Numerical Methods

Xuemei Chen

New Mexico State University

(Last updated on April 9, 2020)

Contents

Notations

Before this course, you need to be familiar with

- Calculus I
- Calculus II: Integrations. Sequences and Series.
- It is not necessary, but helpful to know some multivariate Calculus
- Some basic Linear Algebra.

Notations

- $\{x|$ descriptions of $x\}$ is the set of all x that meets the description after the ":" sign. For example, $\{x|x^2-1<0\}$ is the set of all number x such that $x^2-1<0$. We can solve this inequality further and see that $\{x|x^2-1<0\}=(-1,1)$

  Remark: Some books use colon instead of verticle bar as $\{x:$ descriptions of $x\}$
- $\in$: belongs to/is contained in. For example, $a\in\{x|x^2-1<0\}$ means that a is a number satisfying $a^2-1<0$.
- A vector in $\mathbb{R}^n$ is a column vector in default.
- In all exercises at the end of each chapter, * means extra credit problems.

Chapter 1

Floating Point Arithmetic

The arithmetic performed by a calculator or computer is different from the arithmetic that we use in our algebra or calculus class. In the ideal world, we permit numbers with an infinite number of digits. For example, $\sqrt{2}$ is the unique positive number that, when multiplied by itself, produces the integer 2. It is an irrational number, which means that there are infinitely many digits associated with $\sqrt{2}$ when represented in decimals. In the computational world, we are only able to assign a fixed and finite number of digits to each number.

Try add up 0.1 and 0.2 in Python, you will get

```
>>> 0.1 + 0.2
0.30000000000000004
```

This can cause serious issues and create bugs. See Section 5.1 of [2]. The above error is caused by rounding in floating point numbers, with which numerical analysis is traditionally concerned.

1.1 Floating Point Representation

Floating point numbers are represented in terms of a base β, a precision p, and an exponent e. For example, in base 10, 0.34 can be represented as 3.4×10^{-1} or 34×10^{-2}. In order to guarantee uniqueness, we take a floating point number to have the specific form.

$$\pm \left(d_0 + d_1\beta^{-1} + \cdots + d_{p-1}\beta^{-(p-1)}\right) \times \beta^e, \tag{1.1}$$

where each d_i is an integer in $[0, \beta)$ and $d_0 \neq 0$. Such a representation is said to be a *normalized floating point number*. In this representation, $d_0 + d_1\beta^{-1} + \cdots + d_{p-1}\beta^{-(p-1)}$ is called *mantissa* (or *significand*). Here are a few examples where $\beta = 10, p = 5$ and e is in the range $-10 \leq e \leq 10$:

$$1.2345 \times 10^8, \qquad 3.4000 \times 10^{-2}, \qquad -5.6780 \times 10^5$$

In the above examples, we are using base $\beta = 10$ to ease into the material. Computers work more naturally in binary (base 2). When $\beta = 2$, each digit d_i is 0 or 1 in equation (1.1). I will leave it to the readers to review binary representation, but some examples are listed below:

- Conversion from binary to decimal:
  a. $11.0101_2 = 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} = 2 + 1 + 0.25 + 0.0625 = 3.3125$
  b. $1100_2 = 1.1_2 \times 2^3 = (1 \cdot 2^0 + 1 \cdot 2^{-1}) \cdot 2^3 = 12$
  c. $0.01_2 = 1.00_2 \times 2^{-2} = 0.25$

- Conversion from decimal to binary:

  a. $41 = 2^5 + 9 = 1 * 2^5 + 1 * 2^3 + 1 * 2^0 = 101001_2$

  b. To figure out the binary representation for 0.1, we first observe that $2^{-4} = 0.0625$ is the biggest component of 0.1. The leftover $0.1 - 0.0625 = 0.0375 > 2^{-5}$. So $0.1 = 2^{-4} + 2^{-5} + 0.00625 = 2^{-4} + 2^{-5} + 2^{-4} * 0.1 = 2^{-4} + 2^{-5} + 2^{-4}(2^{-4} + 2^{-5} + 2^{-4} * 0.1)$. This can keep going, so

$$0.1 = 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + \cdots = 0.0\overline{0011}_2 = 1.\overline{1001}_2 \times 2^{-4}. \tag{1.2}$$

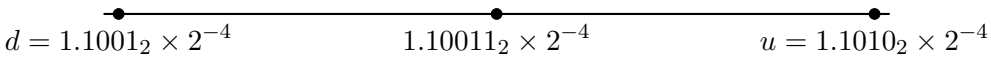
As seen, if a base (subscript) is not specified, then it is understood that it is a decimal number.

The representation in (1.2) has to be rounded at some point, and this is causing the computation inaccuracy at the beginning of this chapter.

There are usually 4 rounding models: rounding down, rounding up, rounding towards 0, and rounding to nearest. rounding to the nearest is, either round down or round up, whichever is closer. In case of a tie, we choose the one whose least significant (rightmost) bit is smaller. Given a real number x, we denote its floating point representation by $\text{fl}(x)$. For convenience of illustration, we use examples where we retain 4 digits after the binary point (this means the representation has precision $p = 5$).

Example 1.1. Round $0.1 = 1.1001100\cdots_2 \times 2^{-4}$ to the nearest so that there are only 4 digits after the binary point. This means that in this machine, $p = 5$.

Rounding up will be $u = 1.1010_2 \times 2^{-4}$. Rounding down will be $d = 1.1001_2 \times 2^{-4}$. To see which one is nearest, we can compute the midpoint of u and d.



0.1 is clearly bigger than the middle point, so we pick $1.1010_2 \times 2^{-4}$ with rounding to the nearest. We can write $\text{fl}(0.1) = 1.1010_2 \times 2^{-4}$ in this particular machine.

Question: In this same machine, meaning we still have $p = 5$ and we still use rounding to the nearest, what is $\text{fl}(1.10011_2 \times 2^{-4})$? (Hint: in binary, we break the tie by choosing the one whose least significant (rightmost) bit is 0.)

In the model where $\beta = 10, p = 5$, 1 is represented as 1.0000×10^0, the number that is closest to 1 (and bigger than 1) is 1.0001×10^0. There is a gap 10^{-4}. This gap is relative. The closest number to 1.0000×10^4 is 1.0001×10^4 (taking the bigger one again). This means we cannot represent any number between 10000 and 10001 with this machine. The gap (=1) is a lot bigger now, but the relative gap is still $1/10000 = 10^{-4}$.

In general, with base β and precision level p, this gap (between 1 and its closest number) is defined to be the *machine precision* or *machine epsilon* ϵ_m.[1] We have $\epsilon_m = \beta^{1-p}$. Machine epsilon is related to the *round-off error*, which is the error that results from replacing a number with its floating point form. The approximation x^* to x has *absolute error* $|x - x^*|$ and *relative error* $\dfrac{|x - x^*|}{|x|}$, provided that $x \neq 0$.

Example 1.2. Let us use base 10 with $p = 3$, in which case the machine precision is $\epsilon_m = 10^{-2}$. If we use the rounding to nearest method, 1.345 becomes 1.34 (both 1.34 and 1.35 are equally

[1] Some sources define the machine precision to be half of this gap.

close, we break the tie by choosing the number whose right most bit is smaller). Relative error $= \left|\frac{1.345 - 1.34}{1.345}\right| < \frac{0.005}{1} = 0.5 * \epsilon_m$ (The relative error is about 0.003.)

The number 1.345×10^4 will be rounded to 1.34×10^4. The absolute error is very big, but the relative error is $\left|\frac{1.345 \times 10^4 - 1.34 \times 10^4}{1.345 \times 10^4}\right| = \left|\frac{1.345 - 1.34}{1.345}\right|$, the same as before.

Example 1.3. Following Example 1.1, the absolute error in representing 0.1 is $|0.1 - \text{fl}(0.1)|$, which is less than half of the difference between u and d. The relative error is

$$\frac{|0.1 - \text{fl}(0.1)|}{0.1} < \frac{2^{-1}(u-d)}{d} = \frac{2^{-5} \cdot 2^{-4}}{1.1001_2 \times 2^{-4}} < 2^{-5}.$$

If we use rounding to the nearest, one has

$$\frac{|\text{fl}(x) - x|}{|x|} \leq 0.5\epsilon_m. \tag{1.3}$$

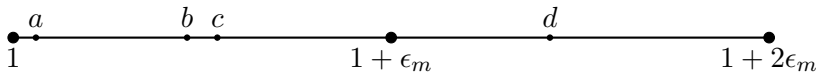
In other words, we may write

$$\text{fl}(x) = x(1+\epsilon), \quad \text{where } |\epsilon| < 0.5\epsilon_m.$$

The idea of machine epsilon is that

- Two numbers may be indistinguishable in a machine if their relative error is less than $0.5\epsilon_m$;
- Two numbers are definitely distinguishable in a machine if their relative error is more than ϵ_m.

Let's consider four numbers a, b, c, d as indicated below:



Using rounding to the nearest, $\text{fl}(a) = \text{fl}(b) = 1$ and a, b are within $0.5\epsilon_m$ relative error. $\text{fl}(b) \neq \text{fl}(c)$ even though they are within $0.5\epsilon_m$ relative error. $\text{fl}(c) = \text{fl}(d)$ even though they are almost ϵ_m apart. The only way to guarantee that two numbers have different representations is make sure they are relatively ϵ_m apart, like b and d.

1.2 Floating Point Operations

On a computer, all computations are reduced to $+, -, \times$, or $/$. Due to rounding, floating point arithmetic are different from the classical ones, and we will denote these *floating point operations (flop)* by $\oplus, \ominus, \otimes, \oslash$.

Example 1.4 (Addition)**.** Suppose we retain 4 digits after the binary point. To add up two numbers with different powers, we first need to adjust one of the numbers to align the mantissa (use bigger power), as:

$1.1000_2 \times 2^1 \oplus 1.1001_2 \times 2^{-1} = 1.1000_2 \times 2^1 \oplus \text{fl}(0.011001_2) \times 2^1 = 1.1000_2 \times 2^1 \oplus 0.0110_2 \times 2^1 = \text{fl}(1.1110_2) \times 2^1 = 1.1110_2 \times 2^1$.

Fundamental axiom of floating point arithmetic: Let $*$ be one of the operations $(+, -, \times, /)$, and let $\circledast$ be its floating point analogue, then for all **floating point numbers** x, y, we have that

$$x \circledast y = (x * y)(1 + \epsilon), \text{ for some } \epsilon \text{ such that } |\epsilon| \leq \epsilon_m.$$

or equivalently

$$\left| \frac{x \circledast y - x * y}{x * y} \right| \leq \epsilon_m.$$

We will not prove this, but we will quickly check this axiom in one example.

Example 1.5. We assume a machine, base 2, can only have 4 digits precision after the binary point, rounding to the nearest. Let $x = 1.1010_2 \times 2^{-4}$ and $y = 1.1010_2 \times 2^{-3}$ be two floating point numbers. (x is the floating point representation of 0.1 by Example 1.1 and y is the floating point representation of 0.2.)

Similar to Example 1.4, we first rewrite x as $0.11010_2 \times 2^{-3}$, which rounds to $0.1110_2 \times 2^{-3}$ (break the tie by picking the one whose right most bit is 0).

$x \oplus y = \text{fl}(0.1110_2 \times 2^{-3} + 1.1010_2 \times 2^{-3}) = \text{fl}(10.1000_2 \times 2^{-3}) = 1.0100_2 \times 2^{-2} = 0.3125$.

The relative error for this addition is

$$\left| \frac{x \oplus y - (x + y)}{x + y} \right| = \left| \frac{1.0100_2 \times 2^{-2} - 1.001110_2 \times 2^{-2}}{1.001110_2 \times 2^{-2}} \right| = \frac{0.000010_2}{1.001110_2} < 2^{-5} < 2^{-4} = \epsilon_m.$$

Finally, we present the example of adding 0.1 and 0.2, but in a "very outdated" machine.

Example 1.6. If we assume a machine, base 2, precision 5. This is how it adds up 0.1 and 0.2.

Step 1: 0.1 is rounded to $\text{fl}(0.1) = 1.1010_2 \times 2^{-4}$ as shown in Example 1.1.

Step 2: $\text{fl}(0.2) = 1.1010_2 \times 2^{-3}$ since 0.2 is twice of 0.1.

Step 3: $\text{fl}(0.1) \oplus \text{fl}(0.2) = 0.3125$ as shown in Example 1.5.

The overall relative error is $|0.3 - 0.3125|/0.3 \approx 0.042$.

To sum up, if we use $m(\cdot)$ to indiate the machine output using floating pont numbers, then $m(x + y) = \text{fl}(x) \oplus \text{fl}(y) = (x(1 + \epsilon_1) + y(1 + \epsilon_2))(1 + \epsilon_3)$

1.3 IEEE 754 Standard

In 1985, the Institute for Electrical and Electronic Engineers (IEEE) published the *Binary Floating Point Arithmetic Standard 754-1985* which is followed by all microcomputer manufacturers. The double precision real numbers require a 64-bit representation. The first bit is a sign indicator, denoted s. This is followed by an 11-bit exponent c, and a 52-bit binary fraction f (also named mantissa or significand). See the following example:

$$0.1 = +1.\underbrace{1001100110011001100110011001100110011001100110011010_2}_{\text{52 digits mantissa}} \times 2^{-4},$$

This is `0.1000000000000000055511151231257827021181583404541015625` in decimal. In the IEEE 754 standard, $p = 53, \epsilon_m = 2^{-52} \approx 2.22 \times 10^{-16}$. To retrieve this machine epsilon, type `eps = numpy.finfo(float).eps` in Python. Moreover, execute `1 + 0.4*eps`, what do you get?

The default IEEE standard is rounding to nearest. The 11-bit exponent can represent $2^{11} = 2048$ numbers. The numbers 0 and 2047 are reserved for other use (like $0, \infty$, NaN), so c can only take on integers ranging from 1 to 2046. In order to include both positive and negative exponents,

we will subtract 1023 from c and the actual exponent can be all integers from -1022 to 1023. To summarize, a double precision number has the form

$$(-1)^s * 2^{c-1023} * (1+f) \tag{1.4}$$

Consider the machine number

$$\underbrace{0}_{s} \quad \underbrace{10000000011}_{\text{11-bit } c} \quad \underbrace{1011100100010000000000000000000000000000000000000000}_{\text{52-bit } f},$$

converted to decimal, $c = 2^{10} + 2^1 + 2^0 = 1027$, $f = 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-12}$, so this number in decimal form is

$$(-1)^0 * 2^{1027-1023} * (1 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-12}) = 27.56640625.$$

The smallest positive number that can be represented is $s = 0, c = 1, f = 0$, which is $2^{-1022} \approx 2.225 \times 10^{-308}$.

1.4 Errors in Scientific Computing

Example 1.7. The quadratic formula states that the roots of $ax^2 + bx + c = 0$ are

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}. \tag{1.5}$$

Let us apply this formula to the equation $x^2 + 80x + 1 = 0$, whose roots are approximately $x_1 = -0.01250195, x_2 = -79.98750$.

If the roots are computed in a machine with $\beta = 10, p = 4$. We would get

$$\text{fl}(x_1) = \frac{-80.00 + \text{fl}(\sqrt{6400 - 4.000})}{2.000} = \frac{-80.00 + 79.97}{2.000} = -0.015$$

whose relative error is about 0.0025/0.0125=20%. This is quite large of an error. This big roundoff error is caused by subtracting two nearly equal number. (In this example, we have b^2 much larger than $4ac$, so b and $\sqrt{b^2 - 4ac}$ are nearly equal.)

On the other hand, the calculation of x_2 involves the addition of two nearly equal numbers, which presents no problem as

$$\text{fl}(x_2) = \frac{-80.00 - \text{fl}(\sqrt{6400 - 4.000})}{2.000} = \frac{-80.00 - 79.97}{2.000} = -79.98$$

with a small relative error $\approx 0.01/79.99 = 1.25 \times 10^{-4}$.

There is a way to get a more accurate approximation for x_1 in this same machine!

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \cdot \frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}} = \frac{-2c}{b + \sqrt{b^2 - 4ac}}. \tag{1.6}$$

This is an equivalent formula for x_1, but makes a big difference in a computer since (1.6) **avoids subtracting two nearly equal numbers**. By (1.6), we have

$$\text{fl}(x_1) = \frac{-2.000}{80.00 + \text{fl}(\sqrt{6400 - 4.000})} = \frac{-2.000}{80.00 + 79.97} = -0.01250.$$

This is a much better approximation.

Due to rounding, subtracting two nearly equal numbers will cause a big relative error. Such an error is called cancellation error and we should try to avoid it if possible. As shown in Example 1.7, rationalizing a formula could be used when appropriate.

The following example is from Example 6 of Section 1.2 of [1].

Example 1.8. Evaluate $f(x) = x^3 - 6.1x^2 + 3.2x + 1.5$ at $x = 4.71$ using $\beta = 10, p = 3$.

The exact value is $f(4.71) = -14.263899$. With rounding, we will have

$$\text{fl}(4.71^3) - 6.1\text{fl}(4.71^2) + 3.2\text{fl}(4.71) + 1.5 = 104 - 135 + 15.1 + 1.50 = -14.4 \tag{1.7}$$

which produces a relative error $|-14.4 + 14.263899|/14.263899 \approx 0.00954$.

As an alternative approach, $f(x)$ can be written in a nested manner as

$$f(x) = ((x - 6.1)x + 3.2)x + 1.5. \tag{1.8}$$

Evaluating (1.8) at $x = 4.71$ using $p = 3$ gives

$$\text{fl}\{(\text{fl}[(4.71 - 6.1)4.71] + 3.2)4.71\} + 1.5 = \text{fl}\{(-6.55 + 3.2)4.71\} + 1.5 = -15.8 + 1.5 = -14.3$$

whose relative error is reduced to 0.0025.

Why does the nested form provide a more accurate evaluation? This is because the nested form requires fewer flops to evaluate, and consequently fewer roundings. Indeed, the original evaluation (1.7) requires 2+2+1 multiplications and 3 additions, so a total of 8 flops. The nested form (1.8) requires only 5 flops.

Let $p(x) = \sum_{i=0}^{n} a_i x^i$ be its original form, then it requires $n + (n-1) + \cdots + 2 + 1 = \dfrac{n(n+1)}{2}$ multiplications and n additions, so a total of $\dfrac{n(n+1)}{2} + n = \dfrac{n(n+3)}{2}$ flops are needed.

It is easy to see that the nested form of $p(x) = \sum_{i=0}^{n} a_i x^i$ is

$$p(x) = a_0 + x\Big(a_1 + x\big(a_2 + x(a_3 + \cdots + x(a_{n-1} + xa_n)\cdots)\big)\Big).$$

In this form, the flop count is

$$\underbrace{2 + 2 + \cdots + 2}_{n} = 2n,$$

which is much less than the original form.

Nested multiplication should be performed whenever a polynomial is evaluated. This is because the nested form minimizes the number of flops needed, hence minimizing the rounding off error. The nested form is faster (fewer flops) and more accurate.

Chapter 1 Exercises

1. Convert binary to decimal.
   (a) 110_2
   (b) 11.0101_2
2. Convert decimal to binary

(a) 11.5

(b) 1.3125

3. Find the **normalized** floating point representation

   (a) 10.345, base 10, $p = 4$, rounding to the nearest.

   (b) 10.395, base 10, $p = 4$, rounding to the nearest.

   (c) 295, base 10, $p = 2$, rounding to the nearest.

   (d) 1.0101_2, base 2, $p = 3$, rounding up.

   (e) 1.0101_2, base 2, $p = 3$, rounding down.

   (f) 1.0101_2, base 2, $p = 3$, rounding to the nearest.

   (g) 1.0101_2, base 2, $p = 4$, rounding to the nearest.

4. Let us use base 10 with $p = 2$. If we use the rounding to the nearest method, what is the relative error of 1.23? What is the relative error of 10.61?

5. Following Exercise 3(g), what is the machine epsilon? what is the relative error in rounding this number? Does the rule (1.3) check out?

6. Directly add in binary (No rounding involved).

   (a) $0.111_2 + 11.101_2$

   (b) $1.1010_2 \times 2^{-4} + 1.1010_2 \times 2^{-3}$.

7. What is the biggest number that can be represented in IEEE 754 standard using double precision? Express this number in the form of $2^a - 2^b$.

8. In a machine of $\beta = 2, p = 3$, compute the machine result of 0.1+0.2 (rounding to the nearest).

9. In a machine of $\beta = 10, p = 4$, compute the machine result of $6400.1 + 4.12$ (rounding to the nearest).

10. This problem is from Chapter 5 problem 15 of [2].

    In the 1991 Gulf War, the Patriot misslie defense system failed due to roundoff error. The troubles stemmed from a computer that performed the tracking calculations with an internal clock whose integer values in tenths of a second were converted to seconds by multiplying by a 24-bit binary approximation to 0.1:

    $$0.1 \approx 0.00011001100110011001100_2$$

    (a) Convert the binary number above to a fraction. Call it x.

    (b) Compute the absolute difference between 0.1 and x.

    (c) What is the time error in seconds after 100 hours of operation (i.e., the value of $|360,000 - 3,600,000x|$)?

    On February 25, 1991, a Patriot battery system, which was to protet the Dhahran Air Base, had been operating for over 100 consecutive hours. The roundoff error caused the system not to track an incoming Scud missle, which slipped through the defense system and detonated on US Army barracks, killing 28 American soldiers.

11. Find the most accurate approximation of the roots of $x^2 - 75.7x + 1$ with $p = 4$. Compute the relative errors for both roots.

12. Given $p(x) = 1 + 2x + 3x^2 + 3x^3 + 4x^4$,

    (a) Find the nested form of $p(x)$.

    (b) How many flops are needed to evaluate this polynomial at one point, in the original form, and in the nested form?

Chapter 1 Python Exercises

13. (a) Define a vector v whose entries are [1, 0.01, 0.02, ... , 5]. Do not print.

    (b) Define a 10×10 matrix whose entries are the integers 1 to 100 ordered column-wise. Print the matrix. [Hint: look up `np.reshape`]

    (c) Plot $y = \cos x$ and $y = x$ on the interval [0,1] in the same graph.

14. Given the equation $x^2 - 100000x + 1 = 0$, use formula (1.5) and formula (1.6) to compute two different approximations of x_1. Print both values in the exponential notation with 15 digits after the decimal point. Which formula produces a better approximation of x_1?

15. Given $p(x) = x^9 - 18x^8 + 144x^7 - 672x^6 + 2016x^5 - 4032x^4 + 5376x^3 - 4608x^2 + 2304x - 512$.

    (a) Let $n(x)$ be the nested form of $p(x)$. Define $p(x), n(x)$, and $s(x) = (x - 2)^9$. All 3 functions need to be able to take in arrays.

    (b) $p(x) = n(x) = s(x)$ algebraically, but makes a difference in computers. Plot all three functions on the interval [1.92, 2.08] (with 0.001 increment). You need to place all 3 plots in a 1×3 grid. Be sure to give a title to each subplot to indicate which is which.

    (c) Explain the difference in three plots in (b).

Chapter 2

Solving nonlinear equations

If we recall the equations that we are able to solve, probably we would realize that there are very few. We can surely solve $x^2 - x - 1 = 0$ using the famous quadratic formula, but have you ever wondered formulas for finding roots of polynomials whose degree is higher than 2? Unfortunately, even as simple as a polynomial equation $x^5 - x - 1 = 0$, a math Ph.D feels hopeless to find an exact answer by analysis. In fact, there are no formula for finding roots of polynomials of degree bigger than 4. Equations like $\cos x = x$ do not have a closed form solution either.

A linear equation has the form $ax = b$. It is trivial to solve if x is a scalar. When x is a vector, we often write $Ax = b$, and is a major topic itself.

In this chapter, we discuss several iterative methods for solving nonlinear equations when x is a scalar. Direct methods are formula or procedures that will produce the exact solution. Iterative methods produce a sequence $\{x_1, x_2, \cdots, x_n, \cdots\}$ that (hopefully) converges to the true solution x.

2.1 The Bisection Method

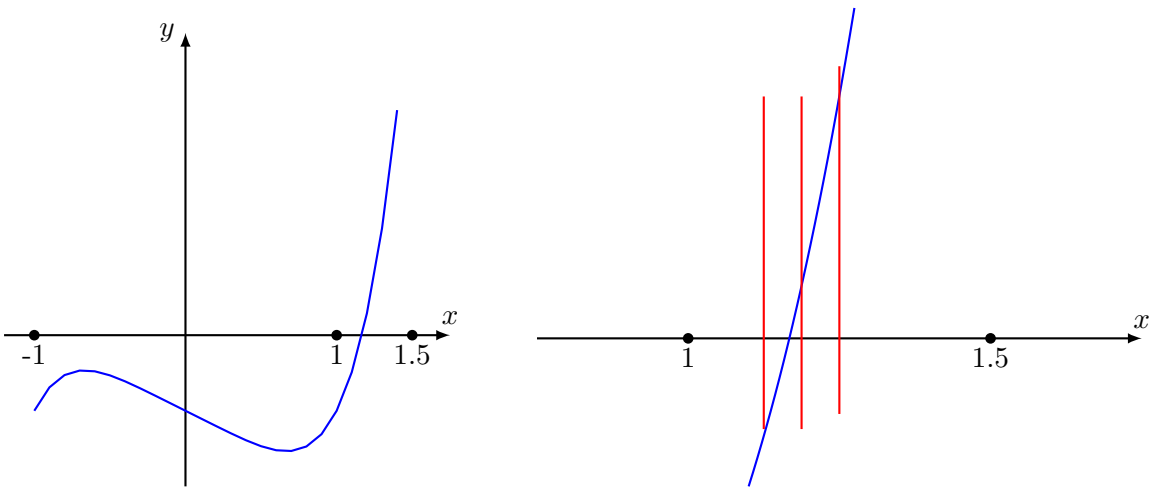


Figure 2.1: Bisection Method. The right is an enlargement of the left.

This is a simple, yet efficient method that is often covered in Calculus I as an application of the Intermediate Value Theorem (IVT). See Theorem 6.1.

Take $f(x) = x^5 - x - 1$ as an example, whose graph is shown on the left of Figure 2.1. Since $f(1) = -1 < 0$ and $f(1.5) \approx 5.09 > 0$, then by the IVT, there is $c \in [1, 1.5]$ such that $f(c) = 0$, or

equivalently, f has a root c in [1,1.5].

To have a more precise estimate of c, we evaluate f at the middle point $\frac{1+1.5}{2} = 1.25$. $f(1.25) \approx 0.8 > 0$. By the IVT again, we conclude that the root $c \in [1, 1.25]$. We can keep bisecting the interval and eventually find the root to a sufficient precision. See the right of Figure 2.1.

Note that the key of the Bisection method is:

1. The function needs to be continuous.
2. One needs to be given initial points a, b such that $f(a)$ and $f(b)$ have different signs.

The Bisection Algorithm

Input:	a_0, b_0 such that $f(a_0), f(b_0)$ have different signs.
Output:	an approximation of a root of f
Repeat:	for $n \geq 0$
	1. $p_n = \frac{a_n + b_n}{2}$
	2. if $f(a_n)f(p_n) < 0$, set $a_{n+1} = a_n, b_{n+1} = p_n$
	otherwsie, set $a_{n+1} = p_n, b_{n+1} = b_n$
	until a stopping criteria has been reached

Example 2.1. Find a solution of $2^{-x} = x$.

This is equivalent to finding a root of $f(x) = (\frac{1}{2})^x - x$. We have $f(0) = 1 - 0 > 0, f(1) = 0.5 - 1 < 0$, so we will let $a_0 = 0, b_0 = 1$. Therefore $p_0 = 1/2$.

$f(1/2) = \frac{1}{\sqrt{2}} - \frac{1}{2} > 0$, so $a_1 = 1/2, b_1 = 1, p_1 = 3/4$.

$f(3/4) = (0.5)^{3/4} - 3/4 < 0$, so $a_2 = 1/2, b_2 = 3/4, p_2 = 5/8$

After only two iterations, we get a coarse approximate of the root: $p_2 = 5/8 = 0.625$. For the reference, the actual root is 0.6411857445.

If we predetermine a tolerance level tol so that the algorithm will terminate if $|p_n - p_{n+1}| < \text{Tol}$, then this can be achieved when

$$|p_n - p_{n+1}| = \frac{b_n - a_n}{4} = \frac{b_0 - a_0}{2^{n+2}} < \text{Tol} \iff n \geq \log_2(\frac{b_0 - a_0}{4\text{Tol}}) \tag{2.1}$$

If p is the actual root, then we have

$$|p_n - p| \leq \frac{b_n - a_n}{2} = 2|p_n - p_{n+1}| < 2\text{Tol}.$$

Therefore the number of iterations needed can be computed via (2.1).

2.2 The Newton's Method

Newton's method (also called the Newton-Raphson method) is a little more sophisticated: it involves derivatives, but it is still Calculus material. This is usually how a calculator or a computer finds a root.

If we are to find the root of $y = f(x)$, we start with a random initial guess x_0. Consider the the tangent line to the curve $y = f(x)$, at $P_0 = (x_0, f(x_0))$. The idea behind Newton's method is linearization. Since the tangent line (linearization of f) is close to f, then its x-intercept, x_2, is close to the x-intercept of the curve $y = f(x)$. We can easily find x_2.

The tangent line at the point $(x_0, f(x_0))$ is

$$y - f(x_0) = f'(x_0)(x - x_0).$$

Plugging in $y = 0$ to solve for the x-intercept:

$$0 - f(x_0) = f'(x_0)(x - x_0) \Longrightarrow x = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

So $x_1 = x_0 - \dfrac{f(x_0)}{f'(x_0)}$. We use x_1 as the next approximation and keep repeating this process, as shown in Figure 2.2, where we gain this iteration formula

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k \geq 0. \tag{2.2}$$

Note that we require $f'(x_k) \neq 0$ which is not a trivial condition.

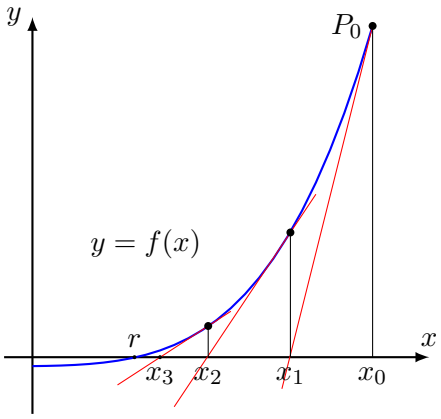


Figure 2.2: Newton's method

Example 2.2. Use the Newton's method to approximate $\sqrt{2}$.

Let $f(x) = x^2 - 2$, which makes $\sqrt{2}$ a root of $f(x)$. We can compute that $f'(x) = 2x$. Let $x_0 = 4$. We can simplify (2.2) to

$$x_{k+1} = x_k - \frac{x_k^2 - 2}{2x_k} = \frac{x_k}{2} + \frac{1}{x_k}.$$

$x_1 = 2 + 1/4 = 9/4 = 2.25$
$x_2 = 9/8 + 4/9 = 113/72 = 1.569\dot{4}$
$x_3 = 1.569\dot{4}/2 + 1/1.569\dot{4} \approx 1.42189$

If we use the Bisection method for this problem starting at $a_0 = 0, b_0 = 4$, we then have $p_0 = 2, p_1 = 1, p_2 = 1.5, p_3 = 1.25$. This is a much worse approximation after the same number of iterations. See Table 2.1 for error comparison. This table also includes the secant method that will be introduced later.

The two drawbacks of the Newton's method are: (1) It requires the knowledge of the derivative function; (2) It may not converge if you have an unlucky initial guess x_0. For example, if one picks $x_0 = 0$ for finding root of $y = x^5 - x - 1$, the sequence will not converge to the root (try to draw tangent lines on your own). However, if the initial guess is close enough, the Newton's method does enjoy fast convergence (faster than the Bisection method as shown in Example 2.2) .

n	Bisection $\lvert p_n - \sqrt{2}\rvert$	Newton $\lvert x_n - \sqrt{2}\rvert$	Secant
0	0.585786438	2.585786438	
1	0.414213562	0.835786438	
2	0.085786438	0.155230882	0.414213562
3	0.164213562	0.007676801	0.080880229
4	0.039213562	0.000020723	0.014357866
5	0.023286438	0.000000000	0.000420459

Table 2.1: Accuracy Comparison of Example 2.2

Definition 2.3. A method that produces a sequence $\{x_n\}$ that converges to a number x *linearly* if there exists $L \in (0,1)$ such that for large values of n

$$|x_{n+1} - x| \leq L|x_n - x|.$$

The sequence converges to x *quadratically* if there exists $Q > 0$ such that for large values of n

$$|x_{n+1} - x| \leq Q|x_n - x|^2.$$

For example, the sequence $\{3^{-n}\}_{n=1}^{\infty}$ converges to 0 linearly. The sequence $\{b_n = 3^{-2^n}\}_{n=1}^{\infty}$ converges quadratically to 0.

Theorem 2.4. *Let $f(x)$ be a function whose second derivative is continuous, and r be a root of $f(x)$. Let I be an interval contains r such that $|f''(x)| \leq M$ for any $x \in I$. If $x_k \in I$, then the Newton's method converges to r quadratically as*

$$|x_{k+1} - r| \leq \frac{M}{2|f'(x_k)|}|x_k - r|^2. \tag{2.3}$$

Proof. Expand f in the first Taylor polynomial ($n = 1$ in (6.1)) at x_k gives

$$0 = f(r) = f(x_k) + f'(x_k)(r - x_k) + \frac{f''(\xi)}{2}(r - x_k)^2,$$

where ξ lies between x_k and r. Consequently, if $f'(x_k) \neq 0$, we have

$$r - x_k + \frac{f(x_k)}{f'(x_k)} = -\frac{f''(\xi)}{2f'(x_k)}(r - x_k)^2.$$

Combined with (2.2), we have

$$r - x_{k+1} = -\frac{f''(\xi)}{2f'(x_k)}(r - x_k)^2 \Longrightarrow |r - x_{k+1}| \leq \frac{M}{2|f'(x_k)|}|r - x_k|^2$$

since ξ is in between x_k and r, hence $\xi \in I$. □

This implies that the Newton's method has the tendency to approximately double the number of digits of accuracy with each successive approximation, as demonstrated in Table 2.1.

The order of convergence for linear methods (like Bisection) is 1 and the order of convergence for quadratic methods (like Newton) is 2.

2.3 The Secant Method

As mentioned, one of the drawbacks of the Newton's method is that we have to know its derivative. For the simple example presented above, this is not difficult. In many problems, however, f can get complicated, or worse, cannot be written down analytically. In such scenarios, differentiating f analytically is difficult or impossible. For such problems, we approximate the derivatives. Iterations of the form

$$x_{k+1} = x_k - \frac{f(x_k)}{g(x_k)}, \quad \text{where } g_k \approx f'(x_k)$$

are called *quasi-Newton* methods.

Among many quasi-Newton methods, the *secant method* is defined by taking $f'(x_k)$ to be

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}.$$

So the iteration formula for the secant method is:

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}, \quad k \geq 1. \tag{2.4}$$

To begin the secant method, one needs two starting points x_0, x_1.

It turns out that the order of convergence of the secant method is in between Bisection and Newton. In fact, the order is $\dfrac{1+\sqrt{5}}{2} \approx 1.62$ the Golden ratio as for the secant method the iterates satisfy

$$|x_{n+1} - x| \leq M|x_n - x|^{\frac{1+\sqrt{5}}{2}}.$$

Interested readers can find a proof in [2, Section 4.3.3].

We will use the secant method on the same example.

Example 2.5. Use the secant method to find the positive root of $f(x) = x^2 - 2$ with initial $x_0 = 0, x_1 = 2$.

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})} = x_k - \frac{(x_k^2 - 2)(x_k - x_{k-1})}{x_k^2 - x_{k-1}^2} = x_k - \frac{x_k^2 - 2}{x_k + x_{k-1}} = \frac{x_k x_{k-1} + 2}{x_k + x_{k-1}}.$$

$$x_2 = \frac{0+2}{2} = 1.$$

$$x_3 = \frac{2+2}{1+2} = 4/3.$$

$$x_4 = \frac{4/3+2}{7/3} = 10/7.$$

$$x_5 = \frac{\frac{10}{7}\frac{4}{3}+2}{\frac{10}{7}+\frac{4}{3}} = \frac{40+42}{30+28} = \frac{41}{29}.$$

The errors are listed in the third column of Table 2.1.

Chapter 2 Exercises

Python Exercises

1. Bisection and Newton comparison.

(a) Use the Bisection method to do Example 2.1, with the stopping criteria $|p_{n+1} - p_n| \leq$1e-5, and initial interval $a_0 = 0, b_0 = 1$. How many iterations was run? (meaning what is the n value when $|p_{n+1} - p_n| \leq$1e-5 is reached?) How is this compared to the theoretical value in (2.1)?

(b) Use the Newton's method to do Example 2.1, with the stopping criteria $|x_{n+1} - x_n| \leq$1e-5, and initial value $x_0 = 1$. How many iterations was run?

2. Newton's method fails for finding a root of $y = x^5 - x - 1$ with $x_0 = 0$. Print the first 6 approximations to illustrate this.

Other Exercises

3. Use the Bisection method to find a solution of $\cos x = x$ with initial interval $[0, \pi]$. Find $p_i, i = 0, 1, 2$. Use (2.1) to compute how many iterations are needed if tol=0.01.

4. Let $f(x) = 3(x+1)(x-1/2)(x-1)$. Use the Bisection method to find p_3 on the interval [-2, 1.5]. [Since f is factored, we know its roots right away. This problem is more of an exercise such that the function's sign can be determined easily.]

5. Write down the first 3 iterations (x_1, x_2, x_3) of the Newton's method for solving $x^2 - 3 = 0$, starting with $x_0 = 1$. Write your answers in terms of fractions.

6. Prove that Newton's method will converge to 0 given any initial value x_0 if we are solving $x^2 = 0$.

7. Write down the first three iterates (2, 3, 4) of the secant method for solving $x^2 - 3 = 0$, starting with $x_0 = 0$ and $x_1 = 1$. Write your answer in terms of fractions.

8. Show that $\{3^{-n}\}_{n=1}^{\infty}$ converges to 0 linearly, and the sequence $\{b_n = 3^{-2^n}\}_{n=1}^{\infty}$ converges to 0 quadratically.

9. We can compute 1/3 by solving $f(x) = 0$ with $f(x) = x^{-1} - 3$.

   (a) Write down the Newton iteration for this problem, and compute by hand the first 2 Newton iterates for approximating 1/3, starting with $x_0 = 0.5$.

   (b) What happens if you start with $x_0 = 1$?

   (c) *In the case of (b), show that the iterates $x_k \to -\infty$ as $k \to \infty$.

10. Problems involving the amount of money required to pay off a mortgage over a fixed period of time involve the formula

$$A = \frac{P}{i}[1 - (1+i)^{-n}],$$

known as an *ordinary annuity equation.* In this equation A is the amount of the mortgage, P is the amount of each payment, and i is the interest rate per period for the n payment periods. Suppose that a 30-year home mortgage in the amount of $135,000 is needed and that the borrower can afford house payments of at most $1000 per month. What is the maximal interest rate the borrower can afford to pay? (You can use `scipy.optimize.newton` to solve the equation in the end.)

Chapter 3

Solving system of linear equations: Direct methods

Solving a linear system $Ax = b$ is everywhere in many applications. There are generally two approaches: Direct methods and iterative (indirect) methods. Direct methods give a direct answer at the end of the algorithm, whereas iterative methods follow an iterative formula where the iterates are approaching the real solution.

We also care about how efficient an algorithm is. This is to count how many flops are needed in a computer. Each flop is one of the following operations: $+, -, \times, \div$. For example, the inner product $x^T y$ requires n multiplications and $n-1$ summations, given both vectors have n coordinates. So $x^T y$ needs $2n-1$ flops.

- Matrix-Vector multiplication: Given $A \in \mathbb{R}^{m\times n}, x \in \mathbb{R}^n$, Ax is computing m inner products, so needs $m(2n-1)$ flops.
- Matrix-Matrix multiplication: Given $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times p}$, AB is computing p matrix-vector multiplications. So the calculation of AB requires $pm(2n-1)$ flops. In particular, if both A and B are square matrices, then AB needs $2n^3 - n^2 \approx 2n^3$ flops.

3.1 Gaussian Elimination and LU factorization

Gaussian elimination is a core material in linear algebra and is the procedure one performs when finding key features like linear dependence, rank, null space, basis, dimension, etc. Gauss elimination is the simplest way to solve linear systems of equations by hand, and also the standard method for solving them on computers.

The following notations are standard:

$$Ax = b \Longleftrightarrow \begin{array}{cccccc} a_{11}x_1 & +a_{12}x_2 & +\cdots & +a_{1n}x_n & = b_1 \\ a_{21}x_1 & +a_{22}x_2 & +\cdots & +a_{2n}x_n & = b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{m1}x_1 & +a_{m2}x_2 & +\cdots & +a_{mn}x_n & = b_m \end{array} \Longleftrightarrow [A|b].$$

The matrix A is called the coefficient matrix and the matrix $[A|b]$ is called the augmented matrix.

Gaussian elimination is suitable for systems of all sizes, but **in this section our examples and analysis will focus on $Ax = b$ where A is invertible.** When solving $Ax = b$, the Gaussian elimination consists of two procedures: forward elimination and backward substitution.

Example 3.1. Solve $\left[\begin{array}{ccc|c} 1 & 2 & 2 & 1 \\ 2 & 7 & 7 & 5 \\ 2 & 7 & 9 & 5 \end{array}\right]$.

Forward Elimination:
$\left[\begin{array}{ccc|c} 1 & 2 & 2 & 1 \\ 2 & 7 & 7 & 5 \\ 2 & 7 & 9 & 5 \end{array}\right] \xrightarrow{\rho 1(-2)+\rho 2} \left[\begin{array}{ccc|c} 1 & 2 & 2 & 1 \\ 0 & 3 & 3 & 3 \\ 2 & 7 & 9 & 5 \end{array}\right] \xrightarrow{\rho 1(-2)+\rho 3} \left[\begin{array}{ccc|c} 1 & 2 & 2 & 1 \\ 0 & 3 & 3 & 3 \\ 0 & 3 & 5 & 3 \end{array}\right] \xrightarrow{\rho 2(-1)+\rho 3} \left[\begin{array}{ccc|c} 1 & 2 & 2 & 1 \\ 0 & 3 & 3 & 3 \\ 0 & 0 & 2 & 0 \end{array}\right]$.

Backward substitution:
(1) Equation form:

$$2x_3 = 0 \Rightarrow x_3 = 0$$
$$3x_2 + 3 * 0 = 3 \Rightarrow x_2 = 1$$
$$x_1 + 2 * 1 + 2 * 0 = 1 \Rightarrow x_1 = -1$$

(2) Matrix form:
$\left[\begin{array}{ccc|c} 1 & 2 & 2 & 1 \\ 0 & 3 & 3 & 3 \\ 0 & 0 & 2 & 0 \end{array}\right] \xrightarrow{\rho 3/2} \left[\begin{array}{ccc|c} 1 & 2 & 2 & 1 \\ 0 & 3 & 3 & 3 \\ 0 & 0 & 1 & 0 \end{array}\right] \xrightarrow[\rho 3(-2)+\rho 1]{\rho 3(-3)+\rho 2} \left[\begin{array}{ccc|c} 1 & 2 & 0 & 1 \\ 0 & 3 & 0 & 3 \\ 0 & 0 & 1 & 0 \end{array}\right] \xrightarrow{\rho 2/(3)} \left[\begin{array}{ccc|c} 1 & 2 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{array}\right]$
$\xrightarrow{\rho 2(-2)+\rho 1} \left[\begin{array}{ccc|c} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{array}\right]$.

As seen in Example 3.1, the Gaussian elimination consists of two procedures: Forward elimination and backward substitution. Next, we will describe them generally and also count the flops used in each procedure.

The following two formula will be useful:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}. \tag{3.1}$$

Forward elimination of Gaussian elimination: This is to get to an upper triangular matrix with row operations.

Column 1

We eliminate all entries below a_{11}.

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{array}\right] \longrightarrow \left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ 0 & a'_{22} & \cdots & a'_{2n} & b'_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & a'_{n2} & \cdots & a'_{nn} & b'_n \end{array}\right] \tag{3.2}$$

At each row, we need $n+1$ multiplications and n additions. [for example, to obtain the new row 2, we first compute $-\frac{a_{21}}{a_{11}}$, which takes one multiplication/division. Then we add $-\frac{a_{21}}{a_{11}}$ copy of row 1 to row 2.] There are $n-1$ rows to perform, so a total of $(n-1)\cdot(2n+1) = 2n^2 - n - 1$ flops are needed to get to the right hand side of (3.2).

Column 2: Now we use a'_{22} to eliminate all entries below. This is the same work except we have an $(n-1)\times(n-1)$ system now, so we need $2(n-1)^2 - (n-1) - 1$ flops.

Total: We keep doing this until the coefficient matrix becomes upper triangular, so there are a total of $\sum_{i=1}^{n}(2i^2 - i - 1) = \frac{n(n+1)(2n+1)}{3} - \frac{n(n+1)}{2} - n = \frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n \approx \frac{2}{3}n^3$ flops.

Backward substitution of Gaussian Elimination: After forward elimination, we obtain an upper triangular system

$$\left[\begin{array}{cccc|c} c_{11} & c_{12} & \cdots & c_{1n} & y_1 \\ 0 & c_{22} & \cdots & c_{2n} & y_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & c_{nn} & y_n \end{array}\right]. \tag{3.3}$$

The backward substitution can be done in either the equation form or the matrix form, both of which are illustrated in Example 3.1. We will use the equation form in our operation counts here. We will start with last row and work our way up.

Row n: $c_{nn}x_n = y_n$. Takes 1 division to solve x_n

Row $n-1$: $c_{n-1,n-1}x_{n-1} + c_{n-1,n}x_n = y_{n-1}$. Takes 1 multiplication, 1 addition, 1 division to solve x_{n-1}.

Row $n-2$: $c_{n-2,n-2}x_{n-2} + c_{n-2,n-1}x_{n-1} + c_{n-2,n}x_n = y_{n-2}$. Takes 2 multiplication, 2 addition, 1 division to solve x_{n-2}.

...

Row 1: $c_{11}x_1 + c_{12}x_2 + \cdots + c_{1n}x_n = y_1$. Takes $n-1$ multiplications, $n-1$ additions, 1 division to solve x_1.

Add up all counts, we have

$\sum_{i=1}^{n}[(i-1) + (i-1) + 1] = n^2$.

Gaussian elimination operation counts:

$$\underbrace{\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n}_{forward\ elim.} + \underbrace{n^2}_{backward\ sub.} = \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n \approx \frac{2}{3}n^3.$$

3.1.1 Forward elimination = LU factorization

Each row operation corresponds to a matrix (often called an elementary row operation matrix). In the forward elimination process of Example 3.1, let $A = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 7 & 7 \\ 2 & 7 & 9 \end{bmatrix}$, then $L_1 A = \begin{bmatrix} 1 & 2 & 2 \\ 0 & 3 & 3 \\ 2 & 7 & 9 \end{bmatrix}$, where L_1 represents the first row operation. To figure out L_1, we perform the very row operation on the identity matrix: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \xrightarrow{\rho 1(-2)+\rho_2} \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = L_1$. For the rest two row operations, we can compute in the same fashion that $L_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix}, L_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix}$.

The forward elimination process of Example 3.1 can be written as

$$L_3L_2L_1A = U = \begin{bmatrix} 1 & 2 & 2 \\ 0 & 3 & 3 \\ 0 & 0 & 2 \end{bmatrix}.$$

So

$$A = (L_3L_2L_1)^{-1}U = L_1^{-1}L_2^{-1}L_3^{-1}U := LU.$$

L_1, L_2, L_3 are very special matrices, and their inverses and L can be figured out easily:

$$L_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, L_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix}, L_3^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}, L = L_1^{-1}L_2^{-1}L_3^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix}.$$

So we can decompose/factor A as

$$A = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 7 & 7 \\ 2 & 7 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 2 \\ 0 & 3 & 3 \\ 0 & 0 & 2 \end{bmatrix} = LU.$$

In general, if there is no row exchange involved in forward elimination, then A can always be written as the product of a lower triangular matrix and an upper triangular matrix. This is called the LU factorization of A. It requires $\frac{2}{3}n^3$ flops as well because it is done through forward elimination. In fact, the operation counts is slightly less because we don't need to consider the right hand side b, but it is still on the order of $\frac{2}{3}n^3$.

3.1.2 A 'second way' to solve $Ax = b$

If the LU factorization of A is already done, then it is very easy to solve $LUx = b$. We set $Ux = y$. We first solve y from $Ly = b$, which is very straightforward as we will see in Example 3.2. Then we solve x from $Ux = y$. We will use the same example to illustrate.

Example 3.2. Suppose we are given $\begin{bmatrix} 1 & 2 & 2 \\ 2 & 7 & 7 \\ 2 & 7 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 2 \\ 0 & 3 & 3 \\ 0 & 0 & 2 \end{bmatrix}$. In order to solve

$$\begin{bmatrix} 1 & 2 & 2 \\ 2 & 7 & 7 \\ 2 & 7 & 9 \end{bmatrix} x = \begin{bmatrix} 1 \\ 5 \\ 5 \end{bmatrix},$$

(1) We first solve $\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix} y = \begin{bmatrix} 1 \\ 5 \\ 5 \end{bmatrix}$ $(Ly = b)$.

$$\begin{aligned} y_1 &= 1 \\ 2 * 1 + y_2 = 5 &\Rightarrow y_2 = 3 \\ 2 * 1 + 1 * 3 + y_3 = 5 &\Rightarrow y_3 = 0 \end{aligned}$$

(2) Then we solve $Ux = y$, which is already done in the backward substitution of Example 3.1. It is not a coincidence that the y we solved in (1) is the same as the right hand side of the system after forward elimination.

It is obvious that solving $Ly = b$ and $Ux = y$ cost the same number of operations because the only difference is that $Ly = b$ is solved forwardly, and $Ux = y$ is solved backwardly.

Solve $Ax = b$ via LU decomposition	
Step 1: $A = LU$. (Now equation becomes $LUx = b$.)	$\frac{2}{3}n^3$ flops
Step 2: Solve $Ly = b$. (y will equal to Ux.)	n^2 flops
Step 3: Solve $Ux = y$.	n^2 flops

LU decomposition approach is better when we need to solve multiple systems with the same coefficient matrix A.

We compare the flop counts of these two approaches for solving m systems:

$$Ax = b_1, Ax = b_2, \cdots, Ax = b_m.$$

Regular Gauss Elimination:	$\frac{2}{3}n^3 m$ flops
LU approach:	$\frac{2}{3}n^3 + 2mn^2$ flops

If $m = n$, then regular GE approach takes around $\frac{2}{3}n^4$ operations, whereas LU approach operation count is $\frac{8}{3}n^3$, still on the order of n^3. This makes a huge difference when the system is large (This summary is also included in Table 3.2).

We list in Table 3.1 the flop count of various matrix algebra.

Table 3.1: Flop count for various matrix operations: $x, y \in \mathbb{R}^n, A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times p}$.

Operation	flop count	complexity	complexity if $m = n$
$x^T y$ (x dot product y)	$= 2n - 1$	$O(n)$	$O(n)$
Ax	$= m(2n - 1)$	$O(mn)$	$O(n^2)$
AB	$= mp(2n - 1)$	$O(mpn)$	$O(n^3)$

We summarize in Table 3.2 the flop count related to solving a linear system.

3.2 Pivoting (row exchange)

LU decomposition is not the full story because it does not consider row exchanges, which is necessary in many scenarios.

Consider computing the LU factorization of $A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 2 & 1 \\ 2 & 7 & 9 \end{bmatrix}$. We run into problem right away because $a_{11} = 0$ is not able to eliminate entries below. In this case, a permutation of rows are necessary, and we can, for instance, switch row 1 and row 2. Switching row 1 and row 2 is the same as being left multiplied by the permutation matrix $P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, i.e. $PA = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 1 \\ 2 & 7 & 9 \end{bmatrix}$. Then we can do the same old LU factorization to PA.

Table 3.2: Flop count related to solving a linear system: $A \in \mathbb{R}^{n\times n}, B = [b_1, \cdots, b_m] \in \mathbb{R}^{n\times m}$.

Operation	flop count	complexity
Forward Elim. on A	$\approx \frac{2}{3}n^3$	$O(n^3)$
Backward Sub.	$= n^2$	$O(n^2)$
LU decomp of A	$\approx \frac{2}{3}n^3$	$O(n^3)$
Solve $Ax = b$ via Gauss Elim./LU	$\approx \frac{2}{3}n^3$	$O(n^3)$
Solve $AX = B$ by LU	$\approx \frac{2}{3}n^3 + 2mn^2$	$O(n^3 + mn^2)$
Finding A^{-1} by reducing $[A\|I]$ to $[I\|A^{-1}]$	$\approx 2n^3$	$O(n^3)$

In the following example, it is not necessary to switch rows because the entry in the first row/first column is nonzero. But we will find out that switching rows will largely increase the accuracy of the solution.

Example 3.3. Let us look at another system $\left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 1 & 1 & 2 \end{array}\right]$. The solution should be very close to $x_1 = x_2 = 1$, and we are able to get this solution if the computer had infinite precision. But with the forward elimination procedure and with double-precision arithmetic, we get

$$\left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 1 & 1 & 2 \end{array}\right] \xrightarrow{\rho_1(-10^{20})+\rho 2} \left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 0 & -10^{20}+1 & -10^{20}+2 \end{array}\right] \xrightarrow{round} \left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 0 & -10^{20} & -10^{20} \end{array}\right]$$

The relative difference between 1 and -10^{20} is $1/10^{20}$, which is smaller than the machine epsilon. In other words, 1 is negligible compared to 10^{20} (or -10^{20}). You are encouraged to type `-1e20 + 1` in Python and see what happens.

From there, we solve $x_2 = 1$, $10^{-20}x_1 + 1 = 1 \Longrightarrow x_1 = 0$, which is far from the actual solution.

A remedy of this is to exchange these two rows:

$$\left[\begin{array}{cc|c} 1 & 1 & 2 \\ 10^{-20} & 1 & 1 \end{array}\right] \to \left[\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & -10^{-20}+1 & -2\times 10^{-20}+1 \end{array}\right] \xrightarrow{round} \left[\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & 1 & 1 \end{array}\right]$$

And we can get a much more accurate solution $x_1 = x_2 = 1$.

This row exchange strategy is the content of the next subsection.

3.2.1 Partial pivoting

Partial pivoting is a straightforward method of row exchanging in the forward process of Gaussian elimination. At stage k (column k), we will look for the biggest entry in absolute value of that column, and use that entry as a pivot. In Example 3.3, at stage 1, we compare all entries in column 1, and 1 is bigger than 10^{-20}, so we exchange row 1 and row 2 to use 1 as the pivot. The next example is 3×3 so this algorithm can be explained more clearly

Example 3.4. Use partial pivoting to solve the system $\left[\begin{array}{ccc|c} 2 & 1 & 1 & 1 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & -1 \end{array}\right]$.

First we focus on column 1, 8 is the biggest number in absolute value, so we switch row 1 and row 3, and then do two row operations.

$$\left[\begin{array}{ccc|c} 2 & 1 & 1 & 1 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & -1 \end{array}\right] \xrightarrow{\rho_1 \leftrightarrow \rho_3} \left[\begin{array}{ccc|c} 8 & 7 & 9 & -1 \\ 4 & 3 & 3 & 1 \\ 2 & 1 & 1 & 1 \end{array}\right] \xrightarrow[\rho_1(-1/4)+\rho_3]{\rho_1(-1/2)+\rho_2} \left[\begin{array}{ccc|c} 8 & 7 & 9 & -1 \\ 0 & -1/2 & -3/2 & 3/2 \\ 0 & -3/4 & -5/4 & 5/4 \end{array}\right]$$

Now we compare the entries of column 2. Keep in mind that we only focus on the smaller system so the first row is out of the picture. We compare the magnitude of -1/2 and -3/4. -3/4 is bigger so we have to switch rows again.

$$\left[\begin{array}{ccc|c} 8 & 7 & 9 & -1 \\ 0 & -1/2 & -3/2 & 3/2 \\ 0 & -3/4 & -5/4 & 5/4 \end{array}\right] \xrightarrow{\rho_2 \leftrightarrow \rho_3} \left[\begin{array}{ccc|c} 8 & 7 & 9 & -1 \\ 0 & -3/4 & -5/4 & 5/4 \\ 0 & -1/2 & -3/2 & 3/2 \end{array}\right] \xrightarrow{\rho_2(2/3)+\rho_3} \left[\begin{array}{ccc|c} 8 & 7 & 9 & -1 \\ 0 & -3/4 & -5/4 & 5/4 \\ 0 & 0 & -2/3 & 2/3 \end{array}\right]$$

The rest backward substitution is the same as before:

$x_3 = -1$

$-\frac{3}{4}x_2 + \frac{5}{4} = \frac{5}{4} \Longrightarrow x_2 = 0$

$8x_1 + 0 - 9 = -1 \Longrightarrow x_1 = 1$

Note that when solving this problem by hand, partial pivoting does make the arithmetic harder due to all the fractions, but it is the same to a computer.

3.2.2 PLU factorization

With row exchanges, we have PLU factorization instead of LU factorization. For every matrix A, we are able to write

$$PA = LU,$$

where P is a permutation matrix, L is lower triangular and U is upper triangular.

First of all, we need to talk more about permutation matrices. A matrix P is a permutation matrix if it is a row permutation of the identity matrix. $\begin{bmatrix} & & 1 \\ & 1 & \\ 1 & & \end{bmatrix}$ is a 3×3 permutation matrix because it is switching row 1 and row 3 of the identity matrix. $\begin{bmatrix} & 1 & & \\ & & & 1 \\ 1 & & & \\ & & 1 & \end{bmatrix}$ is a 4×4 permutation matrix and it represents "place row 1 to row 3; place row 2 to row 1; place row 3 to row 4; place row 4 to row 2", since this is the result of the identity matrix after doing these row operations.

Another useful fact is that for any permutation matrix P, we have $P^{-1} = P^T$.

Before figuring out the PLU factorization, it is also very useful to discuss the difference between left and right multiplications. In short, **left multiplication by an elementary matrix is to operate on the rows; right multiplication by an elementary matrix is to operate on the columns.** We have been left multiplying a matrix since we have been doing row operations. Take a permutation matrix for example,

$$\begin{bmatrix} & & 1 \\ & 1 & \\ 1 & & \end{bmatrix} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} g & h & i \\ d & e & f \\ a & b & c \end{bmatrix}, \quad \text{and} \quad \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} & & 1 \\ & 1 & \\ 1 & & \end{bmatrix} = \begin{bmatrix} c & b & a \\ f & e & d \\ i & h & g \end{bmatrix}.$$

Same thing happens with scaling matrices (diagonal matrix):

$$\begin{bmatrix} 2 & & \\ & 3 & \\ & & 4 \end{bmatrix} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} 2a & 2b & 2c \\ 3d & 3e & 3f \\ 4g & 4h & 4i \end{bmatrix}, \quad \text{and} \quad \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} 2 & & \\ & 3 & \\ & & 4 \end{bmatrix} = \begin{bmatrix} 2a & 3b & 4c \\ 2d & 3e & 4f \\ 2g & 3h & 4i \end{bmatrix}.$$

Example 3.5. Find the PLU decomposition of $A = \begin{bmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{bmatrix}$.

This is the same matrix as the coefficient matrix in Example 3.4, so we will use the work there, but specifying all the row operation matrices.

First we exchanged row 1 and row 3 since 8 is the biggest entry in first column:

$$P_1 A = \underbrace{\begin{bmatrix} & & 1 \\ & 1 & \\ 1 & & \end{bmatrix}}_{P_1} \begin{bmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 \\ 4 & 3 & 3 \\ 2 & 1 & 1 \end{bmatrix}.$$

Second we use the pivot 8 to eliminate the entries below:

$$L_1 P_1 A = \underbrace{\begin{bmatrix} 1 & & \\ -\frac{1}{2} & 1 & \\ -\frac{1}{4} & & 1 \end{bmatrix}}_{L_1} \begin{bmatrix} 8 & 7 & 9 \\ 4 & 3 & 3 \\ 2 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 \\ & -\frac{1}{2} & -\frac{3}{2} \\ & -\frac{3}{4} & -\frac{5}{4} \end{bmatrix}$$ here L_1 combines two row operations.

Now we are at stage 2 (2nd column), -3/4 is bigger than -1/2 (remember we compare numbers in absolute value), so we exchange row 2 and row 3:

$$P_2 L_1 P_1 A = \underbrace{\begin{bmatrix} 1 & & \\ & & 1 \\ & 1 & \end{bmatrix}}_{P_2} \begin{bmatrix} 8 & 7 & 9 \\ & -\frac{1}{2} & -\frac{3}{2} \\ & -\frac{3}{4} & -\frac{5}{4} \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 \\ & -\frac{3}{4} & -\frac{5}{4} \\ & -\frac{1}{2} & -\frac{3}{2} \end{bmatrix}.$$

Finally we perform the last row operation to eliminate the entry below -3/4.

$$L_2 P_2 L_1 P_1 A = \underbrace{\begin{bmatrix} 1 & & \\ & 1 & \\ & -\frac{2}{3} & 1 \end{bmatrix}}_{L_2} \begin{bmatrix} 8 & 7 & 9 \\ & -\frac{3}{4} & -\frac{5}{4} \\ & -\frac{1}{2} & -\frac{3}{2} \end{bmatrix} = \begin{bmatrix} 8 & 7 & 9 \\ & -\frac{3}{4} & -\frac{5}{4} \\ & & -\frac{2}{3} \end{bmatrix} = U.$$

Now we have $L_2 P_2 L_1 P_1 A = U$. We will rewrite the left hand side as $L_2 L_1' P_2 P_1 A$, which means that $P_2 L_1 = L_1' P_2$, so

$$L_1' = P_2 L_1 P_2^{-1} = P_2 L_1 P_2^T = \begin{bmatrix} 1 & & \\ & & 1 \\ & 1 & \end{bmatrix} \begin{bmatrix} 1 & & \\ -\frac{1}{2} & 1 & \\ -\frac{1}{4} & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & & 1 \\ & 1 & \end{bmatrix} = \begin{bmatrix} 1 & & \\ -\frac{1}{4} & 1 & \\ -\frac{1}{2} & & 1 \end{bmatrix}.$$

It is crucial that L_1' is still lower triangular.

With $\underbrace{L_2 L_1'}_{L^{-1}} \underbrace{P_2 P_1}_{P} A = U$, we have

$$P \begin{bmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{bmatrix} = LU, \text{ where } P = \begin{bmatrix} & & 1 \\ 1 & & \\ & 1 & \end{bmatrix}, L = \begin{bmatrix} 1 & & \\ \frac{1}{4} & 1 & \\ \frac{1}{2} & \frac{2}{3} & 1 \end{bmatrix}, U = \begin{bmatrix} 8 & 7 & 9 \\ & -\frac{3}{4} & -\frac{5}{4} \\ & & -\frac{2}{3} \end{bmatrix}. \quad (3.4)$$

Example 3.6. Given the PLU decomposition in (3.4), solve $\begin{bmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{bmatrix} x = \begin{bmatrix} -3 \\ -3 \\ -1 \end{bmatrix}$.

Multiply both sides of the system by P, then we get $LUx = \begin{bmatrix} -1 \\ -3 \\ -3 \end{bmatrix}$. The rest is the same as Example 3.2. We solve $Ly = \begin{bmatrix} -1 \\ -3 \\ -3 \end{bmatrix}$ and get $y = [-1, -11/4, -2/3]^T$. We then solve $Ux = y$ to get $x = [-3, 2, 1]^T$.

You can find more details about PLU factorization in [4, Lecture 21].

3.2.3 Methods and software

The direct methods are the methods of choice for most linear systems. To be more specific, Gaussian elimination with pivoting are recommended.

LAPACK (Linear Algebra Package) is a standard software library for numerical linear algebra. It provides routines for solving systems of linear equations and linear least squares, eigenvalue problems, and singular value decomposition. MATLAB started its life in the late 1970s as an interactive calculator built on top of LAPACK[1]. Python also uses the LAPACK routine. In Python, the command `P, L, U = scipy.linalg.lu(A)` provides the decomposition $A = PLU$. Many other functions of Python, like computing the determinant or finding the inverse, are based on the PLU factorization.

It can be very difficult to solve a large dense linear system. By difficult, we mean it takes too long using Gaussian elimination. But if a large system is sparse (most entries are 0), it can be solved efficiently using iterative methods. Systems of this type arise naturally, for example, when finite-difference techniques are used to solve boundary-value problems, a common application in the numerical solution of partial differential equations.

Chapter 3 Exercises

1. Write out the full matrix $A = [a_{ij}]_{3\times 4}$ where $a_{ij} = i^{j-1}$.
2. Let $A = \begin{bmatrix} 3 & 0 \\ -1 & 2 \\ 1 & 1 \end{bmatrix}, B = \begin{bmatrix} 1 & 5 & 2 \\ -1 & 0 & 1 \\ 3 & 2 & 4 \end{bmatrix}, C = \begin{bmatrix} 1 & 4 & 2 \\ 3 & 1 & 5 \end{bmatrix}$. Compute the following (write undefined if operation is not legal).
   (a) $2A^T + C$ (b) CBB (c) BA
3. Compute $x^T Ax$, where $x = (x_1, x_2, x_3)$, and $A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}$ (A is symmetric.)
4. Perform forward elimination.
   (a) $\left[\begin{array}{ccc|c} 2 & 2 & 2 & 0 \\ -2 & 5 & 2 & 1 \\ 8 & 1 & 4 & -1 \end{array}\right]$ (b) $\left[\begin{array}{ccc|c} 0 & -2 & 3 & 1 \\ 3 & 6 & -3 & -2 \\ 6 & 6 & 3 & 5 \end{array}\right]$

[1] It was called LINPACK and EISPACK then, the predecessor of LAPACK.

5. Find scalars c_i for which the equation $c_1(-1,0,2) + c_2(2,2,-2) + c_3(1,-2,1) = (-6,12,4)$ holds.

6. How many **exact** flops are needed for the following computations?

   (a) x dot product y, where x, y are vectors in $\mathbb{R}^9$.

   (b) $A + B$, where A, B are both $m \times n$.

   (c) ABC, where A, B, C are 5×5 matrices.

   (d) Ux, where U is $n \times n$ upper triangular matrix and x is a vector in $\mathbb{R}^n$.

   (e) ABC, where A is 10×100, B is 100×100, C is 100×100.

   *(f) LU, where L is $n \times n$ lower triangular, U is $n \times n$ upper triangular.

7. Find the 4×4 elementary row matrices that correspond to

   (a) $\rho 2(3) + \rho 3$ (b) $\rho 1(-3) + \rho 4$

8. Find the inverse of the matrix in 7(a). What row operation does this inverse correspond to?

9. Find the inverse of $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$.

10. Is $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 2 & 3 & 2 \end{bmatrix}$ invertible?

11. Find the LU decomposition of the coefficient matrix in 4(a).

12. Does the system in 4(b) solvable? Is the matrix $A = \begin{bmatrix} 0 & -2 & 3 \\ 3 & 6 & -3 \\ 6 & 6 & 3 \end{bmatrix}$ invertible?

13. Given $\begin{bmatrix} 3 & -6 & -3 \\ 2 & 0 & 6 \\ -4 & 7 & 4 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 0 \\ 2 & 4 & 0 \\ -4 & -1 & 2 \end{bmatrix} \begin{bmatrix} 1 & -2 & -1 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}$, solve the system

$$\begin{array}{rrrl} 3x_1 & -6x_2 & -3x_3 & = -3 \\ 2x_1 & & +6x_3 & = -22 \\ -4x_1 & +7x_2 & +4x_3 & = 3 \end{array}.$$

14. Solve $\begin{bmatrix} 2 & 6 & 2 \\ -3 & -8 & 0 \\ 4 & 9 & 2 \end{bmatrix} x = \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix}$ $(Ax = b)$ by

   (a) First find the LU factorization of A.

   (b) Second solve $Ly = b$.

   (c) Third solve $Ux = y$.

15. Consider the linear system $\left[\begin{array}{cc|c} 1 & 1 & 2 \\ 2 & 2 \times 10^{20} & 2 \times 10^{20} \end{array}\right]$. Find its solution using partial pivoting, in a machine where numbers are in standard IEEE double-precision format. Do you think the solution you find is a fairly accurate approximation to the real solution?

16. The LINPACK Benchmarks are a measure of a system's floating point computing power. Introduced by Jack Dongarra, they measure how fast a computer solves a dense $n \times n$ system of linear equations $Ax = b$. This is the standard measure for computer speed. Supercomputers (which usually have linux operating systems) can do about 10 petaflops in one second. One petaflop is 10^{15} flops. Titan, a supercomputer in the Oak Ridge National lab, is the fastest computer in the US. It can perform 17.59 petaflops every second. We assume that it takes $\frac{2}{3}n^3$ flops for Gauss Elimination.

    (a) Calculate how much time (in seconds) does Titan need to solve a 50000 by 50000 system $Ax = b$.

    (b) Suppose a laptop performs 20 gigaflops (a gigaflop is 10^9 flops) per second. Calculate how much time (in seconds) does this laptop need to solve a 50000 by 50000 system $Ax = b$.

17. Let $V = \begin{bmatrix} 0 & -2 & 4 & 2 \\ 0 & 0 & -2 & 7 \\ 2 & 1 & 3 & 1 \\ 0 & 0 & 0 & -43/2 \end{bmatrix}$. Find the permutation matrix P so that PV is upper triangular.

18. State the matrix that you need to multiply for the following transformations. Specify left or right multiplication.

    (a) $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \to \begin{bmatrix} 100 & 200 \\ -30 & -40 \end{bmatrix}$ (b) $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \to \begin{bmatrix} 100 & -20 \\ 300 & -40 \end{bmatrix}$ (c) $\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \to \begin{bmatrix} 2 & 1 & 3 \\ 2 & 1 & 3 \\ 2 & 1 & 3 \end{bmatrix}$

19. Solve the system $\left[\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 0 & 2 \end{array}\right]$ using partial pivoting.

20. Given the following row operations

$$A = \begin{bmatrix} 1 & \frac{5}{2} & \frac{5}{2} \\ 2 & 3 & 1 \\ \frac{2}{3} & \frac{5}{4} & \frac{17}{6} \end{bmatrix} \xrightarrow{\rho_1 \leftrightarrow \rho_2} \begin{bmatrix} 2 & 3 & 1 \\ 1 & \frac{5}{2} & \frac{5}{2} \\ \frac{2}{3} & \frac{5}{4} & \frac{17}{6} \end{bmatrix} \xrightarrow[\rho_1(-1/3)+\rho_3]{\rho_1(-1/2)+\rho_2} \begin{bmatrix} 2 & 3 & 1 \\ 0 & 1 & 2 \\ 0 & \frac{1}{4} & \frac{5}{2} \end{bmatrix} \xrightarrow{\rho_2(-1/4)+\rho_3} \begin{bmatrix} 2 & 3 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 2 \end{bmatrix}$$

    (a) List the three matrices (P_1, L_1, L_2) that correspond to these three row operations.

    (b) Find the PLU factorization of A. You need to list P, L, U explicitly.

21. Given the following row operations

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 4 & -6 & 0 \\ -2 & 8 & 2 \end{bmatrix} \xrightarrow{\rho_1 \leftrightarrow \rho_2} \begin{bmatrix} 4 & -6 & 0 \\ 2 & 1 & 1 \\ -2 & 8 & 2 \end{bmatrix} \xrightarrow[\rho_1 \frac{1}{2}+\rho_3]{\rho_1(-\frac{1}{2})+\rho_2} \begin{bmatrix} 4 & -6 & 0 \\ 0 & 4 & 1 \\ 0 & 5 & 2 \end{bmatrix} \xrightarrow{\rho_2 \leftrightarrow \rho_3} \begin{bmatrix} 4 & -6 & 0 \\ 0 & 5 & 2 \\ 0 & 4 & 1 \end{bmatrix} \xrightarrow{\rho_2(-\frac{4}{5})+\rho_3} \begin{bmatrix} 4 & -6 & 0 \\ 0 & 5 & 2 \\ 0 & 0 & -0.6 \end{bmatrix}$$

    (a) List the four matrices (P_1, L_1, P_2, L_2) that correspond to these four row operations.

    (b) Find the PLU factorization of A. You need to list P, L, U explicitly.

22. Given the PLU factorization of A as $PA = LU$, where

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ \frac{1}{3} & \frac{1}{4} & 1 \end{bmatrix}, U = \begin{bmatrix} 2 & 3 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 2 \end{bmatrix}.$$

Using these factors, without forming the matrix A or taking any inverses, to solve the system $Ax = b$, where $b = (2, 10, -12)$.

Python Exercises

23. Basic matrix operations:

    (a) Create a random 4×4 matrix A (each entry follows standard normal distribution). Print A.

    (b) Display the 3rd column of A.

    (c) Display the upper right 2×2 submatrix of A.

    (d) Compute the sum of all entries of A.

    (e) Define B to be the transpose of A.

24. How powerful is your computer?

    (a) Create a 5000 by 5000 random matrix `R`, where each entry of `R` is standard normal distribution. Define `r` to be the product of `R` and a 5000×1 vector of all 1's.

    (b) Measure how much time does it take to solve `Rx=r` using Python's built-in function. You may want to run several times so that the time stabilizes. Use `time1` to store the elapsed time. Print it. Verify that the solution is the vector of all 1's.

    (c) Calculate how many gigaflops your computer can do in a second (see Problem 16.) Print your answer.

25. It is never necessary to compute the inverse of a matrix in practice. For example, some people may want to solve a linear system $Ax = b$ by finding the inverse of A and then $x = A^{-1}b$. Let's try this with the system `Rx=r`, where `R` and `r` are from Problem 24. To be precise, you are going to measure how many seconds it takes to "first finding the inverse of `R`, and then performing the multiplication $\texttt{R}^{-1}\texttt{r}$". Use `time2` to store the elapsed time. Both Problem 24(b) and this problem is solving the same linear system. Which method takes more time?

26. Write a function `x = backsub(U, b)` to do backward substitution from scratch. The input `U` must be an invertible upper triangular matrix (of any size) and `b` is a column vector of appropriate length. The function returns the solution of `Ux=b`. You only need one loop in your code. Test your function with the backward substitution step in Example 3.1.

27. More time comparison.

    (a) Define `U` to be the upper triangular portion of a 5000×5000 matrix full of 1's. Define `b` to be the product of `U` and a 5000×1 vector of all 1's.

    (b) Solve `Ux=b` using `scipy.linalg.solve_triangular`. Use `time3` to store the elapsed time.

(c) Solve `Ux=b` using your own function `backsub`. Use `time4` to store the elapsed time. Check that the solution is the same as the one from (b) to make sure your code is correct.

(d) Finally run the following command:

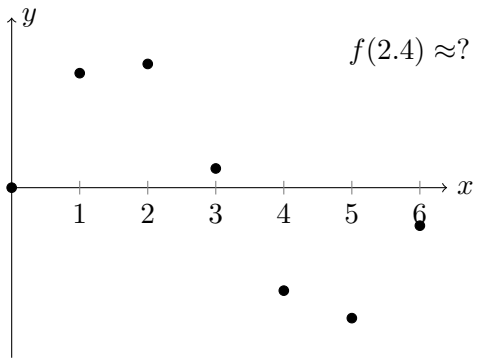
```
print ("For a 5000 by 5000 dense system, it took",time1,"seconds to solve via the
Gaussian elimination. It took",time2,"seconds to solve via finding the inverse.")
print ("For a 5000 by 5000 upper triangular system, it took",time3,"seconds to
solve via using Python's built-in function. It took",time4,"seconds to solve via my code.")
```

Comment on the differences among these time span.

Chapter 4

Polynomial Interpolation

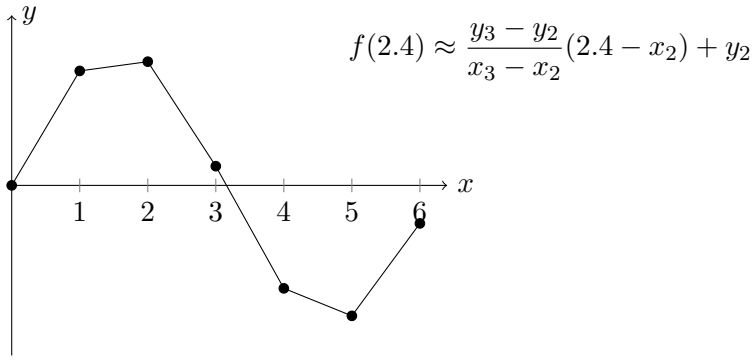
In many applications of engineering and science, one often has a number of data points, obtained by sampling or experimentation, say $f(x_0), f(x_1), \cdots, f(x_n)$, and would like to estimate the value of this function at an intermediate value of the x_i's. See the following figure for example.



This is the problem of interpolation. An estimation of the value at another point is often given by finding a function that at least agrees with f at these $n+1$ points. In general, given $n+1$ points in $\mathbb{R}^2$: $\{(x_i, y_i)\}_{i=0}^n$, any function that goes through these $n+1$ points is called an *interpolant* of these points. This is a classical topic from approximation theory, and we will only present some important theories here.

4.1 Linear Interpolation

The simplest way is linear interpolation, that is, connect nearby points by lines which makes the interpolant a piecewise linear function. See below:



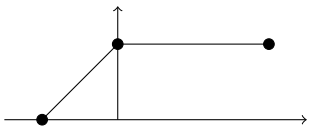
This is quick and easy, but not precise and the whole function is not differentiable at the nodes x_i. But linear interpolation is the simplest example of spline, about which we will talk later.

Example 4.1. Find the linear interpolant of the three points $(-1, 0), (0, 1), (2, 1)$, and then estimate $f(-0.8)$.

Solution: Let $L(x)$ be the linear interpolant and it is a 2-piece line, that is

$$L(x) = \begin{cases} x + 1, & x \in [-1, 0] \\ 1, & x \in [0, 2] \end{cases}.$$

So $f(-0.8) \approx L(-0.8) = 0.2$.



4.2 Interpolation by One Polynomial

Computers can do additions and multiplications really well, therefore making polynomials a friendly function to a computer. For this reason, we would like to interpolate a given set of data points with polynomials.

We can find a line (degree 1 polynomial) going through two points, and we can find a quadratic function (degree 2 polynomial) going through 3 given points. In general, it is believable that we are able to find a polynomial of degree n to fit $n + 1$ points because such polynomial has $n + 1$ coefficients ($n + 1$ degrees of freedom) and we have exactly $n + 1$ pieces of information. We will often use p_n for a polynomial of degree n.

Theorem 4.2. *Given $\{(x_i, y_i)\}_{i=0}^n$ where the $n + 1$ values of x_i are all different, then there is a **unique** polynomial p of degree at most n that goes through these $n + 1$ points.*

The proof is easy. Subsection 4.2.1 can be viewed as a proof.

We will introduce three well-known methods for finding this unique p.

4.2.1 Vandermonde matrix

Let $p(x) = \sum_{i=0}^{n} c_i x^i$. In order to figure out the coefficients c_i, we plug in the known points (x_j, y_j), which generates $n+1$ equations $\sum_{i=0}^{n} c_i x_j^i = y_j, j = 0, 1, \cdots, n$. These equations are all linear, so we get an $(n + 1) \times (n + 1)$ system

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}. \tag{4.1}$$

This is a straightforward method. The coefficient matrix in the system (4.1) is in the form of a Vandermonde matrix.

In general, a $k \times l$ *Vandermonde matrix* is a matrix generated by a vector $v = (v_1, v_2, \cdots, v_k)$ in the following fashion:

$$\text{For } 1 \le i \le l, \text{ the } i\text{th column is } [v_1^{i-1}, v_2^{i-1}, \cdots, v_k^{i-1}]^T.$$

For example, the matrix in Chapter 3 Exercise 1 is a 3×4 Vandermonde matrix generated by the vector (1,2,3).

It can be proven that as long as $x_i \neq x_j$ for any $i \neq j$, then the Vandermonde matrix in (4.1) is invertible, thus (4.1) has a unique solution. System (4.1) will then be solved via Gaussian elimination as described in Chapter 3. In the solution, c_n (or any other coefficient) could be 0, therefore the polynomial we get is of degree at most n.

Example 4.3. Find the quadratic that goes through the three points $(-1, 0), (0, 1), (2, 1)$ (with partial pivoting). Then estimate $f(-0.8)$.

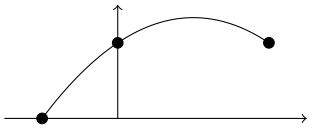
$$\left[\begin{array}{ccc|c} 1 & -1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 2 & 4 & 1 \end{array}\right] \to \left[\begin{array}{ccc|c} 1 & -1 & 1 & 0 \\ 0 & 1 & -1 & 1 \\ 0 & 3 & 3 & 1 \end{array}\right] \to \left[\begin{array}{ccc|c} 1 & -1 & 1 & 0 \\ 0 & 3 & 3 & 1 \\ 0 & 1 & -1 & 1 \end{array}\right] \to \left[\begin{array}{ccc|c} 1 & -1 & 1 & 0 \\ 0 & 3 & 3 & 1 \\ 0 & 0 & -2 & 2/3 \end{array}\right]$$

$c_2 = -1/3; \qquad 3c_1 - 1 = 1 \Rightarrow c_1 = 2/3; \qquad c_0 - 2/3 - 1/3 = 0 \Rightarrow c_0 = 1.$

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

So $f(-0.8) \approx p(-0.8) = 0.2533$.

This can be compared to Example 4.1.



4.2.2 Lagrange interpolation

Lagrange solved this problem in a different approach, where no linear system is involved. First we generate $n + 1$ polynomials $\varphi_i(x)$ such that

$$\varphi_i(x_j) = \delta_{ij} := \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases}.$$

δ_{ij} is a useful notation and is heavily used in mathematics and engineering.

The construction of φ_i is immediate. For example, if $x_0 = 2, x_1 = 2.5, x_2 = 4$, then φ_0 has roots 2.5 and 4, so $\varphi_0(x) = c(x - 2.5)(x - 4)$. c is determined by $\varphi_0(x_0) = 1$, so we can write $\varphi_0(x) = \frac{(x-2.5)(x-4)}{(2-2.5)(2-4)}$. Similarly, $\varphi_1(x) = \frac{(x-2)(x-4)}{(2.5-2)(2.5-4)}$ and $\varphi_2(x) = \frac{(x-2)(x-2.5)}{(4-2)(4-2.5)}$.

Note that φ_i's only depend on the nodes $\{x_0, x_1, \cdots, x_n\}$. In general, they can be written as

$$\varphi_i(x) = \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)}. \tag{4.2}$$

The second step of the Lagrange method is even easier, the answer that we are looking for is then

$$p(x) = \sum_{i=0}^{n} y_i \varphi_i(x) \tag{4.3}$$

Example 4.4. Redo Example 4.3 using Lagrange's method.

$x_0 = -1, x_1 = 0, x_2 = 2$.

Form $\varphi_0(x) = \frac{(x-0)(x-2)}{(-1-0)(-1-2)}, \varphi_1(x) = \frac{(x+1)(x-2)}{(0+1)(0-2)}, \varphi_2(x) = \frac{(x+1)(x-0)}{(2+1)(2-0)}$. After simplification, we have $\varphi_0(x) = \frac{1}{3}x(x-2), \varphi_1(x) = \frac{-1}{2}(x+1)(x-2), \varphi_2(x) = \frac{1}{6}x(x+1)$.

So $p(x) = 0 \cdot \varphi_0(x) + 1 \cdot \varphi_1(x) + 1 \cdot \varphi_2(x) = \frac{-1}{2}(x+1)(x-2) + \frac{1}{6}x(x+1)$. One can verify that this is the same polynomial as the one from Example 4.3.

4.2.3 Newton's interpolation

The Newton's interpolation is similar to the first method. We only formulate $p(x)$ differently as

$$p(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_n \prod_{i=0}^{n-1}(x - x_i).$$

Plugging in $y_0 = p(x_0)$ implies $y_0 = a_0$. Plugging in $y_1 = p(x_1)$ implies $y_1 = a_0 + a_1(x_1 - x_0)$. In the end, $p(x_i) = y_i$ gives us a lower triangular system

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & x_1 - x_0 & 0 & 0 & \cdots & 0 \\ 1 & x_2 - x_0 & (x_2 - x_0)(x_2 - x_1) & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ 1 & x_n - x_0 & (x_n - x_0)(x_n - x_1) & \cdots & & \prod_{i=0}^{n-1}(x_n - x_i) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}. \tag{4.4}$$

It is very efficient to solve (4.4) because it is a lower triangular system.

Example 4.5. Redo Example 4.3 using Newton's interpolation.

We let $p(x) = a_0 + a_1(x+1) + a_2(x+1)(x-0)$. Once again, in this method, The formulation of p depends on all, but the last nodes. We still have $p(x_i) = y_i$ for $i = 0, 1, 2$, which produces three equations:

$$\begin{aligned} a_0 &= 0 \\ a_0 + a_1(0+1) &= 1 \\ a_0 + a_1(2+1) + a_2(2+1)(2-0) &= 1 \end{aligned}. \tag{4.5}$$

This is the same as $\left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 1 & 0+1 & 0 & 1 \\ 1 & 2+1 & (2+1)(2-0) & 1 \end{array}\right]$, which matches (4.4). Solving this lower triangular system, we get $a_0 = 0, a_1 = 1, a_2 = -1/3$.

So $p(x) = (x+1) - \dfrac{1}{3}(x+1)(x-0)$.

One benefit of using Newton's formulation is that the computation is minimal for adding a data point. Suppose now we are trying to find the degree 3 polynomial going through $(-1, 0), (0, 1), (1.5, 3), (2, 1)$. We observe that $(1.5, 3)$ is the extra point, so we will reorder the four points as $(-1, 0), (0, 1), (2, 1), (1.5, 3)$. [The ordering of the points has no effect on the polynomial interpolant. These points are considered collectively.]

Letting $q(x) = a_0 + a_1(x+1) + a_2(x+1)(x-0) + a_3(x+1)(x-0)(x-2)$. Plugging these four points, we get 4 equations:

$$\begin{aligned} a_0 &= 0 \\ a_0 + a_1(0+1) &= 1 \\ a_0 + a_1(2+1) + a_2(2+1)(2-0) &= 1 \\ a_2 + a_2(1.5+1) + a_2(1.5+1)(1.5-0) + a_3(1.5+1)(1.5-0)(1.5-2) &= 3 \end{aligned}. \tag{4.6}$$

Note that the first three equations are exactly the same as (4.5)! This means we can reuse the answers a_0, a_1, a_2 solved earlier. We only need to solve for a_3 as the additional work.

Among all three methods presented, Newton's interpolation is the only one that is easy to add a node.

4.2.4 Flop count

	Vandermonde	Largange	Newton
Compute coefficients	$O(n^3)$	0	$O(n^2)$
Evaluate at one point	$O(n)$	$O(n^2)$	$O(n)$

Table 4.1: Flop count for different methods of polynomial interpolation

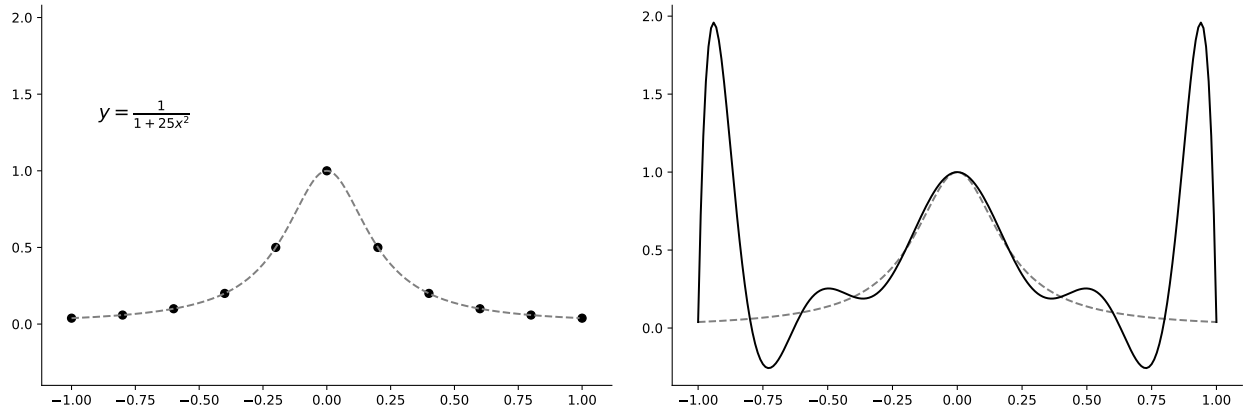
For the Vandermonde method, it takes $O(n^3)$ flops to compute the coefficients, since this is how much you need for Gaussian elimination. To evaluate $p(x) = \sum_{i=0}^{n} c_i x^i$ at a point, we mentioned at the end of Chapter 1 that we need to use the nested form, which takes $2n$ flops.

For Lagrange interpolation, we simply write down the interpolation polynomial (4.3) so it takes no work to generate the polynomial. Evaluation at one point will take $O(n^2)$ flops.

For Newton's interpolation, it takes $O(n^2)$ flops to compute the a_i's since we have a lower triangular system. Evaluation at one point only takes $O(n)$ as well with the nested form. For example, $a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + a_3(x - x_0)(x - x_1)(x - x_2)$ should be rewritten as $a_0 + (x - x_0)\Big[a_1 + (x - x_1)\Big(a_2 + a_3(x - x_2)\Big)\Big]$.

4.2.5 Limitation of polynomial interpolant

Consider the function $R(x) = \dfrac{1}{1 + 25x^2}$ on the interval $I = [-1, 1]$. This is called the Runge function (scaled). Suppose we are interpolating 11 equally spaced points that we get from sampling $R(x)$ (Figure 4.1 (a)). Figure 4.1 (b) is the graph of the degree 10 polynomial that goes through these 11 points. Call this polynomial $P(x)$. Notice that there are big oscillations near the ends of the interval. In fact, as degree increases, the oscillations near the end will get even larger.



(a) Runge function

(b) Interpolate at 11 equally spaced points

Figure 4.1: Runge function effect

In the next two sections, we introduce two approaches to reduce this Runge function effect. The first approach is to pick a better set of nodes. This is Section 4.3. The second approach, discussed in Section 4.4, is to use multiple lower degree polynomials.

4.3 Chebyshev nodes

The polynomial interpolant depends on the nodes $\{x_i\}_{i=0}^n$. In the Runge function example, we used equally spaced points, which is a common choice. However, we will show that there are better nodes in the sense that the corresponding polynomial interpolant is a better approximate. For this we will have to first find a way to measure distances between functions.

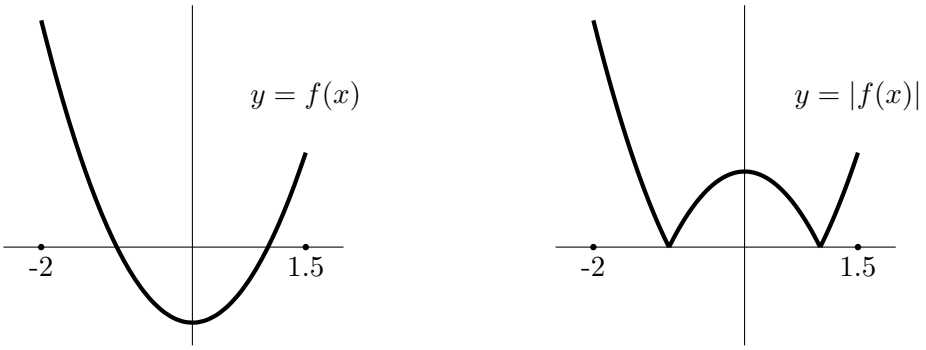
Given f, who is continuous on [a,b], we define its ∞-norm by

$$\|f\|_\infty := \max_{x\in[a,b]} |f(x)|,$$

and its 2-norm by

$$\|f\|_2 := \sqrt{\int_a^b |f(x)|^2 dx}.$$

Example 4.6. Let $f(x) = x^2 - 1$ on [-2, 1.5]. Since this function is easy to graph, we will figure out its norm by first graphing it. Note that to draw $|f(x)|$, we flip the negative portion up.



It is obvious that the maximum value of $|f(x)|$ is obtained at $x = -2$, so $\|f\|_\infty = |f(-2)| = 3$.

$\|f\|_2^2 := \int_{-2}^{1.5} (x^2-1)^2 dx = \int_{-2}^{1.5} x^4 - 2x^2 + 1 dx = \frac{1.5^5 + 2^5}{5} - 2\frac{1.5^3 + 2^3}{3} + 3.5 \approx 3.835417.$

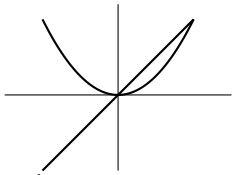
$\|f\|_2 = \sqrt{3.835417} \approx 1.96.$

Correspondingly, we can define two different distances between two continuous functions f, g on $[a, b]$:

$$\infty\text{-distance: } \|f-g\|_\infty = \max_{x\in[a,b]} |f(x) - g(x)|$$

$$\text{2-distance: } \|f-g\|_2 = \sqrt{\int_a^b |f(x) - g(x)|^2 dx}$$

Example 4.7. Let $f(x) = x^2, g(x) = x$ on [-1,1].



We can see from the graph on the left that f and g differ the most at $x = -1$, so $\|f - g\|_\infty = |f(-1) - g(-1)| = 2$.

$\int_{-1}^{1} (f-g)^2 dx = \int_{-1}^{1} (x^2 - x)^2 dx = \int_{-1}^{1} x^4 + x^2 - 2x^3 dx = 16/15.$

$\|f - g\|_2 = \sqrt{16/15}.$

The ∞ distance between the Runge function $R(x)$ and its equally spaced interpolant $P_{10}(x)$ (as shown in Figure 4.1(b)) is $\|R - P_{10}\|_\infty = 1.9157$. (The maximum occurs at $x = \pm 0.94$.)

In general, it is not easy to find the maximum of a function. As learned in Calculus I, a common method is to find the critical points by taking its derivative, and then compare all values at the critical points and end points. This is why we also introduce the 2-distance, which is a lot easier to compute than the ∞-distance.

The following theorem illustrates the point-wise error of polynomial interpolation.

Theorem 4.8. *Assume that $f \in C^{n+1}[a,b]$ ($n+1$st derivative is continuous) and that $x_0, \cdots, x_n$ are in $[a,b]$. Let $p(x)$ be the polynomial of degree n that interpolates f at $\{x_i\}_{i=0}^n$. Then for any point $x \in [a,b]$,*

$$f(x) - p(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i) \tag{4.7}$$

for some point $\xi \in [a,b]$.

The proof only needs Rolle's theorem, see Theorem 8.4.1 of [2] for details.

Let's suppose the interval is [-1,1] for now. This gives us an estimate on the L_∞ distance between f and p on the interval [-1,1]:

$$\|f - p\|_\infty = \max_{x \in [-1,1]} \frac{|f^{(n+1)}(\xi)|}{(n+1)!} \prod_{i=0}^{n} |x - x_i| \le \max_{x \in [-1,1]} \frac{\|f^{(n+1)}\|_\infty}{(n+1)!} \max_{x \in [-1,1]} \prod_{i=0}^{n} |x - x_i|.$$

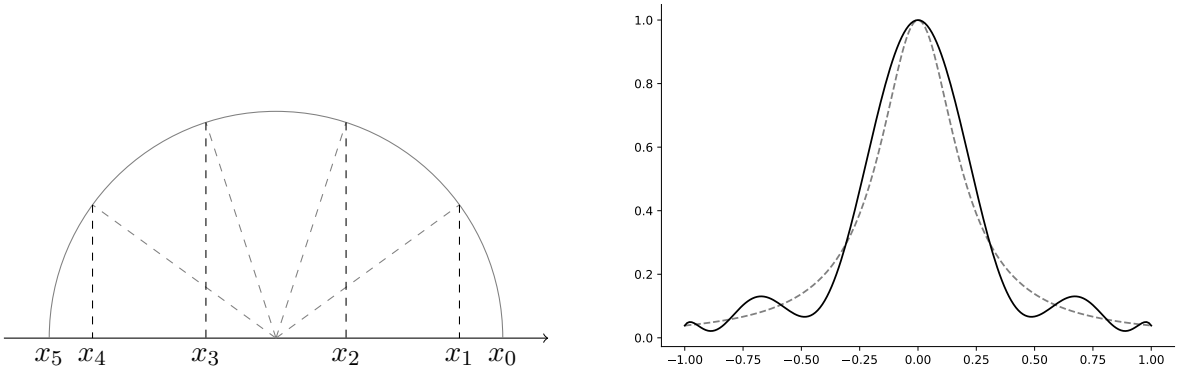
Suppose f is given (like the runge function), in order to minimize the error bound, we need to solve

$$\min_{\{x_0, \cdots, x_n\} \subset [-1,1]} \max_{x \in [-1,1]} \prod_{i=0}^{n} |x - x_i|. \tag{4.8}$$

The solution (not trivial) for this minimization problem is

$$\text{Chebyshev nodes:} \quad x_i = \cos \frac{\pi i}{n}, i = 0, 1, \cdots, n \tag{4.9}$$

which are the x coordinates of n equally spaced angles on the positive half unit circle, see Figure 4.2(a). These are the roots of the Chebyshev polynomial of the first kind. These nodes are not equally spaced, and are more clustered around the ends.



(a) Chebyshev nodes for $n = 5$ (b) Interpolation at Chebyshev nodes for $n = 10$

Figure 4.2: Interpolation with Chebyshev nodes

Figure 4.2 (b) is the Chebyshev interpolant, which approximates the runge function much better. We call this polynomial $C_{10}(x)$. The ∞ distance $\|R - C_{10}\|_\infty = 0.1322$, is only about 1/15 of $\|R - P_{10}\|_\infty$, when equally spaced points were used. Therefore if there is a choice on where to interpolate, picking the Chebyshev nodes is optimal.

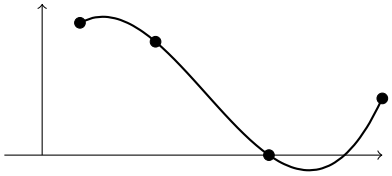
The definition $x_i = \cos\dfrac{\pi i}{n}$ only works for the interval [-1,1]. What happens if we have [-2,2], or in general $[a, b]$? This is left as an exercise.

4.4 Spline

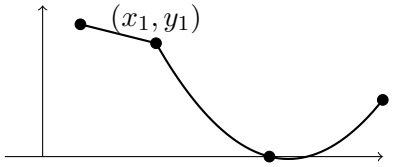
Spline[1] means piecewise polynomials, meaning it is a piecewisely defined function and on each piece it is a different polynomial. Moreover, we generally require it to be continuous. This simply means that different pieces connect at the same point. There is no jump or drop. Linear interpolant from Section 4.1 is an instance of spline. As a spline, each piece does not have to be degree 1 polynomial, and their degrees can be different. The following function is another example of spline.

$$s(x) = \begin{cases} -x^2 + x + 1, & 0 \le x \le 1 \\ x^3, & 1 \le x \le 2 \\ 2x + 4, & 2 \le x \le 3 \end{cases}$$

The purpose of spline is to interpolate the same number of data points, with lower degrees, at the cost of multiple polynomials. Lower degree polynomials are not only faster to be evaluated by computers, they also tend to have less oscillations, which potentially reduces the Runge function effect. Figure 4.3(a) is to interpolate 4 points with one degree 3 polynomial (cubic), and Figure 4.3(b) is to interpolate the first two points with a line (degree 1) and the last three with a quadratic (degree 2).



(a) One polynomial interpolation as shown in Section 4.2



(b) Spline interpolation: a not desirable case

Figure 4.3

However, Figure 4.3 (b) is a situation that we want to avoid. It is not smooth at the second point x_1. To be precise, this piecewise function is not differentiable at x_1 because the left derivative, which is the slope of the line, does not coincide with the right derivative.

Because spline is multiple polynomials piecing together, we need to make sure they are "well-connected" at x_i's. In real applications, we often require it to be twice differentiable. Below is an example of a (one time) differentiable spline.

Example 4.9. Given three points $(0, 1), (1, -1), (2, 0)$, find an interpolant

$$s(x) = \begin{cases} P_1(x), & 0 \le x \le 1 \\ P_2(x), & 1 \le x \le 2 \end{cases}$$

[1]The root of the word "spline" is the same as that of splint. It was originally a small strip of wood that could be used to join two boards. Later the word was used to refer to a long flexible strip, generally of metal, that could be used to draw continuous smooth curves by forcing the strip to pass through specified points and tracing along the curve.

such that $s(x)$ is differentiable. P_1 is a polynomial of degree 1, and P_2 is a polynomial of degree 2.

This boils down to

(1) $P_1(0) = 1, P_1(1) = -1; P_2(1) = -1, P_2(2) = 0$

(2) $P_1'(1) = P_2'(1)$.

Condition (1) already completely determines P_1: $P_1(x) = -2x + 1$. Therefore by (2) we know $P_2'(1) = P_1'(1) = -2$. Since P_2 is of degree 2, we can let $P_2'(x) = 2a(x-1) - 2$, which means that $P_2(x) = a(x-1)^2 - 2(x-1) + b$. Note that $b = -1$ can be determined right away by $P_2(1) = -1$.

All that is left is to determine a. Plugging $P_2(2) = 0$ yields $a - 2 - 1 = 0$. So $a = 3$.

In the end, $P_1(x) = -2x + 1, P_2(x) = 3(x-1)^2 - 2(x-1) - 1$. It is graphed in Figure 4.4.

In order to simply fulfill the continuity requirement (1), there could be many other choices of P_2 (like the gray dashed lines in Figure 4.4), but only the black one connects with the line smoothest.

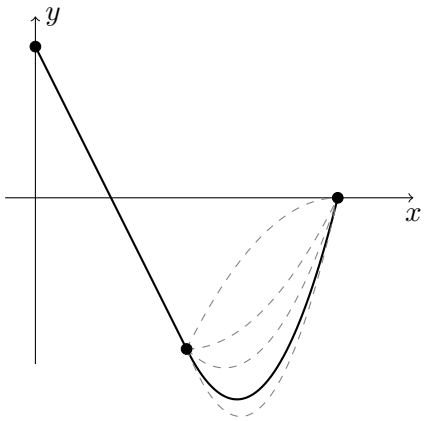


Figure 4.4: Spline interpolation: a somewhat desirable case

If a function is differentiable in higher order, then it is considered to be "smoother". Let us consider another piecewisely defined function $f(x) = q_2(x) := -x^2 + x$ on [0,1] and $f(x) = q_3(x) := (x-1)(x^2 - 3x + 1)$ on [1,2], see Figure 4.5. The quadratic q_2 and the cubic q_3 share the 0th, 1st, and 2nd derivative at $x = 1$ as $q_2(1) = q_3(1) = 0, q_2'(1) = q_3'(1) = -1, q_2''(1) = q_3''(1) = -2$. In fact, there are infinitely many other cubics will do this without specifying the value at $x = 2$ (left as an exercise). As a result, $f(x)$ is twice differentiable, and it transits even smoother at $x = 1$ than $s(x)$ in Example 4.9.

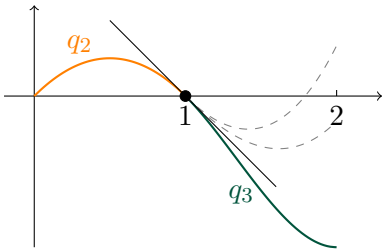


Figure 4.5: Twice differentiable spline: a desirable case

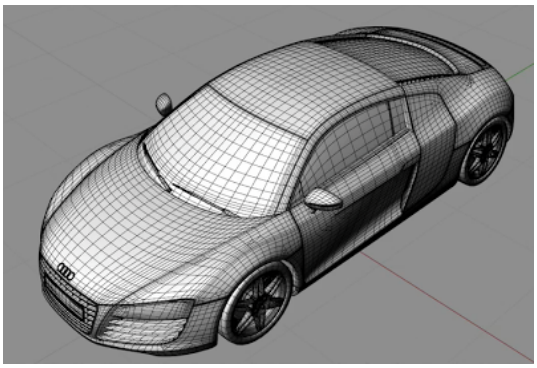
To show the difference between smoothness of once differentiable and twice differentiable functions, we draw in Figure 4.5 some other cubic pieces that will have the same first derivative as q_2 at $x = 1$, but fails to have the second derivative equal (in gray dashed lines). Notice that the second derivative reveals the concavity of a function. In Figure 4.5, q_3 is keeping the concavity near 1,

whereas the gray dashed lines are changing the concavity.

Next section will specifically talk about how to use multiple degree 3 polynomials (cubic) to interpolate given points smoothly (twice differentiable).

4.4.1 Cubic Spline

Spline is used heavily in 2D and 3D computer graphics where one wishes to design a smooth curve/surface based on a collection of nodes. There are numerous examples including design of fonts, car manufacture, and animation.



For $n \geq 2$, given $\{(x_i, y_i)\}_{i=0}^n$, a Cubic spline is to find n cubics $q_i(x)$ such that the interpolant

$$s_c(x) = \begin{cases} q_1(x), & x_0 \leq x \leq x_1 \\ q_2(x), & x_1 \leq x \leq x_2 \\ \vdots & \\ q_n(x) & x_{n-1} \leq x \leq x_n \end{cases}$$

is twice differentiable.

According to the discussion earlier, this precisely means

(a) Each cubic goes through two points: $q_i(x_{i-1}) = y_{i-1}, \quad q_i(x_i) = y_i, \quad i = 1, 2, \cdots, n$

(b) First derivative match: $q_i'(x_i) = q_{i+1}'(x_i), \quad i = 1, 2, \cdots, n-1$

(c) Second derivative match: $q_i''(x_i) = q_{i+1}''(x_i) = z_i, \quad i = 1, 2, \cdots, n-1$. ($z_i$ to be determined.)

Each cubic has 4 coefficients, so there are a total of $4(n+1)$ unknowns. We get $2n+n-1+n-1 = 4n-2$ constraints from (a)(b)(c). This leaves 2 degrees of freedom, which can be used to enforce various conditions on the endpoints of the spline, resulting in different types of cubic splines.

It is not hard to derive a formula for s_c. Let $h_i = x_i - x_{i-1}$ be the length of the ith subinterval. In the case of equally spaced points, $h_i = h = \dfrac{b-a}{n} = \dfrac{x_n - x_0}{n}$.

We start with constraint (c): For $i = 1, \cdots, n$, q_i'' is a line, and it goes through (x_{i-1}, z_{i-i}) and (x_i, z_i), so

$$i = 1, ..., n \qquad q_i''(x) = \frac{z_i - z_{i-1}}{h_i}(x - x_{i-1}) + z_{i-1}. \tag{4.10}$$

Integrating (4.10) twice, we obtain

$$i = 1, ..., n \qquad q_i'(x) = \frac{z_i - z_{i-1}}{2h_i}(x - x_{i-1})^2 + z_{i-1}(x - x_{i-1}) + C_i. \tag{4.11}$$

$$i = 1, ..., n \qquad q_i(x) = \frac{z_i - z_{i-1}}{6h_i}(x - x_{i-1})^3 + \frac{z_{i-1}}{2}(x - x_{i-1})^2 + C_i(x - x_{i-1}) + y_{i-1}. \tag{4.12}$$

The constant term in (4.12) is y_{i-1} since we used the condition that $q_i(x_{i-1}) = y_{i-1}$. Constraint (a) generates $\frac{z_i - z_{i-1}}{6}h_i^2 + \frac{z_{i-1}}{2}h_i^2 + C_i h_i + y_{i-1} = y_i$, so

$$C_i = \frac{y_i - y_{i-1}}{h_i} - h_i(\frac{z_i}{6} + \frac{z_{i-1}}{3}), \quad i = 1, \cdots, n. \tag{4.13}$$

Plugging in C_i in (4.11),

$$q_i'(x) = \frac{z_i - z_{i-1}}{2h_i}(x - x_{i-1})^2 + z_{i-1}(x - x_{i-1}) + \frac{y_i - y_{i-1}}{h_i} - h_i(\frac{z_i}{6} + \frac{z_{i-1}}{3}).$$

Finally we use constraint (b) to get, $i = 1, \cdots, n-1$,

$$\frac{z_i - z_{i-1}}{2h_i}h_i^2 + z_{i-1}h_i + \frac{y_i - y_{i-1}}{h_i} - h_i(\frac{z_i}{6} + \frac{z_{i-1}}{3}) = \frac{y_{i+1} - y_i}{h_{i+1}} - h_{i+1}(\frac{z_{i+1}}{6} + \frac{z_i}{3})$$

which simplifies to $n-1$ linear equations about z_i:

$$i = 1, ..., n-1, \quad \frac{h_i}{6}z_{i-1} + \frac{h_i + h_{i+1}}{3}z_i + \frac{h_{i+1}}{6}z_{i+1} = \frac{y_{i+1} - y_i}{h_{i+1}} - \frac{y_i - y_{i-1}}{h_i}. \tag{4.14}$$

There are $n-1$ equations, and $n+1$ unknowns $(z_0, z_1, \cdots, z_n)$, so we are free to choose z_0 and z_n. The choice $z_0 = z_n = 0$ gives what is called the *natural cubic spline*, in which case (4.14) becomes a symmetric tridiagonal system

$$\begin{bmatrix} \frac{h_1+h_2}{3} & \frac{h_2}{6} & & & \\ \frac{h_2}{6} & \frac{h_2+h_3}{3} & \frac{h_3}{6} & & \\ & \frac{h_3}{6} & \frac{h_3+h_4}{3} & \ddots & \\ & & \ddots & \ddots & \frac{h_{n-1}}{6} \\ & & & \frac{h_{n-1}}{6} & \frac{h_{n-1}+h_n}{3} \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ \vdots \\ z_{n-1} \end{bmatrix} = \begin{bmatrix} \frac{y_2 - y_1}{h_2} - \frac{y_1 - y_0}{h_1} \\ \frac{y_3 - y_2}{h_3} - \frac{y_2 - y_1}{h_2} \\ \vdots \\ \vdots \\ \frac{y_n - y_{n-1}}{h_n} - \frac{y_{n-1} - y_{n-2}}{h_{n-1}} \end{bmatrix}. \tag{4.15}$$

Once z_i's are solved, we use (4.13) to compute the constants C_i, and then find the n cubics using (4.12).

Example 4.10. Find the natural cubic spline that interpolates $(0,0), (1,0), (2,2), (3,2), (4,-1)$. Then use it to approximate $f(0.5)$ and $f'(0.5)$.

In this example, we have $n = 4$. The points are equally spaced points so $h_i = 1$. We already know the boundary condition $z_0 = z_4 = 0$. The system (4.15) becomes a 3×3 system

$$\begin{bmatrix} 2/3 & 1/6 & 0 \\ 1/6 & 2/3 & 1/6 \\ 0 & 1/6 & 2/3 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 2 \\ -2 \\ -3 \end{bmatrix}$$

whose solution is $z_1 = 15/4, z_2 = -3, z_3 = -15/4$. We can use (4.12) and (4.13) to compute q_i. For example, $C_1 = y_1 - y_0 - (z_1/6 + z_0/3) = -5/8$, so

$$q_1(x) = \frac{z_1 - z_0}{6h_1}(x - x_0)^3 + 0 + C_1(x - x_0) + y_0 = \frac{5}{8}(x-0)^3 - \frac{5}{8}(x-0).$$

Similarly, $C_2 = y_2 - y_1 - (z_2/6 + z_1/3) = 2 - (-1/2 + 5/4) = 5/4$, so

$$q_2(x) = \frac{z_2 - z_1}{6h_2}(x - x_1)^3 + \frac{z_1}{2}(x - x_1)^2 + C_2(x - x_1) + y_1 = -\frac{9}{8}(x-1)^3 + \frac{15}{8}(x-1)^2 + \frac{5}{4}(x-1).$$

To approximate $f(0.5)$ and $f'(0.5)$, we only need q_1 since 0.5 falls in the first subinterval.

$$f(0.5) \approx q_1(0.5) = -0.2344, \qquad f'(0.5) \approx q_1'(0.5) = \frac{15}{8}(0.5)^2 - \frac{5}{8} = -0.1562.$$

Remark 4.11. It is the best that we compute the coefficients in Example 4.10 in a vectorized way. We have

index	x	y	h	z
0	0	0	0	0
1	1	0	1	15/4
2	2	2	1	-3
3	3	2	1	-15/4
4	4	-1	1	0

The formula (4.13), $C_i = \dfrac{y_i - y_{i-1}}{h_i} - h_i(\dfrac{z_i}{6} + \dfrac{z_{i-1}}{3})$ translates to

$$C = \left(\begin{bmatrix} 0 \\ 2 \\ 2 \\ -1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 2 \\ 2 \end{bmatrix} \right) ./ \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} .* \left(\frac{1}{6} \begin{bmatrix} 15/4 \\ -3 \\ -15/4 \\ 0 \end{bmatrix} + \frac{1}{3} \begin{bmatrix} 0 \\ 15/4 \\ -3 \\ -15/4 \end{bmatrix} \right) = \begin{bmatrix} -5/8 \\ 5/4 \\ 13/8 \\ -7/4 \end{bmatrix}$$

The formula (4.12), $q_i(x) = \dfrac{z_i - z_{i-1}}{6h_i}(x - x_{i-1})^3 + \dfrac{z_{i-1}}{2}(x - x_{i-1})^2 + C_i(x - x_{i-1}) + y_{i-1}$ translates to

$$\text{degree 3 coeff: } \frac{1}{6} \left(\begin{bmatrix} 15/4 \\ -3 \\ -15/4 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 15/4 \\ -3 \\ -15/4 \end{bmatrix} \right) ./ \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 5/8 \\ -9/8 \\ -1/8 \\ 5/8 \end{bmatrix} \tag{4.16}$$

$$\text{degree 2 coeff: } \frac{1}{2} \begin{bmatrix} 0 \\ 15/4 \\ -3 \\ -15/4 \end{bmatrix} = \begin{bmatrix} 0 \\ 15/8 \\ -3/2 \\ -15/8 \end{bmatrix} \tag{4.17}$$

$$\text{degree 1 coeff: } C = \begin{bmatrix} -5/8 \\ 5/4 \\ 13/8 \\ -7/4 \end{bmatrix} \tag{4.18}$$

$$\text{constant term: } \begin{bmatrix} 0 \\ 0 \\ 2 \\ 2 \end{bmatrix} \tag{4.19}$$

Chapter 4 Exercises

1. Find the polynomial interpolant of (0,-1), (1,1), (2,1) using the Vandermonde matrix. Estimate the value at $x = 0.5$.

2. Find the polynomial interpolant of (1,7/4), (2,3/2), (3.5,0), and (5,3/4) by

   (a) Vandermonde method (feel free to use Python in finding the coefficients.)

   (b) Lagrange method (handwritten, don't simplify)

3. Directly write down the polynomial interpolant for two points $(x_0, y_0), (x_1, y_1)$ using Lagrange's interpolation.

4. Given four points $p_1 = (-1, 0), p_2 = (0, -1), p_3 = (1, 1), p_4 = (2, 1)$. Use Newton's interpolation to find the polynomial interpolant going through

   (a) p_1, p_2, p_3

   (b) p_1, p_2, p_3, p_4

5. Find $\|f\|_\infty$ and $\|f\|_2$ where $f(x) = e^x - 2$ and the interval is [0,1].

6. Let $f(x) = 3/2 - x/2, g(x) = \dfrac{1}{x}$, defined on [1,2]. Find $\|f - g\|_\infty$ and $\|f - g\|_2$. (Hint: for ∞-norm, you can use derivative to find maximum value (material from Calc I).)

7. The Chebyshev nodes defined in (4.9) is only for interval [-1,1]. You need to scale/shift your nodes correspondingly with a different interval. For example, for [-2,2], the interval is scaled by 2, so the nodes should be $2\cos\dfrac{\pi i}{n}$. For [0,2], the nodes should be $\cos\dfrac{\pi i}{n} + 1$, shifted by 1. In general, we have to do both scaling and shifting.

   (a) Write down the Chebyshev nodes for the interval [-1,3]. $n = 6$.

   (b) Write down the Chebyshev nodes for a general interval $[a, b]$.

8. Find the linear interpolant of the three points (0,-1), (1,1), (2,1).

9. Determine a differentiable interpolant of (0,1), (1,0), (2,0) P such that

$$P(x) = \begin{cases} P_2(x)(\text{degree} \leq 2), & 0 \leq x \leq 1 \\ P_1(x)(\text{degree} \leq 1), & 1 \leq x \leq 2 \end{cases}$$

10. If $s(x) = -x^2 + x$ when $0 \leq x \leq 1$ and $s(x) = P_3(x)$ on $1 \leq x \leq 2$.

    (a) Find all possible cubic P_3 that makes $s(x)$ twice differentiable on [0,2].

    (b) Among the ones in (a), find the specific cubic that goes through (2,-1).

11. Given the data points $f(-1) = 0, f(0) = 1, f(2) = 1$ (same data points as in Example 4.1 and Example 4.3).

    (a) Determine the natural cubic spline that interpolates these points.

    (b) Use this cubic spline to estimate $f(-0.8)$.

    (c) Draw this cubic spline.

12. Given the five points (1, 0), (2, 1), (3, 0), (4,1), (5,0),

    (a) find the natural cubic spline that interpolate . You should follow Remark 4.11 and your final answer should look like (4.16)-(4.19).

(b) These five points are coming from an unknown function $y = f(x)$. Estimate $f'(3.2)$.

Python Exercises

13. Draw the cubic spline found in Example 4.10. You can directly use the coefficients found in (4.16)-(4.19). You need to use the command `np.polyval`, and each piece needs to have a different color.

14. (a) Write a function `pp = myspline(x, y)` to return the natural cubic spline that interpolates the data `x` $= (x_0, x_1, \cdots, x_n)$, `y` $= (y_0, y_1, \cdots, y_n)$. The output `pp` should be a $4 \times n$ matrix where one column corresponds to the coefficients of one piece.

    Remark: You will get extra credit if you do not use any loop (for/while) in your function.

    (b) Test `myspline` on Example 4.10. Your test is successful if your function returns

```
array([[ 0.625, -1.125, -0.125,  0.625],
       [ 0.   ,  1.875, -1.5  , -1.875],
       [-0.625,  1.25 ,  1.625, -1.75 ],
       [ 0.   ,  0.   ,  2.   ,  2.   ]])
```

    Remark: You will need to use `myspline` in all the subsequent problems. **If your function is not able to pass the test, use Python's `CubicSpline(x,y,bc_type='natural')` instead for the rest of the problems.**

15. Use `myspline` to construct the natural cubic spline for the following data. Print the coefficient matrix.

(a)

x	$f(x)$
-0.5	-0.0247500
-0.25	0.3349375
0	1.1010000

(b)

x	$f(x)$
0.1	0.90483742
0.2	0.81873075
0.3	0.74081822
0.5	0.60653066

16. The data in Exercise 15 were generated using the following functions. Use the cubic spline constructed to do the following approximation and **print the true value as well**.

    (a) $f(x) = x^3 + 4.001x^2 + 4.002x + 1.101$; approximate $f(-1/3)$ and $f'(-1/3)$.

    (b) $f(x) = e^{-x}$; approximate $f(0.25)$ and $f'0.25)$.

17. Interpolate the Runge function on [-1,1] at 11 equally spaced points using `myspline`. Plot the interpolant and the Runge function in ONE figure, but with Runge in dash and the spline in solid line. How is this spline performing compared to Figure 4.1(b) or Figure 4.2(b)?

18. The following table is the number of cumulative coronavirus cases in New Mexico in 2020.[2]

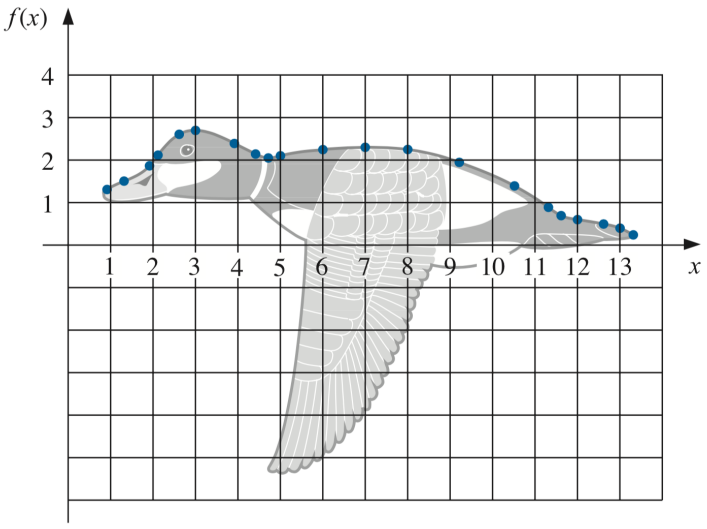
Date	3/23	3/24	3/25	3/26	3/27	3/28	3/30	3/31
Cases	83	100	112	136	191	208	281	315
Date	4/1	4/2	4/3	4/4	4/5	4/6	4/7	4/8
Cases	363	403	495	543	624	686	794	865

[2] source: covidtracking.com

(a) Note that the data on 3/29 is missing. Find an estimation of the cumulative number of cases on 3/29 by interpolating these 16 points via `myspline`.

(b) Redo (a) only using the data in March. How does your estimation compared to that of (a)? Comment on it.

19. (An application in computer graphics) Below is a picture of a ruddy duck in flight. Use `myspline` to reconstruct its back with the 21 points listed. To save your work, coordinates are already saved in the data file `duck.npy`. If you execute data = np.load("duck.npy"), then the first row will be the x coordinates and the second row will be the y coordinates. Use the command `plt.axes().set_aspect('equal')` to show the true aspect ratio of a plot.



x	y
0.9	1.3
1.3	1.5
1.9	1.85
2.1	2.1
2.6	2.6
3.0	2.7
3.9	2.4
4.4	2.15
4.7	2.05
5.0	2.1
6.0	2.25

x	y
7.0	2.3
8.0	2.25
9.2	1.95
10.5	1.4
11.3	0.9
11.6	0.7
12.0	0.6
12.6	0.5
13.0	0.4
13.3	0.25

Chapter 5

Numerical Integration

It is crucial to be able to compute a definite integral $\int_a^b f(x)dx$ in many scientific applications. Numerical integration is necessary when it is difficult or impossible to find an antiderivative of f that is an elementary function. An example of such an integrand is $f(x) = e^{-x^2}$, which is widely used in probability (normal distribution), signal processing, differential equations, etc.

A numerical integration formula is also called a *Quadrature* formula, which is a historical mathematical term that means calculating area. The name comes from the Latin word quadratus meaning square.

5.1 Basic Quadrature Rules

One way to approximate $\int_a^b f(x)dx$ is to replace f by its polynomial interpolant and integrate the polynomial. Formulas derived in this way, using equally-spaced nodes, are called *Newton-Cotes formulas*.

When $n = 1$, then integrating f is replaced by integrating its degree 1 polynomial, i.e., linear interpolant. In other words, the area below the graph of $y = f(x)$ on $[a, b]$ is approximated by the trapezoid. See Figure **??**(a). So the Newton-Cotes formula for two nodes is also called the *Trapezoidal rule*.

To analyze the error, we use formula (4.7) with $n = 1$, so

$$f(x) - p_1(x) = \frac{1}{2}f''(\xi_x)(x-a)(x-b).$$

Integrating both sides, we have

$$\begin{aligned}\int_a^b f(x)dx - \int_a^b p_1(x)dx &= \int_a^b \frac{1}{2}f''(\xi_x)(x-a)(x-b)dx \\ &= \frac{1}{2}f''(\eta)\int_a^b (x-a)(x-b)dx, \qquad \text{for some } \eta \in [a,b] \\ &= -\frac{1}{12}(b-a)^3 f''(\eta).\end{aligned}$$

Note that the second line is justified by the Mean Value Theorem for integrals since $(x-a)(x-b)$ has the same sign throughout $[a, b]$.

$p_1(x) = \dfrac{f(b)-f(a)}{b-a}(x-a) + f(a)$, so in summary

Trapezoidal Rule:

$$\int_a^b f(x)dx \approx \int_a^b p_1(x)dx = \frac{f(a)+f(b)}{2}(b-a). \tag{5.1}$$

$$\text{Error bound } \left|\int_a^b f(x)dx - \int_a^b p_1(x)dx\right| = \frac{(b-a)^3}{12}|f''(\eta)| \le \frac{(b-a)^3}{12}\|f''\|_\infty. \tag{5.2}$$

Example 5.1. Use Trapezoidal rule to approximate $\int_0^2 \sqrt{1+x^2}dx$ and provide an error bound.

Let $f(x) = \sqrt{x^2+1}$.

$\int_0^2 \sqrt{1+x^2}dx \approx \frac{f(0)+f(2)}{2}(2-0) = 1+\sqrt{5} \approx 3.2361.$

To find the error bound, we compute $f'(x) = \frac{x}{(1+x^2)^{1/2}}, \quad f''(x) = \frac{1}{(1+x^2)^{3/2}}.$

It is easy to see that $|f''(x)| = f''(x)$ is decreasing on [0,2], so $\|f''\|_\infty = \max_{x\in[0,2]} |f''(x)| = f''(0) =$ 1. Therefore the error bound is $\frac{(b-a)^3}{12}\|f''\|_\infty = \frac{2^3}{12} * 1 = \frac{2}{3}$.

For the record, the exact absolute error is 0.2782, which is less than the error bound as expected. See Table **??**.

Chapter 6

Preliminaries

6.1 Calculus

Theorem 6.1 (Intermediate Value Theorem)**.** *If $f(x)$ is continuous on $[a, b]$, then for any m that is in between $f(a)$ and $f(b)$, there exists a number $c \in [a, b]$ such that $f(c) = m$.*

Theorem 6.2 (Taylor's Theorem)**.** *Suppose $f \in C^n[a, b]$ and $f^{(n+1)}$ exists on [a,b]. Let $x_0 \in [a, b]$. For every $x \in [a, b]$, there exists a number ξ between x_0 and x such that*

$$f(x) = f(x_0)+f'(x_0)(x-x_0)+\frac{f''(x_0)}{2!}(x-x_0)^2+\cdots+\frac{f^{(n)}(x_0)}{n!}(x-x_0)^n+\frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)^{n+1}. \quad (6.1)$$

Theorem 6.3 (Mean Value Theorem)**.** *If $f \in C[a, b]$ and f is differentiable on (a, b), then a number c in (a, b) exists such that*

$$f'(c) = \frac{f(b) - f(a)}{b - a}.$$

MVT is often in the form of $f(b) - f(a) = f'(c)(b-a)$, which can be rewritten as $\int_a^b f'(x)dx = f'(c)(b - a)$. This becomes the MVT for integrals:

$$\int_a^b h(x)dx = h(c)(b - a),$$

but we will present a more general version below.

Theorem 6.4 (Mean Value Theorem for Integrals)**.** *If $f \in C[a, b]$, g is integrable on [a,b] and $g(x)$ does not change sign on [a,b], then there exists a number c in (a,b) such that*

$$\int_a^b f(x)g(x)dx = f(c)\int_a^b g(x)dx.$$

6.2 Linear Algebra

6.2.1 Vector/Matrix algebra

The notation for a matrix will be a capital letter, such as A, with its corresponding lower letter (with double subscripts), such as a_{ij} to refer to ith row, jth column entry; that is

$$A = [a_{ij}] = [a_{ij}]_{m\times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

is an $m \times n$ matrix. The *transpose* of A is

$$A^T = [a_{ji}]_{n\times m} = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \vdots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}.$$

The letters towards the end of the alphabet, like x, y, z, usually indicate vectors. **They are considered as column vectors unless otherwise stated.** For a vector x, x_i will denote its ith coordinate. The square brackets $[\cdot]$ is for matrices. Since vectors are a special kind of matrices, we have

- $x = [x_1, x_2, \cdots, x_n]$ is a row vector.
- $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$ is a column vector.
- $x = [x_1, x_2 \cdots, x_n]^T$ is a column vector.
- $x = (x_1, x_2 \cdots, x_n)$ is a column vector. Yes, if we use parenthesis, it is a column vector!

In consideration of saving space, you will see the last two expressions often.

Given $x = (x_1, x_2 \cdots, x_n), y = (y_1, y_2, \cdots, y_n)$, and a scalar $a \in \mathbb{R}$.

$$x + ay = (x_1 + ay_1, x_2 + ay_2, \cdots, x_n + ay_n).$$

Let A be an $m \times n$ matrix and $x = (x_1, x_2 \cdots, x_n)$. There are two equivalent ways to compute Ax (a matrix times a vector). First we write

$$A = [a_{ij}]_{m\times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} = \begin{bmatrix} r_1^T \\ r_2^T \\ \vdots \\ r_m^T \end{bmatrix} = [c_1, c_2, \cdots, c_n],$$

where r_i^T is the ith row of this matrix and c_j is the jth column of this matrix.

1. $Ax = \begin{bmatrix} r_1^T \\ r_2^T \\ \vdots \\ r_m^T \end{bmatrix} x = \begin{bmatrix} r_1^T x \\ r_2^T x \\ \vdots \\ r_m^T x \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + a_{mn}x_n \end{bmatrix}$.

2. $Ax = [c_1, c_2, \cdots, c_n] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \sum_{i=1}^{n} x_i c_i$. Ax is a linear combination of columns of A.

A very important special case is when A is reduced to a $1 \times n$ matrix (a row vector). Given $y = (y_1, y_2, \cdots, y_n)$,

$$y^T x = [y_1, y_2, \cdots, y_n] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \sum_{i=1}^{n} x_i y_i = x^T y = x \cdot y = \langle x, y \rangle. \tag{6.2}$$

The number in (6.2) is called the *inner product or dot product* of x and y.

For two general matrices $A = [a_{ij}]_{m \times n}, B = [b_{ij}]_{n \times p}$, their product AB is $m \times p$, and is defined as

$$(AB)_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{im} b_{mj}.$$

For example, given $y = (y_1, y_2, \cdots, y_n), z = (z_1, z_2, \cdots, z_m)$. y can be viewed as $n \times 1$ matrix and z^T is $1 \times m$, then yz^T is $n \times m$ and

$$yz^T = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} [z_1, z_2, \cdots, z_m] = [y_i z_j]_{n \times m} = \begin{bmatrix} y_1 z_1 & y_1 z_2 & \cdots & y_1 z_n \\ y_2 z_1 & y_2 z_2 & \cdots & y_2 z_n \\ \vdots & \vdots & \vdots & \vdots \\ y_n z_1 & y_n z_2 & \cdots & y_n z_m \end{bmatrix}$$

For more examples, please refer to [3, Section 1.2].

A matrix $U = [u_{ij}]$ is called *upper (lower) triangular* if $u_{ij} = 0$ for $i > j$ ($i < j$). A matrix $D = [d_{ij}]$ is *diagonal* if $d_{ij} = 0$ for $i \neq j$. The *identity* matrix of order n, $I_n = [\delta_{ij}]$, is an $n \times n$ diagonal matrix with entries $\delta_{ii} = 1$. Moreover, the columns of I_n are often denoted e_i, that is $I_n = [e_1, e_2, \cdots, e_n]$.

A square matrix A is called *symmetric* if $A = A^T$. Visually, it appears that the entries of such matrices are symmetric about the diagonal.

6.2.2 Matrix Inversion

An $n \times n$ matrix A is *invertible* if there exists B such that $AB = I_n$ or $BA = I_n$.[1] The matrix B is called the *inverse* of A, denoted by A^{-1}. A matrix is called *singular* if it is not invertible; an invertible matrix is also called *nonsingular*.

Gaussian Elimination is the standard way to find out whether a matrix is invertible, and to compute its inverse when invertible. **If we only need to decide invertibility, we stop at the**

[1] The common definite requires $AB = BA = I_n$, but it can be shown that either $AB = I$ or $BA = I$ is sufficient.

end of forward elimination. If number of pivots equals n, then it's invertible; if number of pivots is less than n, then it's singular. Taking $A = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 7 & 7 \\ 2 & 7 & 9 \end{bmatrix}$ for example, we already know from Example 3.1 that it's invertible because there are 3 pivots. To find its inverse, we want to find a matrix $B = [b_1, b_2, b_3]$ such that $AB = I_3$, which is $[Ab_1, Ab_2, Ab_3] = [e_1, e_2, e_3]$. This is solving three linear systems with the same coefficient matrix, so the row operations can be done in parallel, i.e., perform Gaussian elimination on $[A|e_1, e_2, e_3]$. We recycle all the row operations done in Example 3.1:

$$\left[\begin{array}{ccc|ccc} 1 & 2 & 2 & 1 & 0 & 0 \\ 2 & 7 & 7 & 0 & 1 & 0 \\ 2 & 7 & 9 & 0 & 0 & 1 \end{array}\right] \xrightarrow[\rho 1(-2)+\rho 3]{\rho 1(-2)+\rho 2} \left[\begin{array}{ccc|ccc} 1 & 2 & 2 & 1 & 0 & 0 \\ 0 & 3 & 3 & -2 & 1 & 0 \\ 0 & 3 & 5 & -2 & 0 & 1 \end{array}\right] \xrightarrow{\rho 2(-1)+\rho 3} \left[\begin{array}{ccc|ccc} 1 & 2 & 2 & 1 & 0 & 0 \\ 0 & 3 & 3 & -2 & 1 & 0 \\ 0 & 0 & 2 & 0 & -1 & 1 \end{array}\right]$$

$$\xrightarrow[\rho 2/3]{\rho 3/2} \left[\begin{array}{ccc|ccc} 1 & 2 & 2 & 1 & 0 & 0 \\ 0 & 1 & 1 & -\frac{2}{3} & \frac{1}{3} & 0 \\ 0 & 0 & 1 & 0 & -\frac{1}{2} & \frac{1}{2} \end{array}\right] \xrightarrow[\rho 3(-2)+\rho 1]{\rho 3(-1)+\rho 2} \left[\begin{array}{ccc|ccc} 1 & 2 & 0 & 1 & 1 & -1 \\ 0 & 1 & 0 & -\frac{2}{3} & \frac{5}{6} & -\frac{1}{2} \\ 0 & 0 & 1 & 0 & -\frac{1}{2} & \frac{1}{2} \end{array}\right] \xrightarrow{\rho 2(-2)+\rho 1} \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & \frac{7}{3} & -\frac{2}{3} & 0 \\ 0 & 1 & 0 & -\frac{2}{3} & \frac{5}{6} & -\frac{1}{2} \\ 0 & 0 & 1 & -1 & -\frac{1}{2} & \frac{1}{2} \end{array}\right].$$

Remark 6.5. Cancellation DOES NOT hold for matrix algebra, that is $AB = AC$ DOES NOT imply $B = C$. What is true in matrix multiplication is that we can cancel invertible matrices: if A is invertible and $AB = AC$, then we can apply A^{-1} on both sides to get $B = C$.

If A, B are both invertible, then AB is also invertible and

$$(AB)^{-1} = B^{-1}A^{-1}.$$

6.2.3 Determinant

The determinant is a number associated to a square matrix and can be intuitively understood as a signed area/volume of the parallelepiped associated with the matrix. For our purpose, we need to know

- The best way (for humans or computers) to compute determinant is to use forward elimination. In Example 3.1, the determinant of the original coefficient matrix is the same as the determinant of $\begin{bmatrix} 1 & 2 & 2 \\ 0 & 3 & 3 \\ 0 & 0 & 2 \end{bmatrix}$, which is just the product of the pivots, which is 6. Scaling changes the determinant and row exchange makes the determinant the opposite.
- $\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$.
- Determinant of a 3×3 matrix can be found by row/column expansion. For example, if we expand along first row, then

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}.$$

6.2.4 Linear Independence and span

Given vectors $v_1, v_2, \cdots, v_n$ and let $A = [v_1, v_2, \cdots, v_n]$, the following are equivalent:

1. $v_1, v_2, \cdots, v_n$ are linearly independent;
2. The solution of $Ax = 0$ is unique (only $x = 0$: the trivial solution).
3. rank$A$ = number of pivots after doing forward elimination = n.

If in addition that A is square, then there are 2 additional equivalent conditions:

4. A is invertible.
5. $\det A \neq 0$.

Given a set of vectors $\{v_1, v_2, \cdots, v_m\} \subset \mathbb{R}^n$,

$\text{span}\{v_1, v_2, \cdots, v_m\}$ = all linear combinations of $v_1, \cdots, v_n = \{c_1 v_1 + \cdots + c_n v_n : c_i \in \mathbb{R}\}$

6.2.5 Eigenvalues

Given $A \in \mathbb{R}^{n \times n}$, if $Ax = \lambda x$ for some $x \neq 0$, then x is an eigenvector of A, and λ is the corresponding eigenvalue. For example, we have $\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 3 \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, which means that $[x, 1]^T$ is an eigenvector of $A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$, with the corresponding eigenvalue 3.

To figure out all the eigenvalues and its corresponding eigenvectors of A, we need to first find roots of $\det(A - \lambda I)$, and second solve $(A - \lambda I)x = 0$ for each root λ.

Example 6.6. Find eigenvalues and eigenvectors of $A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$.

$\det \begin{bmatrix} 1-\lambda & 2 \\ 2 & 1-\lambda \end{bmatrix} = (1-\lambda)^2 - 4 = (1-\lambda-2)(1-\lambda+2) = (-1-\lambda)(3-\lambda)$.

$\lambda_1 = -1$, solve $\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} x = 0$, $u_1 = [1, -1]^T$.

$\lambda_2 = 3$, solve $\begin{bmatrix} -2 & 2 \\ 2 & -2 \end{bmatrix} x = 0, u_2 = [1, 1]^T$.

6.2.6 Norm

Let $x \in \mathbb{R}^n$, then its (Euclidean) *norm* is defined as

$$\|x\| := \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2} = \sqrt{x_1^2 + x_2^2 + \cdots x_n^2}. \tag{6.3}$$

Note that $x^T x = \langle x, x \rangle = \|x\|^2$. x is called a unit (norm) vector if $\|x\| = 1$.

Given an arbitrary vector x, $\dfrac{x}{\|x\|}$ is a unit vector that has the same direction as x. This is called normalization of x.

6.3 Complexity

Big O notation is frequently used in mathematics and computer science.

Let $f(x), g(x)$ be both real functions. We write

$$f(x) = O(g(x)) \text{ as } x \to \infty$$

if and only if there exist M and x_0 such that $|f(x)| \leq Mg(x)$ for all $x \geq x_0$.

This notation is used a lot in algorithm analysis. For example, for a polynomial, we have $\sum_{k=1}^{M} a_k x^k = O(x^M)$.

Bibliography

[1] Richard L. Burden, and J. Douglas Faires. *Numerical analysis.* Cengage Learning, 9th (2010).

[2] Anne, Greenbaum, and Timothy P. Chartier. *Numerical methods: design, analysis, and computer implementation of algorithms.* Princeton University Press, 2012.

[3] Ab Mooijaart, Matthijs Warrens, and Eeke van der Burg, *An introduction to matrix algebra with Matlab*, manuscript, 2006

[4] Lloyd N. Trefethen, and David Bau III. *Numerical linear algebra.* Vol. 50. Siam, 1997.