What do you do with all this data?

- ▶ then you ‘mine’ it → Data Mining
- ▶ search for meaning in the vast amount of data (extraction of knowledge or ‘actionable insight’ from data)
- ▶ look for patterns and trends in the data and make predictions
 - ▶ profiling customer behaviors, needs and disposable income in order to offer targeted ads,
 - ▶ in financial institutions, tracking customer transactions for unusual behaviors, and flagging fraudulent transactions,
 - ▶ predicting a patient’s likelihood for specific health conditions and prioritizing treatment,
 - ▶ data assimilation

What do you do with all this data?

- ▶ then you ‘mine’ it → Data Mining
- ▶ search for meaning in the vast amount of data (extraction of knowledge or ‘actionable insight’ from data)
- ▶ look for patterns and trends in the data and make predictions
 - ▶ profiling customer behaviors, needs and disposable income in order to offer targeted ads,
 - ▶ in financial institutions, tracking customer transactions for unusual behaviors, and flagging fraudulent transactions,
 - ▶ predicting a patient’s likelihood for specific health conditions and prioritizing treatment,
 - ▶ data assimilation
 - ▶ helping investigation agencies deploy police force where the likelihood of crime is higher.

How do you mine data?

- ▶ standard statistics tools and data visualization, e.g. scatter plots, joint plots, histograms, etc.

How do you mine data?

- ▶ standard statistics tools and data visualization, e.g. scatter plots, joint plots, histograms, etc.

- ▶ Machine Learning

What is Machine Learning?

- ▶ ML is getting computers to learn without being explicitly programmed

What is Machine Learning?

- ▶ ML is getting computers to learn without being explicitly programmed
- ▶ ML is a branch of AI (Artificial Intelligence)

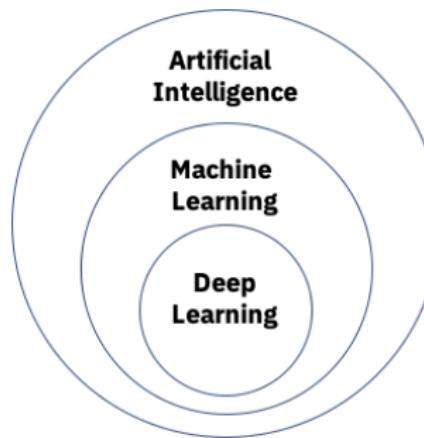


Figure: [www.ibm.com/cloud/learn/what-is-artificial-intelligence]

Machine Learning

The following are all machine learning algorithms:

- ▶ Linear Regression
- ▶ Logistic Regression
- ▶ Decision Tree
- ▶ SVM (support vector machine)
- ▶ Naive Bayes
- ▶ kNN (k nearest neighbours)
- ▶ K-Means Clustering
- ▶ Random Forests
- ▶ Dimensionality Reduction Algorithms
- ▶ Gradient Boosting algorithms
- ▶ Artificial Neural Networks (Deep Learning)

Machine Learning

Broadly, there are three types of machine learning algorithms:

Machine Learning

Broadly, there are three types of machine learning algorithms:

- ▶ Supervised Learning
 - ▶ learns from 'labelled' data

Machine Learning

Broadly, there are three types of machine learning algorithms:

- ▶ Supervised Learning
 - ▶ learns from 'labelled' data
- ▶ Unsupervised Learning
 - ▶ data is not labelled

Machine Learning

Broadly, there are three types of machine learning algorithms:

- ▶ Supervised Learning
 - ▶ learns from 'labelled' data
- ▶ Unsupervised Learning
 - ▶ data is not labelled
- ▶ Reinforcement Learning
 - ▶ learns from past experience; uses past experience to improve its performance

Supervised Learning

- ▶ learns from 'labelled' data

Supervised Learning

- ▶ learns from 'labelled' data
- ▶ generates a function that map inputs to desired outputs

Supervised Learning

- ▶ learns from 'labelled' data
- ▶ generates a function that map inputs to desired outputs
- ▶ examples of tasks: classification, regression

Supervised Learning

- ▶ learns from ‘labelled’ data
- ▶ generates a function that map inputs to desired outputs
- ▶ examples of tasks: classification, regression
- ▶ examples of algorithms: Neural Networks, Linear/Logistic Regression, Decision Tree, Random Forest, KNN

Supervised Learning: Linear Regression

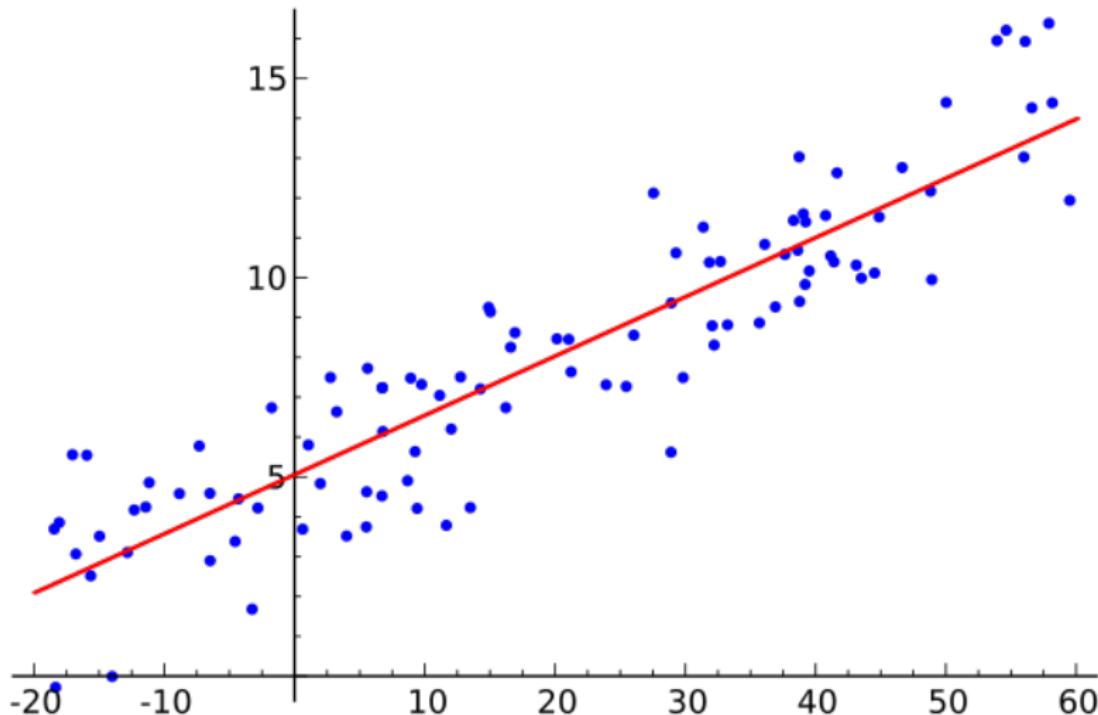


Figure: Wikipedia!

Supervised Learning: Polynomial Regression

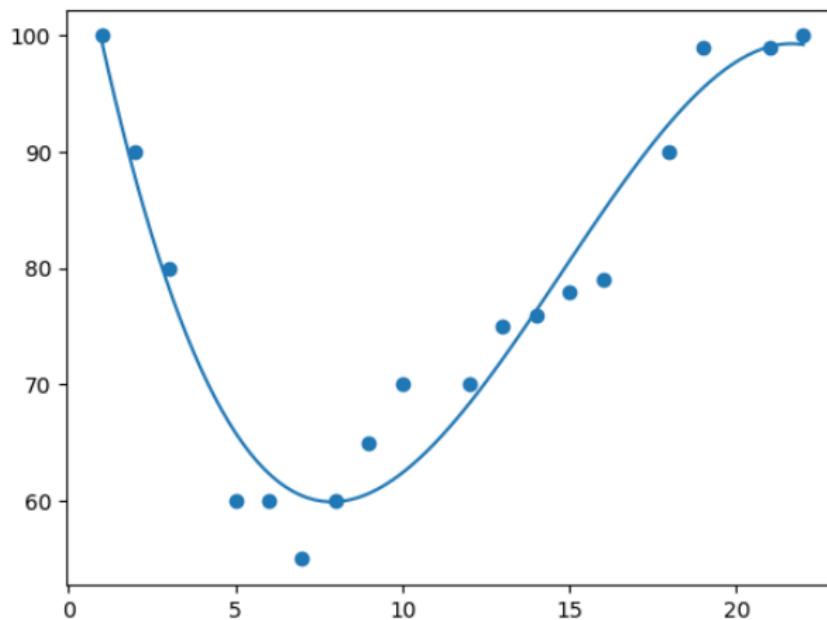


Figure: www.w3schools.com/l

Supervised Learning: K-Nearest Neighbours

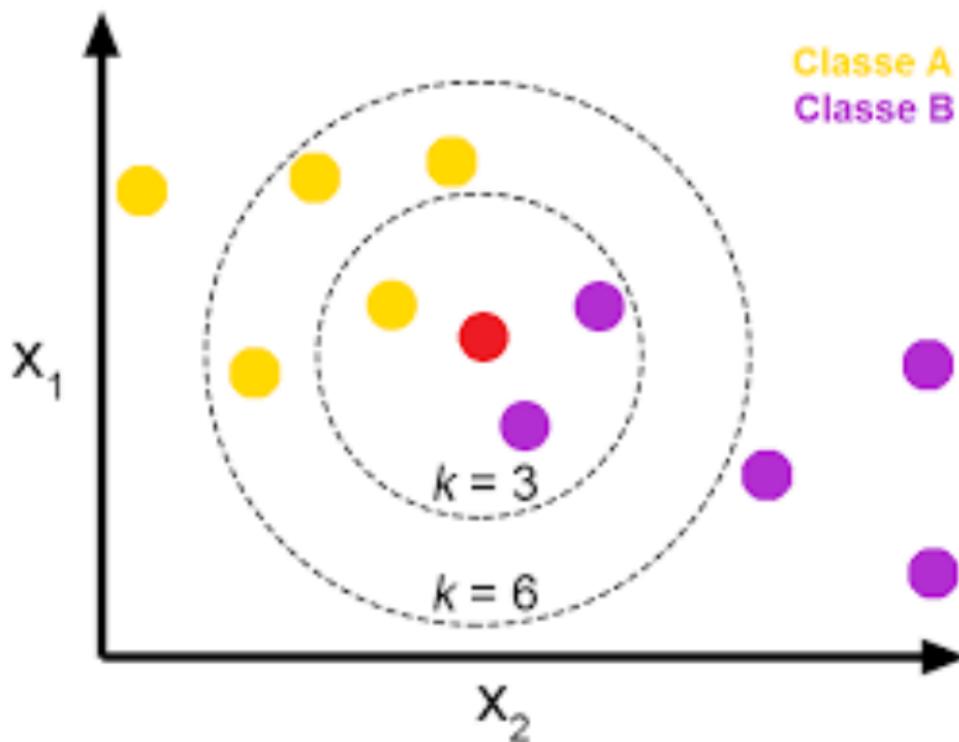


Figure: towardsdatascience.com

Supervised: Image Classification

dog



dog



cat



cat



Figure: Is it a cat or a dog?

Unsupervised Learning

- ▶ data is not labelled

Unsupervised Learning

- ▶ data is not labelled
- ▶ example of tasks: clustering, dimension reduction

Unsupervised Learning

- ▶ data is not labelled
- ▶ example of tasks: clustering, dimension reduction
- ▶ examples of algorithms: K-Means, PCA

Unsupervised: K-Means Clustering

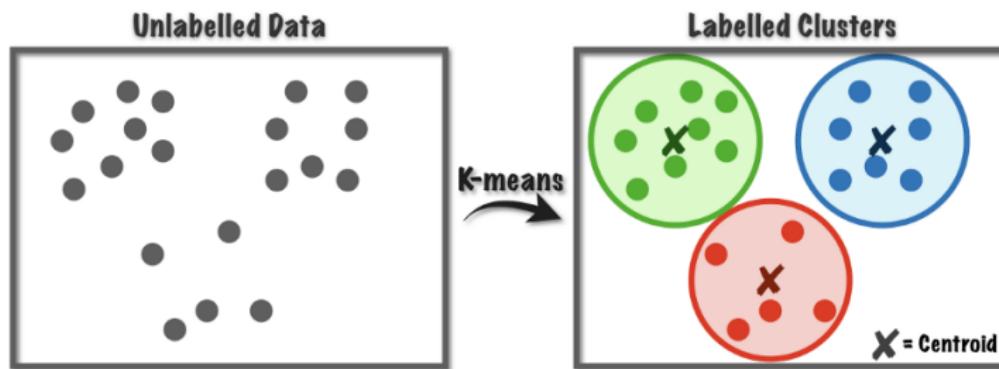


Figure: [towardsdatascience.com]

Unsupervised: Principal Component Analysis

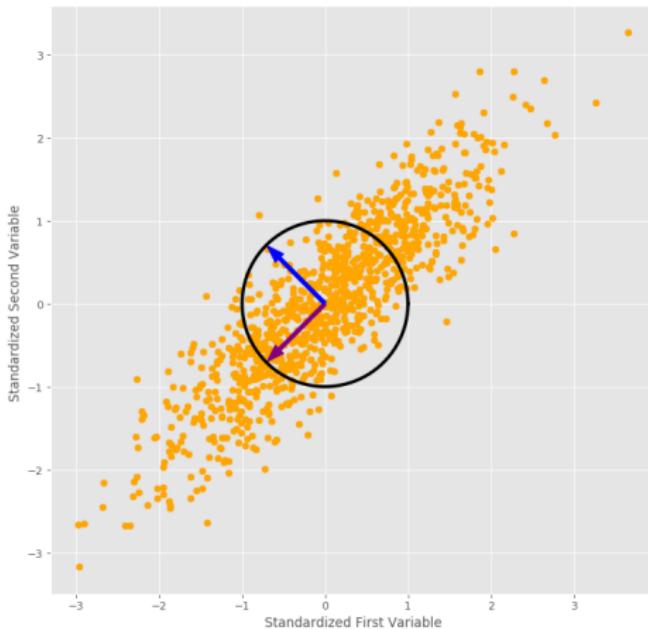


Figure: [towardsdatascience.com]

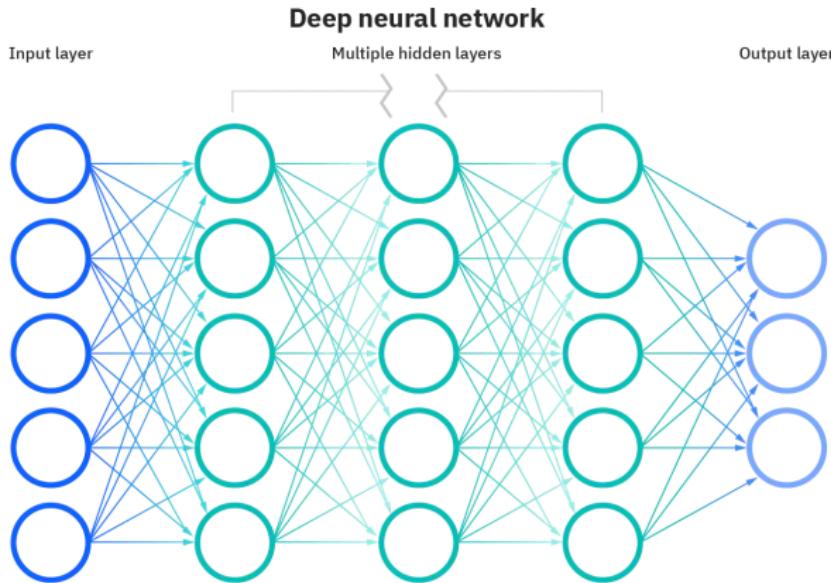
Reinforcement Learning

- ▶ learns from past experience; uses past experience to improve its performance
- ▶ example of tasks: self-driving car, robotics, chess, roomba
- ▶ examples of algorithms: Markov Decision Process



Artificial Neural Networks (Deep Learning)

- ▶ mimics biological neural networks
- ▶ a 'deep neural network' is an ANN with multiple 'hidden layers'



Nodes of an ANN

- ▶ each node or ‘neuron’ looks like:

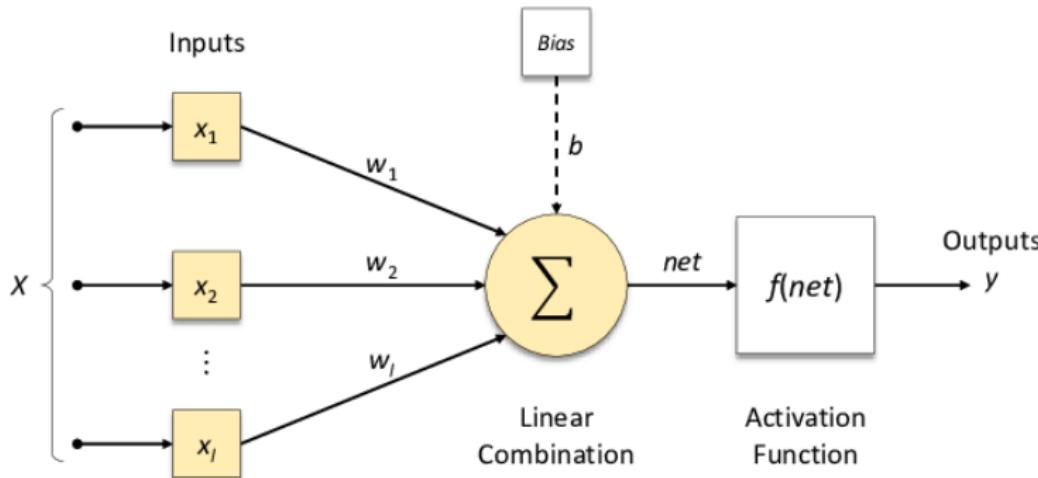


Figure: [Parmezan, Alves de Souza, Batista, Gustavo.
10.1016/j.ins.2019.01.076]

- ▶ Linear combination of inputs: $w_1x_1 + w_2x_2 + \cdots + w_Ix_I + b$

Principal Component Analysis

- ▶ Unsupervised learning technique

Ref. Jose Portilla - PrincipalComponentAnalysis.pdf
[\(https://www.udemy.com/python-for-data-science-and-machine-learning-bootcamp/learn/v4/overview\)](https://www.udemy.com/python-for-data-science-and-machine-learning-bootcamp/learn/v4/overview)

Principal Component Analysis

- ▶ Unsupervised learning technique
- ▶ Used for dimension reduction (by finding most important combinations of features of data)
- ▶ Used to examine the interrelations among a set of variables (features) in order to identify the underlying structure of those variables

Ref. Jose Portilla - PrincipalComponentAnalysis.pdf
(<https://www.udemy.com/python-for-data-science-and-machine-learning-bootcamp/learn/v4/overview>)

Principal Component Analysis

- ▶ Unsupervised learning technique
- ▶ Used for dimension reduction (by finding most important combinations of features of data)
- ▶ Used to examine the interrelations among a set of variables (features) in order to identify the underlying structure of those variables
- ▶ Used to find patterns in a large data set

Ref. Jose Portilla - PrincipalComponentAnalysis.pdf
(<https://www.udemy.com/python-for-data-science-and-machine-learning-bootcamp/learn/v4/overview>)

Principal Component Analysis

- ▶ Where regression determines a line of best fit to a data set, PCA determines several orthogonal lines of best fit to the data set.

Principal Component Analysis

- ▶ Where regression determines a line of best fit to a data set, PCA determines several orthogonal lines of best fit to the data set.
- ▶ Recall: Orthogonal means “at right angles” (perpendicular) in n-dimensional space.
- ▶ n-Dimensional Space is the variable sample space.
 - ▶ There are as many dimensions as there are variables, so in a data set with 4 variables the sample space is 4-dimensional.

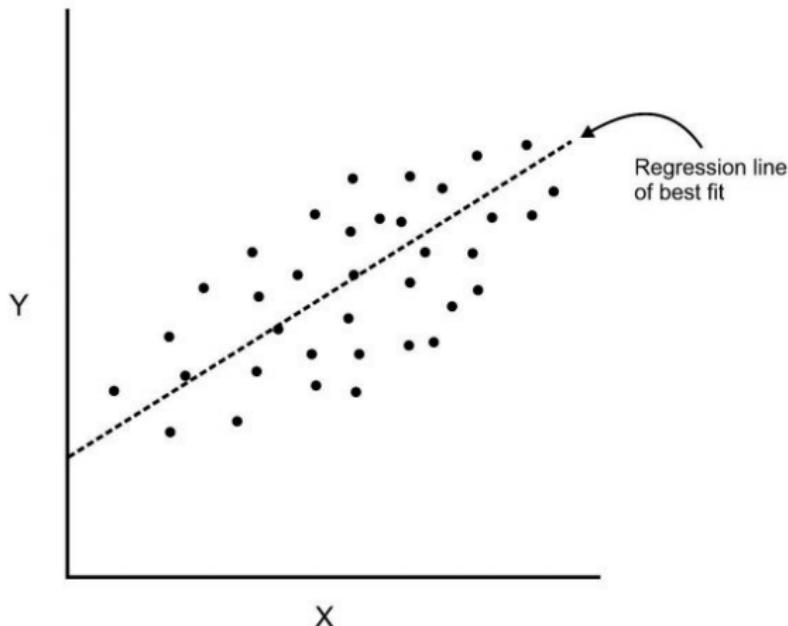


Figure: [udemy.com]

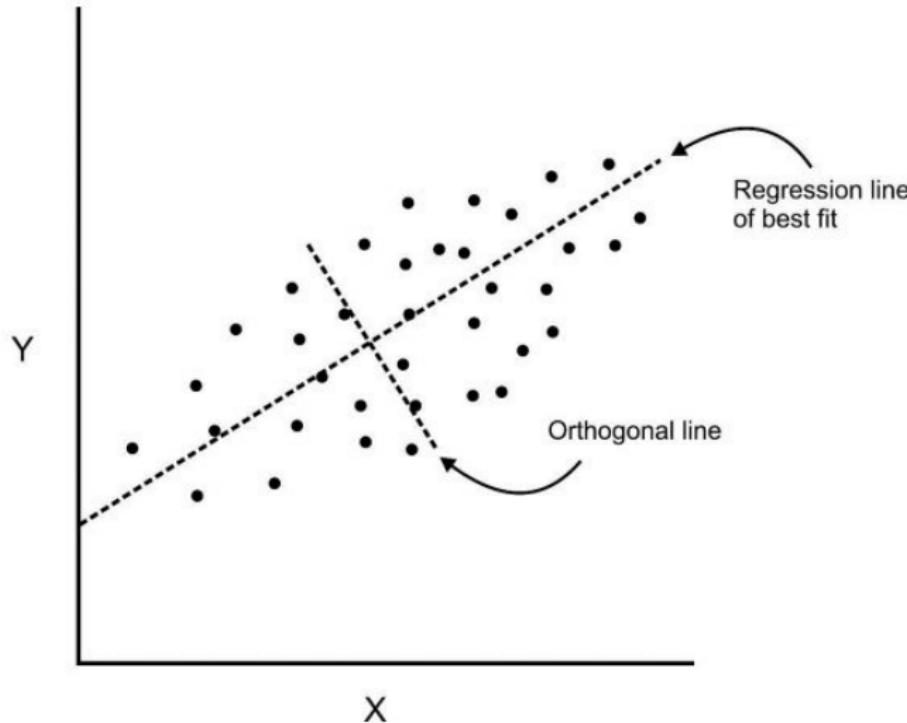


Figure: [udemy.com]

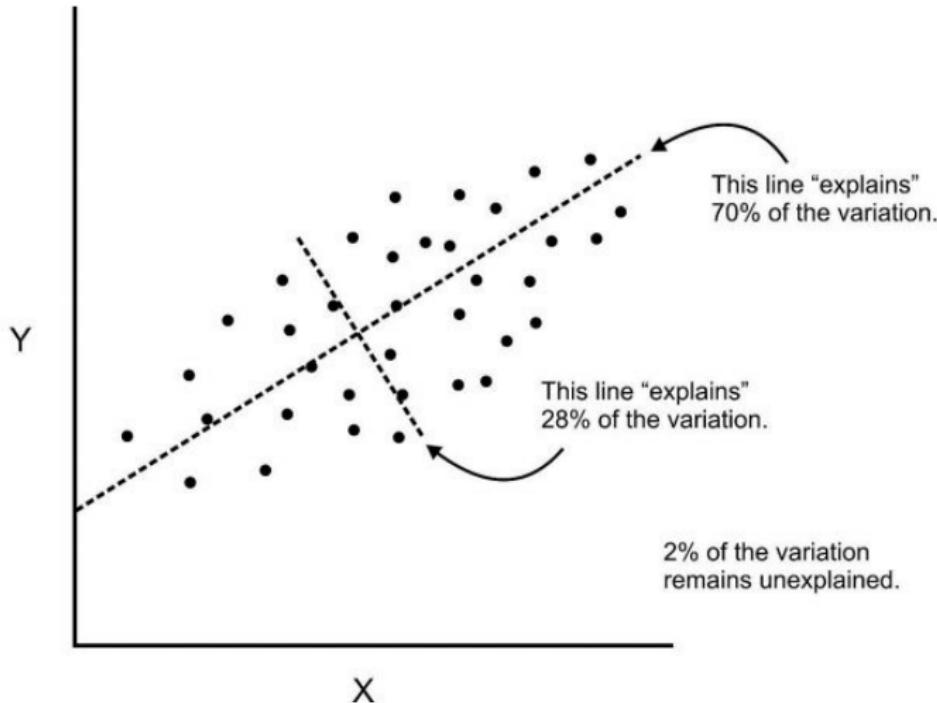


Figure: [udemy.com]

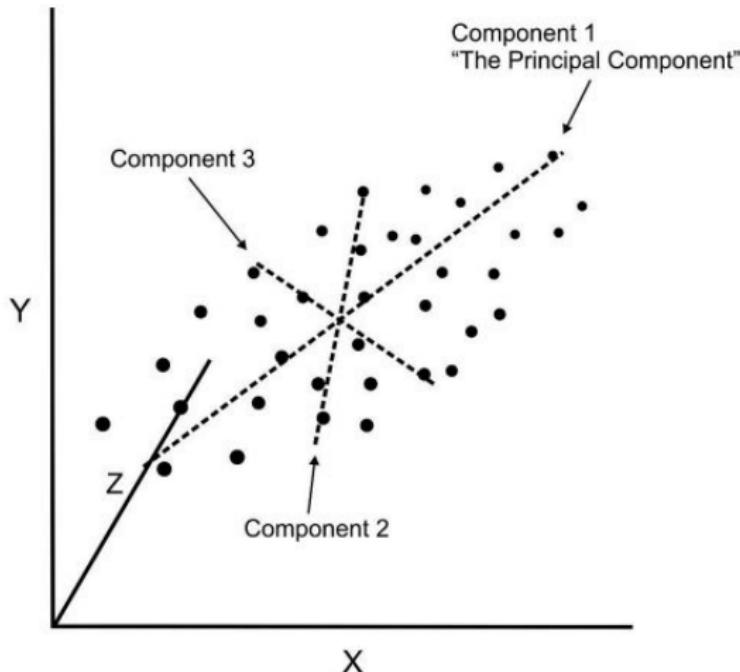


Figure: [udemy.com]

- ▶ PCA is essentially the construction of a coordinate transformation that is chosen such that the greatest variance of the data set comes to lie on the first axis
- ▶ The second greatest variance on the second axis, and so on ...
- ▶ This process allows us to reduce the number of variables used in an analysis.
- ▶ Note that components are uncorrelated, since in the sample space they are orthogonal to each other.

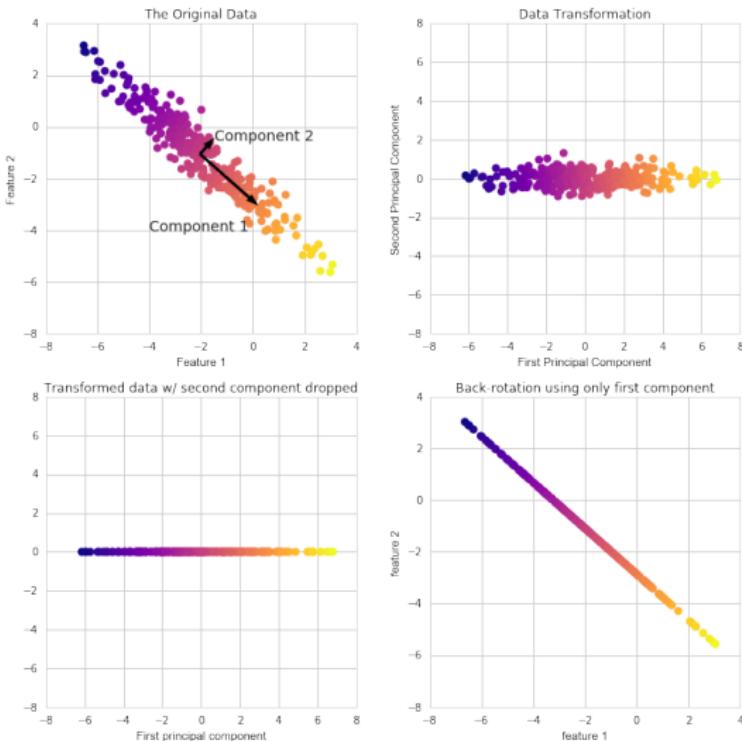


Figure: [udemy.com]

- ▶ If we use this technique on a data set with a large number of variables, we may capture a large portion of the variance (i.e. most of the important features) with just a few components.
- ▶ The most challenging part of PCA is interpreting the components.

PCA uses Singular Value Decomposition (SVD)

Definition (SVD)

Given $A \in \mathbb{R}^{m \times n}$, a singular value decomposition is a factorisation

$$A = U\Sigma V^T$$

- ▶ $U \in \mathbb{R}^{m \times m}$ orthogonal
- ▶ $\Sigma \in \mathbb{R}^{m \times n}$ diagonal
- ▶ $V \in \mathbb{R}^{n \times n}$ orthogonal

Diagonal elements of Σ are singular values of A

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$$

where $p := \min\{m, n\}$.

Examples

$$\begin{aligned} A &= \begin{bmatrix} 0 & 1 \\ 1 & 3/2 \end{bmatrix} \\ &= \frac{1}{\sqrt{5}} \begin{bmatrix} 1 & -2 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} \textcolor{red}{2} & 0 \\ 0 & \textcolor{red}{1/2} \end{bmatrix} \cdot \frac{1}{\sqrt{5}} \begin{bmatrix} 1 & 2 \\ 2 & -1 \end{bmatrix}^T = U \Sigma V^T \end{aligned}$$

Examples

$$\begin{aligned} A &= \begin{bmatrix} 0 & 1 \\ 1 & 3/2 \end{bmatrix} \\ &= \frac{1}{\sqrt{5}} \begin{bmatrix} 1 & -2 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0 \\ 0 & 1/2 \end{bmatrix} \cdot \frac{1}{\sqrt{5}} \begin{bmatrix} 1 & 2 \\ 2 & -1 \end{bmatrix}^T = U\Sigma V^T \end{aligned}$$

$$\begin{aligned} A &= \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \\ &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \sqrt{2} & 0 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}^T = U\Sigma V^T \end{aligned}$$

Example

$$A = \begin{bmatrix} 0 & 3 \\ 2 & 0 \\ 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & 2 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^T = U_1 \Sigma V_1^T$$

$$= \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & 2 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}^T = U_2 \Sigma V_2^T$$

Remarks

- ▶ For A real, matrices U, V real and orthogonal
- ▶ Unlike eigenvalue decomposition, always exists
- ▶ Singular values unique, real, nonnegative
- ▶ U, V not generally unique
- ▶ Reduction of matrix to rotation/reflections and diagonal scaling
- ▶ In PYTHON: `numpy.linalg.svd` (note: output gives V^T not V)

SVD vs. eigenvalue decomposition

Eigenvalue decomposition

$$A = X^{-1} \Lambda X$$

Singular value decomposition

$$A = U \Sigma V^T$$

SVD vs. eigenvalue decomposition

Eigenvalue decomposition

$$A = X^{-1} \Lambda X$$

A must be square

Singular value decomposition

$$A = U \Sigma V^T$$

A can be rectangular/square

SVD vs. eigenvalue decomposition

Eigenvalue decomposition

$$A = X^{-1} \Lambda X$$

A must be square

$$A = X^{-1} \Lambda X \text{ iff } A \text{ nondefective}$$

Singular value decomposition

$$A = U \Sigma V^T$$

A can be rectangular/square

$$A = U \Sigma V^T \text{ always exists}$$

SVD vs. eigenvalue decomposition

Eigenvalue decomposition

$$A = X^{-1} \Lambda X$$

A must be square

$A = X^{-1} \Lambda X$ iff A nondefective

Λ can be real or complex

Singular value decomposition

$$A = U \Sigma V^T$$

A can be rectangular/square

$A = U \Sigma V^T$ always exists

Σ must be real, nonnegative

SVD vs. eigenvalue decomposition

Eigenvalue decomposition

$$A = X^{-1} \Lambda X$$

A must be square

$A = X^{-1} \Lambda X$ iff A nondefective

Λ can be real or complex

$A \in \mathbb{R}^{n \times n} \Rightarrow X, \Lambda$ real or complex

Singular value decomposition

$$A = U \Sigma V^T$$

A can be rectangular/square

$A = U \Sigma V^T$ always exists

Σ must be real, nonnegative

$A \in \mathbb{R}^{n \times n} \Rightarrow U, V$ real

SVD vs. eigenvalue decomposition

Eigenvalue decomposition

$$A = X^{-1} \Lambda X$$

A must be square

$A = X^{-1} \Lambda X$ iff A nondefective

Λ can be real or complex

$A \in \mathbb{R}^{n \times n} \Rightarrow X, \Lambda$ real or complex

columns of X possibly nonorthogonal

Singular value decomposition

$$A = U \Sigma V^T$$

A can be rectangular/square

$A = U \Sigma V^T$ always exists

Σ must be real, nonnegative

$A \in \mathbb{R}^{n \times n} \Rightarrow U, V$ real

columns of U, V orthogonal

Rank, nullity, and SVDs

Theorem

The rank of A is r , the number of nonzero singular values of A .

Rank, nullity, and SVDs

Theorem

The rank of A is r , the number of nonzero singular values of A .

Theorem

Let $A \in \mathbb{R}^{m \times n}$ have SVD $A = U\Sigma V^H$ with $U = [\vec{u}_1 | \cdots | \vec{u}_m]$ and $V = [\vec{v}_1 | \cdots | \vec{v}_n]$. Then,

$$\text{range}(A) = \text{span} \{ \vec{u}_1, \vec{u}_2, \dots, \vec{u}_r \}$$

$$\text{nullspace}(A) = \text{span} \{ \vec{v}_{r+1}, \vec{v}_{r+2}, \dots, \vec{v}_n \}.$$

Example

$$\begin{aligned} A &= \begin{bmatrix} -350 & 775 & -950 \\ 1824 & -336 & 1248 \\ -532 & 98 & -364 \\ 1300 & -950 & 1600 \end{bmatrix} \\ &= \frac{1}{75} \begin{bmatrix} -25 & 50 & 0 & 50 \\ 48 & 48 & 7 & -24 \\ -14 & -14 & 24 & 7 \\ 50 & -25 & 0 & 50 \end{bmatrix} \cdot \begin{bmatrix} 3375 & 0 & 0 \\ 0 & 900 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \cdot \frac{1}{3} \begin{bmatrix} 2 & -1 & 2 \\ 2 & 2 & -1 \\ -1 & 2 & 2 \end{bmatrix} \end{aligned}$$

$$\text{nullspace}(A) = \text{span} \left\{ \begin{bmatrix} -1 \\ 2 \\ 2 \end{bmatrix} \right\}, \quad \text{range}(A) = \text{span} \left\{ \begin{bmatrix} -25 \\ 48 \\ -14 \\ 50 \end{bmatrix}, \begin{bmatrix} 50 \\ 48 \\ -14 \\ -25 \end{bmatrix} \right\}$$

Right and left singular vectors

- Columns of U and V are **left** and **right singular vectors**

$$U := \begin{bmatrix} \vec{u}_1 & | & \vec{u}_2 & | & \cdots & | & \vec{u}_m \end{bmatrix}, V := \begin{bmatrix} \vec{v}_1 & | & \vec{v}_2 & | & \cdots & | & \vec{v}_n \end{bmatrix}$$

- Decomposition as sum of rank one matrices

$$A = \sum_{\ell=1}^r \sigma_\ell \vec{u}_\ell \vec{v}_\ell^T$$

Decomposition as sum of rank one matrices

Can show by direct computation:

$$A_{i,j}$$

Decomposition as sum of rank one matrices

Can show by direct computation:

$$A_{i,j} = [U\Sigma V^T]_{i,j}$$

Decomposition as sum of rank one matrices

Can show by direct computation:

$$A_{i,j} = [U\Sigma V^T]_{i,j} = \sum_{k=1}^m U_{i,k} \sum_{\ell=1}^n \Sigma_{k,\ell} [V^T]_{\ell,j}$$

Decomposition as sum of rank one matrices

Can show by direct computation:

$$\begin{aligned} A_{i,j} &= [U\Sigma V^T]_{i,j} = \sum_{k=1}^m U_{i,k} \sum_{\ell=1}^n \Sigma_{k,\ell} [V^T]_{\ell,j} \\ &= \sum_{k=1}^m U_{i,k} \sum_{\ell=1}^n (\delta_{k,\ell} \sigma_k) [V^T]_{\ell,j} \end{aligned}$$

Decomposition as sum of rank one matrices

Can show by direct computation:

$$\begin{aligned} A_{i,j} &= [U\Sigma V^T]_{i,j} = \sum_{k=1}^m U_{i,k} \sum_{\ell=1}^n \Sigma_{k,\ell} [V^T]_{\ell,j} \\ &= \sum_{k=1}^m U_{i,k} \sum_{\ell=1}^n (\delta_{k,\ell} \sigma_k) [V^T]_{\ell,j} \\ &= \sum_{\ell=1}^r U_{i,\ell} \sigma_\ell [V^T]_{\ell,j} \end{aligned}$$

Decomposition as sum of rank one matrices

Can show by direct computation:

$$\begin{aligned} A_{i,j} &= [U\Sigma V^T]_{i,j} = \sum_{k=1}^m U_{i,k} \sum_{\ell=1}^n \Sigma_{k,\ell} [V^T]_{\ell,j} \\ &= \sum_{k=1}^m U_{i,k} \sum_{\ell=1}^n (\delta_{k,\ell} \sigma_k) [V^T]_{\ell,j} \\ &= \sum_{\ell=1}^r U_{i,\ell} \sigma_\ell [V^T]_{\ell,j} \\ &= \sum_{\ell=1}^r \sigma_\ell [\vec{u}_\ell \vec{v}_\ell^T]_{i,j} \end{aligned}$$

Decomposition as sum of rank one matrices

Can show by direct computation:

$$\begin{aligned} A_{i,j} &= [U\Sigma V^T]_{i,j} = \sum_{k=1}^m U_{i,k} \sum_{\ell=1}^n \Sigma_{k,\ell} [V^T]_{\ell,j} \\ &= \sum_{k=1}^m U_{i,k} \sum_{\ell=1}^n (\delta_{k,\ell} \sigma_k) [V^T]_{\ell,j} \\ &= \sum_{\ell=1}^r U_{i,\ell} \sigma_\ell [V^T]_{\ell,j} \\ &= \sum_{\ell=1}^r \sigma_\ell [\vec{u}_\ell \vec{v}_\ell^T]_{i,j} \end{aligned}$$

i.e., A decomposes as sum of $r = \text{rank}(A)$ outer products

Example

$$\begin{aligned} A &= \begin{bmatrix} -350 & 775 & -950 \\ 1824 & -336 & 1248 \\ -532 & 98 & -364 \\ 1300 & -950 & 1600 \end{bmatrix} \\ &= \frac{1}{75} \begin{bmatrix} -25 & 50 & 0 & 50 \\ 48 & 48 & 7 & -24 \\ -14 & -14 & 24 & 7 \\ 50 & -25 & 0 & 50 \end{bmatrix} \cdot \begin{bmatrix} 3375 & 0 & 0 \\ 0 & 900 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \cdot \frac{1}{3} \begin{bmatrix} 2 & -1 & 2 \\ 2 & 2 & -1 \\ -1 & 2 & 2 \end{bmatrix} \\ &= 3375 \left(\frac{1}{75} \begin{bmatrix} -25 \\ 48 \\ -14 \\ 50 \end{bmatrix} \right) \left(\frac{1}{3} [2 \quad -1 \quad 2] \right) + 900 \left(\frac{1}{75} \begin{bmatrix} 50 \\ 48 \\ -14 \\ -25 \end{bmatrix} \right) \left(\frac{1}{3} [2 \quad 2] \right) \\ &= \sigma_1 \vec{u}_1 \vec{v}_1^T + \sigma_2 \vec{u}_2 \vec{v}_2^T \end{aligned}$$

Low rank approximations

Can approximate A by truncating sum of rank one matrices.
That is, for some $p < r$, we have:

$$A \approx \sum_{\ell=1}^p \sigma_\ell \vec{u}_\ell \vec{v}_\ell^T$$

Image compression

Original image: 200x320 entries

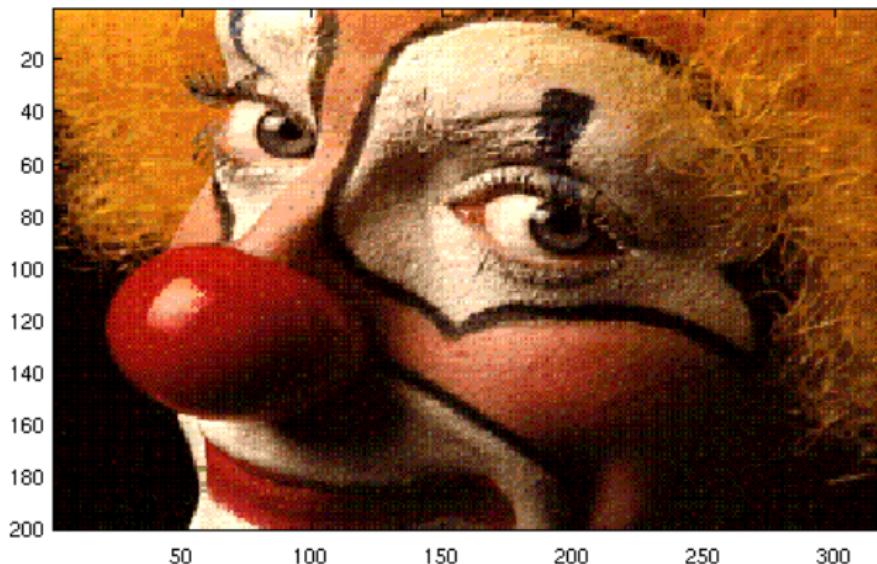


Image compression

$r=10$: 200x20 entries

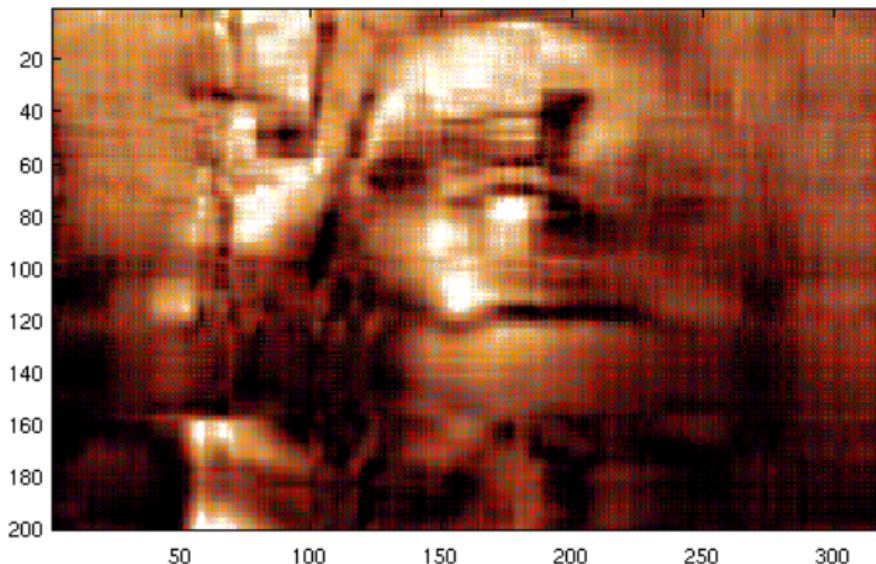
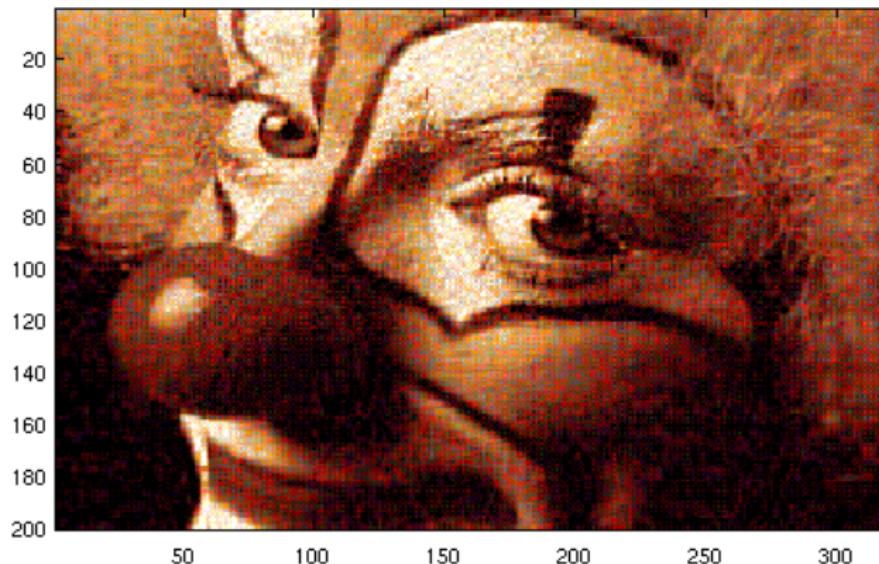
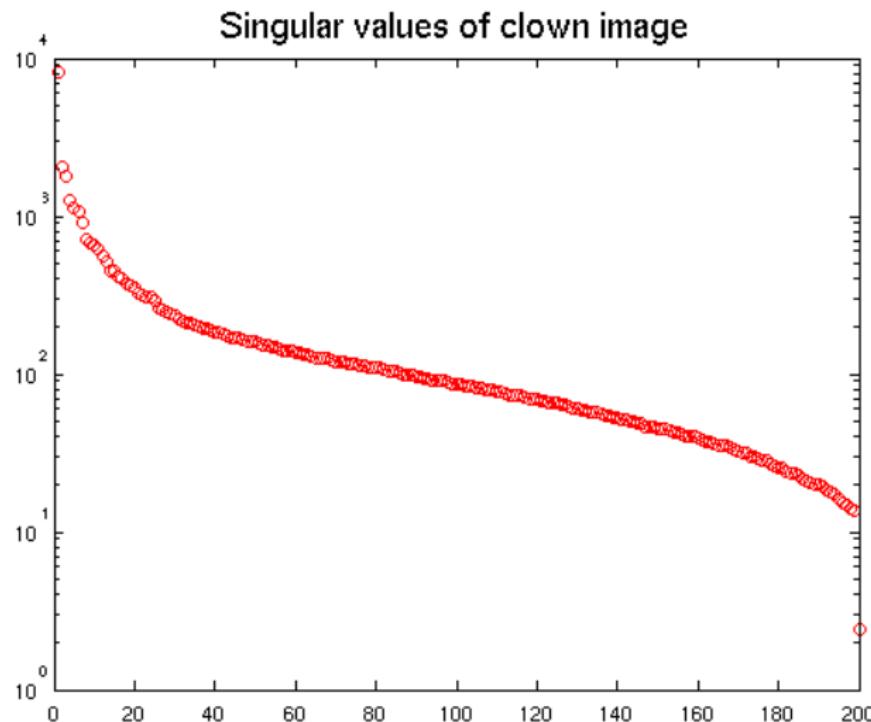


Image compression

$r=50$: 200x100 entries





What does SVD have to do with PCA?

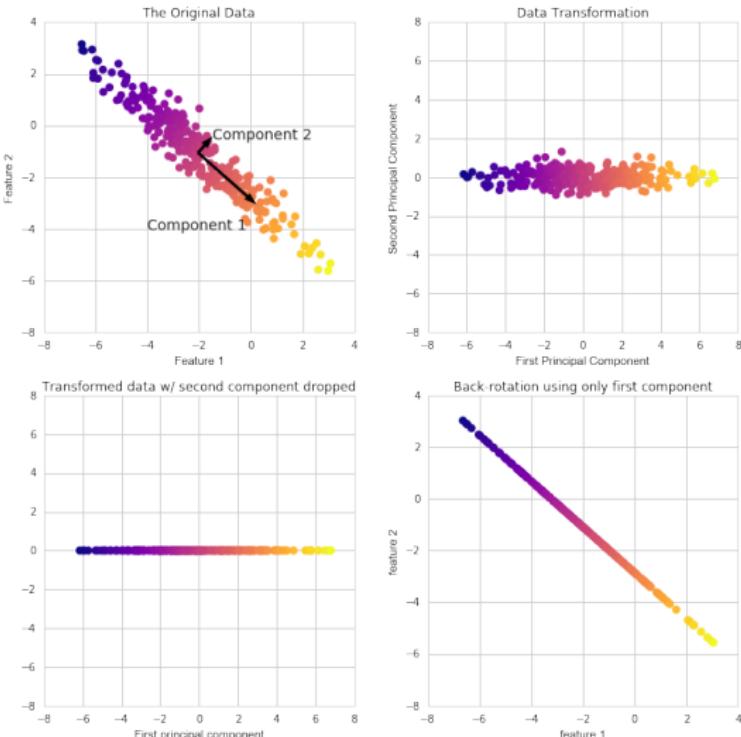


Figure: [udemy.com]

What does SVD have to do with PCA?

- ▶ First principal component is first left singular vector \vec{u}_1
- ▶ Second principal component is second left singular vector \vec{u}_2 , etc.
- ▶ and so on...

Example

Given $n = 6$ people in a class. For each person construct a vector \vec{p}_j containing the person's deviation of height and weight from the mean (average): $p_j = (\text{height}, \text{weight})$:

$$\begin{aligned}\vec{p}_1 &= (3, 7), & \vec{p}_2 &= (-4, -6), & \vec{p}_3 &= (7, 8), & \vec{p}_4 &= (1, -1), \\ \vec{p}_5 &= (-4, -1), & \vec{p}_6 &= (3, 3)\end{aligned}$$

Write into a matrix: Heights in first row, weights in second row:

$$A = \begin{bmatrix} 3 & -4 & 7 & 1 & -4 & 3 \\ 7 & -6 & 8 & -1 & -1 & 3 \end{bmatrix}$$

Construct SVD →

$$\sigma_1 = 15.71, \quad \vec{u}_1 = (-0.61, -0.79),$$
$$\sigma_2 = 3.61, \quad \vec{u}_2 = (-0.79, 0.61)$$

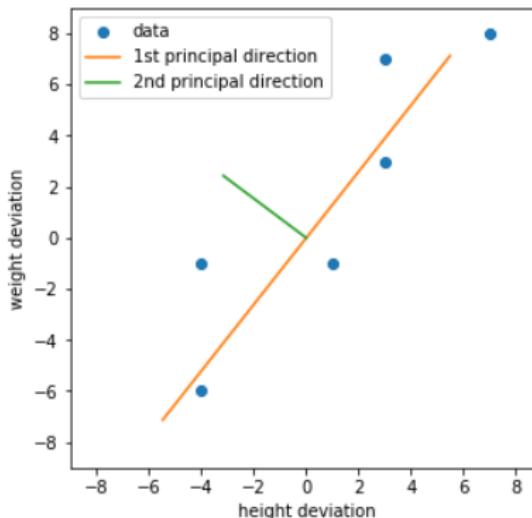


Figure: