Numerical Methods Lecture Notes

Lucas Bouck

September 25, 2025

Contents

1 Numerical Methods: Course Overview (August 25, 2025)

Main Goal: Learn how to use computers to solve scientific problems and mathematical problems of a continuous nature. Problems of a discrete nature typically are in computer science courses

1.1 Example Problems

- **Solving nonlinear equations**

  Given a function $f : \mathbb{R} \to \mathbb{R}$, we seek to find $x \in \mathbb{R}$ such that $f(x) = y$.

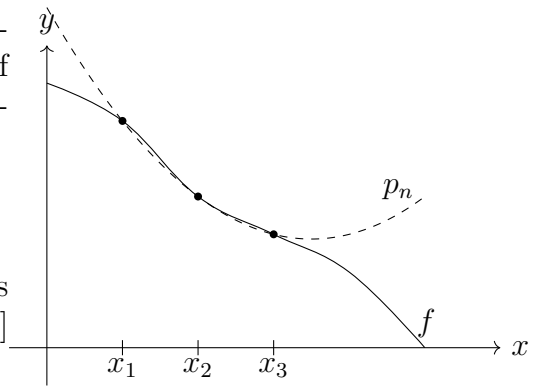
  Example To compute $\sqrt{y}$, we want to find x such that $x^2 = y$. One such method is the Babylonian method (or Heron's method), which we'll learn as Newton's method for solving $x^2 = y$.

- **Polynomial interpolation**

  For a given function $f : [0, 1] \to \mathbb{R}$, it might be extremely expensive or impossible to compute at some points. The goal of polynomial interpolation is to find a polynomial

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

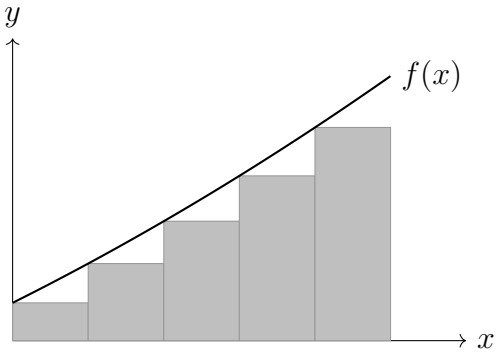
  so that $f(x_i) = p_n(x_i)$ for $n + 1$ points x_i. Hopefully $f(x) \approx p_n(x)$ for $x \in [0, 1]$ and p_n is cheap to evaluate.



- **Numerical integration/quadrature**

Given $f : [0, 1] \to \mathbb{R}$, we would like to compute

$$\int_0^1 f(x)dx.$$



Solving integrals by hand is very difficult, so we approximate by

$$\int_0^1 f(x)dx \approx \sum_{j=0}^{n} w_j f\left(x_j\right).$$

For example if $w_j = \frac{1}{n}$ and $x_j = \frac{j}{n+1}$, we have a Riemann sum.

- **Solving differential equations**

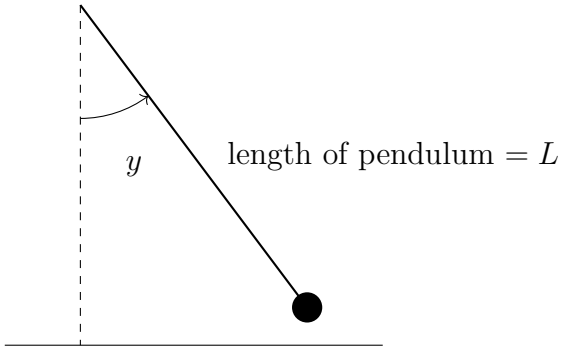
  Many problems in physics/engineering can be written as a differential equation

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

  These problems become too difficult to solve by hand quickly.

  Example: Swinging pendulum $m = 1$

  $$y''(t) = -\frac{g}{L}\sin(y(t))$$



- **Solving systems of linear equations** Numerous problems can be reduced to solving

$$\mathbf{Ax} = \mathbf{b} \quad \text{or} \quad \begin{aligned} A_{11}x_1 + A_{12}x_2 + \cdots + A_{1n}x_n &= b_1 \\ \vdots \qquad\qquad\qquad\qquad &\quad \vdots \\ A_{n1}x_1 + A_{n2}x_2 + \cdots + A_{nn}x_n &= b_n \end{aligned}$$

  Such problems arise in statistics/data science (like linear regression) solving partial differential equations and more. In my own research, I regularly have to solve systems where $n \approx 100,000$ or $n \approx 1,000,000$.

1.2 Why should we care?

Modern science relies more heavily on computation and modern engineering/business also leverages computation.

- **National Weather Service forecasts** regularly require solving systems of linear and nonlinear equations and solving differential equations.
- **Data science** Many online services and data science tools use linear algebra. Some of the higher performing models for recommending movies in the Netflix Prize Problem computed a Singular Value Decomposition.
- **Large scale scientific experiments** leverage more computational tools. The LIGO detector for gravitational waves relies heavily on numerical integration techniques.

2 Floating Point and Machine Representation of Numbers (August 27, 2025)

Goals: Understand how machines represent numbers and how this impacts computing.

Example 2.1. What is $0.1 + 0.2$?

- In exact arithmetic, $0.1 + 0.2 = 0.3$
- In Python, $0.1 + 0.2 = 0.30000000000000004$

Why is there an extra 4?

2.1 Binary representation of numbers

We first review how to write numbers in binary (or base 2).

Example 2.2 (converting binary to base 10)**.** Take $(110.11011)_2$. The $(\cdot)_2$ means it is written in base 2. We have

$$(110)_2 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6$$
$$(0.11011)_2 = 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + 1 \times 2^{-5} = \frac{1}{2} + \frac{1}{4} + \frac{1}{16} + \frac{1}{32} = \frac{27}{32}.$$

Hence,

$$(110.11011)_2 = \left(6\,\frac{27}{32}\right)_{10}.$$

2.1.1 Converting base 10 to binary

The general procedure is for converting base to to binary for integers is below.

- Divide by 2
- Keep track of remainder (0 or 1)
- Keep going into the number = 0.
- remainders are digits in reverse order of binary number

Example 2.3 (converting base 10 to binary for integers)**.** We want to find $(71)_{10} = (???)_2$. We follow the procedure outlined above

Number	Remainder
71	
35	1
17	1
8	1
4	0
2	0
1	0
0	1

- Go down each row dividing by 2
- Right column is remainder

Answer

$$(71)_{10} = (1000111)_2$$

What is the procedure for fractions? We outline the procedure below.

- Multiply by 2
- Keep track of integer part (0 or 1) and fractional part
- Multiply fractional part by 2
- Repeat
- Integer part are binary expansion away from decimal

Example 2.4 (converting base 10 to binary for fractions)**.** We want to find $\left(\frac{5}{8}\right)_{10} = (???)_2$. We follow the procedure outlined above.

$$\frac{5}{8} \times 2 = 1 + \frac{1}{4}$$
$$\frac{1}{4} \times 2 = 0 + \frac{1}{2}$$
$$\frac{1}{2} \times 2 = 1 + 0$$
$$0 \times 2 = 0 + 0$$
stop

- Go down each row multiplying by 2
- Split with integer part and fractional part

Answer

$$\left(\frac{5}{8}\right)_{10} = (0.101)_2$$

Example 2.5 (repeating decimal expansion)**.** What about $\left(\frac{1}{3}\right)_{10} = (???)_2$?

$$\frac{1}{3} \times 2 = 0 + \frac{2}{3}$$
$$\frac{2}{3} \times 2 = 1 + \frac{1}{3}$$
$$\frac{1}{3} \times 2 = 0 + \frac{2}{3}$$
$$\frac{2}{3} \times 2 = 1 + \frac{1}{3}$$
$$\vdots$$

It repeats! The answer is

$$\left(\frac{1}{3}\right)_{10} = (0.\overline{01})_2$$

2.1.2 Going in reverse to find number in base 10

How do we find numbers in base 10 given a repeating binary expansion? There are a few tricks.

Example 2.6 (base 10 number from repeating binary decimal)**.** We want to find $x = (0.\overline{01})_2 = (???)_{10}$. The trick is to multiply x by 4 and subtract in order to cancel the repeating decimal part

$$x = (0.\overline{01})_2$$
$$4x = (1.\overline{01})_2$$
$$3x = 4x - x = (1)_2 = 1 \implies x = \frac{1}{3}$$

2.2 Double Precision Floating Point Representation

Definition 2.1 (double precision floating pt representation)**.** Let x be a real number. It's double precision floating point representation is

$$\texttt{fl}(x) = (-1)^s \times 1.\underbrace{b_1 b_2 \dots b_{52}}_{52 \text{ bits}} \times 2^e$$

where e is an integer exponent in binary and $s = 0$ or 1 is a sign bit. The 52 bits in the main chunk are called the mantissa. The integer exponent is represented with 11 bits and is stored as in the computer as $(e + 1023)_2$.

Example 2.7 (examples of floating point representations)**.** Below are a few examples

- $x = 13 = (1101)_2$

$$\texttt{fl}(13) = \texttt{fl}((1101)_2) = +1.101 \times 2^3 = +1.101 \times 2^{(11)_2}$$

- $x = 2\frac{5}{8} = (10.101)_2$

$$\texttt{fl}\left(2\frac{5}{8}\right) = \texttt{fl}((10.101)_2) = +1.0101 \times 2^1$$

- $x = \frac{1}{3} = (0.\overline{01})_2$

$$\texttt{fl}\left(\frac{1}{3}\right) = \texttt{fl}((0.\overline{01})_2) = +1.0101\ldots \times 2^{-2} = +1.0101\ldots \times 2^{-(10)_2}$$

The last example of $x = 1/3$ leads to some questions about what do we do at the 52nd bit?

2.2.1 Rounding rules for floating point representation

For our example of $x = 1/3$, the mantissa looks like.

$$\text{mantissa} = 0101\ldots010\underbrace{1}_{=b_{52}} \mid \underbrace{0}_{=b_{53}}1\ldots$$

For floating point, we want to round the number to the nearest possible floating point number, which leads to the following rules

- If $b_{53} = 0$, we round down (or chop)

- If $b_{53} = 1$, we round up (add 1 to b_{52} and then chop)
- The special case of $b_{53} = 1$ with all other remaining bits $= 0$, we round up or down so that $b_{52} = 0$. This is known as round to even.

Going back to the example of $x = 1/3$, the rules tell us to chop, so

$$\texttt{fl}\left(\frac{1}{3}\right) = +1.0101\ldots0101 \times 2^{-(10)_2}.$$

Exercise 2.1. Show that the double precision floating point representation of $x = 0.1$ is

$$\texttt{fl}(0.1) = 1.100110011\ldots0011010 \times 2^{-(100)_2}.$$

2.3 Rounding error

Question: How big on an error do we make when rounding?

Let's consider x to be some real number and let x_-, x_+ be the closest floating point numbers above and below x. That is,

$$x_- = \texttt{fl}(x_-) \leq x \leq \texttt{fl}(x_+) = x_+$$

We say that x_-, x_+ are *machine numbers* because they are represented exactly in floating point.

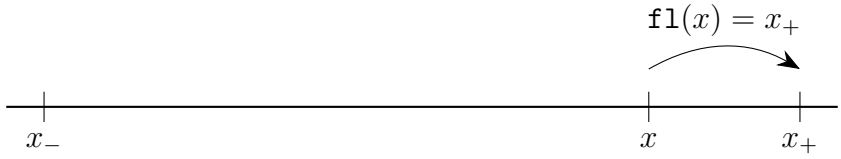


Figure 1: Rounding x to x_+.

Notice that we can write using the floating point conventions without rounding just yet:

$$\begin{aligned} x &= (1.\text{xxxxxx}\ldots) \times 2^e \\ x_+ &= (1.\text{xxxxxx1}\ldots) \times 2^e \\ x_- &= (1.\text{xxxxxx0}\ldots) \times 2^e. \end{aligned}$$

Notice that in this case the midpoint $\frac{x_+ + x_-}{2}$ is the worst case scenario and the absolute value of x satisfies $|x| \geq 2^e$. As a result, we can bound the relative rounding error

$$\underbrace{\frac{|x - \texttt{fl}(x)|}{|x|}}_{\text{called a relative error}} \leq \frac{1}{2}\frac{|x_+ - x_-|}{|x|} \leq \frac{1}{2}\frac{2^{-52} \times 2^e}{2^e} = 2^{-53}$$

The left hand side is a relative error since we divided by $|x|$, the numerator $|x - \texttt{fl}(x)|$ is an *absolute* error. In the end, we have that the largest relative rounding error we expect to make is

$$\frac{1}{2}\varepsilon = 2^{-53} \approx 1.1 \times 10^{-16}.$$

Here, ε is known as *machine epsilon* or *unit round off* for double precision floating point.

3 Floating Point and Machine Representation of Numbers Cont. (August 29, 2025)

3.1 Double Precision Representation:Practice with shifting the exponent

Recall the double precision floating point number

$$\texttt{fl}(x) = (-1)^s 1.b_1 b_2 \dots b_{52} \times 2^e$$

How is this represented in the computer? One way is to represent it as

s	$(e + 1023)_2$	b_1	b_2	b_3	$\dots$	b_{52}

Example 3.1 (some more floating point examples)**.** Let's review some examples

- $\texttt{fl}(2) = 1.000\dots \times 2^1$

0	$\underbrace{10000000000}_{=(\cdot)_2=e+1023=1024}$	0	0	0	$\dots$	0

- $\texttt{fl}(1/3) = 1.0101\ldots01 \times 2^{-2}$

0	$\underbrace{01111111101}_{=(\cdot)_2=e+1023=1021}$	0	1	$\ldots$	0	1

Exercise 3.1 ($\texttt{fl}(.1)$)**.** Show that the double precision floating point representation of $x = 0.1$ in the above table format is

0	01111111011	100110011$\ldots$0011010

3.2 Single Precision Floating Point

The book goes over single precision quite a bit, so we quickly go over single precision.

Definition 3.1 (single precision floating point)**.** Let x be a real number. Its single precision floating point representation is

$$\texttt{fl}(x) = (-1)^s \times 1.b_1 b_2 \cdots b_{23} \times 2^e$$

where the rounding rules are same as double precision. Note that the exponent is stored in 8 bits as the binary expansion of $e + 127$.

We also note that there is a machine epsilon for single precision too. Machine ε for single precision is

$$\varepsilon = 2^{-23} \approx 1.2 \times 10^{-7},$$

so the largest relative rounding error we expect to make is $2^{-24} \approx 6 \times 10^{-8}$.

3.3 Round-off Errors and Where Floating Point Can Go Wrong

We now go back to double precision. Recall from last time that we showed

$$\frac{|x - \texttt{fl}(x)|}{|x|} \leq \frac{\varepsilon}{2} \approx 1.1 \times 10^{-16}$$

so if x is not represented exactly, then we make a small error. If $x = \texttt{fl}(x)$, then we say x is a *machine number*.

3.4 Arithmetic on a Computer

Even if x, y are machine numbers, their addition may not be i.e.

$$x + y \neq \texttt{fl}(x + y)$$

In this case, we can think of the computer as doing the following

- Adjust numbers so exponents are the same (known as padding). For example if $x = 1.0000\ldots \times 2^0$, and $y = 1.00000 \times 2^{-53}$, then we write

$$\begin{aligned} x &= 1.0000\cdots 0 \times 2^0 \\ y &= 0.\underbrace{000\cdots 1}_{\text{53 bits}} \times 2^0 \end{aligned}$$

- We then add x and y.

$$x + y = 1.0000\cdots 1 \times 2^0$$

- We finally find a floating point representation (or round)

$$\texttt{fl}(x + y) = 1.000 \times 2^0 = x$$

  The error we make is due to rounding error, so we can write

$$\texttt{fl}(x + y) = (x + y)(1 + \delta) = x(1 + \delta) + y(1 + \delta)$$

  where δ is some error such that $\delta \leq \frac{\varepsilon}{2}$.

3.4.1 Error of Adding Two Numbers

Now suppose x, y are note machine numbers, i.e. $x \neq \texttt{fl}(x), y \neq \texttt{fl}(y)$. We then have the computer first round:

$$\texttt{fl}(x) = x\,(1 + \delta_1)\,, \quad \texttt{fl}(y) = y\,(1 + \delta_2)$$

and then add and round again

$$\texttt{fl}(\texttt{fl}(x) + \texttt{fl}(y)) = (\texttt{fl}(x) + \texttt{fl}(y))\,(1 + \delta_3)$$

Combining leads to

$$\begin{aligned} \texttt{fl}(\texttt{fl}(x) + \texttt{fl}(y)) &= \left[x\,(1 + \delta_1) + y\,(1 + \delta_2)\right](1 + \delta_3) \\ &= x + y + \left[\delta_1 x + \delta_2 y + \delta_3(x + y) + \underbrace{x\delta_1\delta_3 + y\delta_2\delta_3}_{\substack{\text{We drop these terms because } \delta_1\delta_3 \\ \text{is very small compared to } \delta_1 \text{ or } \delta_3.}}\right] \\ &\approx x + y + \big(\delta_1 x + \delta_2 y + \delta_3(x + y)\big) \end{aligned}$$

Hence, the relative error is approximately

$$\frac{|\texttt{fl}(\texttt{fl}(x)+\texttt{fl}(y))-(x+y)|}{|x+y|} \approx \left|\delta_3 + \frac{\delta_1 x + \delta_2 y}{x+y}\right| \tag{1}$$

where $|\delta_i| \leq \frac{\varepsilon}{2}$ for $i = 1, 2, 3$. We now go over two examples, but first we review some useful properties of absolute value.

Remark 3.1 (properties of $|\cdot|$)**.** Let x, y be real numbers. Two useful properties of absolute value are

$$\begin{aligned} |x+y| &\leq |x| + |y| && \text{(triangle inequality)} \\ |xy| &= |x|\,|y| && \text{(multiplicativity)} \end{aligned}$$

Example 3.2 (adding $.1 + .2$)**.** If we replace $x = .1$, $y = .2$ into (1), we get

$$\begin{aligned} \frac{|\texttt{fl}(\texttt{fl}(.1)+\texttt{fl}(.2))-.3|}{.3} &\approx \left|\delta_3 + \frac{\delta_1 0.1 + \delta_2 0.2}{0.3}\right| \\ &\leq |\delta_3| + \left|\frac{\delta_1 0.1 + \delta_2 0.2}{0.3}\right| \\ &\leq |\delta_3| + \frac{\delta_1}{3} + \frac{2\delta_2}{3} \\ &\leq \frac{\varepsilon}{2} + \frac{\varepsilon}{6} + \frac{2\varepsilon}{6} = \varepsilon. \end{aligned}$$

This upper bound is actually quite close to the true error. The relative error of computing $.1 + .2$ in Python is 1.85×10^{-16}, while our upper bound calculated above is 2.22×10^{-16}.

Example 3.3 (catastrophic cancelation)**.** Say we leave $x > 0$ but now set $y = -x + \eta$ for some small η. Using (1), we have

$$\begin{aligned} \frac{|\texttt{fl}(\texttt{fl}(x)+\texttt{fl}(-x+\eta))-0|}{|x|} &\approx \left|\delta_3 + \frac{\delta_1 x + \delta_2(-x+\eta)}{\eta}\right| \\ &\leq |\delta_3| + \frac{|x|}{|\eta|}\,|\delta_1| + \frac{|\eta - x|}{|\eta|}\,|\delta_2| \\ &\leq \left(1 + \frac{|x| + |\eta - x|}{|\eta|}\right)\frac{\varepsilon}{2} \end{aligned}$$

Notice that

$$\lim_{\eta \to 0}\left(1 + \frac{|x| + |\eta - x|}{|\eta|}\right)\frac{\varepsilon}{2} = \infty,$$

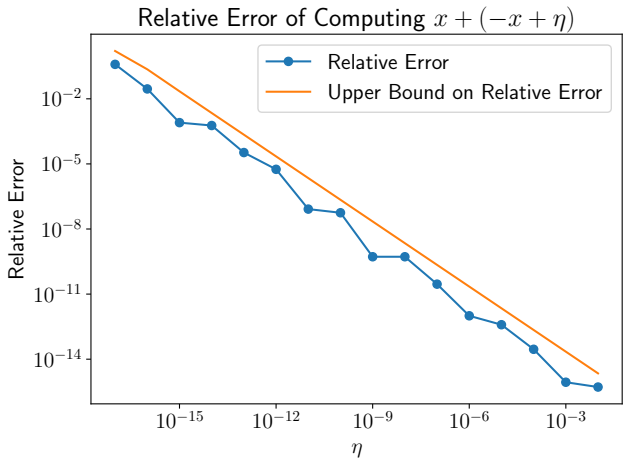


Figure 2: Loss of significance during catastrophic cancelation of computing $x+(-x+\eta)$ in double precision floating point.

which means we could potentially have very large relative error. This is known as **loss of significance**. The particular issue is we tried to subtracting two nearly equal numbers, which is known as **catastrophic cancelation**. Figure 2 is a plot of the relative error of computing $x + (-x + \eta)$ vs the upper bound obtained above for different values of η. We can see that our upper bound is quite accurate and there true is a loss of significance. Notice that even for $\eta = 10^{-7}$, we have lost almost half our digits of accuracy!

4 Round-off Errors and Computer Arithmetic Continued (September 3, 2025)

Let's recap from last time. We showed

$$\texttt{fl}(\texttt{fl}(x) + \texttt{fl}(y)) \approx x + y + \left(\delta_1 x + \delta_2 y + \delta_3 (x + y)\right),$$

so the *absolute error* is

$$\left|\texttt{fl}(\texttt{fl}(x) + \texttt{fl}(y)) - (x + y)\right| = \left|\delta_1 x + \delta_2 y + \delta_3 (x + y)\right|$$

and the *relative error* is

$$\frac{\left|\texttt{fl}(\texttt{fl}(x)+\texttt{fl}(y))-(x+y)\right|}{|x+y|}=\frac{\left|\delta_1 x+\delta_2 y+\delta_3(x+y)\right|}{|x+y|}.$$

We also showed that in the case $y=-x+\eta$, we have

$$\text{relative error}\leq\frac{\varepsilon}{2}\left(1+\frac{|\eta-x|+|x|}{|\eta|}\right),$$

so as $\eta\to 0$, we have RHS $\to\infty$. The blow up of relative error is called **loss of significance**, which arose from catastrophic cancelation when subtracting two nearly equal numbers.

4.1 Strategies to Avoid Loss of Significance

There are two strategies we'll discuss to avoid loss of significance and present examples on how they work.

4.1.1 Rationalize an quantity

Example 4.1 (quadratic formula)**.** To find x such that $ax^2+bx+c=0$, we know

$$x=\frac{-b+\sqrt{b^2-4ac}}{2a},\quad \frac{-b-\sqrt{b^2-4ac}}{2a}.$$

If $b>0$, and b is much larger than $|4ac|$, then we might experience loss of significance. The trick is to multiply by $\frac{-b-\sqrt{b^2-4ac}}{-b-\sqrt{b^2-4ac}}$:

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

Notice that the above formula does not involve subtracting two nearly equal numbers. We'll see this in a code demo.

4.1.2 Taylor series

Example 4.2 (Taylor series formula to avoid loss of significance)**.** Let $f(x) = \sin x - x$. If $x \approx 0$, then $\sin x \approx x$ and we expect some loss of significance. The trick is to write a Taylor series for $\sin x$:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots,$$

so that

$$\begin{aligned} f(x) = \sin x - x &= \cancel{x} + \cancel{x} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots, \\ &= -\frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \end{aligned}$$

There is a general procedure of when to apply Taylor series. Say we want to compute

$$f(x) - g(x).$$

- Find points z such that for $x \approx z$, we have $f(x) - g(x) \approx 0$.
- More specifically, find points z such that for $x \approx z$, we have $f(x) - g(x) \approx c(x-z)^n$ for $n > 1$. The formula $f(x) - g(x)$ is fine for x far away from z, so we don't need to modify it.
- For $x \approx z$, we apply Taylor series to f, g. Say the exponent n was 2. We have the following Taylor expansions around z

  $$\begin{aligned} f(x) &= a_0 + a_1(x-z) + a_2(x-z)^2 + a_3(x-z)^3 + \cdots \\ g(x) &= a_0 + a_1(x-z) + b_2(x-z)^2 + b_3(x-z)^3 + \cdots \end{aligned}$$

  Hence, a better formula for computing $f(x) - g(x)$ is

  $$f(x) - g(x) = (a_2 - b_2)(x-z)^2 + (a_3 - b_3)(x-z)^3 + \cdots$$

A natural question is when should we use the Taylor series formula vs the normal formula? It has to do with quantifying the loss of significance.

Theorem 4.1 (loss of significance in subtraction)**.** Let x, y be machine numbers i.e. $x = \texttt{fl}(x), y = \texttt{fl}(y)$. If

$$2^{-p} \leq 1 - \frac{y}{x} \leq 2^{-q}$$

for positive integers p, q. Then we have

$$q \leq \text{ number of bits lost when computing } \text{fl}(x-y) \leq p.$$

If we only want to lose 1 bit computing $f(x) = \sin x - x$, then we want

$$\frac{1}{2} \leq 1 - \frac{\sin x}{x}.$$

We use the Taylor expansion for $\sin x$:

$$\sin x = x - \frac{1}{6}x^3 + \cdots$$

to solve for values of x

$$\begin{aligned} \frac{1}{2} \leq 1 - \frac{\sin x}{x} &= 1 - \frac{x - \frac{1}{6}x^3 + \cdots}{x} \\ &= \frac{1}{6}x^3 + \cdots \approx \frac{1}{6}x^3 \end{aligned}$$

If we want to use the standard $\sin x - x$ and only lose 1 bit of accuracy, then we require $|x| \geq \sqrt{3} \approx 1.7$.

4.2 Other issues with floating point

Recall in double precision

$$\texttt{fl}(x) = \pm 1.b_1 b_2 \ldots b_{52} \times 2^e.$$

where the exponent e is stored in 11 bits as $e + 1023$. The range for how $e + 1023$ can be stored is

$$0 \leq e + 1023 \leq \sum_{j=0}^{10} 2^j = 1024 + 1023$$

Hence, our valid range for e is

$$-1023 \leq e \leq 1024.$$

There are two issues that can occur

- **Overflow**: If $e > 1024$, the number is too large to store in double precision, so we get overflow. Trying to put 2^{1025} in the computer when result in an overflow error.
- **Underflow**: If $e < -1023$, the number is too small to store in double precision, so we get underflow. Trying to put 2^{-1024} in the computer when result in an underflow error.

5 Solving Nonlinear Equations or Root Finding with Bisection Method (September 5, 2025)

The problem we are interested in solving is given $f : \mathbb{R} \to \mathbb{R}$, we want to find x such that

$$f(x) = 0.$$

If we wanted to solve $f(x) = c$ for some $c \neq 0$, then we can just set $g(x) = f(x) - c$ and solve $g(x) = 0$. In general, solving nonlinear equations boils down to solving for *roots* of f.

5.1 Bisection Method

Suppose f is continuous and suppose we have $a < b$ such that

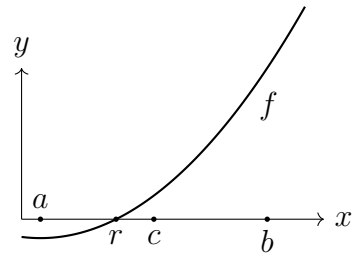
$$f(a) < 0, \quad f(b) > 0.$$

Notice that if $f(a) = 0$ or $f(b) = 0$, then we would have solved the problem.

By Intermediate Value Theorem from calculus, we have that there is a true root $a \leq r \leq b$ such that

$$f(r) = 0,$$

so we know that there is a root in the interval $[a, b]$.



Question: What is a good or natural guess for r given that we don't know where it is?

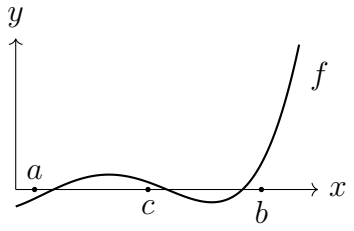
A reasonable guess would be to take the midpoint

$$c = \frac{a + b}{2}.$$

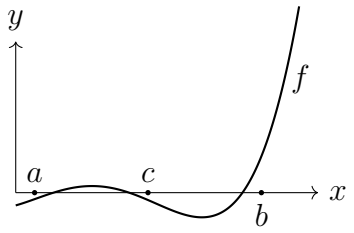
Now that we have a guess c, where do we check to figure out where to guess next? There are 3 cases to consider.

- *Case 1*: If $f(c) = 0$ or $|f(c)| < \texttt{tol}$ for some specified user tolerance. In this case, we're done.

- *Case 2*: If $f(c) > 0$, then by IVT and the fact that $f(a) < 0$, we know there is a root in $[a, c]$.



- *Case 3*: If $f(c) < 0$, then by IVT and the fact that $f(b) > 0$, we know there is a root in $[c, b]$.



Given these three cases, we have the next guesses

- *Case 1*: Next guess $d = c$
- *Case 2*: Next guess $d = \frac{a+c}{2}$
- *Case 3*: Next guess $d = \frac{c+b}{2}$.

Notice that these cases were carried out assuming $f(a) > 0$ and $f(b) < 0$. Is there an easy way to check the cases if $f(a) < 0$ and $f(b) > 0$?

Obviously *Case 1* stays the same. For *Case 2*, we just need to check that

$$f(a) \cdot f(c) < 0.$$

We can see that the above condition covers all the situations

$$\begin{aligned} f(a) < 0 \text{ and } f(c) > 0 \quad &\Rightarrow \quad f(a) \cdot f(c) < 0 \\ f(a) > 0 \text{ and } f(c) < 0 \quad &\Rightarrow \quad f(a) \cdot f(c) < 0. \end{aligned}$$

Similarly, for *Case 3*, we just need to check that

$$f(c) \cdot f(b) < 0.$$

The whole algorithm just repeats this procedure.

Input: Function $f(x)$, interval $[a, b]$, function value tolerance `ftol`, root absolute error tolerance `atol`, maximum iterations N

Output: Approximate root, c, of f guaranteed to satisfy $|f(c)| <$ `ftol` or $|c - r| <$ `atol` or returns message of failure.

if $f(a)f(b) \geq 0$ **then**
  return "Function values at endpoints must have opposite signs";
end
for $i \leftarrow 1$ **to** N **do**
  $c \leftarrow (a + b)/2$;
  if $|f(c)| <$ *ftol* *or* $\frac{|b-a|}{2} <$ *atol* **then**
    return c;
  else
    if $f(a)f(c) < 0$ **then**
      $b \leftarrow c$;
    else
      $a \leftarrow c$;
    end
  end
end
return "Method failed after N iterations";

Algorithm 1: Bisection Method

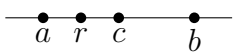
Remark 5.1 (binary search)**.** Notice that bisection method is just a continuous version of binary search.

5.2 Convergence and Error Analysis of Bisection

Notice that we have two convergence conditions of the bisection algorithm in Algorithm 1. The first $|f(c)| <$ `ftol` seems natural since we want to stop when we find an approximate root. We now explain the second condition $\frac{|b-a|}{2} <$ `atol`.

Say that $a_0 \leq r \leq b_0$ is the true root of f. Then, we have the following worst case estimate on the absolute error $|r - c_0|$ below.

$$|r - c_0| \leq \frac{b_0 - a_0}{2}.$$



This means that the length of the bracket divided by 2 is an *error estimator* of the

algorithm, which means that if $\frac{b_0-a_0}{2} < \texttt{atol}$, then $|r - c_0| < \texttt{atol}$. This is why we have the stopping condition on the bracket $\frac{b-a}{2} < \texttt{atol}$.

Continuing the algorithm, let's say that $c_1 = \frac{a_0+c_0}{2}$ is the next guess and the new bracket is $a_1 = a_0$ and $b_1 = c_0$. We can similarly estimate the error again

$$|r - c_1| \leq \frac{b_1 - a_1}{2} = \frac{c_0 - a_0}{2} = \frac{b_0 - a_0}{4} = \frac{b_0 - a_0}{2^{1+1}}$$

If we repeat the iteration, we get after n steps of the algorithm:

$$|r - c_n| \leq \frac{b_0 - a_0}{2^{n+1}}.$$

5.2.1 Plotting Error on Semilog Plot

Often to dispay the error of an algorithm is better to use a log scale for the y or x axis or both. In this case, we'll use a log scale for the y axis, which is like plotting $\log|r - c_n|$ vs n.Recall that the error is

$$|r - c_n| \leq \frac{b_0 - a_0}{2^{n+1}}.$$

Taking the log of both sides yields

$$\log|r - c_n| \leq \log(b_0 - a_0) - (n+1)\log 2,$$

so the error on a plot with log scale for the y axis and linear scale for the x axis (or semilogy plot) looks like a straight line with slope $m = -\log 2$.

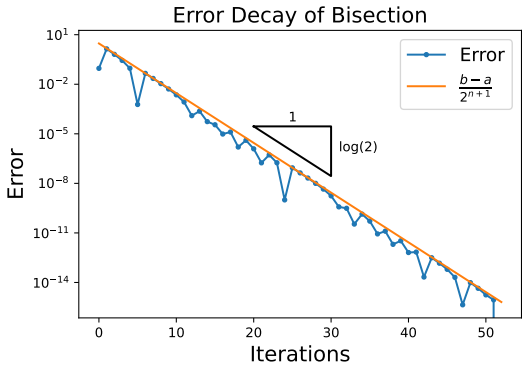
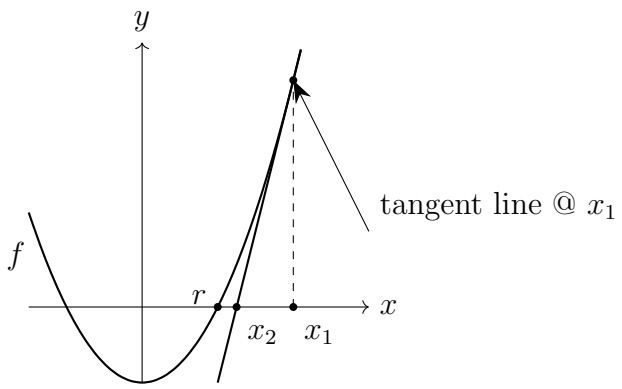


Figure 3: Error Decay of Bisection as number of iterations n increases on semilogy plot.

6 Newton's Method (September 8, 2025)

Today, we'll discuss a method that is much faster than bisection, which is called Newton's method. The main idea of this method is to replace f with a linear approximation and solve for the zero of the linear approximation.



The equation for the tangent line at x_1 is

$$y = \ell(x) = f'(x_1)(x - x_1) + f(x_1).$$

Solving for x_2 such that $\ell(x_2) = 0$, we get

$$\begin{aligned} 0 &= f'(x_1)(x_2 - x_1) + f(x_1) \\ -f(x_1) &= f'(x_1) \cdot (x_2 - x_1) \\ \frac{-f(x_1)}{f'(x_1)} &= (x_2 - x_1) \\ x_1 - \frac{f(x_1)}{f'(x_1)} &= x_2 \end{aligned}$$

As a result, the Newton iteration is the follows. Given x_i, we compute

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

6.1 Error Analysis of Newton Iteration

We still don't have an algorithm just yet. We still need to decide when to stop. However, we can still analyze the errors of the iteration. To do this, we need Taylor's theorem.

Theorem 6.1 (Taylor polynomial)**.** Suppose f has $n+1$ continuous derivatives. Then for real number x and $h > 0$, we have

$$f(x+h) = \sum_{k=0}^{n} \frac{f^{(k)}(x)}{k!} h^k + E_{n+1}$$

where $E_{n+1} = \frac{f^{(n+1)}(c)}{(n+1)!} h^{n+1}$ and $x \le c \le x+h$.

A special case that we'll use is for $n = 1$:

$$f(x+h) = f(x) + h \cdot f'(x) + \frac{f''(c)}{2} h^2.$$

Taylor's theorem is an important theoretical tool for this class, so it is very important to review it. Once we have Taylor, we are ready to prove the following theorem.

Theorem 6.2 (quadratic convergence of Newton)**.** Suppose that f has two continuous derivatives in a neighborhood of a root r. Further assume that $f'(r) \neq 0$. Then there is a $\delta > 0$ such that if $|x_0 - r| < \delta$, then the error $e_n = x_n - r$ of the Newton iteration satisfies

$$|e_{n+1}| \le \left[\frac{1}{2} \frac{\max_{|x-r| \le \delta} |f''(x)|}{\min_{|x-r| \le \delta} |f'(x)|} \right] e_n^2$$

and $e_n \to 0$ as $n \to \infty$.

Proof. We break the proof into several steps. We first begin by assuming $|x_n - r| \le \delta$ for $\delta > 0$ chosen later.
Step 1: (iteration or equation of the error e_n). We first write down an equation to describe the evolution or iteration of the error. Note that the Newton iteration is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

subtracting r from both sides leads to an equation for the error

$$e_{n+1} = x_{n+1} - r = x_n - r - \frac{f(x_n)}{f'(x_n)} = e_n - \frac{f(x_n)}{f'(x_n)}$$

The goal will be to have an error equation of the form $e_{n+1} = C e_n^{\alpha}$, so we do not want to have e_n added to something. To this end, we combine the fractions

$$e_{n+1} = e_n - \frac{f(x_n)}{f'(x_n)} = \frac{e_n f'(x_n) - f(x_n)}{f'(x_n)}.$$

Step 2: (Taylor expansions). We now apply a Taylor expansion to in the numerator. Notice that $r = x_n - (x_n - r) = r - e_n$. Then,

$$0 = f(r) = f(x_n) - f'(x_n)e_n + \frac{f''(c_n)}{2}e_n^2$$

where c_n is in between r and x_n. If we rearrange the above equation as

$$\frac{f''(c_n)}{2}e_n^2 = f'(x_n)e_n - f(x_n),$$

we notice that the RHS is the numerator of our error equation. Hence, the error equation reads

$$e_{n+1} = e_n - \frac{f(x_n)}{f'(x_n)} = \frac{f''(c_n)}{2f'(x_n)}e_n^2.$$

Since x_n and c_n are satisfy $|x_n - r|, |c_n - r| \leq \delta$, we can then write an upper bound

$$|e_{n+1}| = e_n - \frac{f(x_n)}{f'(x_n)} = \left|\frac{f''(c_n)}{2f'(x_n)}\right| e_n^2 \leq \left[\frac{1}{2}\frac{\max_{|x-r|\leq\delta}|f''(x)|}{\min_{|x-r|\leq\delta}|f'(x)|}\right] e_n^2$$

Notice that if we have the desired error estimate. We just now need to pick δ.
Step 3: (choice of δ) All we have so far is that if $|x_n - r| < \delta$, then

$$|e_{n+1}| \leq \left[\frac{1}{2}\frac{\max_{|x-r|\leq\delta}|f''(x)|}{\min_{|x-r|\leq\delta}|f'(x)|}\right] e_n^2.$$

It is possible that the error can be actually increasing if e_n is large the RHS could be ∞ if $\min_{|x-r|\leq\delta}|f'(x)| = 0$. In order to guarantee that $e_n \to 0$, we need to pick δ sufficiently small. In order to guarantee that $|e_{n+1}| \leq |e_n|$, we need to show that $\frac{|e_{n+1}|}{|e_n|}$. Using the error equation, we have

$$\frac{|e_{n+1}|}{|e_n|} \leq \left[\frac{1}{2}\frac{\max_{|x-r|\leq\delta}|f''(x)|}{\min_{|x-r|\leq\delta}|f'(x)|}\right] |e_n| \leq \left[\frac{1}{2}\frac{\max_{|x-r|\leq\delta}|f''(x)|}{\min_{|x-r|\leq\delta}|f'(x)|}\right] \delta.$$

Notice that the RHS only depends on δ. Since f is twice continuously differentiable with $f'(r) \neq 0$, we have

$$\lim_{\delta\to 0}\left[\frac{1}{2}\frac{\max_{|x-r|\leq\delta}|f''(x)|}{\min_{|x-r|\leq\delta}|f'(x)|}\right] = \frac{1}{2}\frac{|f''(r)|}{|f'(r)|},$$

Hence,

$$\lim_{\delta \to 0} \left[\frac{1}{2} \frac{\max_{|x-r| \leq \delta} |f''(x)|}{\min_{|x-r| \leq \delta} |f'(x)|} \right] \delta = \left[\frac{1}{2} \frac{|f''(r)|}{|f'(r)|} \right] \times 0 = 0.$$

Therefore, there is a $\delta > 0$ sufficiently small such that if $|e_n| < \delta$, then $|e_{n+1}| \leq |e_n| < \delta$. Hence, $e_k \to 0$ as $k \to \infty$ and we have the desired error estimate. $\square$

7 Newton's Method Continued (September 10, 2025)

Recall the Newton iteration

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

and the theorem we proved last time about the Newton iteration.

Theorem 7.1 (quadratic convergence of Newton)**.** Suppose that f has two continuous derivatives in a neighborhood of a root r. Further assume that $f'(r) \neq 0$. Then there is a $\delta > 0$ such that if $|x_0 - r| < \delta$, then the error $e_n = x_n - r$ of the Newton iteration satisfies

$$|e_{n+1}| \leq \left[\frac{1}{2} \frac{\max_{|x-r| \leq \delta} |f''(x)|}{\min_{|x-r| \leq \delta} |f'(x)|} \right] e_n^2$$

and $e_n \to 0$ as $n \to \infty$.

Remark 7.1. There are a few points to be made about the proof.

- Notice that there is a constant $C > 0$ such that

  $$|e_{n+1}| \leq C e_n^2.$$

  This is known as **quadratic convergence**. To compare with bisection, we know that error estimator $\tilde{e}_n = (b_n - a_n)/2$ satisfies

  $$\tilde{e}_{n+1} \leq \frac{1}{2} \tilde{e}_n,$$

  which is **linear convergence**.

- When an algorithm has quadratic convergence, the number of correct digits doubles every step. A quadratically converging iteration is very fast!

- The proof and theorem have a problem if $f'(r) = 0$. This issue actually shows up in practice and is called ill-conditioning and the problem of solving $f(x) = 0$ is ill-conditioned.
- Notice that we made an assumption that $|x_n - r| < \delta$, which is assuming that x_n is close enough to r. This kind of theorem is known as a **local convergence** result. Newton may not converge for all possible guesses as we'll see in the code demo.

7.1 Reading Semilogy Plots

We have the following definitions of convergence for root finding problems.

Definition 7.1 (convergence in root finding)**.** Let e_n be a sequence such that $e_n \to 0$ as $n \to \infty$. We say that

- e_n exhibits **linear convergence** if there is a $0 < c < 1$ such that

$$|e_{n+1}| \leq c|e_n|$$

- e_n exhibits **quadratic convergence** if there is a $0 < c < \infty$ such that

$$|e_{n+1}| \leq c|e_n|^2$$

- e_n exhibits **superlinear convergence** if there is a $0 < c < \infty$ and $1 < \alpha < \infty$ such that

$$|e_{n+1}| \leq c|e_n|^\alpha$$

We now repeat a similar argument If a sequence of errors is converging linearly, then we expect

$$|e_n| \leq c^n |e_0|$$

If we repeat the arguments for bisection, we see that

$$\log |e_n| \leq n \log c + \log |e_0|$$

and the error will look like a straight line with slope $\log c$ on a semilogy graph. The steeper the slope, the faster the method converges. For superlinear convergence, the graph will look like an arc bending downwards.

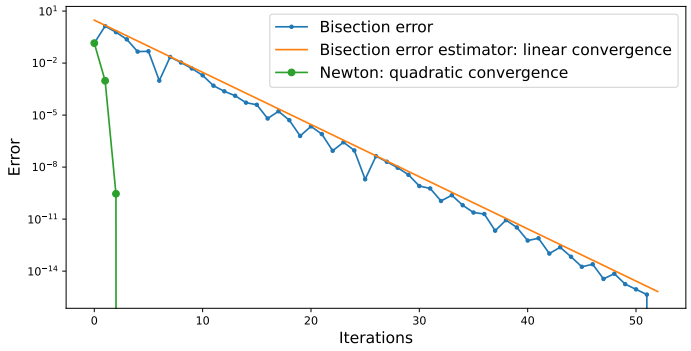


Figure 4: Convergence of Newton and bisection to solve for $\sin(x) = 0$. Newton here converges quadratically and the error estimator for bisection converges linearly.

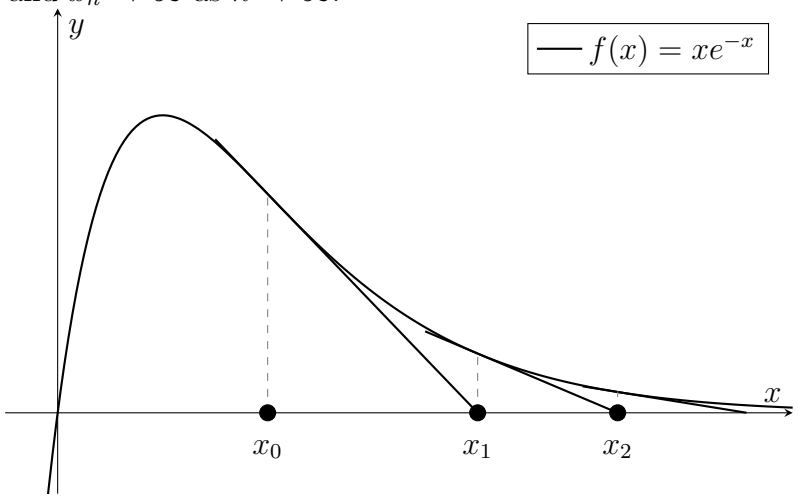
7.2 Where Newton can go wrong

There are a few failure modes for Newton method.

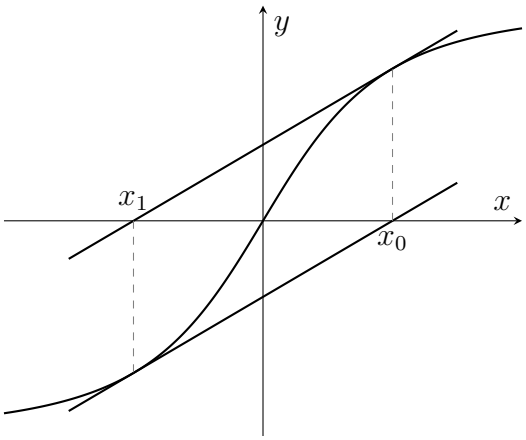
7.2.1 Initial Guess is Not Close to Root

Recall that Newton is only locally convergent. The initial guess may not be close enough to the root.

Example 7.1 (runaway)**.** Let $f(x) = xe^{-x}$. For $x_0 = 2$, we have that $x_1 = 4, x_2 \approx 5.6$ and $x_n \to \infty$ as $n \to \infty$.



Example 7.2 (cycles)**.** Here, we take $f(x) = \frac{x}{\sqrt{x^2+1}}, f'(x) = \frac{1}{(x^2+1)^{3/2}}$ and $x_0 = 1$. When then have that $x_1 = -1, x_2 = 1$ and $x_n = (-1)^n$ for all n.



7.2.2 Slow convergence to a multiple root

Newton method can also be quite slow if $f'(r) = 0$ and the previous theorem does not apply. We start with an example.

Example 7.3 ($f(x) = x^m$ for $m > 1$)**.** Let $f(x) = x^m$. We have that $f'(x) = mx^{m-1}$ There is a root $r = 0$ but $f'(r) = 0$. Consider $x_0 = 1$. We have

$$\begin{array}{llll} n = 0, & x_0 = 1, & f(x_0) = 1, & f'(x_0) = m \\ n = 1, & x_1 = 1 - \dfrac{1}{m}, & f(x_1) = \left(1 - \dfrac{1}{m}\right)^m, & f'(x_1) = m\left(1 - \dfrac{1}{m}\right)^{m-1} \end{array}$$

For $n = 2$, we have that

$$x_2 = 1 - \frac{1}{m} - \frac{(1 - \frac{1}{m})^m}{m\left(1 - \frac{1}{m}\right)^{m-1}} = \left(1 - \frac{1}{m}\right) - \frac{1}{m}\left(1 - \frac{1}{m}\right) = \left(1 - \frac{1}{m}\right)^2 = \left(\frac{m-1}{m}\right)^2.$$

Continuing the pattern, we get

$$x_n = \left(\frac{m-1}{m}\right)^n.$$

Note that

$$e_{n+1} = \left(\frac{m-1}{m}\right)^{n+1} = \left(\frac{m-1}{m}\right) e_n,$$

Here, the convergence is linear and convergence is potentially slower than bisection method! Below is are semilogy plots of the error vs iteration for Newton with $m = 3$. We can see that Newton is slower than bisection in this case.

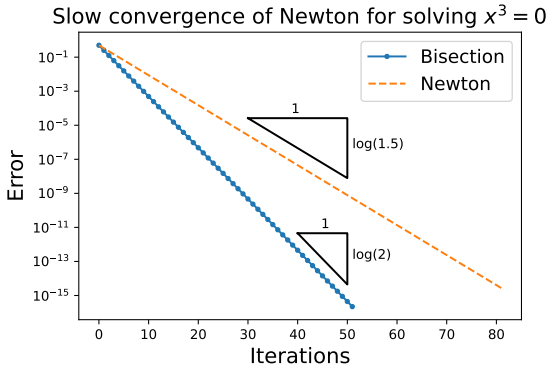


Figure 5: Convergence of Newton and bisection to solve for $x^3 = 0$. Newton here converges linearly and is slower than bisection method.

In the above example, $r = 0$ is what we call a multiple root with multiplicity m. We also say that solving $f(r) = 0$ is an ill-conditioned problem.

Definition 7.2 (multiple root)**.** A root r of f is a multiple root with multiplicity m if

$$f^{(k)}(r) = 0, \text{ for } k \leq m - 1$$
$$f^{(m)}(r) \neq 0$$

Remark 7.2 (ill-conditioned problem)**.** Solving $f(r) = 0$ with a multiple root r is known as an ill-conditioned problem. This means that small changes to inputs lead to large changes in outputs.

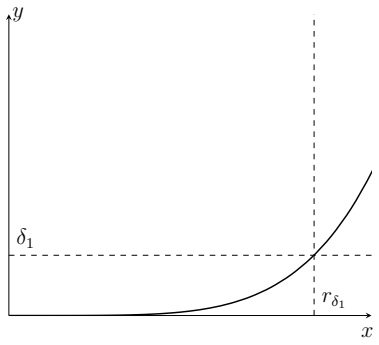
In our problem, the input is the RHS of $f(r) = \delta$ and the output is the solution r_δ. In other words, when solving $f(r_\delta) = \delta$, we compute

$$r_\delta = f^{-1}(\delta).$$

Notice that from a simple Taylor expansion, we have

$$r_\delta - r = f^{-1}(\delta) - f^{-1}(0) \approx \delta \frac{df^{-1}}{dx}(0) = \delta \left(f'(r)\right)^{-1}.$$

If r is a multiple root, then $(f'(r))^{-1}$ is undefined. Hence, perturbing our problem with δ leads to a big change in the root.



Also, bisection can sometimes struggle with ill-conditioned problems when there is floating point error.

8 Secant Method (September 12, 2025)

In this lecture, we'll cover the secant method, which is closely related of Newton. The main idea is to replace $f'(x_n)$ with

$$D_n f(x_n) = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} \approx f'(x_n).$$

The advantage of this method is we no longer need to compute the derivative f' but rather just compute f.

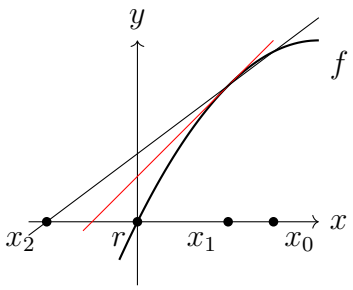


Figure 6: Secant method replaces the tangent line (red) with the secant line of the previous iterates to compute the next guess.

The above approximation is a good one because as $x_{n-1} \to x_n$, we have that $\frac{f(x_n)-f(x_{n-1})}{x_n-x_{n-1}} \to f'(x_n)$.

Like Newton, the secant iteration becomes

$$x_{n+1} = x_n - \left(\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \right) f(x_n) = x_n - \frac{f(x_n)}{D_n f(x_n)}.$$

Is this a good method? We'll see that it can be a good method when we do an error analysis.

8.1 Error Analysis of Secant

In this section, we'll prove the following proposition.

Proposition 8.1 (error inequality for secant)**.** Let r be a root of f and suppose $f'(r) \neq 0$. Let $e_n = x_n - r$, where x_n is generated from running the secant method. Suppose $|e_n| < \delta, |e_{n-1}| < \delta$ for δ sufficiently small. The error satisfies

$$|e_{n+1}| \leq \frac{\max_{|x-r|\leq\delta} |f''(x)|}{\min_{|x-r|\leq\delta} |f'(x)|} \left(\frac{1}{2} |e_n| \, |e_{n-1}| + |e_n|^2 \right).$$

Proof. We proceed as in the first two steps for Newton.
Step 1: (error equation) This procedure is the same for Newton. Recall the secant iteration

$$x_{n+1} = x_n - \left(\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \right) f(x_n) = x_n - \frac{f(x_n)}{D_n f(x_n)}.$$

Subtracting r from both sides and combining like fractions yields

$$e_{n+1} = \frac{D_n f(x_n) e_n - f(x_n)}{D_n f(x_n)}.$$

Step 2: (Taylor expansions) Here, we'd like to do a Taylor expansion like Newton. However, we no longer have the derivative. Here, we'll just add and subtract $f'(x_n)e_n$:

$$\begin{aligned} e_{n+1} &= \frac{f'(x_n)e_n - f(x_n)}{D_n f(x_n)} + \frac{D_n f(x_n) - f'(x_n)}{D_n f(x_n)} e_n \\ &= I + II \end{aligned}$$

Notice from our proof of Newton convergence, we had that there is a c_n in between x_n and r such that

$$f'(x_n)e_n - f(x_n) = \frac{1}{2} f''(c_n) e_n^2.$$

Also, Mean Value Theorem tells us there is a a_n such that

$$D_n f(x_n) = f'(a_n).$$

Hence, the first term I reads

$$I = \frac{f''(c_n)}{2f'(a_n)} e_n^2$$

and the second term now reads

$$II = \frac{D_n f(x_n) - f'(x_n)}{f'(a_n)} e_n.$$

If we stopped here, we'd have a linear convergence result (see HW 2). However, we can do better. If I center a Taylor expansion at x_{n-1}, I have

$$f(x_{n-1}) = f(x_n) - f'(x_n)(x_n - x_{n-1}) + \frac{1}{2} f''(b_n)(x_n - x_{n-1})^2$$

for some b_n in between x_{n-1} and x_n. Hence

$$D_n f(x_n) = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} = f'(x_n) - \frac{1}{2} f''(b_n)(x_n - x_{n-1})$$

and

$$II = \frac{D_n f(x_n) - f'(x_n)}{f'(a_n)} e_n = \frac{1}{2f'(a_n)} f''(b_n)(x_n - x_{n-1}) e_n = \frac{1}{2f'(a_n)} f''(b_n)(e_n - e_{n-1}) e_n.$$

Combining everything together yields

$$e_{n+1} = \frac{f''(c_n) + f''(b_n)}{2f'(a_n)} e_n^2 - \frac{1}{2} f''(b_n) e_{n-1} e_n.$$

We can then take absolute value and repeat the same upper bound procedure as with Newton. □

Say that we have that $|e_n| \leq |e_{n-1}|$, then the error satisfies

$$|e_{n+1}| \leq c|e_n||e_{n-1}|$$

for some constant c. If this is the case, then we'll actually have super linear convergence

$$|e_{n+1}| \leq C|e_n|^\alpha, \quad \alpha = \frac{1}{2}(1 + \sqrt{5})$$

We'll now summarize the convergence of secant in the following result

Theorem 8.1 (superlinear convergence of secant)**.** Let r be a root of f. Suppose f has two continuous derivatives and $f'(r) \neq 0$. Then there there is a constant $\delta > 0$ and $C > 0$ such that if $|x_0 - r| < \delta$, we have that $x_n \to r$ and secant method converges superlinearly

$$|e_{n+1}| \leq C|e_n|^\alpha, \quad \alpha = \frac{1}{2}(1 + \sqrt{5})$$

Remark 8.1 (quasi Newton methods)**.** Secant method is known as a quasi Newton method.

- quasi Newton methods replace $f'(x_n)$ with an approximation that is based on previous iterates
- Often these methods still retain desirable convergence properties like super linear convergence as seen in the example below.

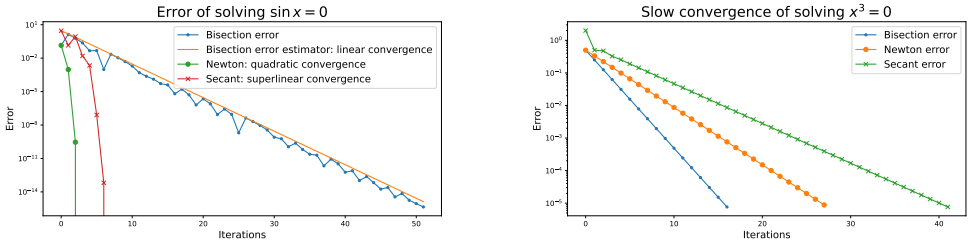


Figure 7: Convergence behavior of secant behavior for a well conditioned problem $\sin x = f(x)$ (left) and a ill-conditioned problem $x^3 = f(x)$ (right). We see that for well conditioned problems, secant converges super linearly. For ill-conditioned problems, secant suffers similarly to Newton method.

9 Fixed Point Iteration (September 15, 2025)

The last root finding method we'll consider is fixed point iteration (FPI). The goal of FPI is to find a fixed point, r, of g. In math terms, we want to solve for r such that

$$g(r) = r$$

If we want to solve $f(x) = 0$, we could write

$$g(x) = f(x) + x$$

and then solve for a fixed point of g. We'll see that there are many (both good and bad) ways of rewriting a root finding problem into a fixed point problem.

To solve the problem $g(x) = x$, the iteration is simple. Given x_n, we compute

$$x_{n+1} = g(x_n).$$

FPI is a wonderfully simple method. Pictorally, what is happening is below

The reason FPI works is that around a fixed point r the behavior of the error is

$$e_{n+1} = x_{n+1} - r = g(x_n) - g(r) \approx g'(r)e_n.$$

Hence, the error is approximately multiplied by $g'(r)$ every step and we might expect that $e_n \to 0$ if

$$|g'(r)| < 1,$$

which means g is known as a contraction mapping. We can actually write a similar theorem for FPI as we have for Newton and Secant.

Theorem 9.1 (local convergence of fixed point iteration)**.** Let g be continuous differentiable and suppose that x_n is given by a fixed point iteration. Also suppose r is a fixed point ($g(r) = r$) with $|g'(r)| < 1$. There is a $\delta > 0$ such that if $|x_0 - r| < \delta$, we have that $e_n \to 0$ and

$$|e_{n+1}| \leq \left(\max_{|x-r| \leq \delta} |g'(x)| \right) \ |e_n|.$$

We'll leave the proof as an exercise.

Exercise 9.1. Prove the above theorem.
Hint: You'll want the following Taylor expansion

$$g(x) = g(r) + g'(c)(x - r)$$

for some c in between x and r.

Remark 9.1. We note the following convergence properties of FPI.

- The convergence of FPI is linear.
- The convergence will be faster than bisection if $|g'(r)| < \frac{1}{2}$ and will be slower than bisection if $|g'(r)| > \frac{1}{2}$.

We now go over how to use FPI for a root finding problem.

Example 9.1 (FPI applied to square roots)**.** We want to compute $\sqrt{R}$ for some $R > 0$. We'll do this be solving the root finding problem

$$f(x) = x^2 - R = 0.$$

- Our first attempt to get a $g(x)$ to apply a fixed point iteration will be to divide both sides by x:

$$\begin{aligned} x^2 - R &= 0 \\ x - \frac{R}{x} &= 0 \\ g(x) = \frac{R}{x} &= x, \end{aligned}$$

  so our g is $g(x) = R/x$. Checking the derivative $g'(x) = -R/x^2$ at $\sqrt{R}$ gives

$$|g'(\sqrt{R})| = \left| \frac{R}{(\sqrt{R})^2} \right| = 1,$$

  so FPI applied to g will not necessarily converge to $\sqrt{R}$.

- Our second attempt to get a $g(x)$ to apply a fixed point iteration will be to subtract x from both sides:

$$\begin{aligned} x^2 - R &= 0 \\ x^2 - x - R &= -x \\ g(x) = -\left(x^2 - x - R\right) &= x, \end{aligned}$$

  so our g is $g(x) = -\left(x^2 - x - R\right)$. Checking the derivative $g'(x) = 1 - 2x$ at $\sqrt{R}$ gives

$$|g'(\sqrt{R})| = \left|1 - 2\sqrt{R}\right|,$$

  so FPI applied to g will converge for $0 \leq \sqrt{R} < 1$.

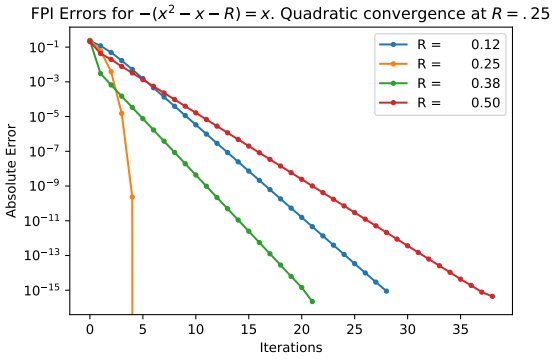


Figure 8: Error, $|e_n|$ of FPI vs n for different values of R in the above example. Note that $g'(\sqrt{R}) = 0$ for $R = 1/4$, and we observe quadratic instead of linear convergence.

Note that in the above example, we have $g'(\sqrt{R}) = 0$ for $R = 1/4$ and we see superlinear convergence for $R = 1/4$. Below is an exercise to prove it

Exercise 9.2. Let g be twice continuous differentiable. Suppose $g(r) = r$ and $g'(r) = 0$. Show that FPI is locally quadratically convergent, i.e. show that there is a $\delta > 0$ such that if $|x_0 - r| < \delta$, then $e_n \to 0$ and there is a $C > 0$ such that

$$|e_{n+1}| \leq Ce_n^2$$

9.1 Systems of Nonlinear Equations

FPI is a good example of a method that generalizes well to systems of nonlinear equations. If we want to solve for a fixed point of the vector valued function $\mathbf{g} : \mathbb{R}^d \to \mathbb{R}^d$:

$$\mathbf{g}(\mathbf{x}) = \mathbf{x},$$

then our iteration is

$$\mathbf{x}^{(n+1)} = \mathbf{g}\left(\mathbf{x}^{(n)}\right).$$

Here, the notation $\mathbf{x}^{(n)}$ means the nth iteration and does not mean a power of something. This is because I denote subscripts as the components of a vector

$$\mathbf{x} = (x_1, x_2, \cdots, x_d)$$

9.1.1 Newton's method for systems of nonlinear equations

Newton's method also generalizes to higher dimensions. Here, I present the method using the 1D method as an analogy.

Newton in 1D	Newton in Multi-D
• **Problem** Solve $f(x) = 0$.	• **Problem** Solve $\mathbf{f}(\mathbf{x}) = \mathbf{0}$.
• **Derivative** of f at x is $f'(x)$	• **Jacobian** of $\mathbf{f}$ at $\mathbf{x}$ is $\mathbf{Jf}(\mathbf{x})$
• **Reciprocal** of $f'(x)$ is $1/f'(x)$	• **Inverse** of $\mathbf{Jf}(\mathbf{x})$is $\left(\mathbf{Jf}(\mathbf{x})\right)^{-1}$
• **Newton iteration** is $x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$	• **Newton iteration** is $\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - \left(\mathbf{Jf}\left(\mathbf{x}^{(n)}\right)\right)^{-1} \mathbf{f}(\mathbf{x}^{(n)})$

Recall that the Jacobian of a function $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$ is a matrix valued function $\mathbf{Jf} : \mathbb{R}^d \to \mathbb{R}^{d\times d}$ given by

$$\mathbf{Jf}\left(\mathbf{x}^{(n)}\right) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_d} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \\ \vdots & & & \end{bmatrix}$$

or in component form is

$$\mathbf{Jf}\left(\mathbf{x}^{(n)}\right)_{ij} = \frac{\partial f_i}{\partial x_j}.$$

The -1 power in Newton iteration denotes matrix inverse. For large systems of equations, we don't compute the inverse matrix $\left(\mathbf{Jf}(\mathbf{x})\right)^{-1}$ directly. Rather, the Newton iteration is

$$\text{Solve linear system for } \mathbf{d}: \quad \left(\mathbf{Jf}\left(\mathbf{x}^{(n)}\right)\right)\mathbf{d} = -\mathbf{f}(\mathbf{x}^{(n)})$$

$$\text{Compute } \mathbf{x}^{(n+1)}: \quad \mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \mathbf{d}.$$

This motivates the study of solving systems of linear equations like in the solve step above. Next lecture looks at solving

$$\mathbf{Ax} = \mathbf{b}$$

10 Gaussian Elimination (September 17, 2025)

The goal of this lecture is the solve the system of equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

We can write this in matrix vector form as

$$\mathbf{Ax} = \mathbf{b}, \quad \mathbf{x}, \mathbf{b} \text{ in } \mathbb{R}^n, \quad \mathbf{A} \text{ in } \mathbb{R}^{n\times n}.$$

We start with a simple example

Example 10.1. Say I want to solve

$$\begin{aligned} x_1 + x_2 &= 1 \\ x_1 - x_2 &= 0 \end{aligned}$$

We'll multiply the first 1st equation by 1 and subtract it from the second equation to get

$$\begin{aligned} x_1 + x_2 &= 1 \\ -2x_2 &= -1 \end{aligned}.$$

We can now solve for x_2

$$x_2 = \frac{-1}{-2} = \frac{1}{2},$$

and also solve for x_1

$$x_1 = 1 - x_2 = \frac{1}{2}.$$

The point of this example is that this procedure of eliminating variables in equations can be generalized.

10.1 Special case: upper triangular matrix

Say I have a linear system to solve of the form

$$\mathbf{Ux} = \mathbf{b}$$

where

$$\mathbf{U} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet & \bullet \\ 0 & 0 & 0 & \bullet & \bullet \\ 0 & 0 & 0 & 0 & \bullet \end{bmatrix}$$

is an upper triangular matrix. Here, the dots $\bullet$ signify that the entry is likely nonzero. The system of equations looks like

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + \cdots + u_{1n}x_n &= b_1 \\ u_{22}x_2 + \cdots + u_{2n}x_n &= b_2 \\ &\vdots \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= b_{n-1} \\ u_{nn}x_n &= b_n \end{aligned}$$

To solve for $\mathbf{x}$, we first can easily solve for x_n

$$x_n = b_n/u_{nn}.$$

Once we know x_n we can then solve for x_{n-1}:

$$x_{n-1} = \left(b_{n-1} - u_{n-1,n}x_n\right)\frac{1}{u_{n-1,n-1}}.$$

Continuing the pattern, we have

$$x_j = \left(b_j - \sum_{i=j+1}^{n} u_{j,i}x_i\right)\frac{1}{u_{jj}}$$

and we have the following algorithm known as back substitution. This was the second step from our example.

Input: Upper triangular matrix $\mathbf{U}$, right-hand side vector $\mathbf{b}$
Output: Solution vector x

$x_n \leftarrow b_n/u_{nn}$;
for $j \leftarrow n-1$ **to** 1 **do**
  $x_j \leftarrow \frac{1}{u_{jj}}\left(b_j - \sum_{i=j+1}^{n} u_{j,i}x_i\right)$;
end

Algorithm 2: Back Substitution version 1

We can actually do slightly better by saving memory and writing the sum as a for loop to get the back substition algorithm below.

Input: Lower triangular matrix **U**, right-hand side vector **b**

Output: Solution vector **x**

$x_n \leftarrow b_n/u_{nn}$;
for $j \leftarrow n-1$ **to** 1 **do**
  $s \leftarrow 0$ **for** $i \leftarrow j+1$ **to** n **do**
    $s \leftarrow s + u_{j,i}x_i$
  end
  $x_j \leftarrow \frac{1}{u_{jj}}\big(b_j - s\big)$;
end

Algorithm 3: Back Substitution version 2 (more efficient)

10.2 Elimination step

In order to get the algorithm into the form that we need to apply back substitution, we need to do the elimination of variables that we did in the first step of the example. This is known as naive Gaussian elimination. The goal is to get the set of equations into an upper triangular form. Say we have the following structure of the matrix after eliminating the zeros below the diagonal for the first few columns

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \cdots \\ 0 & \bullet & \bullet & \bullet & \bullet & \cdots \\ 0 & 0 & a_{jj} & \bullet & \bullet & \cdots \\ 0 & 0 & 0 & \bullet & \bullet & \cdots \\ 0 & 0 & a_{ij} & \bullet & \bullet & \cdots \\ 0 & 0 & \bullet & \bullet & \bullet & \cdots \\ 0 & 0 & \bullet & \bullet & \bullet & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \cdots \end{bmatrix}$$

The goal here is to eliminate a_{ij} value to get 0 and also eliminate all other entries below the diagonal entry a_{jj}. In order to do this, we follow our example, and we need to find a multiplier p so that

$$a_{ij} - pa_{jj} = 0.$$

Solving for p gives $p = a_{ij}/a_{jj}$. We then use this quantity to subtract p times the jth row from the ith row.

$$a_{ik} \leftarrow a_{ik} - pa_{jk},$$

which results the following naive Gaussian elimination algorithm

Input: Matrix $\mathbf{A}$

Output: Overwrites $\mathbf{A}$ into upper triangular form from Gaussian elimination

```
for j ← 1 to n − 1 (loops over columns for elimination) do
    for i ← j + 1 to n (loops over rows below diagonal) do
        p ← a_ij / a_jj;
        for i ← j + 1 to n (loop over ith row) do
            a_ik ← a_ik − p a_jk
        end
    end
end
```

Algorithm 4: Gaussian elimination

Recall from the example that we also modified $\mathbf{b}$ to solve the problem. To modify the Gaussian elimination algorithm to incorporate $\mathbf{b}$, we just need to add one line to modify $\mathbf{b}$ and the back substitution step.

Input: Matrix $\mathbf{A}$, right hand side vector $\mathbf{b}$

Output: Solution $\mathbf{x}$ to problem $\mathbf{Ax} = \mathbf{b}$

```
for j ← 1 to n − 1 (loops over columns for elimination) do
    for i ← j + 1 to n (loops over rows below diagonal) do
        p ← a_ij / a_jj;
        b_j ← b_j − p b_i;
        for i ← j + 1 to n (loop over ith row) do
            a_ik ← a_ik − p a_jk;
        end
    end
end
x ← back-substitution(A, b);
```

Algorithm 5: Gaussian elimination with back substitution to solve $\mathbf{Ax} = \mathbf{b}$

10.3 Operation count of Gaussian elimination

How many operations does Gaussian elimination (Algorithm 4) take? We count this in terms of Floating Point Operations (FLOPS). This because the FLOPS are what dominate the cost of the algorithm. We go with the inner most loop and work our way out.

- Computing $a_{ik} \leftarrow a_{ik} - pa_{jk}$ takes 2 FLOPS.
- The above computation is repeated $n - j + 1$ times and then we add 1 FLOP from

$p \leftarrow a_{ij}/a_{jj}$. Hence,

$$\text{cost of inside of } i \text{ loop} = 2(n-j+1)+1 \text{ FLOPS}$$

- The ith loop goes from $i = j+1, \ldots, n$, so the above computation is repeated $n-j$ times. The inner portion of the j loop costs

$$\text{cost of inside of } j \text{ loop} = (n-j)(2(n-j+1)+1) \text{ FLOPS}$$

- Finally, we have the loop from $j = 1, \ldots, n-1$, so the total cost of the algorithm is

$$\begin{aligned}
\text{total cost} &= \sum_{j=1}^{n-1}(n-j)(2(n-j+1)+1) \\
&= \sum_{j=1}^{n-1} j(2(j+1)+1) = \sum_{j=1}^{n-1}(2j^2+3j) \\
&= \frac{2(n-1)n(2n-1)}{6} + \frac{3(n-1)n}{2} \\
&= \frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n
\end{aligned}$$

Notice that the leading order term will dominate the cost of the computation, so we can drop the n^2 and n terms. Thus,

$$\text{total cost} \approx \frac{2}{3}n^3 = O(n^3) \text{ FLOPS.}$$

The notation $O(n^3)$ is a way to say that the cost grows like Cn^3 for some constant C. We say that the cost is big-O of n^3 since the algorithm grows like n^3 as $n \to \infty$. This means that if we double the size of our matrix, then Gaussian elimination takes 8x more FLOPS.

Exercise 10.1 (operation count of back substitution)**.** Check that

$$\text{cost of back substitution} \approx O(n^2) \text{ FLOPS.}$$

10.3.1 Reading log-log plots

Often in order to check the computational cost of an algorithm, we time how long it takes for the algorithm to run for different problem sizes. Suppose that the algorithm costs

$$\text{cost} = Cn^3$$

for some $C > 0$. Taking the log of both sides, we have

$$\log(\text{cost}) = \log C + 3 \log n,$$

so if we plot $\log(\text{cost})$ vs $\log n$, we expect a straight line of slope 3. Similarly, if

$$\text{cost} = O(n^p)$$

for some power p, we can expect

$$\log(\text{cost}) \approx p \log n,$$

and we can expect that a plot of $\log(\text{cost})$ vs $\log n$, we shows a straight line of slope p. Below is such a plot for Gaussian elimination (Algorithm 5).

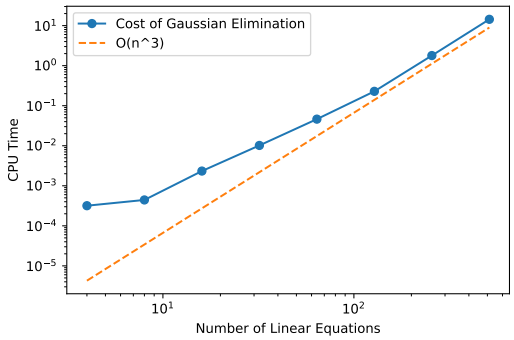


Figure 9: Time to run Algorithm 5 vs n on log-log scale. Notice that the cost follows a straight line of slope 3, which is predicted by our operation counting.

11 LU factorization and Floating Point (September 19, 2025)

We begin lecture by applying the naive Gaussian elimination algorithm from last time to the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 2 \\ 1 & 4 & 5 \\ 3 & 2 & 1 \end{bmatrix}$$

First step is multiply row 1 by 1 and subtract. Notice that this can be written as a matrix multiplication

$$\begin{bmatrix} 1 & 2 & 2 \\ 0 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{A}$$

The second step is to multiply row 1 by 3 and subtract, we again can write this as a matrix multiplication

$$\begin{bmatrix} 1 & 2 & 2 \\ 0 & 2 & 3 \\ 0 & -4 & -5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{A}$$

Finally, the third step is to multiply row 2 by -2 and subtract, which yields the following equation

$$\begin{aligned} \underbrace{\begin{bmatrix} 1 & 2 & 2 \\ 0 & 2 & 3 \\ 0 & 0 & 1 \end{bmatrix}}_{=:\mathbf{U}} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{A} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{A} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -5 & 2 & 1 \end{bmatrix} \mathbf{A} \end{aligned}$$

Notice that the matrix multiplying $\mathbf{A}$ is a lower triangular matrix:

$$\mathbf{L}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -5 & 2 & 1 \end{bmatrix}$$

where

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

Hence, we have a factorization of $\mathbf{A}$ of

$$\mathbf{LU} = \mathbf{A}$$

where $\mathbf{L}$ is lower triangular and $\mathbf{U}$ is upper triangular. Notice that the lower triangular entries of $\mathbf{L}$ are the multipliers from Gaussian elimination. Hence, the Gaussian elimination algorithm provides a factorization of the matrix $\mathbf{A}$. Below is the algorithm for $\mathbf{LU}$ factorization.

Input: Matrix $\mathbf{A}$
Output: Returns triangular matrices $\mathbf{L}, \mathbf{U}$ such that $\mathbf{LU} = \mathbf{A}$.

Initialize $\mathbf{L} \leftarrow \mathbf{I}$;
for $j \leftarrow 1$ **to** $n - 1$ *(loops over columns for elimination)* **do**
  for $i \leftarrow j + 1$ **to** n *(loops over rows below diagonal)* **do**
    $l_{ij} \leftarrow a_{ij}/a_{jj}$;
    for $i \leftarrow j + 1$ **to** n *(loop over ith row)* **do**
      $a_{ik} \leftarrow a_{ik} - pa_{jk}$
    end
  end
end
$\mathbf{U} \leftarrow \mathbf{A}$;
return $(\mathbf{L}, \mathbf{U})$;

Algorithm 6: LU factorization algorithm.

11.1 Computational Cost Advantage of LU Factorization

Suppose I want to solve the problem

$$\mathbf{LUx} = \mathbf{b},$$

where $\mathbf{L}, \mathbf{U}$ are lower and upper triangular matrices. If we multiply $\mathbf{A} = \mathbf{LU}$ and solve using Gaussian elimination we have an $O(n^3)$ algorithm. However, we can actually solve by

- Solve $\mathbf{Lz} = \mathbf{b}$ using forward substitution
- Solve $\mathbf{Ux} = \mathbf{b}$ using backward substitution

Here, forward substitution is just the cousin of backward substitution for lower triangular matrices. Both algorithms are cost $O(n^2)$. Hence, solving this way can be much cheaper if we have the factorization. I leave deriving a forward substitution algorithm as an exercise.

Exercise 11.1 (forward substitution)**.** Derive a forward substitution algorithm to solve

$$\mathbf{Lz} = \mathbf{b}$$

where $\mathbf{L}$ is a lower triangular matrix. Show that your algorithm costs $O(n^2)$ FLOPs.

The advantage of $\mathbf{LU}$ factorization comes in when we want to solve

$$\mathbf{Ax}^{(i)} = \mathbf{b}^{(i)}, \quad i = 1, \ldots, m$$

where m is a large number. This kind of problem comes up in numerically solving differential equations. If we perform Gaussian elimination for each i, our algorithm would cost $O(mn^3)$ FLOPs. If we first perform $\mathbf{A} = \mathbf{LU}$ and solve using the $\mathbf{LU}$ factorization, this algorithm would cost $O(n^3 + mn^2)$ FLOPs. This can make a big difference when m is large like simulating a differential equation for a long time.

11.2 Floating Point Error on Gaussian Elimination

We have so far ignored floating point error in Gaussian elimination. It can be a big issue! We'll see this in an example.

Example 11.1 (naive Gaussian elimination and floating point error)**.** Consider the problem

$$\begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Notice that Gaussian elimination would have to divide by 0 if $\varepsilon = 0$. We'll see that there is an issue when ε is small. We first do our Gaussian elimination step

$$\begin{bmatrix} \varepsilon & 1 \\ 0 & 1 - \frac{1}{\varepsilon} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 - \frac{1}{\varepsilon} \end{bmatrix}.$$

We now apply back substitution to get x_2 and x_1

$$x_2 = \frac{2 - \frac{1}{\varepsilon}}{1 - \frac{1}{\varepsilon}}, \quad x_1 = \frac{1 - x_2}{\varepsilon}$$

Notice that $x_2 \to 1$ as $\varepsilon \to 0$. Hence if ε is small, computing x_1 involves subtracting two nearly equal numbers and we have a catastrophic cancelation. To add insult to injury, we divide by ε and amplify the floating point error!

We can fix this example by swapping rows

Example 11.2 (swapping rows avoids bad floating point errors)**.** We now swap rows

$$\begin{bmatrix} 1 & 1 \\ \varepsilon & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

and do the first step of Gaussian elimination

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 - \varepsilon \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 - 2\varepsilon \end{bmatrix}.$$

When we do back substitution, we get

$$x_2 = \frac{1 - 2\varepsilon}{1 - \varepsilon}, \quad x_1 = 2 - x_2.$$

Notice that we avoid catastrophic cancelation and are no longer dividing by ε.

This solution of swapping rows is known as **partial pivoting**. We'll discuss in more detail next lecture.

12 Partial Pivoting (September 22, 2025)

Recall from last time that in order to solve

$$\begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

and avoid additional error from floating point rounding errors, we swapped rows and then solved

$$\begin{bmatrix} 1 & 1 \\ \varepsilon & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

This technique of swapping rows is known as partial pivoting. One question is how do we know which row to swap? Recall that we want to avoid dividing by small numbers. What we pick is the row with the largest absolute value as our pivot value, which leads to the algorithm below.

Input: Matrix $\mathbf{A}$, right hand side vector $\mathbf{b}$
Output: Solution $\mathbf{x}$ to problem $\mathbf{Ax} = \mathbf{b}$

for $j \leftarrow 1$ **to** $n-1$ *(loops over columns for elimination)* **do**
  $j^* \leftarrow \text{argmax}_{i \geq j} |a_{ij}|$;
  `swap-rows`$(a_{j,:}, a_{j^*,:})$;
  `swap`(b_j, b_{j^*});
  for $i \leftarrow j+1$ **to** n *(loops over rows below diagonal)* **do**
    $p \leftarrow a_{ij}/a_{jj}$;
    $b_j \leftarrow b_j - pb_i$;
    for $i \leftarrow j+1$ **to** n *(loop over ith row)* **do**
      $a_{ik} \leftarrow a_{ik} - pa_{jk}$;
    end
  end
end
$\mathbf{x} \leftarrow$ `back-substitution`$(\mathbf{A}, \mathbf{b})$;

Algorithm 7: Gaussian elimination with partial pivoting with back substitution to solve $\mathbf{Ax} = \mathbf{b}$

A few notes about the algorithm are in order.

- The notation $\text{argmax}_{i \geq j} |a_{ij}|$ means to find the index that corresponds to the largest value of $|a_{ij}|$ for $i \geq j$. This is finding the largest term that we can use to pivot off of.
- $a_{j,:}$ is denoting the whole jth row of $\mathbf{A}$.
- `swap-rows` means to swap the two rows in memory, likewise for `swap`.

12.1 Partial Pivoting and LU Factorization

Since $\mathbf{LU}$ factorization provides a convenient way to resolve $\mathbf{Ax} = \mathbf{b}$ if a new $\mathbf{b}$ arises, a natural question is how does partial pivoting impact the factorization? We first go back to our example of

$$\mathbf{A} = \begin{bmatrix} \varepsilon & 1 \\ 1 & 1 \end{bmatrix}$$

Swapping rows can be written as

$$\begin{bmatrix} 1 & 1 \\ \varepsilon & 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}}_{\mathbf{P}_1} \mathbf{A}.$$

where $\mathbf{P}_1$ is a permutation matrix. Applying an elimination step is the same as multiplying by a lower triangular matrix $\mathbf{L}_1^{-1}$:

$$\begin{bmatrix} 1 & 1 \\ 0 & 1-\varepsilon \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 \\ -\varepsilon & 0 \end{bmatrix}}_{=\mathbf{L}_1^{-1}=\mathbf{L}^{-1}} \mathbf{P}_1\mathbf{A}.$$

We are then left with the factorization

$$\mathbf{PA} = \mathbf{LU}$$

In general, if $\mathbf{A}$ were an $n \times n$ matrix, the above procedure we outlined would result in

$$\mathbf{U} = \mathbf{L}_{n-1}^{-1}\mathbf{P}_{n-1} \cdots \mathbf{L}_2^{-1}\mathbf{P}_2\mathbf{L}_1^{-1}\mathbf{P}_1\mathbf{A}$$

for lower triangular matrices $\mathbf{L}_i$ and permutation matrices $\mathbf{P}_i$. It takes a lot of more work, but one can show that there is a permutation matrix $\mathbf{P}$ and triangular matrices $\mathbf{L}, \mathbf{U}$ such that

$$\mathbf{PA} = \mathbf{LU},$$

and we again have a way to cheaply solve $\mathbf{Ax} = \mathbf{b}$ once we have factorized.

12.2 Partial Pivoting and Floating Point Error

The original motivation for partial pivoting was to avoid unnecessary amplification of floating point error. We now address whether it holds in general.

Before we get into partial pivoting, we go over an important concept in linear algebra called **condition number**. Let us suppose that we have a solution $\mathbf{x}$ to

$$\mathbf{Ax} = \mathbf{b}.$$

Suppose $\hat{\mathbf{x}}$ is our numerical approximation to $\mathbf{x}$ and it satisfies

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

Another way to view the above equation is that $\hat{\mathbf{b}}$ is $\mathbf{b}$ plus additional errors due to floating point, measurement noise, etc. We now look at the ratio between

$$\text{relative forward error} = \frac{\|\mathbf{x} - \hat{\mathbf{x}}\|}{\|\mathbf{x}\|}$$

and

$$\text{relative backward error} = \frac{\|\mathbf{b} - \hat{\mathbf{b}}\|}{\|\mathbf{b}\|}.$$

Here, we are using the double bar notation to mean the typical length of a vector:

$$\|\mathbf{x}\| = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

This ratio of forward and backward errors can be viewed as how errors in $\mathbf{b}$ get amplified by solving $\mathbf{Ax} = \mathbf{b}$. We have

$$\frac{\text{relative forward error}}{\text{relative backward error}} = \frac{\frac{\|\mathbf{x}-\hat{\mathbf{x}}\|}{\|\mathbf{x}\|}}{\frac{\|\mathbf{b}-\hat{\mathbf{b}}\|}{\|\mathbf{b}\|}} = \frac{\|\mathbf{x} - \hat{\mathbf{x}}\| \, \|\mathbf{b}\|}{\|\mathbf{x}\| \, \|\mathbf{b} - \hat{\mathbf{b}}\|}$$

In the numerator, we write $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}, \hat{\mathbf{x}} = \mathbf{A}^{-1}\hat{\mathbf{b}}$, and $\mathbf{b} = \mathbf{Ax}$ to get

$$\begin{aligned}\frac{\text{relative forward error}}{\text{relative backward error}} = \frac{\|\mathbf{A}^{-1}(\mathbf{b} - \hat{\mathbf{b}})\| \, \|\mathbf{Ax}\|}{\|\mathbf{b} - \hat{\mathbf{b}}\| \, \|\mathbf{x}\|} &\leq \left(\max_{\mathbf{z} \neq 0} \frac{\|\mathbf{A}^{-1}\mathbf{z}\|}{\|\mathbf{z}\|} \right) \left(\max_{\mathbf{z} \neq 0} \frac{\|\mathbf{Az}\|}{\|\mathbf{z}\|} \right) \\ &= \|\mathbf{A}^{-1}\| \, \|\mathbf{A}\|\end{aligned}$$

In the above calculation, I used an object called a **matrix norm**. I define it below.

Definition 12.1 (matrix norm)**.** Let $\|\cdot\|$ be some vector norm. The corresponding matrix norm is

$$\|\mathbf{A}\| = \max_{\mathbf{x} \neq \mathbf{0}} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|}$$

Remark 12.1 (singular value)**.** For the case of the standard vector norm $\|\mathbf{x}\| = \sqrt{\sum_{i=1}^{n} x_i^2}$, we have that $\|\mathbf{A}\| = \sigma_{max}(\mathbf{A})$, where σ_{max} denotes the largest singular value.

What we have so far is

$$\frac{\text{relative forward error}}{\text{relative backward error}} \leq \|\mathbf{A}^{-1}\| \, \|\mathbf{A}\|,$$

so $\|\mathbf{A}^{-1}\| \, \|\mathbf{A}\|$ is the maximum relative error amplification factor that could come from solving $\mathbf{Ax} = \mathbf{b}$. This quantity is called the condition number of a matrix.

Definition 12.2 (condition number of a matrix)**.** Let $\|\cdot\|$ be a vector norm. We say that

$$\kappa(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}\|$$

is the condition number of $\mathbf{A}$ with respect to the vector norm $\|\cdot\|$.

Remark 12.2 (condition number in terms of singular values)**.** For the case of the standard vector norm $\|\mathbf{x}\| = \sqrt{\sum_{i=1}^{n} x_i^2}$, we have that the condition number of a matrix $\mathbf{A}$ is

$$\kappa(\mathbf{A}) = \frac{\sigma_{max}(\mathbf{A})}{\sigma_{min}(\mathbf{A})}$$

where $\sigma_{max}, \sigma_{min}$ are the largest and smallest singular values of $\mathbf{A}$.

Our above computations show that

$$\frac{\text{relative forward error}}{\text{relative backward error}} \leq \kappa(\mathbf{A}).$$

We now discuss a few consequences of this fact.

Remark 12.3 (condition number and floating point)**.** The condition number tells us the best an algorithm could possibly do when solving $\mathbf{Ax} = \mathbf{b}$. In the presence of floating point and rounding $\mathbf{b}$ to $\hat{\mathbf{b}}$, we have

$$\frac{\|\mathbf{b} - \hat{\mathbf{b}}\|}{\|\mathbf{b}\|} \leq \frac{\varepsilon}{2}$$

where ε is machine ε. If our algorithm solved $\mathbf{A}\hat{\mathbf{x}} = \hat{\mathbf{b}}$ exactly, we still would expect

$$\frac{\|\hat{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|} \leq \kappa(\mathbf{A})\frac{\varepsilon}{2},$$

so the condition number is a limit on how well our algorithm can do in the presence of floating point.

The next theorem says that Gaussian elimination with partial pivoting achieves this theoretical best for most matrices.

Theorem 12.1 (partial pivoting and floating point)**.** Gaussian elimination with partial pivoting achieves this theoretical best for most matrices in the following sense.

- Gaussian elimination with partial pivoting makes the relative backward error small for most matrices.
- Consequently, for most matrices and in the presence of rounding $\mathbf{A}, \mathbf{b}$ to machine precision, Gaussian elimination with partial pivoting computes an approximate solution $\hat{\mathbf{x}}$ with

$$\text{number of correct digits} = |\log \varepsilon| - \kappa(\mathbf{A})$$

  where ε is machine ε.

Remark 12.4 (ill-conditioned matrices)**.** We say a matrix $\mathbf{A}$ is ill-conditioned if $\kappa(\mathbf{A})$ is large. For instance, if $\kappa(\mathbf{A}) \approx 10^d$, we then expect to lose d digits of precision. Also, $\kappa(\mathbf{A})$ is a measure of how close a matrix is to singular or not invertible. As $\kappa(\mathbf{A}) \to \infty$, the matrix $\mathbf{A}$ becomes harder to invert.

13 Symmetric Positive Definite Matrices and Cholesky Factorization (September 24, 2025)

Today we look at a special class of matrices called symmetric positive definite matrices (SPD) and a useful factorization.

Definition 13.1 (symmetric positive definite matrix)**.** A matrix $\mathbf{A}$ is symmetric positive definite (SPD) if it satisfies

- symmetry $\mathbf{A} = \mathbf{A}^T$
- positive definiteness $\mathbf{x} \cdot \mathbf{A}\mathbf{x} > 0$ for all $\mathbf{x} \neq 0$.

Recall that any symmetric matrix has real eigenvalues. For SPD matrices, we know that the eigenvalues are positive.

Proposition 13.1. Let $A = A^T$. A is SPD if and only if every eigenvalue of $\mathbf{A}$ satisfies $\lambda > 0$.

The eigenvalues can be very useful. Recall the vector and matrix norm from last time.

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}, \quad \|\mathbf{A}\|_2 = \max_{\mathbf{x}\neq 0} \frac{\|\mathbf{A}\mathbf{x}\|_2}{\|\mathbf{x}\|_2}$$

which is a specific case of a more general vector norm the so-called p-norm:

$$\|\mathbf{x}\|_p = \begin{cases} \left(\sum_{i=1}^n |x_i|^p\right)^{1/p}, & 1 \le p < \infty \\ \max_{1\le i\le n} |x_i|, & p = \infty \end{cases}.$$

For SPD matrices, we have specific characterizations of the matrix norm and condition number.

Proposition 13.2. Let $\mathbf{A}$ be an SPD matrix. Then

- the matrix norm satisfies

$$\|\mathbf{A}\|_2 = \lambda_{max}(\mathbf{A}) = \text{ maximum eigenvalue of } \mathbf{A}$$

- and the condition number is

$$\kappa_2(\mathbf{A}) = \|\mathbf{A}\|_2\|\mathbf{A}^{-1}\|_2 = \frac{\lambda_{max}(\mathbf{A})}{\lambda_{min}(\mathbf{A})}.$$

Proof. Left as an exercise but here is a hint. Recall that any symmetric matrix can be diagonalized as

$$\mathbf{A} = \mathbf{Q}^T\mathbf{\Lambda}\mathbf{Q}$$

where

$$\mathbf{Q}^T\mathbf{Q} = \mathbf{Q}\mathbf{Q}^T = \mathbf{I}.$$

In other words, the eigenvectors of $\mathbf{A}$ or orthogonal ($\mathbf{v}^{(i)} \cdot \mathbf{v}^{(j)} = 0$ for eigenvectors $\mathbf{v}^{(i)}, \mathbf{v}^{(j)}$ and $i \neq j$.) □

13.1 Cholesky Factorization

Just like $\mathbf{A} = \mathbf{LU}$, we can decompose an SPD matrix as

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T$$

which is similar to the $\mathbf{LU}$ factorization. This factorization uses the symmetry of the matrix and allows us to save memory.

Let's first consider a 2×2 example

$$A = \begin{bmatrix} a & b \\ b & c \end{bmatrix}$$

We know that

$$\begin{pmatrix}1\\0\end{pmatrix} \cdot \mathbf{A}\begin{pmatrix}1\\0\end{pmatrix} = a > 0$$

and

$$ac - b^2 = \det \mathbf{A} = \lambda_1 \cdot \lambda_2 > 0$$

since $\mathbf{A}$ is SPD. To compute the factorization, we'd like to write

$$\mathbf{A} = \begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} \\ 0 & l_{22} \end{bmatrix} = \begin{bmatrix} l_{11}^2 & l_{11}l_{21} \\ l_{11}l_{21} & l_{21}^2 + l_{22}^2 \end{bmatrix}$$

Solving for $\mathbf{L}$, we have

$$\begin{aligned} l_{11}^2 = a &\implies l_{11} = \sqrt{a}, \\ b = l_{11}l_{21} &\implies l_{21} = b/\sqrt{a}, \\ l_{21}^2 + l_{22}^2 = c &\implies l_{22} = \sqrt{c - b^2/a}. \end{aligned}$$

all of these are well defined because $\mathbf{A}$ is SPD. At least for 2×2 SPD matrices, we have $\mathbf{A} = \mathbf{L}\mathbf{L}^T$.

What about for $n \times n$ matrices? Let

$$\mathbf{A} = \left[\begin{array}{c|c} a_{11} & \mathbf{b}^T \\ \hline \mathbf{b} & \mathbf{C} \end{array}\right]$$

and

$$\mathbf{L} = \left[\begin{array}{c|c} l_{11} & \mathbf{0} \\ \hline \mathbf{v} & \tilde{\mathbf{L}} \end{array}\right]$$

We now compute $\mathbf{L}\mathbf{L}^T$:

$$\begin{aligned} \mathbf{A} &= \mathbf{L}\mathbf{L}^T \\ &= \left[\begin{array}{c|c} l_{11} & \mathbf{0}^T \\ \hline \mathbf{v} & \tilde{\mathbf{L}} \end{array}\right] \left[\begin{array}{c|c} l_{11} & \mathbf{v}^T \\ \hline \mathbf{0} & \tilde{\mathbf{L}}^T \end{array}\right] = \left[\begin{array}{c|c} l_{11}^2 & l_{11}\mathbf{v}^T \\ \hline l_{11}\mathbf{v} & \mathbf{v}\mathbf{v}^T + \tilde{\mathbf{L}}\tilde{\mathbf{L}}^T \end{array}\right] \end{aligned}$$

This multiplication gives us the following system of equations:

$$\begin{aligned} a_{11} = l_{11}^2, \quad \mathbf{b} = l_{11}\mathbf{v} \\ \mathbf{C} = \mathbf{v}\mathbf{v}^T + \tilde{\mathbf{L}}\tilde{\mathbf{L}}^T \end{aligned}$$

and we solve as

$$l_{11} = \sqrt{a_{11}}, \quad \mathbf{v} = \mathbf{b}/\sqrt{a_{11}}$$
$$\tilde{\mathbf{L}}\tilde{\mathbf{L}}^T = \mathbf{C} - \mathbf{v}\mathbf{v}^T$$

We have solve for everything except the lower triangular $(n-1) \times (n-1)$ matrix $\tilde{\mathbf{L}}$. We leave it as an exercise to check that $\mathbf{C} - \mathbf{v}\mathbf{v}^T$ is SPD. Once we know the RHS is SPD, we can actually recursively call Cholesky and compute

$$\tilde{\mathbf{L}} = \texttt{cholesky}\left(\mathbf{C} - \mathbf{v}\mathbf{v}^T\right)$$

where `cholesky` is the cholesky algorithm we are trying to develop. This argument then leads to the following recursive algorithm for Cholesky factorization. Here's an algorithm for Cholesky factorization using induction:

Input: Symmetric positive definite matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$
Output: Lower triangular matrix $\mathbf{L}$ such that $\mathbf{A} = \mathbf{L}\mathbf{L}^T$

if $n = 2$ **then**
  $l_{11} \leftarrow \sqrt{a_{11}}$;
  $l_{21} \leftarrow a_{21}/\sqrt{a_{11}}$;
  $l_{22} \leftarrow \sqrt{a_{22} - \frac{a_{21}^2}{a_{11}}}$;
  return $\mathbf{L}$;
end
else
  $l_{11} \leftarrow \sqrt{a_{11}}$;
  $l_{2:n,1} \leftarrow a_{2:n,1}/\sqrt{a_{11}}$;
  $l_{2:n,2:n} \leftarrow \texttt{cholesky}\left(a_{2:n,2:n} - l_{2:n,1}l_{2:n,1}^T\right)$;
  return $\mathbf{L}$;
end

Algorithm 8: Cholesky Factorization (Recursive)

The notation $l_{2:n,2:n}$ denotes the submatrix of $\mathbf{L}$ that is (i, j) both ranging from 2 to n. The notation $l_{2:n,1}$ denotes the column vector of $\mathbf{L}$ in the first column and rows 2 to n.

This algorithm uses the recursive formulation we derived earlier. It breaks down the problem into smaller subproblems, solving them recursively. The base case is when the matrix is 2×2, in which case we apply the direct solution we found earlier. This algorithm will become very inefficient for large matrices. Depending on the programming language, the recursive call might make copies of the matrix and we

could see an explosion of memory usage. We leave it as an exercise to write an algorithm that does not use recursion. Here's an exercise for students to write a non-recursive Cholesky algorithm:

Exercise 13.1 (non-recursive Cholesky factorization)**.** Develop a non-recursive algorithm for Cholesky factorization of an SPD matrix $\mathbf{A}$ by adapting Algorithm 8 and writing it with nested loops instead.

Hint: Loop from $i = 1, \dots n$ and apply the first two lines of the else block before the recursuve call. Then overwrite the bottom portion of $\mathbf{A}$ with what is inside the `cholesky` function call in the third line of the else block.

14 Introduction to Iterative Methods for Linear Algebra (September 26, 2025)

We now drop the approach of directly computing $\mathbf{x}$ so that $\mathbf{Ax} = \mathbf{b}$ and now try to produce a sequence $\mathbf{x}^{(k)}$ st $\mathbf{x}^{(k)} \to \mathbf{x}$ as $k \to \infty$. Our first method is called Jacobi iteration.

14.1 Jacobi Iteration

If we have $\mathbf{Ax} = \mathbf{b}$, then

$$\sum_{j=1}^{n} a_{ij} x_j = b_i.$$

If we now isolate x_i, we have

$$a_{ii} x_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j = b_i.$$

Rearranging and dividing by a_{ii} yields

$$x_i = \underbrace{\left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j \right) a_{ii}^{-1}}_{=: \mathbf{g}(\mathbf{x})_i}.$$

We now have a fixed point problem $\mathbf{x} = \mathbf{g}(\mathbf{x})$. The Jacobi iteration is the fixed point iteration $\mathbf{x}^{(k+1)} = \mathbf{g}(\mathbf{x}^{(k)})$. More specifically, the Jacobi iteration is

$$x_i^{(k+1)} = \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} \right) a_{ii}^{-1}.$$

Input: Matrix $\mathbf{A}$, right hand side vector $\mathbf{b}$, residual tolerance `rtol`, maximum iterations `maxiter`

Output: Approximate solution $\mathbf{x}$ to problem $\mathbf{Ax} = \mathbf{b}$

Initialize $\mathbf{x}^{(0)}$ (e.g., with zeros or a guess);
$k \leftarrow 0$;
while $k <$ *maxiter* *and* $\|\mathbf{Ax}^{(k)} - \mathbf{b}\|/\|\mathbf{b}\| >$ *rtol* **do**
  for $i \leftarrow 1$ **to** n **do**
    (Loops over components of $\mathbf{x}$);
    $s \leftarrow 0$;
    (Computes sum);
    for $j \leftarrow 1$ **to** n, $j \neq i$ **do**
      $s \leftarrow s + a_{ij} x_j^{(k)}$;
    end
    $x_i^{(k+1)} \leftarrow (b_i - s)/a_{ii}$;
  end
  $k \leftarrow k + 1$;
end
$\mathbf{x} \leftarrow \mathbf{x}^{(k)}$;

Algorithm 9: Jacobi iteration to solve $\mathbf{Ax} = \mathbf{b}$

Remark 14.1 (matrix notation for Jacobi)**.** We set $\mathbf{D}$ to be the diagonal, $\mathbf{U}, \mathbf{L}$ to be the upper and lower triangular portions of $\mathbf{A}$, i.e.

$$\mathbf{A} = \begin{pmatrix} \ddots & & \mathbf{U} \\ & \mathbf{D} & \\ \mathbf{L} & & \ddots \end{pmatrix}$$

The Jacobi iteration then reads

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(\mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{x}^{(k)})$$

This form of Jacobi will be useful for the convergence analysis, while the indexed form above is more useful for computation.

Remark 14.2 (parallelization of Jacobi)**.** Note that the computation of $x_i^{(k+1)}$ only depends on $\mathbf{A}, \mathbf{b}$ and $\mathbf{x}^{(k)}$. The computation doesn't require knowledge of the other $x_j^{(k+1)}$. One advantage of this is it is easy to compute each $x_i^{(k+1)}$ as separate processes and parallelize the Jacobi iteration.

14.2 Gauss Seidel Iteration

Notice that as we loop $i = 1, \ldots, n$, we have in Jacobi

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) a_{ii}^{-1}.$$

In the sum $\sum_{j=1}^{i-1} a_{ij} x_j^{(k)}$ can be replaced with the already computed values $x_j^{(k+1)}$. Hence, we might be able to speed up the method by using these values and compute

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) a_{ii}^{-1}.$$

This new method is called **Gauss Seidel iteration**. Here's an algorithm for Gauss-Seidel iteration, following the structure of the previous Jacobi algorithm:

Input: Matrix $\mathbf{A}$, right hand side vector $\mathbf{b}$, residual tolerance `rtol`, maximum iterations `maxiter`

Output: Approximate solution $\mathbf{x}$ to problem $\mathbf{Ax} = \mathbf{b}$

Initialize $\mathbf{x}^{(0)}$ (e.g., with zeros or a guess);
$k \leftarrow 0$;
while $k <$ *maxiter* *and* $\|\mathbf{Ax}^{(k)} - \mathbf{b}\|/\|\mathbf{b}\| >$ *rtol* **do**
  for $i \leftarrow 1$ **to** n **do**
    $s \leftarrow 0$; **for** $j \leftarrow 1$ **to** n **do**
      $s \leftarrow s + a_{ij} x_j$;
    end
    $x_i \leftarrow (b_i - s)/a_{ii}$;
  end
  $k \leftarrow k + 1$;
end
$\mathbf{x} \leftarrow \mathbf{x}^{(k)}$;

Algorithm 10: Gauss-Seidel iteration to solve $\mathbf{Ax} = \mathbf{b}$

Remark 14.3 (programming Gauss Seidel and Jacobi)**.** Notice that we can just store one vector $\mathbf{x}$ in the above algorithm and the code overwrites $\mathbf{x}$ as it loops over

i. For Jacobi, we needed to have a copy of $\mathbf{x}$ in order to not overwrite it. I have made the mistake of programming Gauss Seidel when I meant to write Jacobi because of this subtle difference.

Exercise 14.1 (matrix notation for Gauss-Seidel)**.** Using the same notation as before with $\mathbf{D}$, $\mathbf{L}$, and $\mathbf{U}$, i.e.

$$\mathbf{A} = \begin{pmatrix} \ddots & & \mathbf{U} \\ & \mathbf{D} & \\ \mathbf{L} & & \ddots \end{pmatrix},$$

show that the Gauss-Seidel iteration can be written as:

$$\mathbf{x}^{(k+1)} = (\mathbf{D} + \mathbf{L})^{-1}(\mathbf{b} - \mathbf{U}\mathbf{x}^{(k)})$$

Remark 14.4 (comparison with Jacobi on parallelization)**.** Unlike Jacobi iteration, Gauss-Seidel uses the most recent values of $x_j^{(k+1)}$ as soon as they are available. This often leads to faster convergence, but it also makes the method inherently sequential and harder to parallelize compared to Jacobi iteration.

14.3 Other methods

We list two other methods but will not go over them in much detail. They are Here are the matrix forms for Successive Over Relaxation (SOR) and Richardson iteration:

14.3.1 Successive Over Relaxation (SOR)

The SOR method is a modification of the Gauss-Seidel method with an additional relaxation parameter ω. In matrix form, it can be written as:

$$\mathbf{x}^{(k+1)} = (\mathbf{D} + \omega\mathbf{L})^{-1}(\omega\mathbf{b} - (\omega\mathbf{U} + (\omega - 1)\mathbf{D})\mathbf{x}^{(k)})$$

where $0 < \omega < 2$ is the relaxation parameter. When $\omega = 1$, SOR reduces to the Gauss-Seidel method.

14.3.2 Richardson Iteration

The Richardson iteration is one of the simplest iterative methods. In matrix form, it can be expressed as:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \omega(\mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}) = (\mathbf{I} - \omega\mathbf{A})\mathbf{x}^{(k)} + \omega\mathbf{b}$$

where $\omega > 0$ is a damping parameter. The term $\mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$ is the residual at step k.

14.4 Convergence of iterative methods

Notice that all of these methods are of the form

$$\mathbf{x}^{(k+1)} = \mathbf{g}(\mathbf{x}^{(k)}) = \mathbf{G}_1\mathbf{x}^{(k)} + \mathbf{G}_2\mathbf{b}$$

for some iteration matrices $\mathbf{G}_1, \mathbf{G}_2$. We can think of each method as a fixed point iteration.

When does FPI converge? We saw from the HW that FPI converges if there is a $\rho < 1$ such that for all $\mathbf{x}, \mathbf{y}$, we have

$$\|\mathbf{g}(\mathbf{x}) - \mathbf{g}(\mathbf{y})\| \leq \rho\|\mathbf{x} - \mathbf{y}\|$$

For our iteration, we have

$$\mathbf{g}(\mathbf{x}) - \mathbf{g}(\mathbf{y}) = \mathbf{G}_1(\mathbf{x} - \mathbf{y}),$$

so we want

$$\frac{\|\mathbf{G}_1(\mathbf{x} - \mathbf{y})\|}{\|\mathbf{x} - \mathbf{y}\|} \leq \rho < 1$$

for all $\mathbf{x} \neq \mathbf{y}$. Setting $\mathbf{z} = \mathbf{x} - \mathbf{y}$, the above condition is equivalent to

$$\max_{\mathbf{z}\neq\mathbf{0}} \frac{\|\mathbf{G}_1\mathbf{z}\|}{\|\mathbf{z}\|} \leq \rho < 1$$

Recall that the LHS above is a matrix norm and our discussion could have been for any norm we put on vectors. We now state an important definition and two important results.

Definition 14.1 (spectral radius)**.** The spectral radius of a matrix $\mathbf{A}$, denoted $\rho(\mathbf{A})$, is defined as

$$\rho(\mathbf{A}) = \max\{|\lambda| : \lambda \text{ is an eigenvalue of } \mathbf{A}\}.$$

Note that in the above definition, the eigenvalues of $\mathbf{A}$ can be complex numbers.

Lemma 14.1. If $\rho(\mathbf{A}) < 1$, then there exists a matrix norm $\|\cdot\|$ such that $\|\mathbf{A}\| < 1$.

Below is the main convergence theorem that we need to study Jacobi, Gauss Seidel, Successive Over Relaxation, and Richardson method.

Theorem 14.1 (spectral radius theorem)**.** Let $\mathbf{x}^{(k+1)} = \mathbf{G}_1\mathbf{x}^{(k)} + \mathbf{G}_2\mathbf{b}$ be an iterative method for solving $\mathbf{A}\mathbf{x} = \mathbf{b}$. If $\rho(\mathbf{G}_1) < 1$, then the iteration converges to the unique solution $\mathbf{A}\mathbf{x}^* = \mathbf{b}$ for any initial guess $\mathbf{x}^{(0)}$. Moreover, if we denote the error as $\mathbf{e}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}$, we have the following linear convergence in some norm given by the previous lemma

$$\|\mathbf{e}^{(k+1)}\| \leq \rho(\mathbf{G}_1)\|\mathbf{e}^{(k)}\|$$

Below is an exercise for Jacobi to establish conditions where the theorem applies Here's an exercise on the convergence of the Jacobi method for strictly diagonally dominant matrices:

Exercise 14.2 (convergence of Jacobi for strictly diagonally dominant matrices)**.** Let $\mathbf{A}$ be a strictly diagonally dominant matrix, i.e., for each row i,

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|.$$

Show that the Jacobi iteration converges for such matrices.

Hint: Recall that for Jacobi iteration, $\mathbf{G}_1 = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$. Show that $\|\mathbf{G}_1\|_\infty < 1$.

15 Convergence of Iterative Methods for Linear Algebra (September 29, 2025)

Last time, we introduced the following iterative methods along with their matrix forms:

- **Jacobi Iteration:**
$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(\mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{x}^{(k)})$$
- **Gauss-Seidel Iteration:**
$$\mathbf{x}^{(k+1)} = (\mathbf{D} + \mathbf{L})^{-1}(\mathbf{b} - \mathbf{U}\mathbf{x}^{(k)})$$
- **Successive Over Relaxation (SOR):**
$$\mathbf{x}^{(k+1)} = (\mathbf{D} + \omega\mathbf{L})^{-1}(\omega\mathbf{b} - (\omega\mathbf{U} + (\omega - 1)\mathbf{D})\mathbf{x}^{(k)})$$
- **Richardson Iteration:**
$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \omega(\mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}) = (\mathbf{I} - \omega\mathbf{A})\mathbf{x}^{(k)} + \omega\mathbf{b}$$

Where:

- $\mathbf{D}$ is the diagonal of $\mathbf{A}$
- $\mathbf{L}$ is the strictly lower triangular part of $\mathbf{A}$

- $\mathbf{U}$ is the strictly upper triangular part of $\mathbf{A}$
- ω is a relaxation parameter (for SOR and Richardson)
- $\mathbf{I}$ is the identity matrix

Recall from last time that we showed

$\mathbf{x}^{(k+1)} = \mathbf{G}_1\mathbf{x}^{(k)} + \mathbf{G}_2\mathbf{b}$ converges if and only if $\|\mathbf{G}_1\| < 1$ for some matrix norm $\|\cdot\|$

Unlike the condition of strict diagonal dominance, it is often difficult to find an appropriate matrix norm. We can instead look at the eigenvalues of $\mathbf{G}_1$.

15.1 Spectral Radius Theorem

We now state an important definition and two important results.

Definition 15.1 (spectral radius)**.** The spectral radius of a matrix $\mathbf{A}$, denoted $\rho(\mathbf{A})$, is defined as

$$\rho(\mathbf{A}) = \max\{|\lambda| : \lambda \text{ is an eigenvalue of } \mathbf{A}\}.$$

Note that in the above definition, the eigenvalues of $\mathbf{A}$ can be complex numbers.

Lemma 15.1. If $\rho(\mathbf{A}) < 1$, then there exists a matrix norm $\|\cdot\|$ such that $\|\mathbf{A}\| < 1$.

Below is the main convergence theorem that we need to study Jacobi, Gauss Seidel, Successive Over Relaxation, and Richardson method.

Theorem 15.1 (spectral radius theorem)**.** Let $\mathbf{x}^{(k+1)} = \mathbf{G}_1\mathbf{x}^{(k)} + \mathbf{G}_2\mathbf{b}$ be an iterative method for solving $\mathbf{A}\mathbf{x} = \mathbf{b}$. If $\rho(\mathbf{G}_1) < 1$, then the iteration converges to the unique solution $\mathbf{A}\mathbf{x}^* = \mathbf{b}$ for any initial guess $\mathbf{x}^{(0)}$. Moreover, if we denote the error as $\mathbf{e}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}$, we have the following linear convergence in some norm given by the previous lemma

$$\|\mathbf{e}^{(k+1)}\| \leq \rho(\mathbf{G}_1)\|\mathbf{e}^{(k)}\|$$

The spectral radius determines the speed of convergence for iterative methods in a manner similar to fixed-point iteration in one dimension. Specifically:

Remark 15.1 (speed of iterative method)**.** The spectral radius $\rho(\mathbf{G}_1)$ determines the asymptotic rate of convergence of the iterative method. A smaller spectral radius leads to faster convergence. In particular:

- If $\rho(\mathbf{G}_1) \approx 0$, the method converges very quickly.
- If $\rho(\mathbf{G}_1) \approx 1$, the method converges slowly.

This is what we had for FPI in 1D. The speed of $x_{n+1} = g(x_n)$ to a fixed point $r = g(r)$ was determined by $g'(r)$ through $e_{n+1} \approx g'(r)e_n$.

15.1.1 Application to SPD matrices

We now look at the special case of SPD matrices $\mathbf{A}$. Recall that the eigenvalues of $\mathbf{A}$ are real and positive, and the eigenvectors of $\mathbf{A}$ are orthogonal. First, is an example of Richardson iteration.

Example 15.1 (convergence of Richardson for SPD matrices)**.** We consider $\mathbf{A}$ to be SPD. Recall that this means $\mathbf{A}$ has positive real eigenvalues. Let $\lambda_{max}, \lambda_{min}$ be the maximum and minimum eigenvalues. The Richardson iteration is

$$\mathbf{x}^{(k+1)} = (\mathbf{I} - \omega\mathbf{A})\mathbf{x}^{(k)} + \omega\mathbf{b} = \mathbf{G}_1\mathbf{x}^{(k)} + \mathbf{G}_2\mathbf{b}$$

where $\mathbf{G}_1 = (\mathbf{I} - \omega\mathbf{A})$. Notice that if $\mathbf{v}$ is an eigenvector of $\mathbf{A}$ with $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$, then

$$\mathbf{G}_1\mathbf{v} = \mathbf{v} - \omega\mathbf{A}\mathbf{v} = (1 - \omega\lambda)\mathbf{v},$$

and $(1-\omega\lambda)$ is an eigenvalue of $\mathbf{G}_1$. Notice that we then have the following inequality for the eigenvalues of $\mathbf{G}_1$ (denoted μ):

$$1 - \omega\lambda_{max} \leq \mu \leq 1 - \omega\lambda_{min},$$

so the spectral radius of $\mathbf{G}_1$ is

$$\rho(\mathbf{G}_1) = \max\{|1 - \omega\lambda_{min}|, |1 - \omega\lambda_{max}|\}.$$

Hence, Richardson iteration converges for $0 < \omega < \frac{2}{\lambda_{max}}$.

We can in fact derive an optimal value for ω that gives the fastest convergence of Richardson. It is left as an exercise below.

Exercise 15.1 (optimizing Richardson iteration convergence)**.** Consider the Richardson iteration for an SPD matrix $\mathbf{A}$ with minimum and maximum eigenvalues λ_{min} and λ_{max}, respectively.

- Find the optimal value of ω that minimizes the spectral radius $\rho(\mathbf{G}_1)$, where $\mathbf{G}_1 = \mathbf{I} - \omega\mathbf{A}$.
- Express the spectral radius $\rho(\mathbf{G}_1)$ at this optimal ω in terms of the condition number $\kappa_2(\mathbf{A}) = \lambda_{max}/\lambda_{min}$.

Hint: For part (a), consider the expression for $\rho(\mathbf{G}_1)$ derived in the example. For part (b), recall the definition of the condition number for SPD matrices.

For Richardson, the choice of ω is crucial for convergence. For SPD matrices, SOR converges for many choices of ω.

Theorem 15.2 (convergence of SOR for SPD matrices)**.** Let $\mathbf{A}$ be a symmetric positive definite matrix. Then the Successive Over Relaxation (SOR) method converges for any relaxation parameter ω satisfying $0 < \omega < 2$.

Corollary 15.1 (convergence of Gauss Seidel for SPD matrices)**.** Let $\mathbf{A}$ be a symmetric positive definite matrix. Then the Gauss Seidel method converges.

16 Descent Methods for SPD Matrices (October 3-6, 2025)

The last set of iterative methods we will look at for SPD matrices, $\mathbf{A}$, are what are known as descent methods. These methods seek to minimize

$$\phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}.$$

Exercise 16.1. Show the following using calculus

- $\nabla\phi(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}$
- Show that minimizing ϕ means we find an $\mathbf{x}$ such that $\nabla\phi(\mathbf{x}) = \mathbf{0}$, which is equivalent to $\mathbf{A}\mathbf{x} = \mathbf{b}$.

A descent method chooses a descent direction $\mathbf{d}^{(k)}$ and step size α_k and sets

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{d}^{(k)}.$$

The direction is called a descent direction if

$$\mathbf{d}^{(k)} \cdot \nabla\phi(\mathbf{x}^{(k)}) \leq 0,$$

which means we are looking in a direction that reduces ϕ.

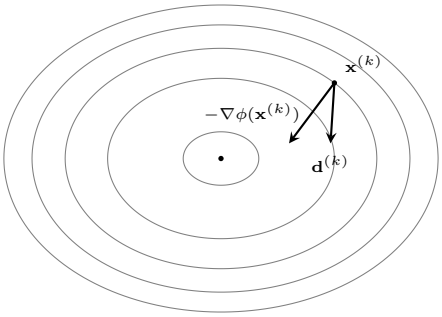


Figure 10: Level sets of ϕ and the negative gradient at a point $\mathbf{x}^{(k)}$. A possible descent direction is labeled as $\mathbf{d}^{(k)}$.

The steepest possible direction that minimizes the LHS above is

$$\mathbf{d}^{(k)} = -\nabla\phi(\mathbf{x}^{(k)}) = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}.$$

If we take a fixed step size $\alpha_k = \omega > 0$, then the update formula is

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{d}^{(k)} = \mathbf{x}^{(k)} + \omega(\mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}),$$

which is Richardson iteration. From now on, we denote

$$\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$$

as the **residual**.

16.1 Steepest descent method

Richardson iteration can be quite slow, what if we choose α_k at every step to minimize ϕ? That is, we set

$$\alpha_k = \text{argmin}_\alpha \phi(\mathbf{x}^{(k)} + \alpha\mathbf{d}^{(k)}).$$

Taking the derivative, we get

$$\begin{aligned}
\frac{d}{d\alpha}\phi(\mathbf{x}^{(k)} + \alpha\mathbf{d}^{(k)}) &= \frac{d}{d\alpha}\left[\frac{1}{2}\left(\mathbf{x}^{(k)} + \alpha\mathbf{d}^{(k)}\right)^T \mathbf{A}\left(\mathbf{x}^{(k)} + \alpha\mathbf{d}^{(k)}\right) - \left(\mathbf{x}^{(k)} + \alpha\mathbf{d}^{(k)}\right)^T \mathbf{b}\right] \\
&= \alpha(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)} + (\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{x}^{(k)} - (\mathbf{d}^{(k)})^T\mathbf{b} \\
&= \alpha(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)} + (\mathbf{d}^{(k)})^T\left(\mathbf{A}\mathbf{x}^{(k)} - \mathbf{b}\right) \\
&= \alpha(\mathbf{d}^{(k)})^T\mathbf{A}\mathbf{d}^{(k)} - (\mathbf{d}^{(k)})^T\mathbf{r}^{(k)}
\end{aligned}$$

Hence, solving $\frac{d}{d\alpha}\phi(\mathbf{x}^{(k)} + \alpha\mathbf{d}^{(k)}) = 0$ yields

$$\alpha_k = \frac{\mathbf{d}^{(k)} \cdot \mathbf{r}^{(k)}}{\mathbf{d}^{(k)} \cdot \mathbf{A}\mathbf{d}^{(k)}},$$

and we have an improved update formula (for $\mathbf{r}^{(k)} = \mathbf{d}^{(k)}$)

$$\begin{aligned} \alpha_k &= \frac{\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)}}{\mathbf{r}^{(k)} \cdot \mathbf{A}\mathbf{r}^{(k)}} \\ \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}. \end{aligned}$$

This iteration is known as **steepest descent method**.

Remark 16.1 (connection to optimization)**.** Richardson and steepest descent methods are essentially gradient descent methods applied to ϕ. Richardson has a fixed step size, while steepest descent method performs what is known as an exact line search to find the optimal step size α_k. We are lucky in the case of ϕ since it is easy to find the best α_k. For minimizing more general functions not coming from linear algebra, exact line search is replaced with backtracking algorithm like Armijo backtracking.

Exercise 16.2 (convergence of steepest descent)**.** Suppose $\mathbf{x}^{(0)}$ is the initial guess for Richardson and steepest descent. Let $\mathbf{e}^{(k)}$ be the error of steepest descent and let $\mathbf{e}^{(k),R}$ be the error for Richardson. We define a new norm called the $\mathbf{A}$-norm:

$$\|\mathbf{x}\|_{\mathbf{A}} = \sqrt{\mathbf{x} \cdot \mathbf{A}\mathbf{x}}.$$

Show that for all k:

$$\|\mathbf{e}^{(k)}\|_{\mathbf{A}} \leq \|\mathbf{e}^{(k),R}\|_{\mathbf{A}}.$$

This shows that steepest descent provides an improvement on Richardson, although it is only modest.
Hint: Show that there is a constant C that depends on $\mathbf{A}, \mathbf{b}$ such that $\phi(\mathbf{x}^{(k)}) + C = \|\mathbf{e}^{(k)}\|_{\mathbf{A}}^2$.

16.2 Conjugate Gradient method

The last iterative method we will cover is Conjugate Gradient (CG) method. The method is complicated to derive, so we will go over the main features and mention the algorithm. Similar to Richardson and steepest descent, our goal is to minimize ϕ.

16.2.1 First step: steepest descent

Given an initial guess $\mathbf{x}^{(0)}$, the first step of CG is to follow the steepest descent direction

$$\mathbf{d}^{(0)} = -\nabla\phi(\mathbf{x}^{(0)}) = \mathbf{r}^{(0)}$$

and step size $\alpha_0 = \frac{\mathbf{r}^{(0)}\cdot\mathbf{d}^{(0)}}{\mathbf{d}^{(0)}\cdot\mathbf{A}\mathbf{d}^{(0)}}$. Recall that the choice of α was so that $\phi(\mathbf{x}^{(0)} + \alpha\mathbf{d}^{(0)})$ was minimized. In other words we minimized ϕ over a line

$$\mathbf{x}^{(1)} = \underset{\mathbf{x}\in\mathbf{x}^{(0)}+\mathcal{S}^0}{\operatorname{argmin}}\ \phi(\mathbf{x}), \quad \mathcal{S}^0 = \operatorname{span}\{\mathbf{r}^{(0)}\}$$

16.2.2 Second step: iterative minimization

The idea of CG is to try to now minimize over a larger subspace. In this case, we pick $\mathbf{x}^{(1)}$ to solve the following 2D minimization problem

$$\mathbf{x}^{(2)} = \underset{\mathbf{x}\in\mathbf{x}^{(0)}+\mathcal{S}^1}{\operatorname{argmin}}\ \phi(\mathbf{x}), \quad \mathcal{S}^1 = \operatorname{span}\{\mathbf{r}^{(0)}, \mathbf{A}\mathbf{r}^{(0)}\}$$

Let's now write $\mathbf{x} = \mathbf{x}^{(0)} + \alpha\mathbf{r} + \beta\mathbf{d}$, where $\mathbf{d}$ is some direction determined later and $\mathbf{r} = \mathbf{r}^{(0)}$. We compute $\phi(\mathbf{x})$:

$$\begin{aligned}\phi(\mathbf{x}) = \phi(\mathbf{x}^{(0)} + \alpha\mathbf{r} + \beta\mathbf{d}) &= \frac{1}{2}\big(\mathbf{x}^{(0)} + \alpha\mathbf{r} + \beta\mathbf{d}\big)^T\mathbf{A}\big(\mathbf{x}^{(0)} + \alpha\mathbf{r} + \beta\mathbf{d}\big) - \big(\mathbf{x}^{(0)} + \alpha\mathbf{r} + \beta\mathbf{d}\big)^T\mathbf{b} \\ &= \frac{1}{2}\mathbf{x}^{(0)}\cdot\mathbf{A}\mathbf{x}^{(0)} - \mathbf{b}\cdot\mathbf{x}^{(0)} \underbrace{-\alpha\mathbf{r}\cdot\mathbf{r} + \frac{\alpha^2}{2}\mathbf{r}\cdot\mathbf{A}\mathbf{r}}_{\text{function of just }\alpha} \underbrace{-\beta\mathbf{d}\cdot\mathbf{r} + \frac{\beta^2}{2}\mathbf{d}\cdot\mathbf{A}\mathbf{d}}_{\text{function of just }\beta} \underbrace{+\alpha\beta\mathbf{d}\cdot\mathbf{A}\mathbf{r}}_{\text{cross term that we want to be 0}}\end{aligned}$$

16.2.3 Third step: orthogonalization procedure

Notice that if $\mathbf{d}\cdot\mathbf{A}\mathbf{r} = 0$, then we could minimize α and β separately. We define

$$\langle\mathbf{d}, \mathbf{r}\rangle_{\mathbf{A}} := \mathbf{d}\cdot\mathbf{A}\mathbf{r}$$

as a new inner product (think of it as a weighted dot product). The equation $\langle\mathbf{d}, \mathbf{r}\rangle_{\mathbf{A}} = 0$ now just means that we want $\mathbf{d} \perp \mathbf{r}$ in the new inner product $\langle\cdot,\cdot\rangle_{\mathbf{A}}$.

In this case, we have freedom of the choice of $\mathbf{d}$, since $\{\mathbf{r}, \mathbf{d}\}$ just need to be a basis for $\operatorname{span}\{\mathbf{r}^{(0)}, \mathbf{A}\mathbf{r}^{(0)}\}$. What we can do is apply a Gram Schmidt orthogonalization

$$\mathbf{d} := \mathbf{A}\mathbf{r}^{(0)} - \frac{\langle\mathbf{A}\mathbf{r}^{(0)}, \mathbf{r}^{(0)}\rangle_{\mathbf{A}}}{\langle\mathbf{r}^{(0)}, \mathbf{r}^{(0)}\rangle_{\mathbf{A}}}\,\mathbf{r}^{(0)}$$

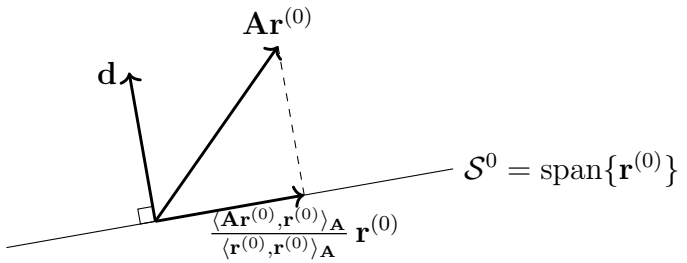


Figure 11: Gram-Schmidt orthogonalization procedure to find $\mathbf{d} = \mathbf{d}^{(1)}$. Note that perpendicular in this picture is with respect to the new $\langle \cdot, \cdot \rangle_{\mathbf{A}}$ inner product.

Once we have this direction $\mathbf{d}$, then minimizing α is actually the α from the previous step $\alpha_0 = \frac{\mathbf{r}^{(0)} \cdot \mathbf{d}^{(0)}}{\mathbf{d}^{(0)} \cdot \mathbf{A}\mathbf{d}^{(0)}}$ and

$$\phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{(0)} \cdot \mathbf{A}\mathbf{x}^{(0)} - \mathbf{b} \cdot \mathbf{x}^{(0)} - \alpha_0 \mathbf{r} \cdot \mathbf{r} + \frac{\alpha_0^2}{2}\mathbf{r} \cdot \mathbf{A}\mathbf{r} - \beta \mathbf{d} \cdot \mathbf{r} + \frac{\beta^2}{2}\mathbf{r} \cdot \mathbf{A}\mathbf{r}.$$

Since the optimal α was α_0, we have that the optimal $\mathbf{x}$ is $\mathbf{x} = \mathbf{x}^{(0)} + \alpha_0 \mathbf{r}^{(0)} + \beta \mathbf{d} = \mathbf{x}^{(1)} + \beta \mathbf{d}$. I'll now call $\mathbf{d}^{(1)} = \mathbf{d}$ as the new direction. Thus, our 2D minimization problem has turned into a 1D minimization problem for β:

$$\phi(\mathbf{x}^{(1)} + \beta \mathbf{d}^{(1)}) = \frac{1}{2}\mathbf{x}^{(1)} \cdot \mathbf{A}\mathbf{x}^{(1)} - \mathbf{x}^{(1)} \cdot \mathbf{b} + \underbrace{\frac{\beta^2}{2}\mathbf{d}^{(1)} \cdot \mathbf{A}\mathbf{d}^{(1)} - \beta \mathbf{d}^{(1)} \cdot \mathbf{r}^{(1)}}_{\text{function of } \beta \text{ to minimize}}$$

whose optimal value is $\beta = \alpha_1 = \frac{\mathbf{d}^{(1)} \cdot \mathbf{r}^{(1)}}{\mathbf{d}^{(1)} \cdot \mathbf{A}\mathbf{d}^{(1)}}$.

16.2.4 Summary: One step of CG and algorithm

Let's now summarize one step of this minimization procedure

- We were given the initial residual and direction $\mathbf{d}^{(0)} = \mathbf{r}^{(0)}$, current guess $\mathbf{x}^{(1)}$, and current residual $\mathbf{r}^{(1)}$.
- We determined a good direction

  $$\mathbf{d}^{(1)} = \mathbf{A}\mathbf{r}^{(0)} - \frac{\langle \mathbf{A}\mathbf{r}^{(0)}, \mathbf{r}^{(0)} \rangle_{\mathbf{A}}}{\langle \mathbf{r}^{(0)}, \mathbf{r}^{(0)} \rangle_{\mathbf{A}}} \mathbf{r}^{(0)} = \mathbf{A}\mathbf{r}^{(0)} - \frac{\langle \mathbf{A}\mathbf{r}^{(0)}, \mathbf{r}^{(0)} \rangle_{\mathbf{A}}}{\langle \mathbf{r}^{(0)}, \mathbf{r}^{(0)} \rangle_{\mathbf{A}}} \mathbf{d}^{(0)}$$

  so that the minimization over $\mathbf{x} + \text{span}\{\mathbf{r}^{(0)}, \mathbf{A}\mathbf{r}^{(0)}\}$ became a 1d minimization problem in the direction $\mathbf{d}^{(1)}$.

Exercise 16.3. Show that

$$\mathbf{d}^{(1)} = \mathbf{r}^{(1)} - \frac{\mathbf{r}^{(1)} \cdot \mathbf{r}^{(1)}}{\mathbf{r}^{(0)} \cdot \mathbf{r}^{(0)}} \mathbf{d}^{(0)}$$

- We found the optimal step size in direction $\mathbf{d}^{(1)}$

$$\alpha_1 = \frac{\mathbf{d}^{(1)} \cdot \mathbf{r}^{(1)}}{\mathbf{d}^{(1)} \cdot \mathbf{A}\mathbf{d}^{(1)}}$$

We then set

$$\mathbf{x}^{(2)} = \mathbf{x}^{(1)} + \alpha_1 \mathbf{d}^{(1)}.$$

We can continue this pattern with the following algorithm.

Input: SPD matrix $\mathbf{A}$, RHS vector $\mathbf{b}$, and initial guess $\mathbf{x}^{(0)}$
Output: Approximate solution $\mathbf{x}$ to $\mathbf{A}\mathbf{x} = \mathbf{b}$

$\mathbf{r}^{(0)} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$;
$\mathbf{d}^{(0)} \leftarrow \mathbf{r}^{(0)}$;
for $k \leftarrow 0$ **to** N_{max} **do**
  $\alpha_k \leftarrow \frac{\mathbf{d}^{(k)} \cdot \mathbf{r}^{(k)}}{\mathbf{d}^{(k)} \cdot \mathbf{A}\mathbf{d}^{(k)}}$; (optimizes over line)
  $\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} + \alpha_k \mathbf{d}^{(k)}$; (update $\mathbf{x}$)
  $\mathbf{r}^{(k+1)} \leftarrow \mathbf{r}^{(k)} - \alpha_k \mathbf{A}\mathbf{d}^{(k)}$; (update $\mathbf{r}$)
  if $\mathbf{r}^{(k+1)}$ *is small* **then**
    return $\mathbf{x}^{(k+1)}$;
  end
  $\beta_k \leftarrow \frac{\mathbf{r}^{(k+1)} \cdot \mathbf{r}^{(k+1)}}{\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)}}$; (comes from Gram Schmidt orthogonalization)
  $\mathbf{d}^{(k+1)} \leftarrow \mathbf{r}^{(k+1)} + \beta_k \mathbf{d}^{(k)}$; (new direction)
end

Algorithm 11: Conjugate gradient method

Conjugate gradient is very fast. We now state two results about how fast CG is.

Theorem 16.1 (minimization property of CG)**.** Let $\mathbf{x}^{(k+1)}$ be the sequence of iterates produced by CG. Then,

$$\mathbf{x}^{(k+1)} = \underset{\mathbf{x} \in \mathbf{x}^{(0)} + \mathcal{S}^k}{\operatorname{argmin}} \phi(\mathbf{x}), \quad \mathcal{S}^k = \operatorname{span}\{\mathbf{r}^{(0)}, \mathbf{A}\mathbf{r}^{(0)}, \ldots, \mathbf{A}^k \mathbf{r}^{(0)}\}$$

The above theorem essentially means that CG terminates in n steps because $\mathcal{S}^{n-1} = \mathbb{R}^n$. We usually stop the iteration much sooner and the next result states that convergence to the solution is quite fast.

Theorem 16.2 (convergence of CG)**.** Let $\mathbf{x}^{(k)}$ be the sequence of iterates produced by CG. Then, the error satisfies

$$\|\mathbf{e}^{(k+1)}\|_{\mathbf{A}} \leq \frac{\sqrt{\kappa_2(\mathbf{A})}-1}{\sqrt{\kappa_2(\mathbf{A})}+1}\|\mathbf{e}^{(k)}\|_{\mathbf{A}}$$

where

$$\|\mathbf{v}\|_{\mathbf{A}} = \sqrt{\langle \mathbf{v}, \mathbf{v}\rangle_{\mathbf{A}}}$$

Remark 16.2 (comparison with Richardson and steepest descent)**.** CG is much faster than Richardson and steepest descent due to $\frac{\sqrt{\kappa_2(\mathbf{A})}-1}{\sqrt{\kappa_2(\mathbf{A})}+1}$ factor multiplying the error rather than the $\frac{\kappa_2(\mathbf{A})-1}{\kappa_2(\mathbf{A})+1}$ factor from Richardson or steepest descent.

17 Least squares (October 6-8, 2024)

This will be the last linear algebra lecture. So far, we have studied how to solve $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is an $n \times n$ matrix. This lecture now considers solving $\mathbf{Ax} \approx \mathbf{b}$, where $\mathbf{A}$ is an $m \times n$ matrix and $m > n$. This is known as an underdetermined system meaning there are more equations than unknowns. It is very likely that in this case, there is no solution to $\mathbf{Ax} = \mathbf{b}$. Here we replace solving $\mathbf{Ax} = \mathbf{b}$ with minimizing

$$\min_{\mathbf{x}} \frac{1}{2}\|\mathbf{Ax}-\mathbf{b}\|_2^2 = \frac{1}{2}\sum_{i=1}^{n}((\mathbf{Ax})_i - b_i)^2$$

which is finding the **least squares** solution. These kinds of problems are ubiquitous but a common application is fitting a curve to data.

Example 17.1 (line of best fit)**.** Let $(t_i, b_i)_{i=1}^m$ be a set of data points. We expect that they have a linear relationship $b_i \approx mt_i + c$. The system of equations that we would like to apply least squares to is

$$\begin{pmatrix} 1 & t_1 \\ 1 & t_2 \\ \vdots & \vdots \\ 1 & t_m \end{pmatrix} \begin{pmatrix} c \\ m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Example 17.2 (power law fit)**.** Let $(t_i, b_i)_{i=1}^m$ be a set of data points. We expect that they have a linear relationship

$$b_i \approx ct_i^p.$$

How might we get this into a linear form? One way is to take the log of both sides

$$\log b_i \approx \log c + p \log t_i$$

The system of equations that we would like to apply least squares to is

$$\begin{pmatrix} 1 & \log t_1 \\ 1 & \log t_2 \\ \vdots & \vdots \\ 1 & \log t_m \end{pmatrix} \begin{pmatrix} \log c \\ p \end{pmatrix} = \begin{pmatrix} \log b_1 \\ \log b_2 \\ \vdots \\ \log b_m \end{pmatrix}$$

In numerical methods, this kind of best fit is useful for evaluating the convergence of a numerical method. Often t_i is replaced by a numerical parameter, h_i, and y_i is replaced by some error e_i. We often have theorems saying $e_i = ch_i^p$ for some power p. We can then test the algorithm by trying different h values and putting a power law fit to the error.

Exercise 17.1 (exponential fit)**.** Show that you can fit data $(t_i, b_i)_{i=1}^m$ with an exponential fit $y_i \approx ce^{kt_i}$ through finding the least squares solution of

$$\begin{pmatrix} 1 & t_1 \\ 1 & t_2 \\ \vdots & \vdots \\ 1 & t_m \end{pmatrix} \begin{pmatrix} \log c \\ k \end{pmatrix} = \begin{pmatrix} \log b_1 \\ \log b_2 \\ \vdots \\ \log b_m \end{pmatrix}$$

17.1 Normal equations

How might we find a least squares solution of $\mathbf{Ax} = \mathbf{b}$? We now look to the optimization problem

$$\min_{\mathbf{x}} \frac{1}{2}\|\mathbf{Ax} - \mathbf{b}\|_2^2 = \min_{\mathbf{x}} \frac{1}{2} \sum_{i=1}^{n} ((\mathbf{Ax})_i - b_i)^2$$

Theorem 17.1 (normal equations)**.** Let $\mathbf{A}$ be an $m \times n$ matrix with $m \geq n$ and with full column rank. The vector $\mathbf{x}$ minimizes $\|\mathbf{Ax} - \mathbf{b}\|_2^2$ if and only if $\mathbf{x}$ solves the **normal equations**:

$$\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$$

Proof. Since $\mathbf{A}$ has full column rank, the columns of $\mathbf{A}$ (denoted $\{\mathbf{a}_1, \mathbf{a}_2, \dots \mathbf{a}_n\}$) form a basis of the range of $\mathbf{A}$ (denoted $\mathcal{R}(\mathbf{A})$). Using this fact of the basis of the range, we have that

$$\mathbf{x} = \text{argmin}_{\mathbf{x}} \frac{1}{2}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 \text{ if and only if } \mathbf{z} = \mathbf{A}\mathbf{x} = \text{argmin}_{\mathbf{v} \text{ in } \mathcal{R}(\mathbf{A})} \frac{1}{2}\|\mathbf{v} - \mathbf{b}\|_2^2$$

I now claim that the minimizer to the problem

$$\min_{\mathbf{v} \text{ in } \mathcal{R}(\mathbf{A})} \frac{1}{2}\|\mathbf{v} - \mathbf{b}\|_2^2$$

solves $\mathbf{A}^T\mathbf{z} = \mathbf{A}^T\mathbf{b}$. The minimizer to the above problem is the *orthogonal projection* of $\mathbf{b}$ onto $\mathcal{R}(\mathbf{A})$:

$$\mathbf{z} = \text{proj}_{\mathcal{R}(\mathbf{A})}(\mathbf{b}).$$

A key property of the orthogonal projection is that the difference between $\mathbf{b}, \mathbf{z}$ is orthogonal to $\mathcal{R}(\mathbf{A})$:

$$\mathbf{y} \cdot (\mathbf{z} - \mathbf{b}) = 0 \text{ for all } \mathbf{y} \text{ in } \mathcal{R}(\mathbf{A})$$

The above equation also holds for each basis vector of $\mathcal{R}(\mathbf{A})$:

$$\mathbf{a}_i \cdot (\mathbf{z} - \mathbf{b}) = 0 \text{ for all } i = 1, \dots, n.$$

In matrix form, this reads $\mathbf{A}^T(\mathbf{z} - \mathbf{b}) = \mathbf{0}$ or $\mathbf{A}^T\mathbf{z} = \mathbf{A}^T\mathbf{b}$. Using the fact that $\mathbf{A}\mathbf{x} = \mathbf{z}$ completes the proof. □

Remark 17.1 (pseudoinverse)**.** We have that the least squares solution is $\mathbf{x} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$. The matrix $\mathbf{A}^\dagger = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$ is the Moore-Penrose pseudoinverse for a matrix with full column rank. It satisfies $\mathbf{A}^\dagger\mathbf{A} = \mathbf{I}$ (but not $\mathbf{A}\mathbf{A}^\dagger = \mathbf{I}$ like a normal inverse would also satisfy).

Remark 17.2 (solving the normal equations)**.** Notice that if $\mathbf{A}$ is an $m \times n$ matrix with m large and n small, then $\mathbf{A}^T\mathbf{A}$ is a small $n \times n$ matrix. Moreover, $\mathbf{A}^T\mathbf{A}$ will be SPD if $\mathbf{A}$ has full column rank. Thus, we can use a bunch of our previously learned linear algebra techniques for SPD matrices to solve $\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b}$.

Remark 17.3 (solving the normal equations: poor conditioning)**.** Notice that if $\mathbf{A}$ is full column rank, then $\mathbf{A}^T\mathbf{A}$ is SPD (left as an exercise). This means we can apply our large array of methods to solve the normal equations $\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{b}$ directly. However, there is a small problem; often the normal equations are poorly conditioned. To see this, we write an SVD of $\mathbf{A}$

$$\mathbf{A} = \mathbf{U}^T\mathbf{\Sigma}\mathbf{V},$$

and compute $\mathbf{A}^T\mathbf{A}$:

$$\mathbf{A}^T\mathbf{A} = \mathbf{V}^T\mathbf{\Sigma}\mathbf{U}\mathbf{U}^T\mathbf{\Sigma}\mathbf{V} = \mathbf{V}^T\mathbf{\Sigma}^2\mathbf{V}$$

The condition number of $\mathbf{A}^T\mathbf{A}$ is

$$\kappa_2(\mathbf{A}^T\mathbf{A}) = \frac{\lambda_{max}(\mathbf{A}^T\mathbf{A})}{\lambda_{min}(\mathbf{A}^T\mathbf{A})} = \frac{\sigma_{max}(\mathbf{A})^2}{\sigma_{min}(\mathbf{A})^2}.$$

In essence, we have just doubled the condition number of $\mathbf{A}$. For larger least squares problems, this can lead to trouble.

Remark 17.4 (**QR** decomposition)**.** Another way to solve a least squares problem $\min_{\mathbf{x}} \frac{1}{2}\|\mathbf{A}\mathbf{x}-\mathbf{b}\|_2^2$ is to perform a **QR** decomposition of $\mathbf{A}$. That is, we write $\mathbf{A} = \mathbf{QR}$ where $\mathbf{Q}$ is an $m \times m$ rotation matrix ($\mathbf{Q}^T\mathbf{Q} = \mathbf{Q}\mathbf{Q}^T = \mathbf{I}$) and $\mathbf{R}$ is an $m \times n$ upper triangular matrix. Writing

$$\mathbf{R} = \begin{pmatrix} \mathbf{R}_1 \\ \mathbf{0} \end{pmatrix}$$

where $\mathbf{R}_1$ is a square $n \times n$ matrix. We have that the least squares solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$ is $\mathbf{x} = \mathbf{R}_1^{-1}(\mathbf{Q}^T\mathbf{b})_1$, where $(\mathbf{Q}^T\mathbf{b})_1$ is the first n components of $\mathbf{Q}^T\mathbf{b}$. Notice this only involves a matrix multiplication and inverting an upper triangular matrix once we know the factorization $\mathbf{A} = \mathbf{QR}$ and conditioning is less of an issue.

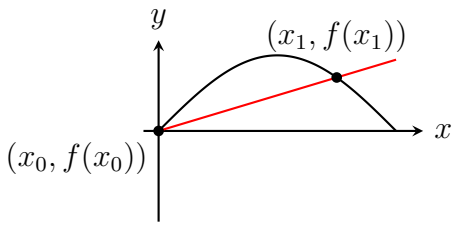
18 Polynomial Interpolation (Lectures October 10-27, 2025)

Suppose we have points (x_i, y_i) for $i = 0, \ldots, n$ where $y_i = f(x_i)$. Here, f may be expensive to compute as it could be from solving a differential equation or from running a lab experiment. It could also be impossible compute for every x. The remedy this problem, we approximate f with some polynomial p_n, which is easy and efficient to compute. This set of lectures is concerned with the following questions

- How do we find a polynomial st $p(x_i) = y_i$?
- Is $p(x) \approx f(x)$ for $x \neq x_i$?

18.1 Lagrange Form of Interpolating Polynomial (Lecture October 10, 2025)

Our first example will be of linear interpolation. Say I have 2 data pts $(x_0, y_0), (x_1, y_1)$ we can just connect them with a straight line.



The equation for the lst degree polynomial interpolating (x_0, y_0) and (x_1, y_1) is

$$\begin{aligned} p(x) &= y_1 \frac{(x - x_0)}{(x_1 - x_0)} + y_0 \frac{(x - x_1)}{(x_0 - x_1)} \\ &= y_1 \ell_1(x) + y_0 \ell_0(x) \end{aligned}$$

Note that ℓ_0, ℓ_1 satisfy

$$\begin{aligned} \ell_0(x_0) = 1, \quad \ell_0(x_1) = 0 \\ \ell_1(x_0) = 0, \quad \ell_1(x_1) = 1, \end{aligned}$$

so the linear interpolant is expressed in terms of functions that activate on each individual x_i point. How do we build an interpolating polynomial for more data points (x_i, y_i) for $i = 0, \ldots, n$?

Following the example of linear interpolation, one approach would be to find polynomials ℓ_i such that they satisfy

$$\begin{cases} \ell_i(x_i) = 1 \\ \ell_i(x_j) = 0, \quad \text{for } j \neq i \end{cases}$$

and then write the interpolating polynomial as

$$p_n(x) = \sum_{i=0}^{n} y_i \ell_i(x). \tag{2}$$

If we can find such ℓ_i, then $p(x_i) = y_i$ for all i. The polynomials ℓ_i do in fact exist and are given by

$$\ell_i(x) = \prod_{\substack{j \neq i \\ j=0}}^{n} \left(\frac{x - x_j}{x_i - x_j} \right). \tag{3}$$

Notice that the product ensures that if $x = x_j$ for $j \neq i$, then $\ell_i(x_j) = 0$. Dividing by $x_i - x_j$ in each term of the product enforces the condition that $\ell_i(x_i) = 1$ because then the formula is just a product of all 1. These polynomials l_i in (3) are known as the Lagrange basis for points $\{x_i\}_{i=0}^n$ and (2) is called the Lagrange form. of an interpolating polynomial.

18.2 Uniqueness of Interpolating Polynomial

The construction of the Lagrange form of interpolating polynomial in (2) shows that if given (x_i, y_i) for $i = 0, \ldots, n$, then there exists an interpolating polynomial p_n. We can see from the Lagrange basis in (3) that since each ℓ_i is the product of n linear functions, then the degree of the polynomial, p_n, is guaranteed to be less than n. The question one may ask is whether this interpolating polynomial is unique? This would be important to know, so that we know there is just one answer when constructing such approximations.

Theorem 18.1 (existence and uniqueness of interpolating polynomial)**.** Let (x_i, y_i) for $i = 0, \ldots, n$ be $n + 1$ points such that $x_i \neq x_j$ for all $i \neq j$, i.e. the x_i's are distinct. There is a unique polynomial of degree at most n such that $p(x_i) = y_i$ for all i.

Proof. Existence comes from our Lagrange form construction in (2). For uniqueness, we want to show that if there are two interpolating polynomials p_n, q_n of degree at most n, then $p_n = q_n$.

Suppose p_n, q_n are interpolating polynomials of degree at most n. Let $r_n = p_n - q_n$, which is a polynomial of degree at most n. Looking at the interpolation points x_i, we have

$$r_n(x_i) = p_n(x_i) - q_n(x_i) = y_i - y_i = 0$$

for each $i = 0, \ldots, n$. Since r_n is a polynomial of degree at most n, r_n has at most n roots if it is nonzero. Since r_n has $n + 1$ roots, $r_n = 0$, and $p_n = q_n$. □

18.3 Newton Algorithm and Newton Form

The Lagrange basisflomis very convenient. I use it often. One disadvantage is it is often hard to incorporate new data easily because we would have to reconstruct the ℓ_is everytime. This is where the Newton algorithm and Newton form of the interpolating polynomial comes in.

18.3.1 Newton Algorithm

Say we have $(x_i, y_i) \quad i = 0, \cdots, n$ with interpolating polynomial p_n, and say I want to add a new data point. we look for

$$p_{n+1}(x) = p_n(x) + c \cdot \prod_{i=0}^{n} (x - x_i)$$

We then use (x_{n+1}, y_{n+1}) to determine c. We want

$$y_{n+1} = p_{n+1}(x_{n+1}) = p_n(x_{n+1}) + c\prod_{i=0}^{n}(x_{n+1} - x_i).$$

Solving for c yields

$$c = \frac{y_{n+1} - p_n(x_{n+1})}{\prod_{i=0}^{n}(x_{n+1} - x_i)}$$

and

$$p_{n+1}(x) = p_n(x) + (y_{n+1} - p_n(x))\prod_{i=0}^{n}\frac{(x - x_i)}{(x_{n+1} - x_i)}. \tag{4}$$

Notice that the formula (4) is a recursive update formula. We can then generate the interpolating polynomial by starting with $p_0(x) = y_0$. Once we have p_k, we update p_{k+1} with formula (4) by replacing $n = k$. We continue the process until we have an interpolating polynomial p_n through the data points (x_i, y_i) for $i = 0, \ldots, n$. This procedure is known as Newton's algorithm and the resulting form is called the Newton form:

$$p_n(x) = \sum_{i=0}^{n} a_i \prod_{j=0}^{i-1}(x - x_j) \tag{5}$$

where $a_i = \frac{y_i - p_{i-1}(x_i)}{\prod_{j=0}^{i-1}(x_i - x_j)}$ and p_i is the interpolating polynomial through the points $x_k = 0, \ldots x_i$. Let's now work through an example.

Example 18.1. Let $x_i = 0, 1, 2$ and $y_i = -5, -3, -15$. We start by interpolating (x_0, y_0) with

$$p_0(x) = -5$$

To compute p_1, we write $p_1(x) = p_0(x) + c(x - x_0)$. We set

$$-3 = p_1(x_1) = -5 + c(x_1 - x_0) = -5 + c \quad \implies c = 2$$

so $p_1(x) = -5 + 2x$. We finally compute p_2 by writing $p_2(x) = p_1(x) + cx(x - 1)$ and solving for c

$$-15 = p_2(2) = p_1(2) + 2c = 1 + 2c \quad \implies c = -7.$$

Hence, the interpolating polynomial in Newton form is

$$p_2(x) = -5 + 2x - 7x(x - 1) \tag{6}$$

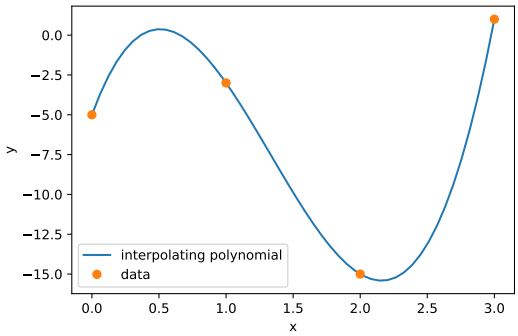


Figure 12: Data and interpolating polynomial from Example 18.1

18.3.2 Nested Multiplication (Lecture October 20, 2025)

Since the goal of polynomial interpolation is to have an efficient approximation of some function f, what is an efficient way to evaluate an interpolating polynomial p_n? Going back to our example of interpolating $x_i = 0, 1, 2$ and $y_i = -5, -3, -15$, we had from (6) that the interpolating polynomial is

$$p_2(x) = -5 + 2x - 7x(x-1).$$

Consider two ways of rewriting p_2. The first is to expand all of the multiplications:

$$P_2(x) = -5 + 9x - 7x^2. \tag{7}$$

Another approach would be to factor the polynomial

$$P_2(x) = -5 + x(2 - 7(x-1)) \tag{8}$$

How do each of these formulas compare in terms of FLOPs? Counting each addition and multiplication, we have that both (7) and (8) cost 5 FLOPs and the original formula (6) takes 6 FLOPs. In general, the nested factoring strategy in (8) is most efficient and is known as **nested multiplication**.

For a general nested formula, we know a Newton polynomial in (5) is

$$\begin{aligned} p_n(x) &= \sum_{i=0}^{n} a_i \prod_{j=0}^{i-1}(x - x_j) \\ &= a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \ldots + a_n(x - x_0)(x - x_1)\cdots(x - x_{n-1}) \end{aligned}$$

We now repeat the factoring process by factoring $(x - x_0)$:

$$p_n(x) = a_0 + (x - x_0)\left[a_1 + a_2(x - x_1) + \cdots + a_n(x - x_1)\cdots(x - x_n)\right]$$

and now factoring $(x - x_1)$

$$p_n(x) = a_0 + (x - x_0)\left[a_1 + (x - x_1)\left[a_2 + \cdots + a_n(x - x_2)\cdots(x - x_n)\right]\right].$$

We continue the pattern to get the new formula

$$p_n(x) = a_0 + (x - x_0)\left[a_1 + (x - x_1)\left[a_2 + (x - x_2)\left[a_3 + \cdots + a_n(x - x_n)\right]\right]\right].$$

This full nested form can be written as a for loop in the following pseudocode

Input: Coefficients $a_0, \ldots, a_n$, interpolation points $x_0, \ldots, x_{n-1}$, evaluation point x

Output: Value of the interpolating polynomial $p_n(x)$

$p \leftarrow a_n$;
for $i \leftarrow n - 1$ **to** 0 *(loop backwards)* **do**
  $p \leftarrow a_i + (x - x_i) \cdot p$;
end
return p;

Algorithm 12: Nested multiplication algorithm for evaluating Newton form polynomial

This code takes 3n FLOPs. The traditional or naive way of evaluating the Newton form of p_n in (5)

$$p_n(x) = \sum_{i=0}^{n} a_i \prod_{j=0}^{i-1}(x - x_j)$$

takes many more FLOPs. To see this, we notice that for each i, there are $i - 1$ multiplications and $i - 1$ subtractions. Adding the whole sum then takes $n + 1$ additions. The total number of FLOPs for the naive evaluation is

$$\text{FLOPs} = (n + 1) + 2\sum_{i=0}^{n}(i - 1) = O(n^2).$$

The nested form is much more efficient in terms of number of FLOPs.

18.3.3 Divided Differences

There is another view of the Newton form of the polynomial from (5), which is divided differences. Recall that the Newton form is

$$p_n(x) = \sum_{i=0}^{n} a_i \prod_{j=0}^{i-1} (x - x_j).$$

This new perspective will shed light on the coefficients a_i and show that the Newton form of an interpolating polynomial is a discrete version of a Taylor expansion.

Suppose that p_n interpolates the function f through the points x_i for $i = 0, \ldots, n$. We'll denote the function values as $f_i = f(x_i)$. The first coefficient is

$$a_0 = p_n(x_0) = f_0.$$

We now look at a_1. We want a_1 to satisfy

$$f_1 = p_n(x_1) = a_0 + a_1(x_1 - x_0).$$

Subtracting $a_0 = f(x_0)$ from both sides and dividing by $(x_1 - x_0)$ yields

$$a_1 = \frac{f_1 - f_0}{x_1 - x_0}.$$

The above formula is known as a finite difference. It mimics the first derivative of f.

Thinking in terms of a Taylor expansion, we might expect that a_2 looks like a second derivative of f. We compute $p_n(x_2)$ to find a_2

$$f_2 = p_n(x_2) = a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1)$$

Solving for a_2 yields

$$a_2 = \frac{f_2 - a_1(x_2 - x_0) - a_0}{(x_2 - x_0)(x_2 - x_1)}$$

Notice that

$$a_1(x_2 - x_0) + a_0 = a_1(x_2 - x_1) + a_1(x_1 - x_0) + a_0 = a_1(x_2 - x_1) + f_1,$$

and inserting into the formula for a_2 leads to

$$a_2 = \frac{f_2 - f_1 - a_1(x_2 - x_1)}{(x_2 - x_0)(x_2 - x_1)} = \frac{f_2 - f_1}{(x_2 - x_0)(x_2 - x_1)} - \frac{a_1}{(x_2 - x_0)}$$

We then use the fact that $a_1 = \frac{f_1 - f_0}{x_1 - x_0}$ to get

$$a_2 = \frac{\frac{f_2 - f_1}{x_2 - x_1} - \frac{f_1 - f_0}{x_1 - x_0}}{x_2 - x_0}.$$

Notice that a_2 is now a difference of differences, and it mimics a second derivative of f.

These coefficients come from what are known as **divided differences**.

Definition 18.1 (divided differences of f)**.** The divided differences of a function f on points $x_0, \ldots, x_n$ are defined recursively as follows:

$$f[x_i] = f(x_i) \quad \text{for } i = 0, \ldots, n \quad \text{(base case)}$$

$$f[x_i, \ldots, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i} \quad \text{for } k = 1, \ldots, n$$

where $f[x_i, \ldots, x_{i+k}]$ denotes the k-th order divided difference of f on the points $x_i, \ldots, x_{i+k}$.

Example 18.2 (lower order divided differences)**.** For three points x_0, x_1, x_2, the divided differences are

- Zeroth order difference as function evaluation

$$f[x_0] = f(x_0), \quad f[x_1] = f(x_1), \quad f[x_2] = f(x_2)$$

- First order difference as a discrete derivative

$$f[x_0, x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0}, \quad f[x_1, x_2] = \frac{f[x_2] - f[x_1]}{x_2 - x_1}$$

- Second order difference as a discrete second derivative

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

For the Newton form of an interpolating polynomial, the coefficients are given by divided differences.

Proposition 18.1 (Newton form of interpolating polynomial and divided differences)**.** Let p_n be the interpolating polynomial of degree at most n interpolating f through the points $x_0, \ldots, x_n$. The Newton form of p_n in terms of divided differences is

$$p_n(x) = \sum_{i=0}^{n} f[x_0, \ldots, x_i] \prod_{j=0}^{i-1} (x - x_j).$$

The divided differences give an efficient way to compute the Newton form of p_n through what is known as a divided differences table. Each row of the table corresponds to a point x_i and each column is the next order divided difference. We show how this works through an example.

18.3.4 Divided differences table (Lecture October 22, 2025)

Example 18.3 (divided differences table)**.** Let $f(x) = 2^x$ and let $x_i = 0, \ldots, 3$. The divided differences table is

x_i	$f[x_i]$	$f[x_i, x_{i+1}]$	$f[x_i, x_{i+1}, x_{i+2}]$	$f[x_0, x_1, x_2, x_3]$
0	1			
1	2	$\frac{2-1}{1-0} = 1$		
2	4	$\frac{4-2}{2-1} = 2$	$\frac{2-1}{2-0} = \frac{1}{2}$	
3	8	$\frac{8-4}{3-2} = 4$	$\frac{4-2}{3-1} = 1$	$\frac{1-0.5}{3-0} = \frac{1}{6}$

We then take the diagonal as the coefficients of the interpolating polynomial in Newton form. The resulting Newton form of the interpolating polynomial p_3 is

$$p_3(x) = 1 + x + \frac{1}{2}x(x-1) + \frac{1}{6}x(x-1)(x-2)$$

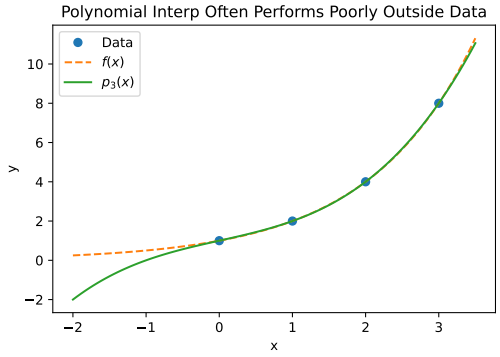


Figure 13: Data and interpolating polynomial from Example 18.3. Notice that the interpolating polynomial does not do well for extrapolating an exponential trend beyond its data.

18.4 Polynomial Interpolation and Root Finding

Polynomial interpolation can be useful in some rootfinding applications. We go over two such techniques.

18.4.1 Inverse Interpolation

The goal of root finding is to find r such that $f(r) = 0$. If f is an invertible function, then we may write $r = f^{-1}(0)$. The idea of inverse interpolation is instead of interpolating $(x_i, f(x_i))$ for $i = 0, \ldots, n$, we interpolate the inverse function by interpolating the points $(f(x_i), x_i)$. Our interpolant $p_n \approx f^{-1}$ and we get an approximate root by writing $r = p_n(0)$. This technique can be used to get an approximate root in one iteration or we can incoporate it into an iterative root finding algorithm, as we'll see in the next example

Example 18.4 (inverse interpolation and secant method)**.** Suppose we have points $(x_0, f(x_0)), (x_1, f(x_1))$. We can construct a linear interpolant of the inverse function using the Newton form:

$$p_1(y) = x_0 + (x_1 - x_0)\frac{y - f(x_0)}{f(x_1) - f(x_0)}$$

To find an approximate root, we evaluate $p_1(0)$:

$$r \approx p_1(0) = x_0 - \frac{(x_1 - x_0)}{f(x_1) - f(x_0)} f(x_0)$$

This formula shows up in the secant method and method of false position.

18.4.2 Companion Matrix

Suppose p_n is a polynomial approximation of f. We hope that the roots of p_n are approximate roots of f. The technique we will use to compute the roots of p_n is called the companion matrix. One advantage of the companion matrix is it enables us to potentially find many distinct roots of a function if there happen to be such roots.

Given a polynomial $p_n(x)$ of degree n in the monomial basis

$$p(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

The companion matrix $\mathbf{C}$ for this polynomial is an $n \times n$ matrix defined as:

$$\mathbf{C} = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & -a_0 \\ 1 & 0 & 0 & \cdots & 0 & -a_1 \\ 0 & 1 & 0 & \cdots & 0 & -a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & -a_{n-1} \end{bmatrix}$$

The companion matrix has an important property that we leave as an exercise.

Exercise 18.1. Show that the eigenvalues of $\mathbf{C}$ are the roots of p_n. *Hint*: Show the characteristic polynomial satisfies $\det(x\mathbf{I} - \mathbf{C}) = p_n(x)$.

Hence, finding the roots of the polynomial p_n is equivalent to finding eigenvalues of $\mathbf{C}$. If we have a good code for solving eigenvalues, then we can find multiple roots of p_n and

These properties make the companion matrix useful for finding roots of polynomials, as the problem of finding roots is transformed into an eigenvalue problem. Standard numerical methods for computing eigenvalues (such as the QR algorithm) can then be applied to find the roots of the original polynomial.

18.5 Vandermonde Matrix

The final topic on constructing polynomials that we will discuss is that of the Vandermonde matrix. Instead of using the Lagrange or Newton form of an interpolating polynomial, we use linear algebra to construct the interpolating polynomial. To construct the interpolating polynomial using the Vandermonde matrix, we start with the monomial basis representation of the polynomial:

Let

$$p_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

be the interpolating polynomial of degree at most n that passes through the points (x_i, y_i) for $i = 0, \ldots, n$. We want to find the coefficients $a_0, a_1, \ldots, a_n$.

For each point (x_i, y_i), we have the equation:

$$y_i = p_n(x_i) = a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_n x_i^n$$

We can write this as a system of linear equations:

$$\underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix}}_{=\mathbf{V}} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

The matrix $\mathbf{V}$ on the left-hand side is called the Vandermonde matrix. The linear system can be written in matrix vector form as

$$\mathbf{Va} = \mathbf{y}$$

where $\mathbf{a} = [a_0, a_1, \ldots, a_n]^T$ is the vector of coefficients we want to find, and $\mathbf{y} = [y_0, y_1, \ldots, y_n]^T$ is the vector of function values.

To solve for the coefficients, we need to solve the above system using techniques we have learned like Gaussian elimination, partial pivoting, etc. Does there exist a solution to the above linear system? The answer is given in the below exercise.

Exercise 18.2. Let $\mathbf{V}$ be the Vandermonde matrix. Show that if the x_i are distinct, then there exists a unique solution $\mathbf{a}$ to $\mathbf{Va} = \mathbf{y}$. *Hint*: Use uniqueness of polynomial interpolation to prove this.

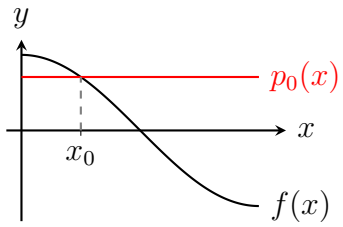
Remark 18.1 (poor conditioning of the Vandermonde matrix)**.** One drawback of this approach is that the Vandermonde matrix is ill-conditioned. This is because the monomials become nearly linearly dependent as n gets larger. We can choose a different basis to represent the polynomial to get a better conditioned matrix.

18.6 Error of Polynomial Interpolation (Lecture October 24, 2025)

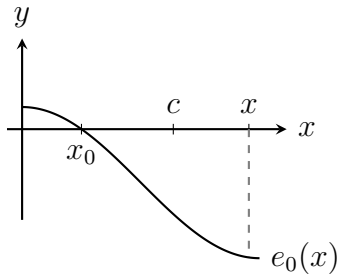
Now that we know a few ways of constructing polynomials, the next question to answer is how accurate is $p_n(x) \approx f(x)$ when x is not an interpolation point? To answer this, we need to develop a theory of polynomial interpolation error.

18.6.1 General theory

We'll build the theory starting from the ground up. Consider the simplest case of interpolating $f(x)$ at x_0 with a degree zero polynomial p_0.



We consider an error function $e_0(x) = f(x) - p_0(x)$. Notice that $e_0(x_0) = 0$.



At any other point x, we have by Mean Value Theorem

$$e_0(x) - \cancel{e_0(x_0)} = e_0'(c)(x - x_0)$$

for some c in between x and x_0. Computing the derivative of e_0, we use the fact that the derivative of p_0 is zero because p_0 is a constant to get

$$e_0'(c) = f'(c) - p_0'(c) = f'(c)$$

Hence,

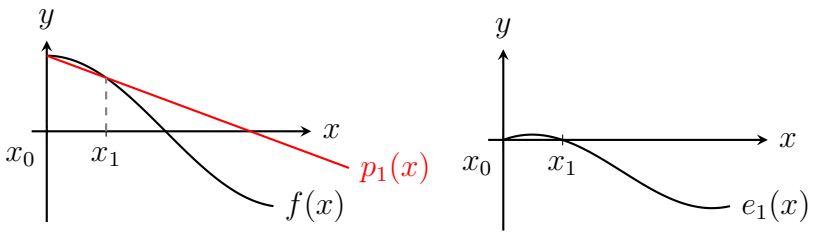
$$f(x) - p_0(x) = e_0(x) = f'(c)(x - x_0),$$

and we can characterize the error between f and p_0 using just f', x, x_0.

Let's highlight a few key features of the argument of what we used.

- We wrote an error function $e_0(x) = f(x) - p_0(x)$.
- We can characterize the error at a point using Mean Value Theorem to get the error in terms of $(x - x_0)$ and e_0'.
- In order to not have any dependence on p_0, we use the fact that $p_0' = 0$.

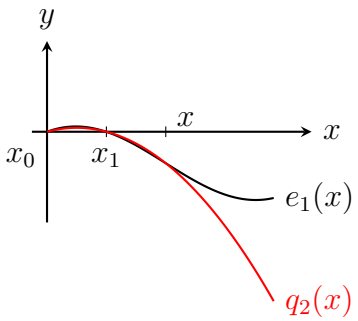
In order to go to the case of a higher polynomial degree, say p_1 interpolates f at x_0, x_1, we repeat the argument by looking at the error function $e_1(x) = f(x) - p_1(x)$.



In order to apply a Mean Value Theorem type argument and lose the dependence on p_1, we need to take 2 derivatives to use $p_1'' = 0$. This will require a repeated application of MVT and requires not looking at the error function but rather an interpolant of the error function.

We look at the error function $e_1(x) = f(x) - p_1(x)$ and now fix a point x to study the error at x. In order to take two derivatives of e_1 and connect it to the error $e_1(x)$, we interpolate e_1 with a degree 2 polynomial q_2 at x_0, x_1, x.

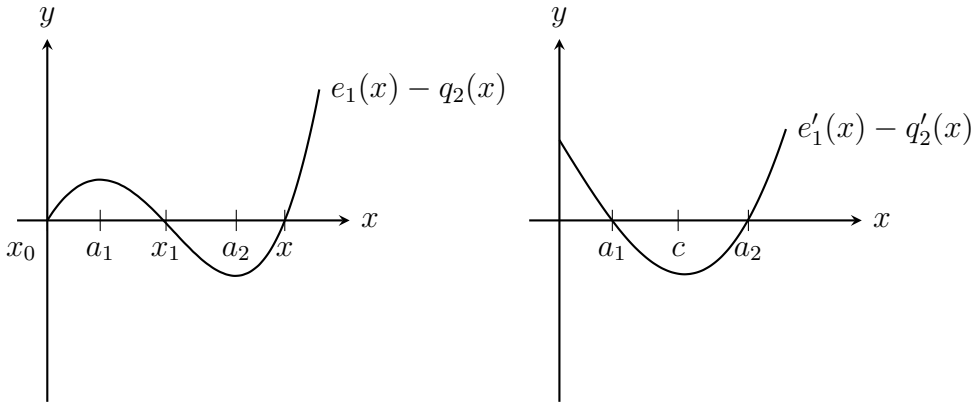
$$q_2(y) = e_1(x)\frac{(y - x_0)(y - x_1)}{(x - x_0)(x - x_1)}.$$



Notice that $q_2''(y) = \frac{2e_1(x)}{(x-x_0)(x-x_1)}$, so taking two derivatives will get us the error at x. Notice that $e_1(y) - q_2(y) = 0$ at $y = x_0, x_1, x$, this means we can apply MVT on each

interval to get a_1, a_2 such that $e_1'(a_i) - q_2'(a_i)$. Repeating MVT again gives a c in between a_1, a_2 such that $e_1''(c) - q_2''(c) = 0$. To summarize, we have

$$f''(c) - \cancel{p_1''(c)} = e_1''(c) = q_2''(c) = \frac{2e_1(x)}{(x - x_0)(x - x_1)}.$$



Rearranging gives

$$f(x) - p_1(x) = \frac{f''(c)}{2}(x - x_0)(x - x_1).$$

where c is some number in between $\min\{x, x_0, x_1\}$ and $\max\{x, x_0, x_1\}$. This pattern holds in general for higher degree polynomials. We summarize it in the following theorem

Theorem 18.2 (error of polynomial interpolation)**.** Let $x_0 < x_1 < \ldots < x_n$ be $n+1$ distinct points. Let p_n interpolate f through the x_i's and assume f is $n+1$ times continuously differentiable. The error at a point x is

$$f(x) - p_n(x) = \frac{f^{(n+1)}(c)}{(n+1)!}(x - x_0)(x - x_1)\cdots(x - x_n) = \frac{f^{(n+1)}(c)}{(n+1)!}\prod_{i=0}^{n}(x - x_i)$$

where $\min\{x, x_0, x_1, \ldots, x_n\} = a \leq c \leq b = \max\{x, x_0, x_1, \ldots, x_n\}$.

18.6.2 Evenly spaced points

The previous theorem depends on the choice of points x_i and the point x and can be hard to find the value on the RHS. One simplification is to consider evenly spaced points. We now go over the special case.

Theorem 18.3 (polynomial interpolation error on evenly spaced points)**.** Thm Let f satisfy the conditions of the last theorem Let p_n be a poly of degree at most n that interpolates f at $n+1$ evenly spaced points in $[a,b]$ with $x_0 = a$ and $x_n = b$. Then for $a \leq x \leq b$ we have

$$|f(x) - p(x)| \leq \frac{1}{4(n+1)} M h^{n+1}$$

where $h = \frac{b-a}{n}$ is the point spacing and $M = \max_{a \leq z \leq b} \left| f^{(n+1)}(z) \right|$

This theorem is often easier to apply to examples.

Example 18.5 (interpolating sin)**.** Say we interpolate $f(x) = \sin(x)$ at $n+1$ points in $[0, \pi]$, then

$$\left| f(x) - p_n(x) \right| \leq \frac{1}{4(n+1)} M h^{n+1} = \frac{h^{n+1}}{4(n+1)}$$

Since $h = \pi/n$, we have

$$\left| f(x) - p_n(x) \right| \leq \frac{\pi^{n+1}}{n^{n+1}} \frac{1}{4(n+1)}$$

If we use 3 points or $n = 2$, we have

$$\left| f(x) - p_n(x) \right| \leq \frac{\pi^3}{2^3} \frac{1}{12} = \frac{\pi^3}{96} \approx .32.$$

Below is a plot of the example for $n = 2$ and a plot of the error as $n \to \infty$, we see that the error goes to zero.

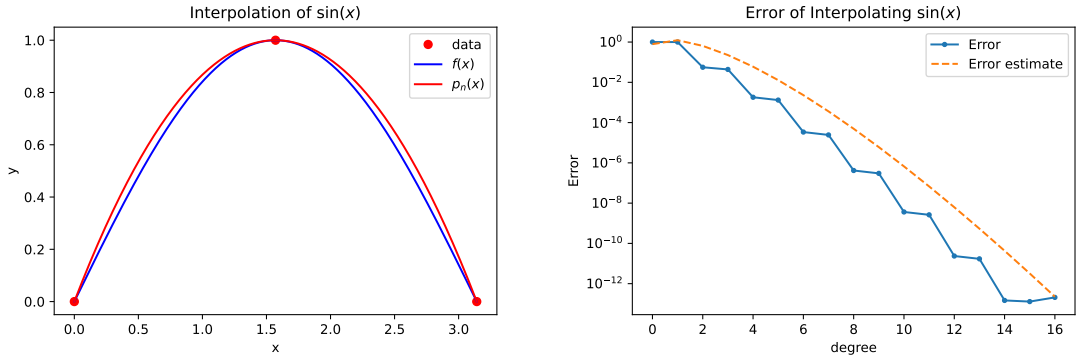


Figure 14: Evenly spaced points interpolation (left) and the error $\to 0$ as $n \to \infty$ (right)

The previous example shows that using evenly spaced points can work well sometimes. However, there are functions where interpolation on evenly spaced points can fail spectacularly. This is called **Runge phenomena**.

Example 18.6 (Runge phenomena)**.** Let $f(x) = \frac{1}{1+x^2}$ and let $[a, b] = [-5, 5]$. Then we have

$$\left|f(x) - p_n(x)\right| \leq \frac{1}{4(n+1)} M h^{n+1} = \frac{10^{n+1} M}{4(n+1)n^{n+1}}.$$

For odd values of n, we have (check this yourself)

$$M = \max_{a \leq z \leq b} \left|f^{(n+1)}(z)\right| \geq f^{(n+1)}(0) = (n+1)!$$

Hence,

$$\left|f(x) - p_n(x)\right| \leq \frac{10^{n+1}(n+1)!}{4(n+1)n^{n+1}} = \frac{10^{n+1} n!}{4n^{n+1}}.$$

Notice that the RHS $\to \infty$ as $n \to \infty$. Our error bound blows up and we get no good approximation of the function by the interpolating polynomial. Is it possible that our error bound is just bad? In this case, no. We can see that the interpolant oscillates with huge oscillations near the boundary.

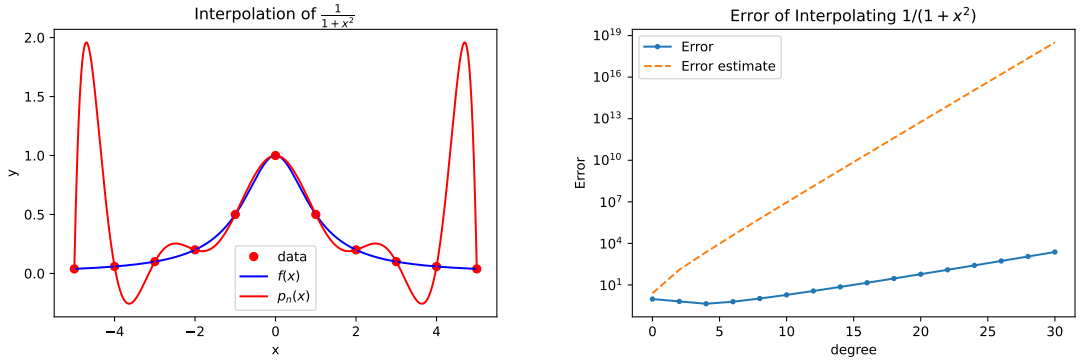


Figure 15: Evenly spaced points interpolation experiencing Runge phenomenon (left) and the error $\to \infty$ as $n \to \infty$ (right)

18.7 Chebyshev Polynomials and Interpolation on the Chebyshev Points (Lecture October 27, 2025)

We now introduce a set of points that will perform much better than evenly spaced points. These are known as the **Chebyshev points**, which are the roots of Chebyshev polynomials, which are defined on $[-1, 1]$.

18.7.1 Chebyshev polynomials

We start by introducing the Chebyshev polynomials and showing some basic properties

Definition 18.2 (Chebyshev polynomials)**.** The nth Chebyshev polynomial T_n is defined for $-1 \leq x \leq 1$ as

$$T_n(x) = \arccos(n \cos x)$$

At first glance, these appear to not be polynomials. As a matter of fact, they are. Let's check the first few

$$\begin{aligned} T_0(x) &= \arccos(0) = 1 \\ T_1(x) &= \arccos(\cos x) = x \\ T_2(x) &= \arccos(2 \underbrace{\cos x}_{=y}) = 2\cos^2 y - 1 = 2x^2 - 1 \end{aligned}$$

To see that T_n is a polynomial in general, we use the following recursion formula

Lemma 18.1 (recursion formula for Chebyshev polynomials)**.** The Chebyshev polynomials on $[-1, 1]$ satisfy

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).$$

Since T_0, T_1 are polynomials, this tells us that T_n is a polynomial for all n.

Proof. Write $y = \arccos x$ and use the angle addition formula

$$\cos((n+1)y) = \cos(ny)\cos(y) \mp \sin(ny)\sin(y).$$

□

18.7.2 Chebyshev points for interpolation

On $[-1, 1]$, the n Chebyshev points are defined as the roots of T_n. We now derive these points

$$0 = T_n(x) = \arccos(n \underbrace{\cos x}_{=y}) \qquad \Longrightarrow \qquad y = \frac{2j+1}{2n}\pi \text{ for } i = 0, \ldots n-1$$

so the n roots of T_n are

$$x_i = \cos\left(\frac{2i+1}{2n}\pi\right) \text{ for } i = 0, \ldots, n-1$$

The values y_i can be viewed as evenly spaced over the arc of a semicircle, and x_i are the projection of these points on the x axis. This bunches more points onto the ends of the interval, which is precisely where evenly spaced points failed last time.

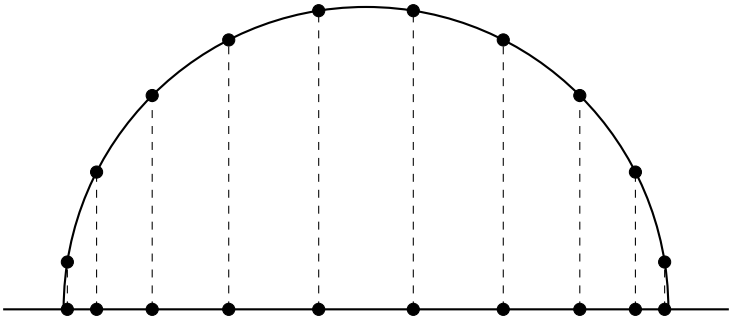


Figure 16: Evenly spaced y_i on the circle and their projections onto the x axis giving the Chebyshev points x_i. Notice that the Chebyshev points concentrate near the edge of the domain.

A key feature of these points is they give better control over the error on the interval $[-1, 1]$. We start by showing an important quantity in the first interpolation error theorem is small.

Lemma 18.2. Let x_i be the $n+1$ Chebyshev points on $[-1, 1]$, then

$$\left|\prod_{i=0}^{n}(x - x_i)\right| \leq \frac{1}{2^n} \text{ for } -1 \leq x \leq 1$$

Combining this with the first interpolation error theorem leads to the following theorem.

Theorem 18.4 (error of interpolating through Chebyshev points on -1 to 1)**.** Let x_i be the $n+1$ Chebyshev points in $[-1, 1]$, and suppose p_n interpolates of through x_i, then

$$\left|f(x) - p_n(x)\right| \leq \frac{1}{2^n(n+1)!}M$$

where $M = \max_{-1\leq x\leq 1}|f^{(n+1)}(x)|$.

18.7.3 Chebyshev interpolation on different intervals

Notice that the whole discussion has focused on the interval $[-1, 1]$. What happens when want to interpolate through f on a different interval? We can use all our previous work and just introduce a change of coordinates.

Let $\hat{x}$ be a value in $[-1, 1]$ and define

$$x = \frac{b-a}{2}\hat{x} + \frac{b+a}{2}.$$

Notice that this is change of coordinates from $[-1, 1]$ to $[a, b]$. To use the new Chebyshev points, all we have to do is interpolate through the transformed points

$$x_i = \frac{b-a}{2}\hat{x}_i + \frac{b+a}{2}, \quad \text{where} \quad \hat{x}_i = \cos\left(\frac{2i+1}{2n}\pi\right) \quad \text{for} \quad i = 0, \ldots, n-1.$$

These new points will give us a reasonable set of points to interpolate through. What is the error in this case? All we need to do is try to transform our interpolation problem on $[a, b]$ to $[-1, 1]$.

Theorem 18.5 (Chebyshev interpolation error on arbitrary interval)**.** Let f be $n+1$ times continuously differentiable on $[a, b]$ and suppose p_n interpolates f through the $n + 1$ Chebyshev points on $[a, b]$. The interpolation error for any $a \leq x \leq b$ satisfies

$$|f(x) - p_n(x)| \leq \left(\frac{b-a}{2}\right)\left(\frac{b-a}{4}\right)^n \frac{1}{(n+1)!}M$$

where $M = \max_{-1\leq x\leq 1}\left|f^{(n+1)}(x)\right|$

Proof. We prove this by transforming our problem to be a new problem on $[-1, 1]$ and applying the previous theorem. Define

$$\hat{f}(\hat{x}) = f\left(\tfrac{b-a}{2}\hat{x} + \tfrac{b+a}{2}\right)$$
$$\hat{p}_n(\hat{x}) = p_n\left(\tfrac{b-a}{2}\hat{x} + \tfrac{b+a}{2}\right)$$

Notice that $\hat{p}_n$ interpolates $\hat{f}$ through the $n + 1$ Chebyshev points on $[-1, 1]$ Hence, we can apply our interpolation error theorem for Chebyshev interpolation on $[-1, 1]$ to get

$$\left|\hat{f}(\hat{x}) - \hat{p}_n(\hat{x})\right| \leq \frac{1}{2^n(n+1)!}\max_{-1\leq\hat{x}\leq 1}\left|\frac{d^{n+1}}{d\hat{x}^{n+1}}\hat{f}(\hat{x})\right|$$

The LHS satisfies

$$\left|\hat{f}(\hat{x}) - \hat{p}_n(\hat{x})\right| = \left|f(x) - p_n(x)\right|,$$

which is what we want. The RHS can be dealt with via chain rule. Notice that if

$$\hat{f}(\hat{x}) = f(x) \quad \text{for} \quad x = \frac{b-a}{2}\hat{x} + \frac{b+a}{2}.$$

Taking one derivative with respect to $\hat{x}$ and applying chain rule yields

$$\frac{d}{d\hat{x}}\hat{f}(\hat{x}) = f'(x)\frac{b-a}{2} \quad \text{for} \quad x = \frac{b-a}{2}\hat{x} + \frac{b+a}{2}.$$

Repeating this $n+1$ times yields

$$\frac{d^{n+1}}{d\hat{x}^{n+1}}\hat{f}(\hat{x}) = f^{(n+1)}(x)\left(\frac{b-a}{2}\right)^{n+1} \quad \text{for} \quad x = \frac{b-a}{2}\hat{x} + \frac{b+a}{2}.$$

Inserting to the initial bound, we have

$$\left|f(x) - p_n(x)\right| \leq \frac{1}{2^n(n+1)!}\left(\frac{b-a}{2}\right)^{n+1}\max_{a\leq x\leq b}\left|f^{(n+1)}(x)\right| = \left(\frac{b-a}{2}\right)\left(\frac{b-a}{4}\right)^n\frac{1}{(n+1)!}M$$

which finishes the proof. □

Remark 18.2. A common strategy with numerical methods is to implement and study the method on some reference interval like $[-1, 1]$. To implement the method for more general situations, we need coordinate transformations to transform the problem back to the original interval. We often need chain rule to analyze the method on the new interval. Be careful with chain rule as it can often get tricky.

Let's now see how Chebyshev compares with evenly spaced interpolation for an example.

Example 18.7. Say $f(x) = \sin x, \quad a = 0, b = \pi$. Then $\left|f(x) - p_n(x)\right| \leqslant \frac{\pi}{2} \cdot \left(\frac{\pi}{4}\right)^n \frac{1}{(n+1)!}$ Using 3 points or $n = 2$, we have

$$\left|f(x) - p_n(x)\right| \leqslant \frac{\pi}{2}\left(\frac{\pi}{4}\right)^2\frac{1}{3!} \approx 0.16$$

Here, using the Chebyshev points only gives a modest improvement over evenly spaced points.

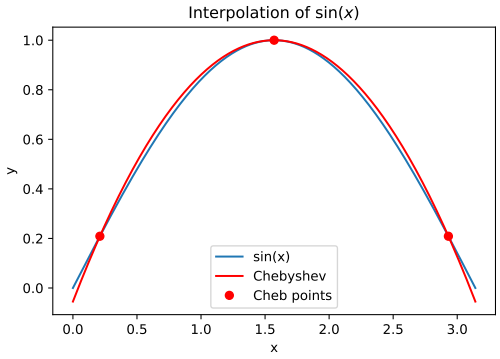


Figure 17: Interpolation of $f(x) = \sin x$ using Chebyshev points.

However, for functions like $f(x) = \frac{1}{1+x^2}$, using the Chebyshev points is much more accurate and will converge.

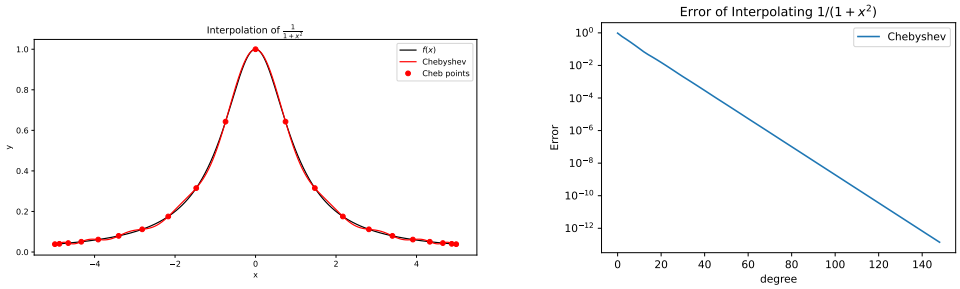


Figure 18: Interpolation of the Runge example $f(x) = 1/(1+x^2), \quad a = -5, b = 5$ using the Chebyshev points. We expect the Chebyshev interpolants to converge as $n \to \infty$. Above are plots showing computational evidence of this.