COMP2860 Theoretical Foundations: Linear Algebra

Thomas Ranner

Table of contents

1 Welcome to LA

1.1 Contents of this submodule

- floating point numbers
  - rounding errors
  - top tips for working with floating point numbers
- linear systems of equations
  - motivation
  - Gaussian-elimination
  - practicalities
  - iterative schemes
- transforming matrices
  - what is a basis? and what does it mean?
  - eigenvectors and eigenvalues
    - definitions
    - small matrix solutions
  - large matrix solutions

TODO add recap of matrices terminology - inverse, transpose, coefficients, size

1.2 Motivations

1.3 Learning outcomes

1.4 Programming

2 (Preliminary topic) Floating point number systems

2.1 Finite precision number systems

Computers store numbers with **finite precision**, i.e. using a finite set of bits (binary digits), typically 32 or 64 of them. You met how to store numbers as floating point numbers last year in the module COMP18XX??.

You will recall that many numbers cannot be stored exactly.

- Some numbers cannot be represented precisely using **any** finite set of digits:
  e.g. $\sqrt{2} = 1.14142\ldots$, $\pi = 3.14159\ldots$, etc.
- Some cannot be represented precisely in a given number base:
  e.g. $\frac{1}{9} = 0.111\ldots$ (decimal), $\frac{1}{5} = 0.00110011\ldots$ (binary).
- Others can be represented by a finite number of digits but only using more than are available: e.g. 1.526374856437 cannot be stored exactly using 10 decimal digits.

The inaccuracies inherent in finite precision arithmetic must be modelled in order to understand:

- how the numbers are represented (and the nature of associated limitations);
- the errors in their representation;
- the errors which occur when arithmetic operations are applied to them.

The examples shown here will be in **decimal** by the issues apply to any base, *e.g.* **binary**.

This is important when trying to solve problems with floating point numbers so that we learn how to avoid the key pitfalls.

2.2 Normalised systems

To understand how this works in practice, we introduce an abstract way to think about the practicalities of floating point numbers, but our examples will have smaller numbers of digits.

Any finite precision number can be written using the floating point representation

$$x = \pm 0.b_1 b_2 b_3 \ldots b_{t-1} b_t \times \beta^e.$$

- The digits b_i are integers satisfying $0 \leq b_i \leq \beta - 1$.
- The **mantissa**, $b_1 b_2 b_3 \ldots b_{t-1} b_t$, contains t digits.
- β is the **base** (always a positive integer).
- e is the integer **exponent** and is bounded ($L \leq e \leq U$).

(β, t, L, U) fully defines a finite precision number system.

Normalised finite precision systems will be considered here for which

$$b_1 \neq 0 \quad (0 < b_1 \leq \beta - 1).$$

Examples:

1. In the case $(\beta, t, L, U) = (10, 4, -49, 50)$ (base 10),

$$10000 = .1000 \times 10^5, \quad 22.64 = .2264 \times 10^2, \quad 0.0000567 = .5670 \times 10^{-4}$$

2. In the case $(\beta, t, L, U) = (2, 6, -7, 8)$ (binary),

$$10000 = .100000 \times 2^5, \quad 1011.11 = .101111 \times 2^4,$$

$$0.000011 = .110000 \times 2^{-4}.$$

3. **Zero** is always taken to be a special case e.g., $0 = \pm .00 \ldots 0 \times \beta^0$.

Our familiar floating point numbers can be representing using this format too:

1. The IEEE single precision standard is $(\beta, t, L, U) = (2, 23, -127, 128)$. This is available via `numpy.single`.
2. The IEEE double precision standard is $(\beta, t, L, U) = (2, 52, -1023, 1024)$. This is available via `numpy.double`.

```
import numpy as np

a = np.double(1.1)
print(type(a))
b = np.single(1.2)
print(type(b))
c = np.half(1.3)
print(type(c))
```

```
<class 'numpy.float64'>
<class 'numpy.float32'>
<class 'numpy.float16'>
```

Example 2.1. Consider the number system given by $(\beta, t, L, U) = (10, 2, -1, 2)$ which gives

$$x = \pm .b_1 b_2 \times 10^e \text{ where } -1 \leq e \leq 2.$$

a. How many numbers can be represented by this normalised system?

- the sign can be positive or negative
- b_1 can take on the values 1 to 9 (9 options)
- b_2 can take on the values 0 to 9 (10 options)
- e can take on the values $-1, 0, 1, 2$ (4 options)

Overall this gives us:

$$2 \times 9 \times 10 \times 4 \text{ options } = 720 \text{ options.}$$

b. What are the two largest positive numbers in this system?

The largest value uses + as a sign, $b_1 = 9$, $b_2 = 9$ and $e = 2$ which gives

$$+0.99 \times 10^2 = 99.$$

The second largest value uses + as a sign, $b_1 = 9$, $b_2 = 8$ and $e = 2$ which gives

$$+0.98 \times 10^2 = 98.$$

c. What are the two smallest positive numbers?

The smallest positive number has + sign, $b_1 = 1$, $b_2 = 0$ and $e = -1$ which gives

$$+0.10 \times 10^{-1} = 0.01.$$

The second smallest positive number has + sign, $b_1 = 1$, $b_2 = 1$ and $e = -1$ which gives

$$+0.11 \times 10^{-1} = 0.011.$$

d. What is the smallest possible difference between two numbers in this system?

The smallest different will be between numbers of the form $+0.10\ times 10^{-1}$ and $+0.11 \times 10^{-1}$ which gives

$$0.11 \times 10^{-1} - 0.10 \times 10^{-1} = 0.011 - 0.010 = 0.001.$$

Alternatively, we can brute force search for this:

```
The minimum difference min_diff=0.0010
at x=+0.57 x 10^{-1} y=+0.58 x 10^{-1}.
```

Exercise 2.1. Consider the number system given by $(\beta, t, L, U) = (10, 3, -3, 3)$ which gives

$$x = \pm .b_1 b_2 b_3 \times 10^e \text{ where } -3 \leq e \leq 3.$$

a. How many numbers can be represented by this normalised system?

b. What are the two largest positive numbers in this system?

c. What are the two smallest positive numbers?

d. What is the smallest possible difference between two numbers in this system?

e. What is the smallest possible difference in this system, x and y, for which $x < 100 < y$?

Example 2.2 (What about in python)**.** We find that even with double-precision floating point numbers, we see sum funniness when working with decimals:

```
a = np.double(0.0)

for _ in range(10):
    a = a + np.double(0.1)
    print(a)

print("Is a = 1?", a == 1.0)
```

```
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
Is a = 1? False
```

Why is this output not a surprise?

We also see that even adding up numbers can have different results depending on what order we add them:

```
x = np.double(1e30)
y = np.double(-1e30)
z = np.double(1.0)

print(f"{(x + y) + z=:.16f}")
```

```
(x + y) + z=1.0000000000000000
```

```
print(f"{x + (y + z)=:.16f}")
```

```
x + (y + z)=0.0000000000000000
```

2.3 Errors and machine precision

From now on $fl(x)$ will be used to represent the (approximate) stored value of x. The error in this representation can be expressed in two ways.

$$\text{Absolute error} = |fl(x) - x|$$
$$\text{Relative error} = \frac{|fl(x) - x|}{|x|}.$$

The number $fl(x)$ is said to approximate x to t **significant digits** (or figures) if t is the largest non-negative integer for which

$$\text{Relative error} < 0.5 \times \beta^{1-t}.$$

It can be proved that if the relative error is equal to β^{-d} then $fl(x)$ has d correct significant digits.

In the number system given by (β, t, L, U), the nearest (larger) representable number to $x = 0.b_1b_2b_3 \ldots b_{t-1}b_t \times \beta^e$ is

$$\tilde{x} = x + .\underbrace{000\ldots01}_{t \text{ digits}} \times \beta^e = x + \beta^{e-t}$$

Any number $y \in (x, \tilde{x})$ is stored as either x or $\tilde{x}$ by **rounding** to the nearest representable number, so

- the largest possible error is $\frac{1}{2}\beta^{e-t}$,
- which means that $|y - fl(y)| \leq \frac{1}{2}\beta^{e-t}$.

It follow from $y > x \geq .100\ldots00 \times \beta^e = \beta^{e-1}$ that

$$\frac{|y - fl(y)|}{|y|} < \frac{1}{2}\frac{\beta^{e-t}}{\beta^{e-1}} = \frac{1}{2}\beta^{1-t},$$

and this provides a bound on the **relative error**: for any y

$$\frac{|y - fl(y)|}{|y|} < \frac{1}{2}\beta^{1-t}.$$

The last term is known as **machine precision** or **unit roundoff** and is often called *eps*. This is obtained in Python with

```
np.finfo(np.double).eps
```

```
np.float64(2.220446049250313e-16)
```

Example 2.3.

1. The number system $(\beta, t, L, U) = (10, 2, -1, 2)$ gives

$$eps = \frac{1}{2}\beta^{1-t} = \frac{1}{2}10^{1-2} = 0.05.$$

2. The number system $(\beta, t, L, U) = (10, 3, -3, 3)$ gives

$$eps = \frac{1}{2}\beta^{1-t} = \frac{1}{2}10^{1-3} = 0.005.$$

3. The number system $(\beta, t, L, U) = (10, 7, 2, 10)$ gives

$$eps = \frac{1}{2}\beta^{1-t} = \frac{1}{2}10^{1-7} = 0.000005.$$

4. For some common types in python, we see the following values:

```
for dtype in [np.half, np.single, np.double]:
    print(dtype.__name__, np.finfo(dtype).eps)
```

```
float16 0.000977
float32 1.1920929e-07
float64 2.220446049250313e-16
```

Machine precision epsilon (*eps*) gives us an upper bound for the error in the representation of a floating point number in a particular system. We note that this is different to the smallest possible numbers that we are able to store!

```
eps = np.finfo(np.double).eps
print(f"{eps=}")

smaller = eps
for i in range(5):
    smaller = smaller / 10.0
    print(f"{smaller=}")
```

```
eps=np.float64(2.220446049250313e-16)
smaller=np.float64(2.2204460492503132e-17)
smaller=np.float64(2.220446049250313e-18)
smaller=np.float64(2.220446049250313e-19)
smaller=np.float64(2.2204460492503132e-20)
smaller=np.float64(2.2204460492503133e-21)
```

Arithmetic operations are usually carried out as though infinite precision is available, after which the result is rounded to the nearest representable number.

This means that arithmetic cannot be completely trusted
e.g. $x + y = ?$,

and the usual rules don't necessarily apply
e.g. $x + (y + z) = (x + y) + z$?

Example 2.4. Consider the number system $(\beta, t, L, U) = (10, 2, -1, 2)$ and take

$$x = .10 \times 10^2, \quad y = .49 \times 10^0, \quad z = .51 \times 10^0.$$

- In exact arithmetic $x + y = 10 + 0.49 = 10.49$ and $x + z = 10 + 0.51 = 10.51$.
- In this number system rounding gives

$$fl(x + y) = .10 \times 10^2 = x, \qquad fl(x + z) = .11 \times 10^2 \neq x.$$

(Note that $\frac{y}{x} < eps$ but $\frac{z}{x} > eps$.)

Evaluate the following expression in this number system.

$$x + (y + y), \quad (x + y) + y, \quad x + (z + z), \quad (x + z) + z.$$

(Also note the benefits of adding the *smallest* terms first!)

Example 2.5 (Computing derivatives with floating point numbers)**.** Suppose we want to compute the derivative of $f(x) = x^3$ at $x = 1$ using the definition of limits and floating point numbers:

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}.$$

We know that $f'(x) = 3x^2$ so $f'(1) = 3$. We hope that using floating point numbers gives something similar:

```
def f(x):
    return x**3


x0 = np.double(1.0)

print("Delta_x   Approx  Abs Error")

for j in range(20):
    Delta_x = 10 ** (-j)

    deriv_approx = (f(x0 + Delta_x) - f(x0)) / Delta_x
    abs_error = abs(3.0 - deriv_approx)

    print(f" {Delta_x:.1e}   {deriv_approx:.4f}  {abs_error:.4e}")
```

```
Delta_x   Approx  Abs Error
 1.0e+00   7.0000  4.0000e+00
 1.0e-01   3.3100  3.1000e-01
 1.0e-02   3.0301  3.0100e-02
 1.0e-03   3.0030  3.0010e-03
 1.0e-04   3.0003  3.0001e-04
 1.0e-05   3.0000  3.0000e-05
 1.0e-06   3.0000  2.9998e-06
 1.0e-07   3.0000  3.0151e-07
 1.0e-08   3.0000  3.9720e-09
```

```
1.0e-09   3.0000  2.4822e-07
1.0e-10   3.0000  2.4822e-07
1.0e-11   3.0000  2.4822e-07
1.0e-12   3.0003  2.6670e-04
1.0e-13   2.9976  2.3978e-03
1.0e-14   2.9976  2.3978e-03
1.0e-15   3.3307  3.3067e-01
1.0e-16   0.0000  3.0000e+00
1.0e-17   0.0000  3.0000e+00
1.0e-18   0.0000  3.0000e+00
1.0e-19   0.0000  3.0000e+00
```

We see that if `Delta_x` is not too small, we do an okay job. But if `Delta_x` is too small we start to have problems!

Exercise 2.2 (More examples)**.**

1. Verify that a similar problem arises for the numbers

$$x = .85 \times 10^0, \quad y = .3 \times 10^{-2}, \quad z = .6 \times 10^{-2},$$

   in the system $(\beta, t, L, U) = (10, 2, -3, 3)$.

2. Given the number system $(\beta, t, L, U) = (10, 3, -3, 3)$ and $x = .100 \times 10^3$, find nonzero numbers y and z from this system for which $fl(x + y) = x$ and $fl(x + z) > x$.

It is sometimes helpful to think of another machine precision epsilon in other way: **Machine precision epsilon** is the smallest positive number *eps* such that $1 + eps > 1$, i.e. it is half the difference between 1 and the next largest representable number.

Examples:

1. For the number system $(\beta, t, L, U) = (10, 2, -1, 2)$,

$$\begin{array}{rll} & .11 \times 10^1 & \leftarrow \text{next number} \\ - & .10 \times 10^1 & \leftarrow 1 \\ \hline & .01 \times 10^1 & \leftarrow 0.1 \end{array}$$

   so $eps = \frac{1}{2}(0.1) = 0.05$.

2. Verify that this approaches gives the previously calculated value for *eps* in the number system given by $(\beta, t, L, U) = (10, 3, -3, 3)$.

2.4 Other "Features" of finite precision

When working with floating point numbers there are other things we need to worry about too!

Overflow the number is too large to be represented, e.g. multiply the largest representable number by 10. This gives `inf` (infinity) with `numpy.double`s and is usually "fatal".

Underflow the number is too small to be represented, e.g. divide the smallest representable number by 10. This gives 0 and may not be immediately obvious.

Divide by zero gives a result of `inf`, but $\frac{0}{0}$ gives `nan` (not a number)

Divide by `inf` gives 0.0 with no warning

2.5 Is this all academic?

No! There are many examples of major software errors that have occurred due to programmers not understanding the issues associated with computer arithmetic…

- In February 1991, a basic rounding error within software for the US Patriot missile system caused it to fail, contributing to the loss of 28 lives.
- In June 1996, the European Space Agency's Ariane Rocket exploded shortly after take-off: the error was due to failing to handle overflow correctly.
- In October 2020, a driverless car drove straight into a wall due to faulty handling of a floating point error.

2.6 Summary

- There is inaccuracy in almost all computer arithmetic.
- Care must be taken to minimise its effects, for example:
  - add the smallest terms in an expression first;
  - avoid taking the difference of two very similar terms;
  - even checking whether $a = b$ is dangerous!
- The usual mathematical rules no longer apply.
- There is no point in trying to compute a solution to a problem to a greater accuracy than can be stored by the computer.

2.6.1 Further reading

- Wikipedia: Floating-point arithmetic
- David Goldberg, What every computer scientist should know about floating-point arithmetic, ACM Computing Surveys, Volume 23, Issue 1, March 1991.
- John D Cook, Floating point error is the least of my worries, *online*, November 2011.

3 Introduction to systems of linear equations

3.1 Definitions of matrices and vectors

There are two important objects we wil work with that were defined in your first year Theoretical Foundations module.

Definition 3.1. A *matrix* is a rectangular array of numbers called *entries* or *elements* of the matrix. A matrix with m rows and n columns is called an $m \times n$ matrix or m-by-n matrix. We may additionally say that the matrix is of order $m \times n$. If $m = n$, then we say that the matrix is *square*.

Example 3.1. A is a 4×4 matrix and B is a 3×4 matrix:

$$A = \begin{pmatrix} 10 & 1 & 0 & 9 \\ 12.4 & 6 & 1 & 0 \\ 1 & 3.14 & 1 & 0 \end{pmatrix} \quad B = \begin{pmatrix} 0 & 6 & 3 & 1 \\ 1 & 4 & 1 & 0 \\ 7 & 0 & 10 & 20 \end{pmatrix} \quad C = \begin{pmatrix} 4 & 1 & 8 & -1 \\ 1.5 & 1 & 3 & 4 \\ 6 & -4 & 2 & 8 \end{pmatrix}$$

Exercise 3.1.

1. Compute, if defined, $A + B$, $B + C$.
2. Compute, if defined, AB, BA, BC (here, by writing matrices next to each other we mean the matrix product).

TODO add example of 2d rotation matrix.

When considering systems of linear equations the entries of the matrix will always be real numbers.

Definition 3.2. A *column vector*, often just called a *vector*, is a matrix with a single column. A matrix with a single row is a *row vector*. The entries of a vector are called *components*. A vector with n-rows is called an n-vector.

Example 3.2. $\vec{a}$ is a row vector, $\vec{b}$ and $\vec{c}$ are (column) vectors.

$$\vec{a} = \begin{pmatrix} 0 & 1 & 7 \end{pmatrix} \quad \vec{b} = \begin{pmatrix} 0 \\ 1 \\ 3.1 \\ 7 \end{pmatrix} \quad \vec{c} = \begin{pmatrix} 4 \\ 6 \\ -4 \\ 0 \end{pmatrix}.$$

Exercise 3.2.

1. Compute, if defined, $\vec{b} + \vec{c}$, $0.25\vec{c}$.
2. What is the meaning of $\vec{b}^T\vec{c}$? (here, we are interpreting the vectors as matrices).
3. Compute, if defined, $B\vec{b}$.

3.2 Definition of systems of linear equations

Given an $n \times n$ matrix A and an n-vector $\vec{b}$, find the n-vector $\vec{x}$ which satisfies:

$$A\vec{x} = \vec{b}. \tag{3.1}$$

We can also write (3.1) as a system of linear equations:

$$\begin{aligned}
\text{Equation 1:} \quad & a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1 \\
\text{Equation 2:} \quad & a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n = b_2 \\
& \vdots \\
\text{Equation i:} \quad & a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3 + \cdots + a_{in}x_n = b_i \\
& \vdots \\
\text{Equation n:} \quad & a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n = b_n.
\end{aligned}$$

Notes:

- The values a_{ij} are known as **coefficients**.
- The **right hand side** values b_i are known and are given to you as part of the problem.
- $x_1, x_2, x_3, \ldots, x_n$ are **not** known and are what you need to find to solve the problem.

Many computational algorithms require the solution of linear equations, e.g. in fields such as

- Scientific computation;
- Network design and optimisation;
- Graphics and visualisation;
- Machine learning.

TODO precise examples

Typically these systems are *very* large ($n \approx 10^9$).

It is therefore important that this problem can be solved

- accurately: we are allowed to make small errors but not big errors;
- efficiently: we need to find the answer quickly;

- reliably: we need to know that our algorithm will give us an answer that we are happy with.

Example 3.3 (Temperature in a sealed room)**.** Suppose we wish to estimate the temperature distribution inside an object:

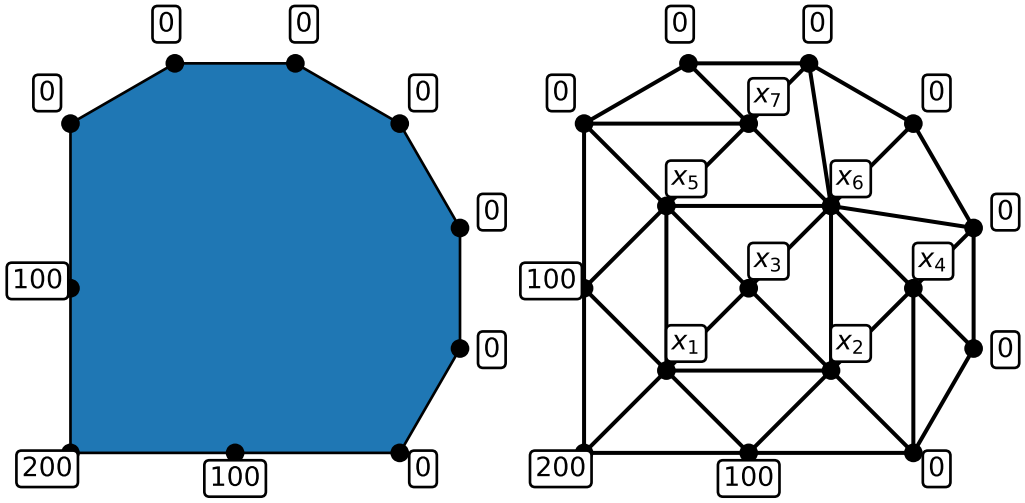


Figure 3.1: Image showing temperature sample points and relations in a room.

We can place a network of points inside the object and use the following model: the temperature at each interior point is the average of its neighbours.

This example leads to the system:

$$\begin{pmatrix} 1 & -1/6 & -1/6 & 0 & -1/6 & 0 & 0 \\ -1/6 & 1 & -1/6 & -1/6 & 0 & -1/6 & 0 \\ -1/4 & -1/4 & 1 & 0 & -1/4 & -1/4 & 0 \\ 0 & -1/5 & 0 & 1 & 0 & -1/5 & 0 \\ -1/6 & 0 & -1/6 & 0 & 1 & -1/6 & -1/6 \\ 0 & -1/8 & -1/8 & -1/8 & -1/8 & 1 & -1/8 \\ 0 & 0 & 0 & 0 & -1/5 & -1/5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} = \begin{pmatrix} 400/6 \\ 100/6 \\ 0 \\ 0 \\ 100/6 \\ 0 \\ 0 \end{pmatrix}.$$

Example 3.4 (Traffic network)**.** Suppose we wish to monitor the flow of traffic in a city centre:

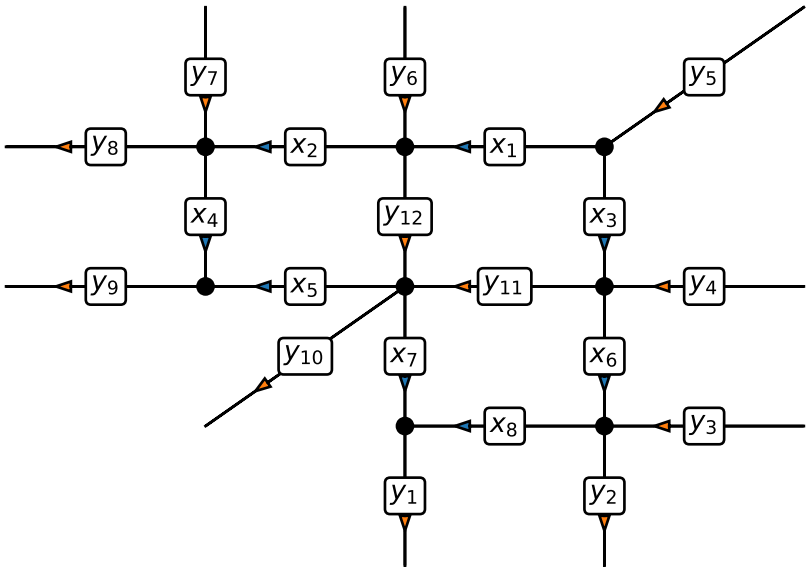


Figure 3.2: Example network showing traffic flow in a city

As the above example shows, it is not necessary to monitor at every single road. If we know all of the y values we can calculate the x values!

This example leads to the system:

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} = \begin{pmatrix} y_5 \\ y_{12} - y_6 \\ y_8 - y_7 \\ y_{11} - y_4 \\ y_{11} + y_{12} - y_{10} \\ y_9 \\ y_2 - y_3 \\ y_1 \end{pmatrix}.$$

3.3 Can we do it?

Our first question might be - is it possible to solve (3.1)?

We know a few simple cases where we can answer this question very quickly:

1. If $A = I_n$, the $n \times n$ *identity matrix*, then we *can* solve this problem:

$$\vec{x} = \vec{b}.$$

2. If $A = O$, the $n \times n$ *zero matrix*, and $\vec{b} \neq \vec{0}$, the zero vector, then we cannot solve this problem:

$$O\vec{x} = \vec{0} \neq \vec{b} \quad \text{for any vector} \quad \vec{x}.$$

3. If A is *invertible*, with inverse A^{-1}, then we *can* solve this problem:

$$\vec{x} = A^{-1}\vec{b}.$$

   But, in general, this is a *very bad* idea and we will see algorithms that are more efficient than finding the inverse of A.

Remark 3.1. One way to solve a system of linear equations is to compute the inverse of A, A^{-1}, directly, then the solution is found through matrix multiplication: $\vec{x} = A^{-1}\vec{b}$. This turns out to be an inefficient approach and we can do better with specialised algorithms.

```
Trying different approaches for problem size 100
Approach 1 - inverting the matrix and applying the inverse. Time = 0.0002560615539550781
Approach 2 - solving the system of linear equations. Time = 0.00013875961303710938
Approach 2 is faster by a factor of  1.8453608247422681
```

```
Trying different approaches for problem size 10000
Approach 1 - inverting the matrix and applying the inverse. Time = 31.876757621765137
Approach 2 - solving the system of linear equations. Time = 8.109544515609741
Approach 2 is faster by a factor of  3.9307704101514984
```

There are tools to help us answer when a matrix is invertible which arise naturally when thinking about what $A\vec{x} = \vec{b}$ means! We have to go back to the basic operations on vectors.

There are two fundamental operations you can do on vectors: addition and scalar multiplication. Consider the vectors:

$$\vec{a} = \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix}, \quad \vec{c} = \begin{pmatrix} 4 \\ 2 \\ 6 \end{pmatrix}.$$

Then, we can easily compute the following *linear combinations*

$$\vec{a} + \vec{b} = \begin{pmatrix} 3 \\ 3 \\ 6 \end{pmatrix} \tag{3.2}$$

$$\vec{c} - 2\vec{a} = \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix} \tag{3.3}$$

$$\vec{a} + 2\vec{b} + 2\vec{c} = \begin{pmatrix} 12 \\ 9 \\ 22 \end{pmatrix}. \tag{3.4}$$

Now if we write A for the 3×3-matrix whose columns are $\vec{a}, \vec{b}, \vec{c}$:

$$A = \begin{pmatrix} \vec{a} & \vec{b} & \vec{c} \end{pmatrix} = \begin{pmatrix} 2 & 1 & 4 \\ 1 & 2 & 2 \\ 2 & 4 & 6 \end{pmatrix},$$

then the three equations (3.2), (3.3), (3.4), can be written as

$$\vec{a} + \vec{b} = 1\vec{a} + 1\vec{b} + 0\vec{c} = A \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix},$$

$$\vec{c} - 2\vec{a} = -2\vec{a} + 0\vec{b} - 2\vec{c} = A \begin{pmatrix} -2 \\ 0 \\ 1 \end{pmatrix},$$

$$\vec{a} + 2\vec{b} + 2\vec{c} = 1\vec{a} + 2\vec{b} + 2\vec{c} = A \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix}.$$

In other words,

> We can write any linear combination of vectors as a matrix-vector multiply,

or if we reverse the process,

> We can write matrix-vector multiplication as a linear combination of the columns of the matrix.

This rephrasing means, solving the system $A\vec{x} = \vec{b}$ is equivalent to finding a linear combination of the columns of A which is equal to b. So, our question about whether we can solve (3.1), can also be rephrased as: does there exist a linear combination of the columns of A which is equal to $\vec{b}$? We will next write this condition mathematically using the concept of *span*.

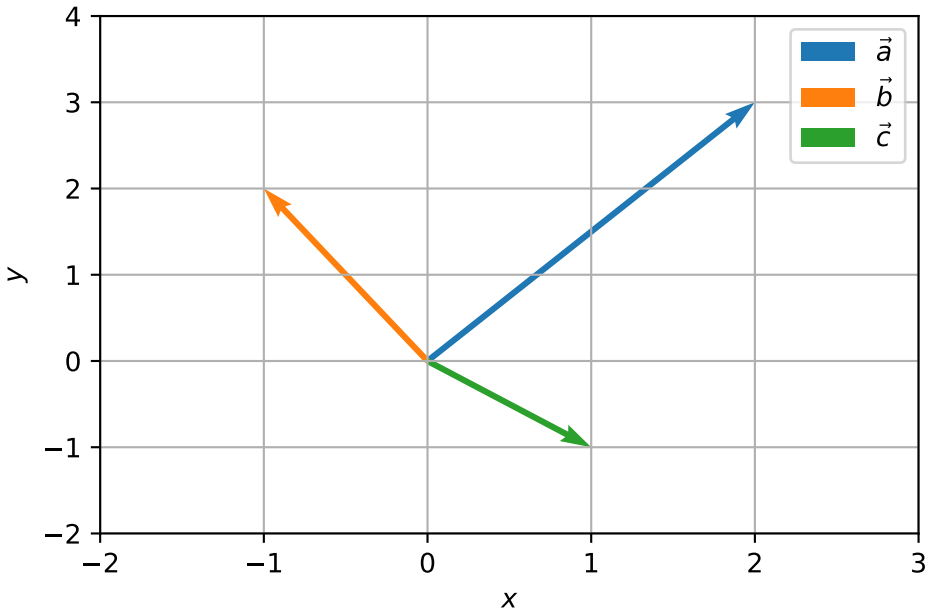
3.3.1 The span of a set of vectors

Definition 3.3. Given a set of vectors of the same size, $S = \{\vec{v}_1, \dots, \vec{v}_k\}$, we say the *span* of S is the set of all vectors which are linear combinations of vectors in S:

$$\text{span}(S) = \left\{ \sum_{i=1}^{k} x_i \vec{v}_i : x_i \in \mathbb{R} \text{ for } i = 1, \dots, k \right\}. \tag{3.5}$$

Example 3.5. Consider three new vectors

$$\vec{a} = \begin{pmatrix} 2 \\ 3 \end{pmatrix} \quad \vec{b} = \begin{pmatrix} -1 \\ 2 \end{pmatrix} \quad \vec{c} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$



1. Let $S = \{\vec{a}\}$, then $\text{span}(S) = \{x\vec{a} : x \in \mathbb{R}\}$. Geometrically, we can think of the span of a single vector to be an infinite straight line which passes through the origin and $\vec{a}$.
2. Let $S = \{\vec{a}, \vec{b}\}$, then $\text{span}(S) = \mathbb{R}^2$. To see this is true, we first see that $\text{span}(S)$ is contained in $\mathbb{R}^2$ since any 2-vectors added together and the scalar multiplication of a 2-vector also form a 2-vector. For the opposite inclusion, consider an arbitrary point

$\vec{y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \in \mathbb{R}^2$ then

$$\frac{2y_1 + y_2}{7}\vec{a} + \frac{-3y_1 + 2y_2}{7}\vec{b} = \frac{2y_1 + y_2}{7}\begin{pmatrix} 2 \\ 3 \end{pmatrix} + \frac{-3y_1 + 2y_2}{7}\begin{pmatrix} -1 \\ 2 \end{pmatrix}$$
$$= \begin{pmatrix} \frac{4y_1+2y_2}{7} + \frac{3y_1-2y_2}{7} \\ \frac{6y_1+3y_2}{7} + \frac{-6y_1+4y_2}{7} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \vec{y}. \tag{3.6}$$

This calculation shows, that we can always form a linear combination of $\vec{a}$ and $\vec{b}$ which results in $\vec{y}$.

3. Let $S = \{\vec{a}, \vec{b}, \vec{c}\}$, then $\text{span}(S) = \mathbb{R}^2$. Since $\vec{c} \in \text{span}(\{\vec{a}, \vec{b}\})$, any linear combination of $\vec{a}, \vec{b}, \vec{c}$ has an equivalent combination of just $\vec{a}$ and $\vec{b}$. In formulae, we can see that by applying the formula for example 2, we have

$$\vec{c} = \frac{1}{7}\vec{a} - \frac{5}{7}\vec{b}.$$

So we have, if $\vec{y} \in \text{span}(S)$, then

$$\vec{y} = x_1\vec{a} + x_2\vec{b} + x_3\vec{c} \Rightarrow \vec{y} = (x_1 + \frac{1}{7}x_3)\vec{a} + (x_2 - \frac{5}{7}x_3)\vec{b},$$

so $\vec{y} \in \text{span}(\{\vec{a}, \vec{b}\})$. Conversely, if $\vec{y} \in \text{span}(\{\vec{a}, \vec{b}\})$, then

$$\vec{y} = x_1\vec{a} + x_2\vec{b} \Rightarrow \vec{y} = x_1\vec{a} + x_2\vec{b} + 0\vec{c}.$$

So the span of $S = \text{span}(\{\vec{a}, \vec{b}\}) = \mathbb{R}^2$. Notice that we this final linear combination of $\vec{a}, \vec{b}$ and $\vec{c}$ to form $\vec{y}$ is not unique.

So our first statement is that (3.1) has a solution if $\vec{b}$ is in the span of the columns of A. However, as we saw with Example 3.5, Part 3, we are not guaranteed that the linear combination is unique! For this we need a further condition.

3.3.2 Linear independence

Definition 3.4. Given a set of vectors of the same size, $S = \{\vec{v}_1, \ldots, \vec{v}_k\}$, we say that S is *linearly dependent*, if there exists numbers $x_1, x_2, \ldots x_k$, not all zero, such that

$$\sum_{i=1}^{k} x_i\vec{v}_i = \vec{0}.$$

The set S is *linearly independent* if it is not linearly dependent.

Exercise 3.3. Can you write the definition of a linearly independent set of vectors explicitly?

Example 3.6. Continuing from Example 3.5.

1. Let $S = \{\vec{a}, \vec{b}\}$, then S is linearly independent. Indeed, let x_1, x_2 be real numbers such that

$$x_1\vec{a} + x_2\vec{b} = \vec{0},$$

then,

$$2x_1 - x_2 = 0 \qquad\qquad 3x_1 + 2x_2 \qquad\qquad = 0$$

The first equation says that $x_2 = 2x_1$, which when substituted into the second equation gives $3x_1 + 4x_1 = 7x_1 = 0$. Together this implies that $x_1 = x_2 = 0$. Put simply this means that if we do have a linear combination of $\vec{a}$ and $\vec{b}$ which is equal zero, then the corresponding scalar multiples are all zero.

2. Let $S = \{\vec{a}, \vec{b}, \vec{c}\}$, then S is linearly dependent. We have previously seen that:

$$\vec{c} = \frac{1}{7}\vec{a} - \frac{5}{7}\vec{b},$$

which we can rearrange to say that

$$\frac{1}{7}\vec{a} - \frac{5}{7}\vec{b} - \vec{c} = \vec{0}.$$

We see that the definition of linear dependence is satisfied for $x_1 = \frac{1}{7}, x_2 = -\frac{5}{7}, x_3 = -1$ which are all nonzero.

So linear independence removes the multiplicity (or non-uniqueness) in how we form linear combinations! This leads us to the final definition of this section.

3.3.3 When vectors form a basis

Definition 3.5 (Definition)**.** We say that a set of n-vectors S is a *basis* of a set of n-vectors V if the span of S is V and S is linearly independent.

Example 3.7.

1. From Example 3.5, we have that $S = \{\vec{a}, \vec{b}\}$ is a basis of $\mathbb{R}^2$.
2. Another (perhaps simpler) basis of $\mathbb{R}^2$ are the coordinate axes:

$$\vec{e}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{and} \quad \vec{e}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

3. When we look at eigenvectors and eigenvalues we will see that there are other convienient bases (plural of basis) to work with.

We phrase the idea that the existence and uniqueness of linear combinations together depend on the underlying set being a basis mathematically in the following Theorem:

Theorem 3.1. *Let S be a basis of V. Then any vector in V can be written* uniquely *as a linear combination of entries in S.*

Example 3.8.

1. From the main examples in this section, we have that $S = \{\vec{a}, \vec{b}\}$ is a basis of $\mathbb{R}^2$ and we already know the formula for how to write $\vec{y}$ as a unique combination of $\vec{a}$ and $\vec{b}$: it's given in (3.6).
2. For the simpler example of the coordinate axes:

$$\vec{e}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{and} \quad \vec{e}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

we have that for any $\vec{y} = (y_1/y_2) \in \mathbb{R}^2$

$$\vec{y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = y_1 \begin{pmatrix} 1 \\ 0 \end{pmatrix} + y_2 \begin{pmatrix} 0 \\ 1 \end{pmatrix} = y_1\vec{e}_1 + y_2\vec{e}_2.$$

Proof of Theorem 1. Let $\vec{y}$ be a vector in V and label $S = \{\vec{v}_1, \dots \vec{v}_k\}$. Since S forms a basis of V, $\vec{y} \in \text{span}(S)$ so there exists numbers $x_1, \dots, x_k$ such that

$$\vec{y} = \sum_{i=1}^{k} x_i\vec{v}_i. \tag{3.7}$$

Suppose that there exists another set of number $z_1, \dots z_k$ such that

$$\vec{y} = \sum_{i=1}^{k} z_i\vec{v}_i. \tag{3.8}$$

Taking the difference of (3.7) and (3.8), we see that

$$\vec{0} = \sum_{i=1}^{k} (x_i - z_i)\vec{v}_i. \tag{3.9}$$

Since S is linearly independent, this implies $x_i = z_i$ for $i = 1, \dots, k$, and we have shown that there is only one linear combination of the vectors $\{\vec{v}_i\}$ to form $\vec{y}$. □

There is a theorem that says that the number of vectors in any basis of a given 'nice' set of vectors V is the same but is beyond the scope of this module!

Theorem 3.2. *Let A be a $n \times n$-matrix. If the columns of A form a basis of $\mathbb{R}^n$, then there exists a unique n-vector $\vec{x}$ which satisfies $A\vec{x} = \vec{b}$.*

We do not give full details of the proof here since all the key ideas are already given above.

3.4 Special types of matrices

The general matrix A before the examples is known as a **full** matrix: any of its components a_{ij} might be nonzero.

Almost always the problem being solved leads to a matrix with a particular structure of entries: Some entries may be known to be zero. If this is the case then it is often possible to use this knowledge to improve the efficiency of the algorithm (in terms of both speed and/or storage).

Example 3.9 (Triangular matrix)**.** One common (and important) structure takes the form

$$\begin{pmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ a_{31} & a_{32} & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}.$$

- A is a **lower triangular** matrix. Every entry above the leading diagonal is zero:

$$a_{ij} = 0 \quad \text{for} \quad j > i.$$

- The *transpose* of this matrix is an **upper triangular** matrix and can be treated in a very similar manner.

Example 3.10 (Sparse matrices)**. Sparse matrices** are extremely common in any application which relies on some form of *graph* structure (see both the temperature (Example 3.3) and traffic network examples (Example 3.4).

- The a_{ij} typically represents some form of "communication" between vertices i and j of the graph, so the element is only nonzero if the vertices are connected.
- There is no generic pattern for these entries, though there is usually one that is specific to the problem solved.

- Usually $a_{ii} \neq 0$ - the diagonal is nonzero.
- A "large" portion of the matrix is zero.
  - A full $n \times n$ matrix has n^2 nonzero entries.
  - A sparse $n \times n$ has αn nonzero entries, where $\alpha \ll n$.
- Many special techniques exist for handling sparse matrices, some of which can be used automatically within Python (`scipy.sparse` documentation)

What is the significance of these special examples?

- In the next section we will discuss a general numerical algorithm for the solution of linear systems of equations.
- This will involve **reducing** the problem to one involving a **triangular matrix** which, as we show below, is relatively easy to solve.
- In subsequent lectures, we will see that, for *sparse* matrix systems, alternative solution techniques are available.

3.5 Further reading

- Wikipedia: Systems of linear equations (includes a nice geometric picture of what a system of linear equations means).
- Maths is fun: Systems of linear equations (very basic!)
- Gregory Gundersen Why shouldn't I invert that matrix?

4 Direct solvers for systems of linear equtaions

4.1 Reminder of the problem

This is the first method we will use to solve systems of linear equations. We will see that by using the approach of Gaussian elimination (and variations of that method) we can solve any system of linear equations that have a solution.

Recall the problem is to solve a set of n **linear** equations for n unknown values x_j, for $j = 1, 2, \ldots, n$.

Notation:

$$\begin{aligned}
\text{Equation } 1: &\quad a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1 \\
\text{Equation } 2: &\quad a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n = b_2 \\
&\vdots \\
\text{Equation } i: &\quad a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3 + \cdots + a_{in}x_n = b_i \\
&\vdots \\
\text{Equation } n: &\quad a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n = b_n.
\end{aligned}$$

We can also write the system of linear equations in *general matrix-vector* form:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}.$$

Recall the $n \times n$ matrix A represents the coefficients that multiply the unknowns in each equation (row), while the n-vector $\vec{b}$ represents the right-hand-side values.

Our strategy will be to reduce the system to a triangular system of matrices which is then easy to solve!

4.2 Elementary row operations

Consider equation p of the above system:

$$a_{p1}x_1 + a_{p2}x_2 + a_{p3}x_3 + \cdots + a_{pn}x_n = b_p,$$

and equation q:

$$a_{q1}x_1 + a_{q2}x_2 + a_{q3}x_3 + \cdots + a_{qn}x_n = b_q.$$

Note three things…

- The order in which we choose to write the n equations is irrelevant
- We can multiply any equation by an arbitrary real number ($k \neq 0$ say):

$$ka_{p1}x_1 + ka_{p2}x_2 + ka_{p3}x_3 + \cdots + ka_{pn}x_n = kb_p.$$

- We can add any two equations:

$$ka_{p1}x_1 + ka_{p2}x_2 + ka_{p3}x_3 + \cdots + ka_{pn}x_n = kb_p$$

added to

$$a_{q1}x_1 + a_{q2}x_2 + a_{q3}x_3 + \cdots + a_{qn}x_n = b_q$$

yields

$$(ka_{p1} + a_{q1})x_1 + (ka_{p2} + a_{q2})x_2 + \cdots + (ka_{pn} + a_{qn})x_n = kb_p + b_q.$$

Example 4.1. Consider the system

$$2x_1 + 3x_2 = 4 \tag{4.1}$$
$$-3x_1 + 2x_2 = 7. \tag{4.2}$$

Then we have:

$$\begin{aligned} 4 \times (4.1) \to \quad & 8x_1 + 12x_2 = 16 \\ -15. \times (4.2) \to \quad & 4.5x_2 - 3x_2 = -10.5 \\ (4.1) + (4.2) \to \quad & -x_1 + 5x_2 = 11 \\ (4.2) + 1.5 \times (4.1) \to \quad & 0 + 6.5x_2 = 13. \end{aligned}$$

Exercise 4.1. Consider the system

$$x_1 + 2x_2 = 1 \tag{4.3}$$
$$4x_1 + x_2 = -3. \tag{4.4}$$

Work out the result of these elementary row operations:

$$\begin{aligned} 2 \times (4.3) \to \\ 0.25 \times (4.4) \to \\ (4.4) + (-1) \times (4.3) \to \\ (4.4) + (-4) \times (4.3) \to \end{aligned}$$

For a system written in matrix form our three observations mean the following:

- We can swap any two rows of the matrix (and corresponding right-hand side entries). For example:

$$\begin{pmatrix} 2 & 3 \\ -3 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 7 \end{pmatrix} \Rightarrow \begin{pmatrix} -3 & 2 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 7 \\ 4 \end{pmatrix}$$

- We can multiply any row of the matrix (and corresponding right-hand side entry) by a scalar. For example:

$$\begin{pmatrix} 2 & 3 \\ -3 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & \frac{3}{2} \\ -3 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 7 \end{pmatrix}$$

- We can replace row q by row $q + k\times$ row p. For example:

$$\begin{pmatrix} 2 & 3 \\ -3 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 2 & 3 \\ 0 & 6.5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 13 \end{pmatrix}$$

  (here we replaced row w by row $2 + 1.5\times$ row 1)

$$\begin{pmatrix} 1 & 2 \\ 4 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -3 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 \\ 0 & -7 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -7 \end{pmatrix}$$

(here we replaced row 2 by row $2 + (-4)\times$ row 1)

Our strategy for solving systems of linear equations using Gaussian elimination is based on the following ideas:

- Three types of operation described above are called **elementary row operations** (ERO).
- We will applying a sequence of ERO to reduce an arbitrary system to a triangular form, which, we will see, can be easily solved.
- The algorithm for reducing a general matrix to upper triangular form is known as **forward elimination** or (more commonly) as **Gaussian elimination**.

4.3 Gaussian elimination

The algorithm of Gaussian elimination is a very old method that you may have already met at school - perhaps by a different name. The details of the method may seem quite confusing at first. Really we are following the ideas of eliminating systems of simultaneous equations, but in a way a computer understands. This is an important point in this section. You will have seen different ideas of how to solve systems of simulations equations where the first step is to "look at the equations to decide the easiest first step". When there are 10^9 equations, its not effective for a computer to try and find an easy way through the problem. It must instead of a simple set of instructions to follow: this will be our algorithm.

The method is so old, in fact we have evidence of Chinese mathematicians using Gaussian elimination in 179CE (From Wikipedia):

> The method of Gaussian elimination appears in the Chinese mathematical text Chapter Eight: Rectangular Arrays of The Nine Chapters on the Mathematical Art. Its use is illustrated in eighteen problems, with two to five equations. The first reference to the book by this title is dated to 179 CE, but parts of it were written as early as approximately 150 BCE. It was commented on by Liu Hui in the 3rd century.
>
> The method in Europe stems from the notes of Isaac Newton. In 1670, he wrote that all the algebra books known to him lacked a lesson for solving simultaneous equations, which Newton then supplied. Carl Friedrich Gauss in 1810 devised a notation for symmetric elimination that was adopted in the 19th century by professional hand computers to solve the normal equations of least-squares problems.

The algorithm that is taught in high school was named for Gauss only in the 1950s as a result of confusion over the history of the subject.

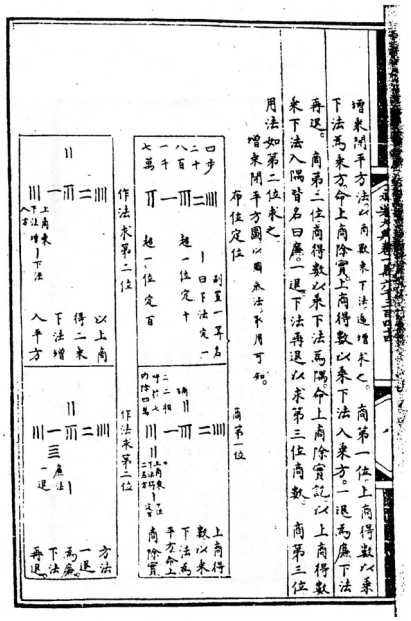


Figure 4.1: Image from Nine Chapter of the Mathematical art. By Yang Hui(1238-1298) - mybook, Public Domain, https://commons.wikimedia.org/w/index.php?curid=10317744

4.3.1 The algorithm

The following algorithm systematically introduces zeros into the system of equations, below the diagonal.

1. Subtract multiples of row 1 from the rows below it to eliminate (make zero) nonzero entries in column 1.
2. Subtract multiplies of the new row 2 from the rows below it to eliminate nonzero entries in column 2.
3. Repeat for row $3, 4, \ldots, n-1$.

After row $n-1$ all entities below the diagonal have been eliminated, so A is now upper triangular and the resulting system can be solved by backward substitution.

Example 4.2. Use Gaussian eliminate to reduce the following system of equations to upper triangular form:

$$\begin{pmatrix} 2 & 1 & 4 \\ 1 & 2 & 2 \\ 2 & 4 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 22 \end{pmatrix}.$$

First, use the first row to eliminate the first column below the diagonal:

- (row 2) $-0.5\times$ (row 1) gives

$$\begin{pmatrix} 2 & 1 & 4 \\ \mathbf{0} & 1.5 & 0 \\ 2 & 4 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 3 \\ 22 \end{pmatrix}$$

- (row 3) $-$ (row 1) then gives

$$\begin{pmatrix} 2 & 1 & 4 \\ \mathbf{0} & 1.5 & 0 \\ \mathbf{0} & 3 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 3 \\ 10 \end{pmatrix}$$

Now use the second row to eliminate the second column below the diagonal.

- (row 3) $-2\times$ (row 2) gives

$$\begin{pmatrix} 2 & 1 & 4 \\ \mathbf{0} & 1.5 & 0 \\ \mathbf{0} & \mathbf{0} & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 3 \\ 4 \end{pmatrix}$$

Exercise 4.2. Use Gaussian elimination to reduce the following system of linear equations to upper triangular form.

$$\begin{pmatrix} 4 & -1 & -1 \\ 2 & 4 & 2 \\ 1 & 2 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 9 \\ -6 \\ 3 \end{pmatrix}.$$

Remark.

- Each row i is used to eliminate the entries in column i below a_{ii}, i.e. it forces $a_{ji} = 0$ for $j > i$.
- This is done by subtracting a multiple of row i from row j:

$$(\text{row } j) \leftarrow (\text{row } j) - \frac{a_{ji}}{a_{ii}}(\text{row } i).$$

- This guarantees that a_{ji} becomes zero because

$$a_{ji} \leftarrow a_{ji} - \frac{a_{ji}}{a_{ii}} a_{ii} = a_{ji} - a_{ji} = 0.$$

Exercise 4.3. Solve the system

$$\begin{pmatrix} 4 & 3 & 2 & 1 \\ 1 & 2 & 2 & 2 \\ 1 & 1 & 3 & 0 \\ 2 & 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 10 \\ 7 \\ 5 \\ 8 \end{pmatrix}.$$

The solution is $\vec{x} = (1, 1, 1, 1)^T$.

4.3.2 Python version

We start with some helper code which determines the size of the system we are working with:

```
def system_size(A, b):
    """
    for a system of linear equations, returns the size and errors if sizes are not compatible
    """
    n, m = A.shape
    nb = b.shape[0]

    assert n == m and n == nb
    return n
```

Then we can implement the elementary row operations

```
def row_swap(A, b, p, q):
    """
    swap the rows p and q for the system of linear equations given by Ax = b.
    updates entries in place
    """
    n = system_size(A, b)
    # swap rows of A
    for j in range(n):
        A[p, j], A[q, j] = A[q, j], A[p, j]
    # swap rows of b
    b[p, 0], b[q, 0] = b[q, 0], b[p, 0]
```

```
def row_scale(A, b, p, k):
    """
    scale the entries in row p by k for the system of linear equations given by Ax = b.
    updates entries in place
    """
    n = system_size(A, b)

    # scale row p of A
    for j in range(n):
        A[p, j] = k * A[p, j]
    # scale row p of b
    b[p, 0] = b[p, 0] * k


def row_add(A, b, p, k, q):
    """
    add rows for the system of linear equations given by Ax = b
    this operation is row p |-> row p + k * row q
    updates entries in place
    """
    n = system_size(A, b)

    for j in range(n):
        A[p, j] = A[p, j] + k * A[q, j]
    b[p, 0] = b[p, 0] + k * b[q, 0]
```

Let's test we are ok so far:

Test 1: swapping rows

```
A = np.array([[2.0, 3.0], [-3.0, 2.0]])
b = np.array([[4.0], [7.0]])

print("starting arrays:")
print_array(A)
print_array(b)
print()

print("swapping rows 0 and 1")
row_swap(A, b, 0, 1)  # remember numpy arrays are indexed starting from zero!
print()
```

```
print("new arrays:")
print_array(A)
print_array(b)
print()
```

```
starting arrays:
A = [  2.0,  3.0 ]
    [ -3.0,  2.0 ]
b = [  4.0 ]
    [  7.0 ]

swapping rows 0 and 1

new arrays:
A = [ -3.0,  2.0 ]
    [  2.0,  3.0 ]
b = [  7.0 ]
    [  4.0 ]
```

Test 2: scaling one row

```
A = np.array([[2.0, 3.0], [-3.0, 2.0]])
b = np.array([[4.0], [7.0]])

print("starting arrays:")
print_array(A)
print_array(b)
print()

print("row 0 |-> 0.5 * row 0")
row_scale(A, b, 0, 0.5)  # remember numpy arrays are indexed started from zero!
print()

print("new arrays:")
print_array(A)
print_array(b)
print()
```

```
starting arrays:
A = [  2.0,  3.0 ]
    [ -3.0,  2.0 ]
```

```
b = [  4.0 ]
    [  7.0 ]

row 0 |-> 0.5 * row 0

new arrays:
A = [  1.0,  1.5 ]
    [ -3.0,  2.0 ]
b = [  2.0 ]
    [  7.0 ]
```

Test 3: replacing a row by that adding a multiple of another row

```
A = np.array([[2.0, 3.0], [-3.0, 2.0]])
b = np.array([[4.0], [7.0]])

print("starting arrays:")
print_array(A)
print_array(b)
print()

print("row 1 |-> row 1 + 1.5 * row 0")
row_add(A, b, 1, 1.5, 0)  # remember numpy arrays are indexed started from zero!
print()

print("new arrays:")
print_array(A)
print_array(b)
print()
```

```
starting arrays:
A = [  2.0,  3.0 ]
    [ -3.0,  2.0 ]
b = [  4.0 ]
    [  7.0 ]

row 1 |-> row 1 + 1.5 * row 0

new arrays:
A = [  2.0,  3.0 ]
    [  0.0,  6.5 ]
b = [  4.0 ]
```

```
[ 13.0 ]
```

Now we can define our Gaussian elimination function. We update the values in-place to avoid extra memory allocations.

```python
def gaussian_elimination(A, b, verbose=False):
    """
    perform Gaussian elimnation to reduce the system of linear equations Ax=b to upper triang
    use verbose to print out intermediate representations
    """
    # find shape of system
    n = system_size(A, b)

    # perform forwards elimination
    for i in range(n - 1):
        # eliminate column i
        if verbose:
            print(f"eliminating column {i}")
        for j in range(i + 1, n):
            # row j
            factor = A[j, i] / A[i, i]
            if verbose:
                print(f"  row {j} |-> row {j} - {factor} * row {i}")
            row_add(A, b, j, -factor, i)

        if verbose:
            print()
            print("new system")
            print_array(A)
            print_array(b)
            print()
```

We can try our code on Example 1:

```python
A = np.array([[2.0, 1.0, 4.0], [1.0, 2.0, 2.0], [2.0, 4.0, 6.0]])
b = np.array([[12.0], [9.0], [22.0]])

print("starting system:")
print_array(A)
print_array(b)
print()
```

```
print("performing Gaussian Elimination")
gaussian_elimination(A, b, verbose=True)
print()

print("final system:")
print_array(A)
print_array(b)
print()

# test that A is really upper triangular
print("Is A really upper triangular?", np.allclose(A, np.triu(A)))
```

```
starting system:
A = [  2.0,  1.0,  4.0 ]
    [  1.0,  2.0,  2.0 ]
    [  2.0,  4.0,  6.0 ]
b = [ 12.0 ]
    [  9.0 ]
    [ 22.0 ]

performing Gaussian Elimination
eliminating column 0
  row 1 |-> row 1 - 0.5 * row 0
  row 2 |-> row 2 - 1.0 * row 0

new system
A = [  2.0,  1.0,  4.0 ]
    [  0.0,  1.5,  0.0 ]
    [  0.0,  3.0,  2.0 ]
b = [ 12.0 ]
    [  3.0 ]
    [ 10.0 ]

eliminating column 1
  row 2 |-> row 2 - 2.0 * row 1

new system
A = [  2.0,  1.0,  4.0 ]
    [  0.0,  1.5,  0.0 ]
    [  0.0,  0.0,  2.0 ]
b = [ 12.0 ]
    [  3.0 ]
```

```
    [  4.0 ]


final system:
A = [  2.0,  1.0,  4.0 ]
    [  0.0,  1.5,  0.0 ]
    [  0.0,  0.0,  2.0 ]
b = [ 12.0 ]
    [  3.0 ]
    [  4.0 ]

Is A really upper triangular? True
```

4.4 Solving triangular systems of equations

A general *lower triangular* system of equations has $a_{ij} = 0$ for $j > i$ and takes the form:

$$\begin{pmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ a_{31} & a_{32} & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}.$$

Note the first equation is

$$a_{11}x_1 = b_1.$$

Then x_i can be found by calculating

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

for each row $i = 1, 2, \ldots, n$ in turn.

- Each calculation requires only previously computed values x_j (and the sum gives a loop for $j < i$.
- The matrix A **must** have nonzero diagonal entries
  i.e. $a_{ii} \neq 0$ for $i = 1, 2, \ldots, n$.
- **Upper triangular** systems of equations can be solved in a similar manner.

Example 4.3. Solve the lower triangular system of equations given by

$$\begin{aligned} 2x_1 &= 2 \\ x_1 + 2x_2 &= 7 \\ 2x_1 + 4x_2 + 6x_3 &= 26 \end{aligned}$$

or, equivalently,

$$\begin{pmatrix} 2 & 0 & 0 \\ 1 & 2 & 0 \\ 2 & 4 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 7 \\ 26 \end{pmatrix}.$$

The solution can be calculated systematically from

$$\begin{aligned} x_1 &= \frac{b_1}{a_{11}} = \frac{2}{2} = 1 \\ x_2 &= \frac{b_2 - a_{21}x_1}{a_{22}} = \frac{7 - 1 \times 1}{2} = \frac{6}{2} = 3 \\ x_3 &= \frac{b_3 - a_{31}x_1 - a_{32}x_2}{a_3 3} = \frac{26 - 2 \times 1 - 4 \times 3}{6} = \frac{12}{6} = 2 \end{aligned}$$

which gives the solution $\vec{x} = (1, 3, 2)^T$.

Exercise 4.4. Solve the upper triangular linear system given by

$$\begin{aligned} 2x_1 + x_2 + 4x_3 &= 12 \\ 1.5x_2 &= 3. \\ 2x_3 &= 4 \end{aligned}$$

Remark 4.1.

- It is simple to solve a lower (upper) triangular system of equations (provided the diagonal is nonzero).
- This process is often referred to as **forward** (**backward**) **substitution**.
- A general system of equations (i.e. a full matrix A) can be solved rapidly once it has been reduced to upper triangular form. This is the idea of using Gaussian elimination with backward subsitution.

We can define functions to solve both upper or lower triangular form systems of linear equations:

```
def forward_substitution(A, b):
    """
    solves the system of linear equationa Ax = b assuming that A is lower triangular
    returns the solution x
    """
    # get size of system
    n = system_size(A, b)

    # check is lower triangular
    assert np.allclose(A, np.tril(A))

    # create solution variable
    x = np.empty_like(b)

    # perform forwards solve
    for i in range(n):
        partial_sum = 0.0
        for j in range(0, i):
            partial_sum += A[i, j] * x[j]
        x[i] = 1.0 / A[i, i] * (b[i] - partial_sum)

    return x


def backward_substitution(A, b):
    """
    solves the system of linear equationa Ax = b assuming that A is upper triangular
    returns the solution x
    """
    # get size of system
    n = system_size(A, b)

    # check is upper triangular
    assert np.allclose(A, np.triu(A))

    # create solution variable
    x = np.empty_like(b)

    # perform backwards solve
    for i in range(n - 1, -1, -1):  # iterate over rows backwards
```

```
        partial_sum = 0.0
        for j in range(i + 1, n):
            partial_sum += A[i, j] * x[j]
        x[i] = 1.0 / A[i, i] * (b[i] - partial_sum)

    return x
```

And we can then test it out!

```
A = np.array([[2.0, 0.0, 0.0], [1.0, 2.0, 0.0], [2.0, 4.0, 6.0]])
b = np.array([[2.0], [7.0], [26.0]])

print("The system is given by:")
print_array(A)
print_array(b)
print()

print("Solving the system using forward substitution")
x = forward_substitution(A, b)
print()

print("The solution using forward substitution is:")
print_array(x)
print()

print("Does x really solve the system?", np.allclose(A @ x, b))
```

```
The system is given by:
A = [  2.0,  0.0,  0.0 ]
    [  1.0,  2.0,  0.0 ]
    [  2.0,  4.0,  6.0 ]
b = [  2.0 ]
    [  7.0 ]
    [ 26.0 ]

Solving the system using forward substitution

The solution using forward substitution is:
x = [  1.0 ]
    [  3.0 ]
    [  2.0 ]
```

```
Does x really solve the system? True
```

We can also do a backward substitution test:

```
A = np.array([[2.0, 1.0, 4.0], [0.0, 1.5, 0.0], [0.0, 0.0, 2.0]])
b = np.array([[12.0], [3.0], [4.0]])

print("The system is given by:")
print_array(A)
print_array(b)
print()

print("Solving the system using backward substitution")
x = backward_substitution(A, b)
print()

print("The solution using backward substitution is:")
print_array(x)
print()


print("Does x really solve the system?", np.allclose(A @ x, b))
```

```
The system is given by:
A = [  2.0,  1.0,  4.0 ]
    [  0.0,  1.5,  0.0 ]
    [  0.0,  0.0,  2.0 ]
b = [ 12.0 ]
    [  3.0 ]
    [  4.0 ]

Solving the system using backward substitution

The solution using backward substitution is:
x = [  1.0 ]
    [  2.0 ]
    [  2.0 ]

Does x really solve the system? True
```

4.5 Combining Gaussian elimination and backward substitution

Our grant strategy can now come together so we have a method to solve systems of linear equations:

Given a system of linear equations $A\vec{x} = \vec{b}$;

- First perform Gaussian elimination to give an equivalent system of equations in upper triangular form;
- Then use backward substitution to produce a solution $\vec{x}$

We can use our code to test this:

```
A = np.array([[2.0, 1.0, 4.0], [1.0, 2.0, 2.0], [2.0, 4.0, 6.0]])
b = np.array([[12.0], [9.0], [22.0]])

print("starting system:")
print_array(A)
print_array(b)
print()

print("performing Gaussian Elimination")
gaussian_elimination(A, b, verbose=True)
print()

print("upper triangular system:")
print_array(A)
print_array(b)
print()

print("Solving the system using backward substitution")
x = backward_substitution(A, b)
print()

print("solution using backward substitution:")
print_array(x)
print()

A = np.array([[2.0, 1.0, 4.0], [1.0, 2.0, 2.0], [2.0, 4.0, 6.0]])
b = np.array([[12.0], [9.0], [22.0]])
print("Does x really solve the original system?", np.allclose(A @ x, b))
```

```
starting system:
```

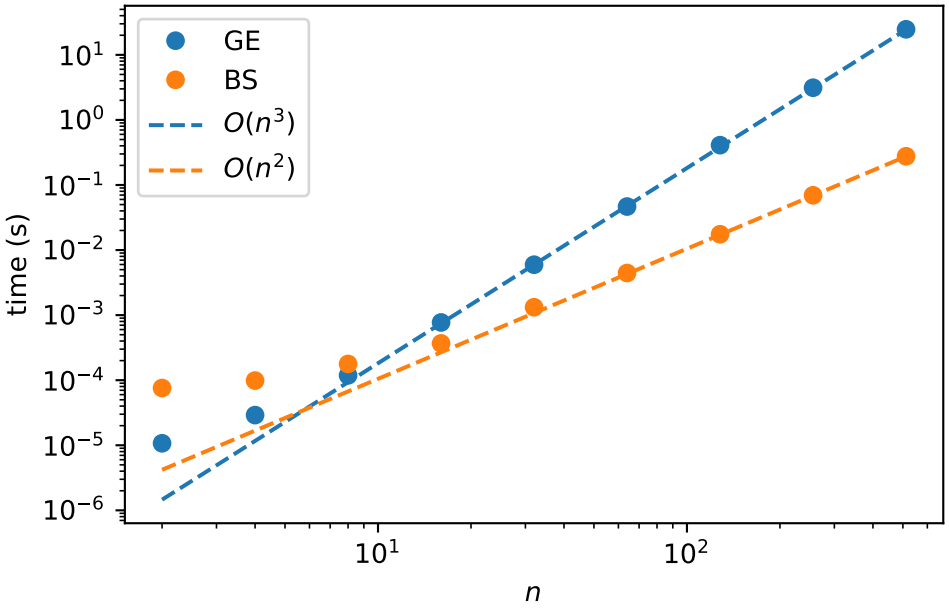
```
A = [  2.0,  1.0,  4.0 ]
    [  1.0,  2.0,  2.0 ]
    [  2.0,  4.0,  6.0 ]
b = [ 12.0 ]
    [  9.0 ]
    [ 22.0 ]

performing Gaussian Elimination
eliminating column 0
  row 1 |-> row 1 - 0.5 * row 0
  row 2 |-> row 2 - 1.0 * row 0

new system
A = [  2.0,  1.0,  4.0 ]
    [  0.0,  1.5,  0.0 ]
    [  0.0,  3.0,  2.0 ]
b = [ 12.0 ]
    [  3.0 ]
    [ 10.0 ]

eliminating column 1
  row 2 |-> row 2 - 2.0 * row 1

new system
A = [  2.0,  1.0,  4.0 ]
    [  0.0,  1.5,  0.0 ]
    [  0.0,  0.0,  2.0 ]
b = [ 12.0 ]
    [  3.0 ]
    [  4.0 ]


upper triangular system:
A = [  2.0,  1.0,  4.0 ]
    [  0.0,  1.5,  0.0 ]
    [  0.0,  0.0,  2.0 ]
b = [ 12.0 ]
    [  3.0 ]
    [  4.0 ]

Solving the system using backward substitution

solution using backward substitution:
```

```
x = [  1.0 ]
    [  2.0 ]
    [  2.0 ]

Does x really solve the original system? True
```

4.6 The cost of Gaussian Elimination

Gaussian elimination (GE) is unnecessarily expensive when it is applied to many systems of equations with the same matrix A but different right-hand sides $\vec{b}$.

- The forward elimination process is the most computationally expensive part at $O(n^3)$ but is exactly the same for any choice of $\vec{b}$.
- In contrast, the solution of the resulting upper triangular system only requires $O(n^2)$ operations.



We can use this information to improve the way in which we solve multiple systems of equations with the same matrix A but different right-hand sides $\vec{b}$.

4.7 LU factorisation

Our next algorithm, called LU factorisation, is a way to try to speed up Gaussian elimination by reusing information. This can be used when we solve systems of equations with the same matrix A but different right hand sides $\vec{b}$ - this is more common than you would think!

Recall the elementary row operations (EROs) from above. Note that the EROs can be produced by left multiplication with a suitable matrix:

- Row swap:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = \begin{pmatrix} a & b & c \\ g & h & i \\ d & e & f \end{pmatrix}$$

- Row swap:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} = \begin{pmatrix} a & b & c & d \\ i & j & k & l \\ e & f & g & h \\ m & n & o & p \end{pmatrix}$$

- Multiply row by α:

$$\begin{pmatrix} \alpha & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = \begin{pmatrix} \alpha a & \alpha b & \alpha c \\ d & e & f \\ g & h & i \end{pmatrix}$$

- $\alpha \times$ row $p +$ row q:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \alpha & 0 & 1 \end{pmatrix} \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ \alpha a + g & \alpha b + h & \alpha c + i \end{pmatrix}$$

Since Gaussian elimination (GE) is just a sequence of EROs and each ERO just multiplication by a suitable matrix, say E_k, forward elimination applied to the system $A\vec{x} = \vec{b}$ can be expressed as

$$(E_m \cdots E_1)A\vec{x} = (E_m \cdots E_1)\vec{b},$$

here m is the number of EROs required to reduce the upper triangular form.

Let $U = (E_m \cdots E_1)A$ and $L = (E_m \cdots E_1)^{-1}$. Now the original system $A\vec{x} = \vec{b}$ is equivalent to

$$LU\vec{x} = \vec{b} \tag{4.5}$$

where U is *upper triangular* (by construction) and L may be shown to be lower triangular (provided the EROs do not include any row swaps).

Once L and U are known it is easy to solve (4.5)

- Solve $L\vec{z} = \vec{b}$ in $O(n^2)$ operations.
- Solve $U\vec{x} = \vec{z}$ in $O(n^2)$ operations.

L and U may be found in $O(n^3)$ operations by performing GE and saving the E_i matrices, however it is more convenient to find them directly (also $O(n^3)$ operations).

4.7.1 Computing L and U

Consider a general 4×4 matrix A and its factorisation LU:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{pmatrix}$$

For the first column,

$$\begin{aligned} a_{11} &= (1,0,0,0)(u_{11},0,0,0)^T &&= u_{11} && \rightarrow u_{11} = a_{11} \\ a_{21} &= (l_{21},1,0,0)(u_{11},0,0,0)^T &&= l_{21}u_{11} && \rightarrow l_{21} = a_{21}/u_{11} \\ a_{31} &= (l_{31},l_{32},1,0)(u_{11},0,0,0)^T &&= l_{31}u_{11} && \rightarrow l_{31} = a_{31}/u_{11} \\ a_{41} &= (l_{41},l_{42},l_{43},1)(u_{11},0,0,0)^T &&= l_{41}u_{11} && \rightarrow l_{41} = a_{41}/u_{11} \end{aligned}$$

The second, third and fourth columns follow in a similar manner, giving all the entries in L and U.

Remark.

- L is assumed to have 1's on the diagonal, to ensure that the factorisation is unique.
- The process involves division by the diagonal entries u_{11}, u_{22}, etc., so they **must** be non-zero.
- In general the factors l_{ij} and u_{ij} are calculated for each column j in turn, i.e.,

```
for j in range(n):
  for i in range(j+1):
      # Compute factors u_{ij}
      ...
  for i in range(j+1, n):
      # Compute factors l_{ij}
      ...
```

Example 4.4. Use LU factorisation to solve the linear system of equations given by

$$\begin{pmatrix} 2 & 1 & 4 \\ 1 & 2 & 2 \\ 2 & 4 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 22 \end{pmatrix}.$$

This can be rewritten in the form $A = LU$ where

$$\begin{pmatrix} 2 & 1 & 4 \\ 1 & 2 & 2 \\ 2 & 4 & 6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}.$$

Column 1 of A gives

$$\begin{aligned} 2 &= u_{11} && \rightarrow && u_{11} = 2 \\ 1 &= l_{21}u_{11} && \rightarrow && l_{21} = 0.5 \\ 2 &= l_{31}u_{11} && \rightarrow && l_{31} = 1. \end{aligned}$$

Column 2 of A gives

$$\begin{aligned} 1 &= u_{12} && \rightarrow && u_{12} = 1 \\ 2 &= l_{21}u_{12} + u_{22} && \rightarrow && u_{22} = 1.5 \\ 4 &= l_{31}u_{12} + l_{32}u_{22} && \rightarrow && l_{32} = 2. \end{aligned}$$

Column 3 of A gives

$$\begin{aligned} 4 &= u_{13} && \rightarrow && u_{13} = 4 \\ 2 &= l_{21}u_{13} + u_{23} && \rightarrow && u_{23} = 0 \\ 6 &= l_{31}u_{13} + l_{32}u_{23} + u_{33} && \rightarrow && u_{33} = 2. \end{aligned}$$

Solve the lower triangular system $L\vec{z} = \vec{b}$:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 1 & 2 & 1 \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 22 \end{pmatrix} \rightarrow \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 3 \\ 4 \end{pmatrix}$$

Solve the upper triangular system $U\vec{x} = \vec{z}$:

$$\begin{pmatrix} 2 & 1 & 4 \\ 0 & 1.5 & 0 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \begin{pmatrix} 12 \\ 3 \\ 4 \end{pmatrix} \rightarrow \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix}.$$

Exercise 4.5. Rewrite the matrix A as the product of lower and upper triangular matrices where

$$A = \begin{pmatrix} 4 & 2 & 0 \\ 2 & 3 & 1 \\ 0 & 1 & 2.5 \end{pmatrix}.$$

Remark. The first example gives

$$\begin{pmatrix} 2 & 1 & 4 \\ 1 & 2 & 2 \\ 2 & 4 & 6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 1 & 2 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 4 \\ 0 & 1.5 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

Note that

- the matrix U is the same as the fully eliminated upper triangular form produced by Gaussian elimination;
- L contains the multipliers that were used at each stage to eliminate the rows.

4.7.2 Python code

We can implement computation of the LU factorisation:

```
def lu_factorisation(A):
    """
    compute the LU factorisation of A
    returns the factors L and U
    """
    n, m = A.shape
    assert n == m
```

```
    # construct arrays of zeros
    L, U = np.zeros_like(A), np.zeros_like(A)

    # fill entries
    for i in range(n):
        L[i, i] = 1
        # compute entries in U
        for j in range(i, n):
            U[i, j] = A[i, j] - sum(L[i, k] * U[k, j] for k in range(i))
        # compute entries in L
        for j in range(i + 1, n):
            L[j, i] = (A[j, i] - sum(L[j, k] * U[k, i] for k in range(i))) / U[i, i]

    return L, U
```

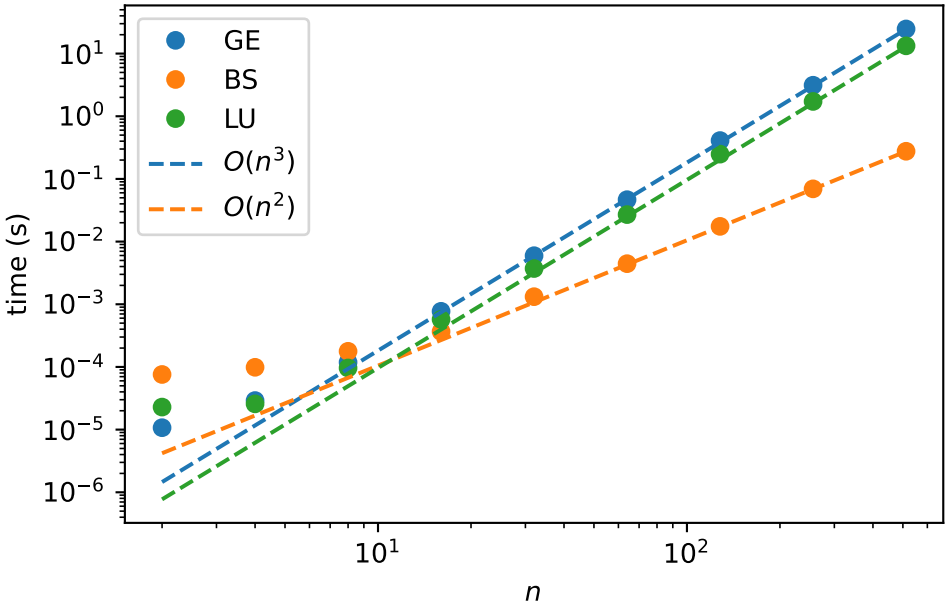
and test our implementation:

```
A = np.array([[2.0, 1.0, 4.0], [1.0, 2.0, 2.0], [2.0, 4.0, 6.0]])

print("matrix:")
print_array(A)
print()

print("performing factorisation")
L, U = lu_factorisation(A)
print()

print("factorisation:")
print_array(L)
print_array(U)
print()

print("Is L lower triangular?", np.allclose(L, np.tril(L)))
print("Is U lower triangular?", np.allclose(U, np.triu(U)))
print("Is LU a factorisation of A?", np.allclose(L @ U, A))
```

```
matrix:
A = [  2.0,  1.0,  4.0 ]
    [  1.0,  2.0,  2.0 ]
    [  2.0,  4.0,  6.0 ]
```

```
performing factorisation

factorisation:
L = [  1.0,  0.0,  0.0 ]
    [  0.5,  1.0,  0.0 ]
    [  1.0,  2.0,  1.0 ]
U = [  2.0,  1.0,  4.0 ]
    [  0.0,  1.5,  0.0 ]
    [  0.0,  0.0,  2.0 ]

Is L lower triangular? True
Is U lower triangular? True
Is LU a factorisation of A? True
```

and then add LU factorisation times to our plot:



We see that LU factorisation is still $O(n^3)$ and that the run times are similar to Gaussian elimination. But, importantly, we can reuse this factorisation more cheaply for different right-hand sides $\vec{b}$.

4.8 Effects of finite precision arithmetic

Example 4.5. Consider the following linear system of equations

$$\begin{pmatrix} 0 & 2 & 1 \\ 2 & 1 & 0 \\ 1 & 2 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 4 \\ 5 \end{pmatrix}$$

Problem. We cannot eliminate the first column by the diagonal by adding multiples of row 1 to rows 2 and 3 respectively.

Solution. Swap the order of the equations!

- Swap rows 1 and 2:

$$\begin{pmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \\ 1 & 2 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 7 \\ 5 \end{pmatrix}$$

- Now apply Gaussian elimination

$$\begin{pmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 1.5 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 7 \\ 3 \end{pmatrix}; \begin{pmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & -0.75 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 7 \\ -2.25 \end{pmatrix}.$$

Example 4.6. Consider another system of equations

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

- Apply Gaussian elimination as usual:

$$\begin{pmatrix} 2 & 1 & 1 \\ 0 & 0 & -1 \\ 2 & 2 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ -1 \\ 2 \end{pmatrix}; \begin{pmatrix} 2 & 1 & 1 \\ 0 & 0 & -1 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ -1 \\ -1 \end{pmatrix}$$

- *Problem.* We cannot eliminate the second column below the diagonal by adding a multiple of row 2 to row 3.
- Again this problem may be overcome simply by swapping the order of the equations - this time swapping rows 2 and 3:

$$\begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & -1 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ -1 \\ -1 \end{pmatrix}$$

- We can now continue the Gaussian elimination process as usual.

In general. Gaussian elimination requires row swaps to avoid breaking down when there is a zero in the "pivot" position. This might be a familiar aspect of Gaussian elimination, but there is an additional reason to apply pivoting when working with floating point numbers:

Example 4.7. Consider using Gaussian elimination to solve the linear system of equations given by

$$\begin{pmatrix} \varepsilon & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2+\varepsilon \\ 3 \end{pmatrix}$$

where $\varepsilon \neq 1$.

- The true, unique solution is $(x_1, x_2)^T = (1, 2)^T$.
- If $\varepsilon \neq 0$, Gaussian elimination gives

$$\begin{pmatrix} \varepsilon & 1 \\ 0 & 1 - \frac{1}{\varepsilon} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2+\varepsilon \\ 3 - \frac{2+\varepsilon}{\varepsilon} \end{pmatrix}$$

- Problems occur not only when $\varepsilon = 0$ but also when it is very small, i.e. when $\frac{1}{\varepsilon}$ is very large, this will introduce very significant rounding errors into the computation.

Use Gaussian elimination to solve the linear system of equations given by

$$\begin{pmatrix} 1 & 1 \\ \varepsilon & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3 \\ 2+\varepsilon \end{pmatrix}$$

where $\varepsilon \neq 1$.

- The true solution is still $(x_1, x_2)^T = (1, 2)^T$.
- Gaussian elimination now gives

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 - \varepsilon \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 - 2\varepsilon \end{pmatrix}$$

- The problems due to small values of ε have disappeared.

This is a genuine problem we see in the code versions too!

```
print("without row swapping:")
for eps in [1.0e-2, 1.0e-4, 1.0e-6, 1.0e-8, 1.0e-10, 1.0e-12, 1.0e-14]:
    A = np.array([[eps, 1.0], [1.0, 1.0]])
    b = np.array([[2.0 + eps], [3.0]])

    gaussian_elimination(A, b)
    x = backward_substitution(A, b)
    print(f"{eps=:.1e}", end=", ")
    print_array(x.T, "x.T", end=", ")

    A = np.array([[eps, 1.0], [1.0, 1.0]])
    b = np.array([[2.0 + eps], [3.0]])
    print("Solution?", np.allclose(A @ x, b))
print()

print("with row swapping:")
for eps in [1.0e-2, 1.0e-4, 1.0e-6, 1.0e-8, 1.0e-10, 1.0e-12, 1.0e-14]:
    A = np.array([[1.0, 1.0], [eps, 1.0]])
    b = np.array([[3.0], [2.0 + eps]])

    gaussian_elimination(A, b)
    x = backward_substitution(A, b)
    print(f"{eps=:.1e}", end=", ")
    print_array(x.T, "x.T", end=", ")

    A = np.array([[1.0, 1.0], [eps, 1.0]])
    b = np.array([[3.0], [2.0 + eps]])
    print("Solution?", np.allclose(A @ x, b))
print()
```

```
without row swapping:
eps=1.0e-02, x.T = [  1.0,  2.0 ], Solution? True
eps=1.0e-04, x.T = [  1.0,  2.0 ], Solution? True
eps=1.0e-06, x.T = [  1.0,  2.0 ], Solution? True
eps=1.0e-08, x.T = [  1.0,  2.0 ], Solution? True
eps=1.0e-10, x.T = [  1.0,  2.0 ], Solution? True
eps=1.0e-12, x.T = [  1.00009,  2.00000 ], Solution? False
eps=1.0e-14, x.T = [  1.0214,  2.0000 ], Solution? False

with row swapping:
eps=1.0e-02, x.T = [  1.0,  2.0 ], Solution? True
eps=1.0e-04, x.T = [  1.0,  2.0 ], Solution? True
```

```
eps=1.0e-06, x.T = [  1.0,  2.0 ], Solution? True
eps=1.0e-08, x.T = [  1.0,  2.0 ], Solution? True
eps=1.0e-10, x.T = [  1.0,  2.0 ], Solution? True
eps=1.0e-12, x.T = [  1.0,  2.0 ], Solution? True
eps=1.0e-14, x.T = [  1.0,  2.0 ], Solution? True
```

Remark 4.2.

- Writing the equations in a different order has removed the previous problem.
- The diagonal entries are now always *relatively* larger.
- The interchange of the order of equations is a simple example of **row pivoting**. This strategy avoids excessive rounding errors in the computations.

4.8.1 Gaussian elimination with pivoting

Key idea:

- Before eliminating entries in column j:
  - find the entry in column j, below the diagonal, of maximum magnitude;
  - if that entry is larger in magnitude than the diagonal entry then swap its row with row j.
- Then eliminate column j as before.

This algorithm will always work when the matrix A is invertible/non-singular. Conversely, if all of the possible pivot values are zero this implies that the matrix is singular and a unique solution does not exist. At each elimination step the row multiplies used are guaranteed to be at most one in magnitude so any errors in the representation of the system cannot be amplified by the elimination process. As always, solving $A\vec{x} = \vec{b}$ requires that the entries in $\vec{b}$ are also swapped in the appropriate way. Pivoting can be applied in an equivalent way to LU factorisation. The sequence of pivots is independent of the vector $\vec{b}$ and can be recorded and reused. The constraint imposed on the row multipliers means that for LU factorisation every entry in L satisfies $|l_{ij}| \leq 1$.

Example 4.8. Consider the linear system of equations given by

$$\begin{pmatrix} 10 & -7 & 0 \\ -3 & 2.1-\varepsilon & 6 \\ 5 & -1 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 9.9+\varepsilon \\ 11 \end{pmatrix}$$

where $0 \leq \varepsilon \ll 1$, and solve it using

1. Gaussian elimination without pivoting
2. Gaussian elimination with pivoting.

The exact solution is $\vec{x} = (0, -1, 2)^T$ for any ε in the given range.

1. Solve the system using Gaussian elimination with no pivoting.

Eliminating the first column gives

$$\begin{pmatrix} 10 & -7 & 0 \\ 0 & -\varepsilon & 6 \\ 0 & 2.5 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 12+\varepsilon \\ 7.5 \end{pmatrix}$$

and then the second column gives

$$\begin{pmatrix} 10 & -7 & 0 \\ 0 & -\varepsilon & 6 \\ 0 & 0 & 5+15/\varepsilon \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 12+\varepsilon \\ 7.5+2.5(12+\varepsilon)/\varepsilon \end{pmatrix}$$

which leads to

$$x_3 = \frac{3+\frac{12+\varepsilon}{\varepsilon}}{2+\frac{6}{\varepsilon}} \qquad x_2 = \frac{(12+\varepsilon)-6x_3}{-\varepsilon} \qquad x_1 = \frac{7+7x_2}{10}.$$

There are many divisions by ε, so we will have problems if ε is (very) small.

2. Solve the system using Gaussian elimination with pivoting.

The first stage is identical (because $a_{11} = 10$ is largest).

$$\begin{pmatrix} 10 & -7 & 0 \\ 0 & -\varepsilon & 6 \\ 0 & 2.5 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 12+\varepsilon \\ 7.5 \end{pmatrix}$$

but now $|a_{22}| = \varepsilon$ and $|a_{32}| = 2.5$ so we swap rows 2 and 3 to give

$$\begin{pmatrix} 10 & -7 & 0 \\ 0 & 2.5 & 5 \\ 0 & -\varepsilon & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 7.5 \\ 12+\varepsilon \end{pmatrix}$$

Now we may eliminate column 2:

$$\begin{pmatrix} 10 & -7 & 0 \\ 0 & 2.5 & 5 \\ 0 & 0 & 6+2\varepsilon \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 7.5 \\ 12+4\varepsilon \end{pmatrix}$$

which leads to the exact answer:

$$x_3 = \frac{12+4\varepsilon}{6+2\varepsilon} = 2 \qquad x_2 = \frac{7.5-5x_3}{2.5} = -1 \qquad x_1 = \frac{7+7x_2}{10} = 0.$$

4.8.2 Python code

```
def gaussian_elimination_with_pivoting(A, b, verbose=False):
    """
    perform Gaussian elimnation with pivoting to reduce the system of linear equations Ax=b t
    use verbose to print out intermediate representations
    """
    # find shape of system
    n = system_size(A, b)

    # perform forwards elimination
    for i in range(n - 1):
        # eliminate column i
        if verbose:
            print(f"eliminating column {i}")

        # find largest entry in column i
        largest = abs(A[i, i])
        j_max = i
        for j in range(i + 1, n):
            if abs(A[j, i]) > largest:
                largest, j_max = abs(A[j, i]), j

        # swap rows j_max and i
        row_swap(A, b, i, j_max)
        if verbose:
            print(f"swapped system ({i} <-> {j_max})")
            print_array(A)
            print_array(b)
            print()
```

```
        for j in range(i + 1, n):
            # row j
            factor = A[j, i] / A[i, i]
            if verbose:
                print(f"row {j} |-> row {j} - {factor} * row {i}")
            row_add(A, b, j, -factor, i)

        if verbose:
            print("new system")
            print_array(A)
            print_array(b)
            print()
```

Gaussian elimination without pivoting following by back subsitution:

```
eps = 1.0e-14
A = np.array([[10.0, -7.0, 0.0], [-3.0, 2.1 - eps, 6.0], [5.0, -1.0, 5.0]])
b = np.array([[7.0], [9.9 + eps], [11.0]])

print("starting system:")
print_array(A)
print_array(b)
print()

print("performing Gaussian elimination without pivoting")
gaussian_elimination(A, b, verbose=True)
print()

print("upper triangular system:")
print_array(A)
print_array(b)
print()

print("performing backward substitution")
x = backward_substitution(A, b)
print()

print("solution using backward substitution:")
print_array(x)
print()

A = np.array([[10.0, -7.0, 0.0], [-3.0, 2.1 - eps, 6.0], [5.0, -1.0, 5.0]])
```

```
b = np.array([[7.0], [9.9 + eps], [11.0]])
print("Does x solve the original system?", np.allclose(A @ x, b))
```

```
starting system:
A = [ 10.0, -7.0,  0.0 ]
    [ -3.0,  2.1,  6.0 ]
    [  5.0, -1.0,  5.0 ]
b = [  7.0 ]
    [  9.9 ]
    [ 11.0 ]

performing Gaussian elimination without pivoting
eliminating column 0
  row 1 |-> row 1 - -0.3 * row 0
  row 2 |-> row 2 - 0.5 * row 0

new system
A = [ 10.0, -7.0,  0.0 ]
    [  0.0, -0.0,  6.0 ]
    [  0.0,  2.5,  5.0 ]
b = [  7.0 ]
    [ 12.0 ]
    [  7.5 ]

eliminating column 1
  row 2 |-> row 2 - -244760849313613.9 * row 1

new system
A = [ 10.0, -7.0,  0.0 ]
    [  0.0, -0.0,  6.0 ]
    [  0.0,  0.0, 1468565095881688.5 ]
b = [  7.0 ]
    [ 12.0 ]
    [ 2937130191763377.0 ]


upper triangular system:
A = [ 10.0, -7.0,  0.0 ]
    [  0.0, -0.0,  6.0 ]
    [  0.0,  0.0, 1468565095881688.5 ]
b = [  7.0 ]
    [ 12.0 ]
```

```
    [ 2937130191763377.0 ]

performing backward substitution

solution using backward substitution:
x = [ -0.030435 ]
    [ -1.043478 ]
    [  2.000000 ]

Does x solve the original system? False
```

Gaussian elimination with pivoting following by back subsitution:

```
eps = 1.0e-14
A = np.array([[10.0, -7.0, 0.0], [-3.0, 2.1 - eps, 6.0], [5.0, -1.0, 5.0]])
b = np.array([[7.0], [9.9 + eps], [11.0]])

print("starting system:")
print_array(A)
print_array(b)
print()

print("performing Gaussian elimination with pivoting")
gaussian_elimination_with_pivoting(A, b, verbose=True)
print()

print("upper triangular system:")
print_array(A)
print_array(b)
print()

print("performing backward substitution")
x = backward_substitution(A, b)
print()

print("solution using backward substitution:")
print_array(x)
print()

A = np.array([[10.0, -7.0, 0.0], [-3.0, 2.1 - eps, 6.0], [5.0, -1.0, 5.0]])
b = np.array([[7.0], [9.9 + eps], [11.0]])
print("Does x solve the original system?", np.allclose(A @ x, b))
```

```
starting system:
A = [ 10.0, -7.0,  0.0 ]
    [ -3.0,  2.1,  6.0 ]
    [  5.0, -1.0,  5.0 ]
b = [  7.0 ]
    [  9.9 ]
    [ 11.0 ]

performing Gaussian elimination with pivoting
eliminating column 0
swapped system (0 <-> 0)
A = [ 10.0, -7.0,  0.0 ]
    [ -3.0,  2.1,  6.0 ]
    [  5.0, -1.0,  5.0 ]
b = [  7.0 ]
    [  9.9 ]
    [ 11.0 ]

row 1 |-> row 1 - -0.3 * row 0
row 2 |-> row 2 - 0.5 * row 0
new system
A = [ 10.0, -7.0,  0.0 ]
    [  0.0, -0.0,  6.0 ]
    [  0.0,  2.5,  5.0 ]
b = [  7.0 ]
    [ 12.0 ]
    [  7.5 ]

eliminating column 1
swapped system (1 <-> 2)
A = [ 10.0, -7.0,  0.0 ]
    [  0.0,  2.5,  5.0 ]
    [  0.0, -0.0,  6.0 ]
b = [  7.0 ]
    [  7.5 ]
    [ 12.0 ]

row 2 |-> row 2 - -4.085620730620576e-15 * row 1
new system
A = [ 10.0, -7.0,  0.0 ]
    [  0.0,  2.5,  5.0 ]
    [  0.0,  0.0,  6.0 ]
b = [  7.0 ]
```

```
    [  7.5 ]
    [ 12.0 ]


upper triangular system:
A = [ 10.0, -7.0,  0.0 ]
    [  0.0,  2.5,  5.0 ]
    [  0.0,  0.0,  6.0 ]
b = [  7.0 ]
    [  7.5 ]
    [ 12.0 ]

performing backward substitution

solution using backward substitution:
x = [  0.0 ]
    [ -1.0 ]
    [  2.0 ]

Does x solve the original system? True
```

4.9 Further reading

Some basic reading:

- Wikipedia: Gaussian elimination
- Joseph F. Grcar. How ordinary elimination became Gaussian elimination. Historia Mathematica. Volume 38, Issue 2, May 2011. (More history)

Some reading on LU factorisation:

- $A = LU$ and solving systems [pdf]
- Wikipedia: LU decomposition
- Wikipedia: Matrix decomposition (Other examples of decompositions).
- Nick Higham: What is an LU factorization? (a very mathematical treatment with additional references)

Some reading on using Gaussian elimination with pivoting:

- Gaussian elimination with Partial Pivoting [pdf]
- Gaussian elimination with partial pivoting example [pdf]

A good general reference for this area:

- Trefethen, Lloyd N.; Bau, David (1997), Numerical linear algebra, Philadelphia: Society for Industrial and Applied Mathematics, ISBN 978-0-89871-361-9.

Some implementations:

- Numpy `numpy.linalg.solve`
- Scipy `scipy.linalg.lu`
- LAPACK Gaussian elimination (uses LU factorisation): `dgesv()`
- LAPACK LU Factorisation: `dgetrf()`.

5 Iterative solutions of linear equations

5.1 Iterative methods

In the previous section we looked at what are known as *direct* methods for solving systems of linear equations. They are guaranteed to produce a solution with a fixed amount of work (we can even prove this in exact arithmetic!), but this fixed amount of work may be **very** large.

For a general $n \times n$ system of linear equations $A\vec{x} = \vec{b}$, the computation expense of all direct methods if $O(n^3)$. The amount of storage required for these approaches is $O(n^2)$ which is dominated by the cost of storing the matrix A. As n becomes larger the storage and computation work required limit the practicality of direct approaches.

As an alternative, we will propose some **iterative methods**. Iterative methods produce a sequence $(\vec{x}^{(k)})$ of approximations to the solution of the linear system of equations $A\vec{x} = \vec{b}$. The iteration is defined recursively and is typically of the form:

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

where $\vec{x}^{(k)}$ is now a vector of values and $\vec{F}$ is some vector function (which needs to be defined to define the method). We will need to choose a starting value $\vec{x}^{(k)}$ but there is often a reasonable approximation which can be used. Once all this is defined, we still need to decide when we need to stop!

Some very bad examples

Example 1
Consider

$$\vec{F}(\vec{x}^{(k)}) = \vec{x}^{(k)}.$$

Each iteration is very cheap to compute but very inaccurate - it never converges!
Example 2
Consider

$$\vec{F}(\vec{x}^{(k)}) = \vec{x}^{(k)} + A^{-1}(\vec{b} - A\vec{x}^{(k)}).$$

Each iteration is very expensive to compute - you have to invert A! - but it converges in just one step since

$$\begin{aligned} A\vec{x}^{(k+1)} &= A\vec{x}^{(k)} + AA^{-1}(\vec{b} - A\vec{x}^{(k)}) \\ &= A\vec{x}^{(k)} + \vec{b} - A\vec{x}^{(k)} \\ &= \vec{b}. \end{aligned}$$

Key idea

The key point here is we want a method which is both cheap to compute but converges quickly to the solution. One way to do this is to construct iteration given by

$$\vec{F}(\vec{x}^{(k)}) = \vec{x}^{(k)} + P(\vec{b} - A\vec{x}^{(k)}). \tag{5.1}$$

for some matrix P such that

- P is easy to compute, or the matrix vector product $P\vec{r}$ is easy to compute,
- P approximates A^{-1} well enough that the algorithm converges in few iterations.

We call $\vec{b} - A\vec{x}^{(k)} = \vec{r}$ the **residual**. Note in the above examples we would have $P = O$ (the zero matrix) or $P = A^{-1}$.

5.2 Jacobi iteration

One simple choice for P is given by the Jacobi method where we take $P = D^{-1}$ where D is the diagonal of A:

$$D_{ii} = A_{ii} \quad \text{and} \quad D_{ij} = 0 \text{ for } i \neq j.$$

The **Jacobi iteration** is given by

$$\vec{x}^{(k+1)} = \vec{x}^{(k)} + D^{-1}(\vec{b} - A\vec{x}^{(k)})$$

D is a *diagonal matrix*, so D^{-1} is trivial to form (as long as the diagonal entries are all nonzero):

$$(D^{-1})_{ii} = \frac{1}{D_{ii}} \quad \text{and} \quad (D^{-1})_{ij} = 0 \text{ for } i \neq j.$$

Remark.

- The cost of one iteration is $O(n^2)$ for a full matrix, and this is dominated by the matrix-vector product $A\vec{x}^{(k)}$.
- This cost can be reduced to $O(n)$ if the matrix A is sparse - this is when iterative methods are especially attractive (TODO add future ref).

- The amount of work also depends on the number of iterations required to get a "satisfactory" solution.
  - The number of iterations depends on the matrix;
  - Fewer iterations are needed for a less accurate solution;
  - A good initial estimate $\vec{x}^{(0)}$ reduces the required number of iterations.
- Unfortunately, the iteration might not converge!

The Jacobi iteration updates all elements of $\vec{x}^{(k)}$ *simultaneously* to get $\vec{x}^{(k+1)}$. Writing the method out component by component gives

$$\begin{aligned}
x_1^{(k+1)} &= x_1^{(k)} + \frac{1}{A_{11}}\left(b_1 - \sum_{j=1}^{n} A_{1j}x_j^{(k)}\right) \\
x_2^{(k+1)} &= x_2^{(k)} + \frac{1}{A_{22}}\left(b_2 - \sum_{j=1}^{n} A_{2j}x_j^{(k)}\right) \\
&\vdots \qquad\quad \vdots \\
x_n^{(k+1)} &= x_n^{(k)} + \frac{1}{A_{nn}}\left(b_n - \sum_{j=1}^{n} A_{nj}x_j^{(k)}\right).
\end{aligned}$$

Note that once the first step has been taken, $x_1^{(k+1)}$ is already known, but the Jacobi iteration does not make use of this information!

Example 1

Take two iterations of Jacobi iteration to approximate the solution of the following system using the initial guess $\vec{x}^{(0)} = (1,1)^T$:

$$\begin{pmatrix} 2 & 1 \\ -1 & 4 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3.5 \\ 0.5 \end{pmatrix}$$

Starting from $\vec{x}^{(0)} = (1,1)^T$, the first iteration is

$$\begin{aligned}
x_1^{(1)} &= x_1^{(0)} + \frac{1}{A_{11}}\left(b_1 - A_{11}x_1^{(0)} - A_{12}x_2^{(0)}\right) \\
&= 1 + \frac{1}{2}(3.5 - 2\times 1 - 1\times 1) = 1.25 \\
x_2^{(1)} &= x_2^{(0)} + \frac{1}{A_{22}}\left(b_2 - A_{21}x_1^{(0)} - A_{22}x_2^{(0)}\right) \\
&= 1 + \frac{1}{4}(0.5 - (-1)\times 1 - 4\times 1) = 0.375.
\end{aligned}$$

So we have $\vec{x}^{(1)} = (1.25, 0.375)^T$. Then the second iteration is

$$\begin{aligned} x_1^{(2)} &= x_1^{(1)} + \frac{1}{A_{11}}\left(b_1 - A_{11}x_1^{(1)} - A_{12}x_2^{(1)}\right) \\ &= 1.25 + \frac{1}{2}(3.5 - 2 \times 1.25 - 1 \times 0.375) = 1.5625 \\ x_2^{(2)} &= x_2^{(1)} + \frac{1}{A_{22}}\left(b_2 - A_{21}x_1^{(1)} - A_{22}x_2^{(1)}\right) \\ &= 0.375 + \frac{1}{4}(0.5 - (-1) \times 1.25 - 4 \times 0.375) = 0.4375. \end{aligned}$$

So we have $\vec{x}^{(2)} = (1.5625, 0.4375)$.
Note the only difference between the formulae for Iteration 1 and 2 is the iteration number, the superscript in brackets. The exact solution is given by $\vec{x} = (1.5, 0.5)^T$.

We note that we can also slightly simplify the way the Jacobi iteration is written. We can expand A into $A = L + D + U$, where L and U are the parts of the matrix from below and above the diagonal respectively:

$$L_{ij} = \begin{cases} A_{ij} & \text{if } i < j \\ 0 & \text{if } i \geq j, \end{cases} \qquad U_{ij} = \begin{cases} A_{ij} & \text{if } i > j \\ 0 & \text{if } i \leq j. \end{cases}$$

The we can calculate that:

$$\begin{aligned} \vec{x}^{(k+1)} &= \vec{x}^{(k)} + D^{-1}(\vec{b} - A\vec{x}^{(k)}) \\ &= \vec{x}^{(k)} + D^{-1}(\vec{b} - (L + D + U)\vec{x}^{(k)}) \\ &= \vec{x}^{(k)} - D^{-1}D\vec{x}^{(k)} + D^{-1}(\vec{b} - (L + U)\vec{x}^{(k)}) \\ &= \vec{x}^{(k)} - \vec{x}^{(k)} + D^{-1}(\vec{b} - (L + U)\vec{x}^{(k)}) \\ &= D^{-1}(\vec{b} - (L + U)\vec{x}^{(k)}). \end{aligned}$$

In this formulation, we do not explicitly form the residual as part of the computations. In practical situations, this may be a simpler formulation we can use if we have knowledge of the coefficients of A, but this is not always true!

5.3 Gauss-Seidel iteration

As an alternative to Jacobi iteration, the iteration might use $x_i^{(k+1)}$ as soon as it is calculated (rather than using the previous iteration), giving

$$
\begin{aligned}
x_1^{(k+1)} &= x_1^{(k)} + \frac{1}{A_{11}}\left(b_1 - \sum_{j=1}^{n} A_{1j}x_j^{(k)}\right)\\
x_2^{(k+1)} &= x_2^{(k)} + \frac{1}{A_{22}}\left(b_2 - A_{21}x_1^{(k+1)} - \sum_{j=2}^{n} A_{2j}x_j^{(k)}\right)\\
x_3^{(k+1)} &= x_3^{(k)} + \frac{1}{A_{33}}\left(b_3 - \sum_{j=1}^{2} A_{3j}x_j^{(k+1)} - \sum_{j=3}^{n} A_{3j}x_j^{(k)}\right)\\
&\vdots \qquad \vdots\\
x_i^{(k+1)} &= x_i^{(k)} + \frac{1}{A_{ii}}\left(b_i - \sum_{j=1}^{i-1} A_{ij}x_j^{(k+1)} - \sum_{j=i}^{n} A_{ij}x_j^{(k)}\right)\\
&\vdots \qquad \vdots\\
x_n^{(k+1)} &= x_n^{(k)} + \frac{1}{A_{nn}}\left(b_n - \sum_{j=1}^{n-1} A_{nj}x_j^{(k+1)} - A_{nn}x_n^{(k)}\right).
\end{aligned}
$$

Consider the system $A\vec{x} = b$ with the matrix A split as $A = L + D + U$ where D is the diagonal of A, L contains the elements below the diagonal and U contains the elements above the diagonal. The componentwise iteration above can be written in matrix form as

$$
\begin{aligned}
\vec{x}^{(k+1)} &= \vec{x}^{(k)} + D^{-1}(\vec{b} - L\vec{x}^{(k+1)} - (D+U)\vec{x}^{(k)})\\
&= \vec{x}^{(k)} - D^{-1}L\vec{x}^{(k+1)} + D^{-1}(\vec{b} - (D+U)\vec{x}^{(k)})\\
&= \vec{x}^{(k)} - D^{-1}L\vec{x}^{(k+1)} + D^{-1}L\vec{x}^{(k)} + D^{-1}(\vec{b} - (L+D+U)\vec{x}^{(k)})\\
\vec{x}^{(k+1)} + D^{-1}L\vec{x}^{(k+1)} &= \vec{x}^{(k)} + D^{-1}L\vec{x}^{(k)} + D^{-1}(\vec{b} - (L+D+U)\vec{x}^{(k)})\\
D^{-1}(D+L)\vec{x}^{(k+1)} &= D^{-1}(D+L)\vec{x}^{(k)} + D^{-1}(\vec{b} - A\vec{x}^{(k)})\\
(D+L)\vec{x}^{(k+1)} &= DD^{-1}(D+L)\vec{x}^{(k)} + DD^{-1}(\vec{b} - A\vec{x}^{(k)})\\
&= (D+L)\vec{x}^{(k)} + (\vec{b} - A\vec{x}^{(k)})\\
\vec{x}^{(k+1)} &= (D+L)^{-1}(D+L)\vec{x}^{(k)} + (D+L)^{-1}(\vec{b} - A\vec{x}^{(k)})\\
&= \vec{x}^{(k)} + (D+L)^{-1}(\vec{b} - A\vec{x}^{(k)}).
\end{aligned}
$$

…and hence the **Gauss-Seidel** iteration

$$
\vec{x}^{(k+1)} = \vec{x}^{(k)} + (D+L)^{-1}(\vec{b} - A\vec{x}^{(k)}).
$$

That is, we use $P = (D+L)^{-1}$ in (5.1).

In general, we don't form the inverse of $D + L$ explicitly here since it is more complicated to do so than for simply computing the inverse of D.

Example 1

Take two iterations of Gauss-Seidel iteration to approximate the solution of the following system using the initial guess $\vec{x}^{(0)} = (1,1)^T$:

$$\begin{pmatrix} 2 & 1 \\ -1 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3.5 \\ 0.5 \end{pmatrix}$$

Starting from $\vec{x}^{(0)} = (1,1)^T$ we have
Iteration 1:

$$\begin{aligned} x_1^{(1)} &= x_1^{(0)} + \frac{1}{A_{11}}(b_1 - A_{11}x_1^{(0)} - A_{12}x_2^{(0)}) \\ &= 2 + \frac{1}{2}(3.5 - 1 \times 2 - 1 \times 1) = 2.25 \\ x_2^{(1)} &= x_2^{(0)} + \frac{1}{A_{22}}(b_2 - A_{21}x_1^{(1)} - A_{22}x_2^{(0)}) \\ &= 1 + \frac{1}{4}(0.5 - (-1) \times 2.25 - 4 \times 1) = 0.6875. \end{aligned}$$

Iteration 2:

$$\begin{aligned} x_1^{(2)} &= x_1^{(1)} + \frac{1}{A_{11}}(b_1 - A_{11}x_1^{(1)} - A_{12}x_2^{(1)}) \\ &= 1.25 + \frac{1}{2}(3.5 - 2 \times 1.25 - 1 \times 0.4375) = 1.53125 \\ x_2^{(2)} &= x_2^{(1)} + \frac{1}{A_{22}}(b_2 - A_{21}x_1^{(2)} - A_{22}x_2^{(1)}) \\ &= 0.4375 + \frac{1}{4}(0.5 - (-1) \times 1.53125 - 4 \times 0.4375) = 0.5078125. \end{aligned}$$

Again, note the changes in the iteration number on the right hand side of these equations, especially the differences against the Jacobi method.

- What happens if the initial estimate is altered to $\vec{x}^{(0)} = (2,1)^T$ (homework).

Example 2 (homework)

Take one iteration of (i) Jacobi iteration; (ii) Gauss-Seidel iteration to approximate the solution of the following system using the initial guess $\vec{x}^{(0)} = (1,2,3)^T$:

$$\begin{pmatrix} 2 & 1 & 0 \\ 1 & 3 & 1 \\ 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 10 \\ 6 \end{pmatrix}.$$

Note that the exact solution to this system is $x_1 = 2, x_2 = 2, x_3 = 2$.

Remark.

- Here both methods converge, but fairly slowly. They might not converge at all!
- We will discuss convergence and stopping criteria in the next lecture.
- The Gauss-Seidel iteration generally out-performs the Jacobi iteration.
- Performance can depend on the order in which the equations are written.
- Both iterative algorithms can be made faster and more efficient for sparse systems of equations (far more than direct methods).

5.4 Python version of iterative methods

```
def jacobi_iteration(A, b, x0, max_iter, verbose=False):
    """
    TODO
    """
    n = system_size(A, b)

    x = x0.copy()
    xnew = np.empty_like(x)

    if verbose:
        print("starting value: ", end="")
        print_array(x.T, "x.T")

    for iter in range(max_iter):
        for i in range(n):
            Axi = 0.0
            for j in range(n):
                Axi += A[i, j] * x[j]
            xnew[i] = x[i] + 1.0 / (A[i, i]) * (b[i] - Axi)
        x = xnew.copy()

        if verbose:
            print(f"after {iter=}: ", end="")
            print_array(x.T, "x.T")
```

```
    return x


def gauss_seidel_iteration(A, b, x0, max_iter, verbose=False):
    """
    TODO
    """
    n = system_size(A, b)

    x = x0.copy()
    xnew = np.empty_like(x)

    if verbose:
        print("starting value: ", end="")
        print_array(x.T, "x.T")

    for iter in range(max_iter):
        for i in range(n):
            Axi = 0.0
            for j in range(i):
                Axi += A[i, j] * xnew[j]
            for j in range(i, n):
                Axi += A[i, j] * x[j]
            xnew[i] = x[i] + 1.0 / (A[i, i]) * (b[i] - Axi)
        x = xnew.copy()

        if verbose:
            print(f"after {iter=}: ", end="")
            print_array(x.T, "x.T")

    return x
```

```
A = np.array([[2.0, 1.0], [-1.0, 4.0]])
b = np.array([[3.5], [0.5]])
x0 = np.array([[1.0], [1.0]])

print("jacobi iteration")
x = jacobi_iteration(A, b, x0, 5, verbose=True)
print()

print("gauss seidel iteration")
x = gauss_seidel_iteration(A, b, x0, 5, verbose=True)
```

```
print()
```

```
jacobi iteration
starting value: x.T = [  1.0,  1.0 ]
after iter=0: x.T = [  1.250,  0.375 ]
after iter=1: x.T = [  1.5625,  0.4375 ]
after iter=2: x.T = [  1.53125,  0.51562 ]
after iter=3: x.T = [  1.49219,  0.50781 ]
after iter=4: x.T = [  1.49609,  0.49805 ]

gauss seidel iteration
starting value: x.T = [  1.0,  1.0 ]
after iter=0: x.T = [  1.2500,  0.4375 ]
after iter=1: x.T = [  1.53125,  0.50781 ]
after iter=2: x.T = [  1.49609,  0.49902 ]
after iter=3: x.T = [  1.50049,  0.50012 ]
after iter=4: x.T = [  1.49994,  0.49998 ]
```

5.5 Sparse Matrices

We met sparse matrices as an example of a special matrix format when we first thought about systems of linear equations. Sparse matrices are very common in applications and have a structure which is very useful when used with iterative methods. There are two main ways in which sparse matrices can be exploited in order to obtain benefits within iterative methods.

- The storage can be reduced from $O(n^2)$.
- The cost per iteration can be reduced from $O(n^2)$.

Recall that a sparse matrix is defined to be such that it has at most αn non-zero entries (where α is independent of n). Typically this happens when we know there are at most α non-zero entries in any row.

The simplest way in which a sparse matrix is stored is using three arrays:

- an array of floating point numbers (`A_real` say) that stores the non-zero entries;
- an array of integers (`I_row` say) that stores the row number of the corresponding entry in the real array;
- an array of integers (`I_col` say) that stores the column numbers of the corresponding entry in the real array.

This requires just $3\alpha n$ units of storage - i.e. $O(n)$.

Given the above storage pattern, the following algorithm will execute a sparse matrix-vector multiplication ($\vec{z} = A\vec{y}$) in $O(n)$ operations:

```
z = np.zeros((n, 1))
for k in range(nonzero):
    z[I_row[k]] = z[I_row[k]] + A_real[k] * y[I_col[k]]
```

- Here `nonzero` is the number of non-zero entries in the matrix.
- Note that the cost of this operation is $O(n)$ as required.

5.5.1 Python experiments

```
def system_size_sparse(A_real, I_row, I_col, b):
    n = len(b)
    nonzero = len(A_real)
    assert nonzero == len(I_row)
    assert nonzero == len(I_col)

    return n, nonzero
```

First let's adapt our implementations to use this sparse matrix format:

```
def jacobi_iteration_sparse(A_real, I_row, I_col, b, x0, max_iter, verbose=False):
    """
    TODO
    """
    n, nonzero = system_size_sparse(A_real, I_row, I_col, b)

    x = x0.copy()
    xnew = np.empty_like(x)

    if verbose:
        print("starting value: ", end="")
        print_array(x.T, "x.T")

    # determine diagonal
    # D[i] should be A_{ii}
    D = np.zeros_like(x)
    for k in range(nonzero):
```

```
        if I_row[k] == I_col[k]:
            D[I_row[k]] = A_real[k]

    for iter in range(max_iter):
        # precompute Ax
        Ax = np.zeros_like(x)
        for k in range(nonzero):
            Ax[I_row[k]] = Ax[I_row[k]] + A_real[k] * x[I_col[k]]

        for i in range(n):
            xnew[i] = x[i] + 1.0 / D[i] * (b[i] - Ax[i])
        x = xnew.copy()

        if verbose:
            print(f"after {iter=}: ", end="")
            print_array(x.T, "x.T")

    return x


def gauss_seidel_iteration_sparse(A_real, I_row, I_col, b, x0, max_iter, verbose=False):
    """
    TODO
    """
    n, nonzero = system_size_sparse(A_real, I_row, I_col, b)

    x = x0.copy()
    xnew = np.empty_like(x)

    if verbose:
        print("starting value: ", end="")
        print_array(x.T, "x.T")

    for iter in range(max_iter):
        # precompute Ax using xnew if i < j
        Ax = np.zeros_like(x)
        for k in range(nonzero):
            if I_row[k] < I_col[k]:
                Ax[I_row[k]] = Ax[I_row[k]] + A_real[k] * xnew[I_col[k]]
            else:
                Ax[I_row[k]] = Ax[I_row[k]] + A_real[k] * x[I_col[k]]
```

```
        for i in range(n):
            xnew[i] = x[i] + 1.0 / (A[i, i]) * (b[i] - Ax[i])
        x = xnew.copy()

        if verbose:
            print(f"after {iter=}: ", end="")
            print_array(x.T, "x.T")

    return x
```

Then we can test the two different implementations of the methods:

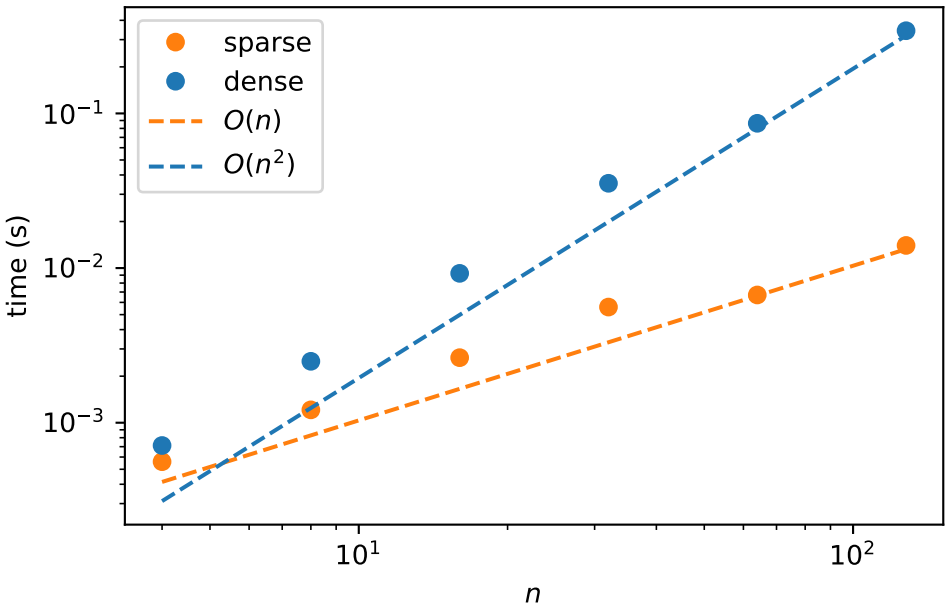
```
# random matrix
n = 4
nonzero = 10
A_real, I_row, I_col, b = random_sparse_system(n, nonzero)
print("sparse matrix:")
print("A_real =", A_real)
print("I_row = ", I_row)
print("I_col = ", I_col)
print()

# convert to dense for comparison
A_dense = to_dense(A_real, I_row, I_col)
print("dense matrix:")
print_array(A_dense)
print()

# starting guess
x0 = np.zeros((n, 1))

print("jacobi with sparse matrix")
x_sparse = jacobi_iteration_sparse(
    A_real, I_row, I_col, b, x0, max_iter=5, verbose=True
)
print()

print("jacobi with dense matrix")
x_dense = jacobi_iteration(A_dense, b, x0, max_iter=5, verbose=True)
print()
```

```
sparse matrix:
```

```
A_real = [10.5  -7.   12.25  8.   -6.   20.   -7.   24.25  8.   21.25 -6.   10.5 ]
I_row =  [0 0 0 1 1 1 1 2 2 3 3 3]
I_col =  [3 1 0 2 3 1 0 2 1 3 1 0]

dense matrix:
A_dense = [ 12.25, -7.00,  0.00, 10.50 ]
          [ -7.00, 20.00,  8.00, -6.00 ]
          [  0.00,  8.00, 24.25,  0.00 ]
          [ 10.50, -6.00,  0.00, 21.25 ]

jacobi with sparse matrix
starting value: x.T = [  0.0,  0.0,  0.0,  0.0 ]
after iter=0: x.T = [  1.28571,  0.75000,  1.32990,  1.21176 ]
after iter=1: x.T = [  0.67563,  1.03157,  1.08247,  0.78824 ]
after iter=2: x.T = [  1.19955,  0.78995,  0.98958,  1.16919 ]
after iter=3: x.T = [  0.73495,  1.12477,  1.06929,  0.84209 ]
after iter=4: x.T = [  1.20665,  0.83214,  0.95884,  1.16619 ]

jacobi with dense matrix
starting value: x.T = [  0.0,  0.0,  0.0,  0.0 ]
after iter=0: x.T = [  1.28571,  0.75000,  1.32990,  1.21176 ]
after iter=1: x.T = [  0.67563,  1.03157,  1.08247,  0.78824 ]
after iter=2: x.T = [  1.19955,  0.78995,  0.98958,  1.16919 ]
after iter=3: x.T = [  0.73495,  1.12477,  1.06929,  0.84209 ]
after iter=4: x.T = [  1.20665,  0.83214,  0.95884,  1.16619 ]
```

We see that we get the same results!

Now let's see how long it takes to get a solution. The following plot shows the run times of using the two different implementations of the Jacobi method. We see that, as expected, the run time of the dense formulation is $O(n^2)$ and the run time of the sparse formulation is $O(n)$.



We say "as expected" because we have already counted the number of operations per iteration and these implementations compute for a fixed number of iterations. In the next section, we look at alternative stopping criteria.

5.6 Convergence of an iterative method

We have discussed the construction of **iterations** which aim to find the solution of the equations $A\vec{x} = \vec{b}$ through a sequence of better and better approximations $\vec{x}^{(k)}$.

In general the iteration takes the form

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

here $\vec{x}^{(k)}$ is a vector of values and $\vec{F}$ is some vector-valued function which we have defined.

How can we decide if this iteration has converged? We need $\vec{x} - \vec{x}^{(k)}$ to be small, but we don't have access to the exact solution $\vec{x}$ so we have to do something else!

How do we decide that a vector/array is small? The most common measure is to use the "Euclidean norm" of an array (which you met last year!). This is defined to be the square root of the sum of squares of the entries of the array:

$$\|\vec{r}\| = \sqrt{\sum_{i=1}^{n} r_i^2}.$$

where $\vec{r}$ is a vector with n entries.

Examples

Consider the following sequence $\vec{x}^{(k)}$:

$$\begin{pmatrix}1\\-1\end{pmatrix}, \begin{pmatrix}1.5\\0.5\end{pmatrix}, \begin{pmatrix}1.75\\0.25\end{pmatrix}, \begin{pmatrix}1.875\\0.125\end{pmatrix}, \begin{pmatrix}1.9375\\-0.0625\end{pmatrix}, \begin{pmatrix}1.96875\\-0.03125\end{pmatrix}, \ldots$$

- What is $\|\vec{x}^{(1)} - \vec{x}^{(0)}\|$?
- What is $\|\vec{x}^{(5)} - \vec{x}^{(4)}\|$?

Let $\vec{x} = \begin{pmatrix}2\\0\end{pmatrix}$.

- What is $\|\vec{x} - \vec{x}^{(3)}\|$?
- What is $\|\vec{x} - \vec{x}^{(4)}\|$?
- What is $\|\vec{x} - \vec{x}^{(5)}\|$?

Rather than decide in advance how many iterations (of the Jacobi or Gauss-Seidel methods) to use stopping criteria:

- This could be a maximum number of iterations.
- This could be the *change* in values is small enough:

$$\|x^{(k+1)} - \vec{x}^{(k)}\| < tol,$$

- This could be the *norm of the residual* is small enough:

$$\|\vec{r}\| = \|\vec{b} - A\vec{x}^{(k)}\| < tol$$

In both cases, we call *tol* the **convergence tolerance** and the choice of *tol* will control the accuracy of the solution.

Discussion

What is a good convergence tolerance?

In general there are two possible reasons that an iteration may fail to converge.

- It may **diverge** - this means that $\|\vec{x}^{(k)}\| \to \infty$ as k (the number of iterations) increases, e.g.:

$$\begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}4\\2\end{pmatrix}, \begin{pmatrix}16\\4\end{pmatrix}, \begin{pmatrix}64\\8\end{pmatrix}, \begin{pmatrix}256\\16\end{pmatrix}, \begin{pmatrix}1024\\32\end{pmatrix}, \ldots$$

- It may *neither* converge nor diverge, e.g.:

$$\begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}2\\0\end{pmatrix}, \begin{pmatrix}3\\1\end{pmatrix}, \begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}2\\1\end{pmatrix}, \begin{pmatrix}3\\0\end{pmatrix}, \ldots$$

In addition to testing for convergence it is also necessary to include tests for failure to converge.

- Divergence may be detected by monitoring $\|\vec{x}^{(k)}\|$.
- Impose a maximum number of iterations to ensure that the loop is not repeated forever!

5.7 Summary

Many complex computational problems simply cannot be solved with today's computers using direct methods. Iterative methods are used instead since they can massively reduce the computational cost and storage required to get a "good enough" solution.

These basic iterative methods are simple to describe and program but generally slow to converge to an accurate answer - typically $O(n)$ iterations are required! Their usefulness for general matrix systems is very limited therefore - but we have shown their value in the solution of sparse systems however.

More advanced iterative methods do exist but are beyond the scope of this module - see Final year projects, MSc projects, PhD, and beyond!

5.8 Further reading

More details on these basic (and related) methods:

- Wikipedia: Jacobi method
- Wikipedia: Gauss-Seidel method
- Wikipedia: Iterative methods
  - see also Richardson method, Damped Jacobi method, Successive over-relaxation method (SOR), Symmetric successive over-relaxation method (SSOR) and Krylov subspace methods

More details on sparse matrices:

- Wikipedia Sparse matrix - including a long detailed list of software libraries support sparse matrices.
- Stackoverflow: Using a sparse matrix vs numpy array

- Jason Brownlee: A gentle introduction to sparse matrices for machine learning, Machine learning mastery

Some related textbooks:

- Jack Dongarra Templates for the solution of linear systems: Stopping criteria
- Jack Dongarra Templates for the solution of linear systems: Stationary iterative methods
- Golub, Gene H.; Van Loan, Charles F. (1996), Matrix Computations (3rd ed.), Baltimore: Johns Hopkins, ISBN 978-0-8018-5414-9.
- Saad, Yousef (2003). Iterative Methods for Sparse Linear Systems (2nd ed.). SIAM. p. 414. ISBN 0898715342.

Some software implementations:

- `scipy.sparse` custom routines specialised to sparse matrices
- SuiteSparse, a suite of sparse matrix algorithms, geared toward the direct solution of sparse linear systems
- `scipy.sparse` iterative solvers: Solving linear problems
- PETSc: Linear system solvers - a high performance linear algebra toolkit

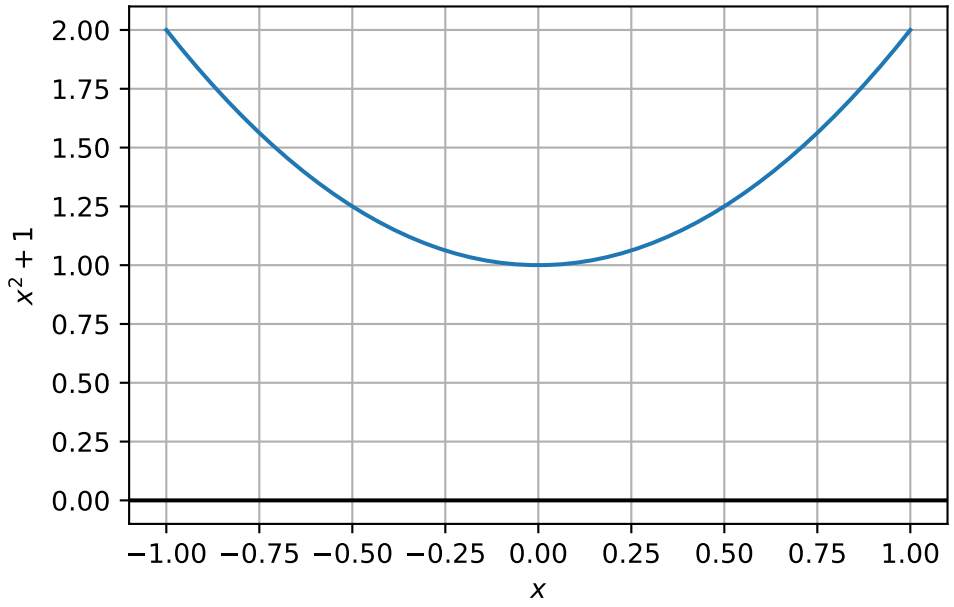
6 (Preliminary topic) Complex numbers

6.1 Basic definitions

A complex number is an element of a number system which extends our familiar real number system. Our motivation will be for finding eigenvalues and eigenvectors which is the next topic in this module. In order to find eigenvalues, we will need to find solutions of polynomial equations and it will turn out to be useful to *always* be able to get a solution to any polynomial equation.

Example 6.1. Consider the polynomial equation

$$x^2 + 1 = 0. \tag{6.1}$$



We can see that this equation has no solution over the real numbers.

The key idea of complex numbers is to create a new symbol, that we will call i, or the **imaginary unit** which satisfies:

$$i^2 = -1 \qquad \sqrt{-1} = i.$$

By taking multiples of this imaginary unit, we can create many more new numbers, like $3i, \sqrt{5}i$ or $-12i$. These are examples of **imaginary numbers**.

We form **complex numbers** by adding real and imaginary numbers whilst keeping each part separate. For example, $2 + 3i$, $\frac{1}{2} + \sqrt{5}i$ or $12 - 12i$.

Definition 6.1. Any number that can be written as $z = a + bi$ with a, b real numbers and i the imaginary unit are called *complex numbers*. In this format, we call a the **real part** of z and b the **imaginary part** of z.

We notice that all real numbers x must also be complex numbers since $x = x + 0i$.

Remark 6.1. Among the first recorded use of complex numbers in European mathematics is by an Italian mathematician Gerolamo Cardano in around 1545. He later described complex numbers as being "as subtle as they are useless" and "mental torture".

The term imaginary was coined by Rene Descartes in 1637:

> … sometimes only imaginary, that is one can imagine as many as I said in each equation, but sometimes there exist no quantity that matches that which we imagine.

Exercise 6.1. What are the real and imaginary parts of these numbers?

1. 3 + 6i
2. -3.5 + 2i
3. 5
4. 7i

6.2 Calculations with complex numbers

We can perform the basic operations, addition, subtraction, multiplication and division, on complex numbers.

Addition and subtraction is relatively straight forward: we simple treat the real and imaginary parts separately.

Example 6.2. We can compute that

$$(2+3i)+(12-12i) = (2+12)+(3-12)i = 14-9i$$
$$(2+3i)-(12-12i) = (2-12)+(3-(-12))i = -10+15i.$$

Exercise 6.2. Compute $(3+6i)+(-3.5+2i)$ and $(3+6i)-(-3.5+2i)$.

For multiplying and dividing, we first say that applying these operations between complex and real numbers again follows the usual rules. Let x be a real number and $z=(a+bi)$ be a complex number, then

$$x\times(a+bi) = (a+bi)\times x = (x\times a)+(x\times b)$$
$$\frac{a+bi}{x} = \frac{a}{x}+\frac{b}{x}i.$$

For multiplication between complex numbers, things are a bit harder. We expand out brackets and apply the rule that $i^2=-1$:

Example 6.3.

$$\begin{aligned}
&(2+3i)\times(12-12i)\\
&= 2\times 12+3i\times 12+2\times -12i+3i\times -12i && \text{(expand brackets)}\\
&= 2\times 12+(12\times 3)i+(2\times -12)i+(3\times -12)\times i^2 && \text{(rearrange)}\\
&= 24+36i-24i-36\times i^2 && \text{(compute products)}\\
&= 24+36i-24i+36 && \text{(use } i^2=-1\text{)}\\
&= 60+12i && \text{(collect terms).}
\end{aligned}$$

We see that we have a general formula:

$$(a+bi)\times(c+di) = (ac-bd)+(ad+bc)i.$$

Exercise 6.3. Compute $(3+6i)\times(-3.5+2i)$.

Division is harder - you may want to skip this on first reading since it is not so important for what follows in these notes. When we divide complex numbers, we try to rewrite the fraction to have a real denominator by "rationalising the denominator".

Example 6.4. Suppose we want to find $(2+3i)/(12-12i)$. Our idea is to find a numbers so that we can write

$$\frac{2+3i}{12-12i} = \frac{2+3i}{12-12i}\times\frac{z}{z} = \frac{(2+3i)z}{(12-12i)z} = \frac{\text{something}}{\text{something real}}.$$

The answer is to use $z = 12 + 12i$ - that is the denominator with the sign of the imaginary part flipped (we will give this a name later on).

We can compute that

$$\begin{aligned}
&(12 - 12i) \times (12 + 12i) \\
&= (12 \times 12) + (12 \times 12i) + (-12i \times 12) + (-12i \times 12i) && \text{(expand bracket)} \\
&= 12 \times 12 + (12 \times 12)i + (-12 \times 12)i + (-12 \times 12)i^2 && \text{(rearrange)} \\
&= 144 + 144i - 144i - 144i^2 && \text{(compute products)} \\
&= 144 + 144i - 144i + 144 && \text{(use } i^2 = -1\text{)} \\
&= 288 + 0i && \text{(collect terms).}
\end{aligned}$$

So we have that $(12 - 12i) \times (12 + 12i) = 288$ is a real number.

We continue by computing that

$$\begin{aligned}
&(2 + 3i) \times (12 + 12i) \\
&= (2 \times 12) + (2 \times 12i) + (3i \times 12) + (3i \times 12i) \\
&= 2 \times 12 + (2 \times 12)i + (3 \times 12)i + (3 \times 12)i^2 \\
&= 24 + 24i + 36i + 36i^2 \\
&= 24 + 24i + 36i - 36 \\
&= -12 + 60i.
\end{aligned}$$

Thus we infer that

$$\begin{aligned}
\frac{2 + 3i}{12 - 12i} &= \frac{2 + 3i}{12 - 12i} \times \frac{12 + 12i}{12 + 12i} \\
&= \frac{(2 + 3i)(12 + 12i)}{(12 - 12i)(12 + 12i)} \\
&= \frac{-12 + 60i}{288} \\
&= \frac{-12}{288} + \frac{60}{288}i \\
&= -\frac{1}{24} + \frac{5}{24}i.
\end{aligned}$$

To check we have not done anything silly, we should also check that $(12 + 12i)/(12 + 12i) = 1$. This is left as an exercise.

Exercise 6.4. Find $(3 + 6i)/(-3.5 + 2i)$ and $(12 + 12i)/(12 + 12i)$.

Remark 6.2. One thing to be careful of when considering products is that the identity $i^2 = -1$ appears to break one rule of arithmetic of square roots:

$$i^2 = (\sqrt{-1})^2 = \sqrt{-1}\sqrt{-1} \neq \sqrt{(-1) \times (-1)} = \sqrt{1} = 1.$$

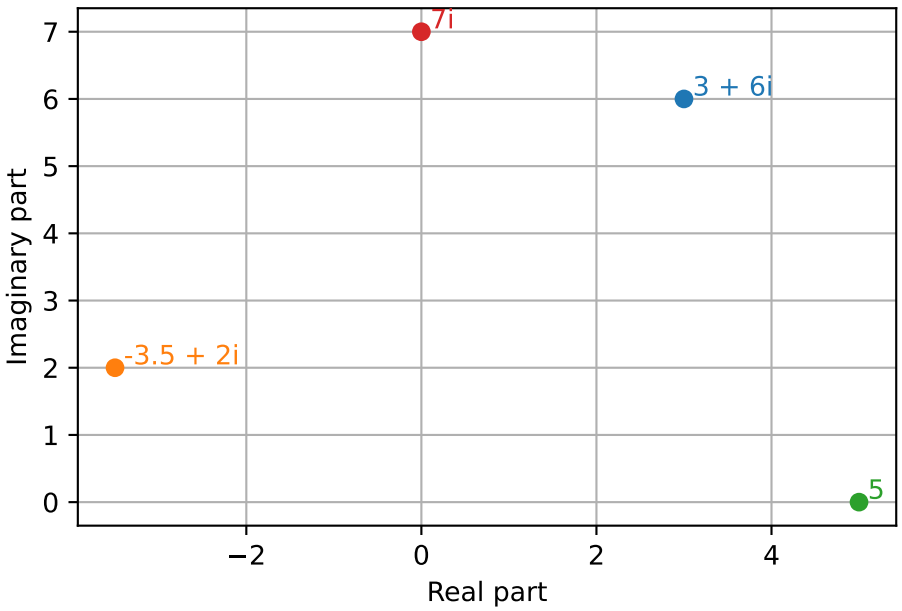
In fact, we have that $\sqrt{x}\sqrt{y} = \sqrt{xy}$ only if $x, y > 0$.

6.3 A geometric picture

The idea of adding complex numbers by considering real and imaginary parts separately is reminiscent of adding two dimensional vectors. For this reason, it is often helpful to think of complex numbers as points in *the complex plane*.

The complex plane is the two dimensional space formed by considering the real and imaginary parts of a complex number as two different coordinate axes.

Example 6.5.



We can see that adding complex numbers looks just like adding two dimensional vectors! We can also use this geometric picture to help with some further operations.

The complex conjugate of a complex number $z = a + bi$ is given by $\bar{z} = a - bi$. (The complex conjugate is that number we used before when working out how to divide complex numbers).

Example 6.6. The complex conjugate of $2+3i$ is $2-3i$. The complex conjugate of $12-12i$ is $12+12i$.

Exercise 6.5. Find the complex conjugates of $3+6i$ and $-3.5+2i$.

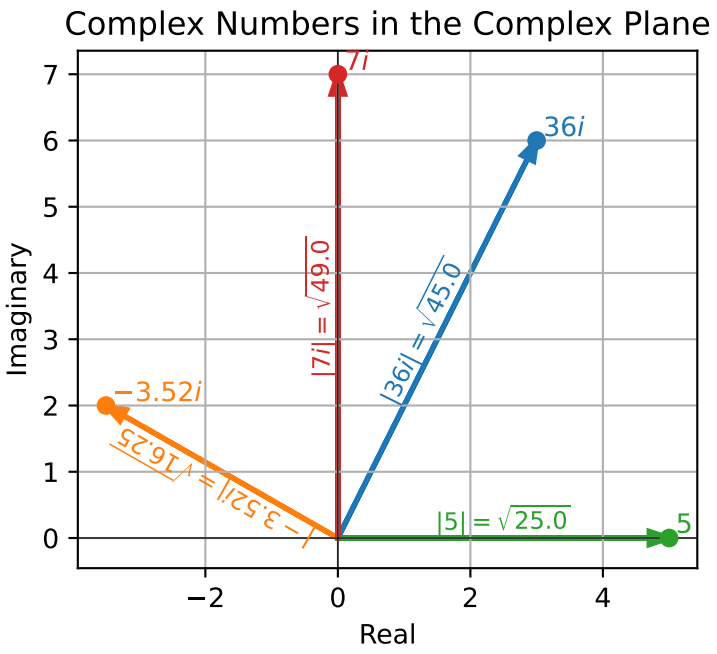
We have already seen that the complex conjugate of a complex number is helpful when performing division of complex numbers. The reason is that computing the product of a number and its conjugate always gives a real, positive number:

$$\begin{aligned}
&(a+bi)\times(a-bi)\\
&=(a\times a)+(a\times -bi)+(bi\times a)+(bi\times -bi)\\
&=(a\times a)+(a\times -b)i+(b\times a)i+(b\times -b)i^2\\
&=(a\times a)+(a\times -b+b\times a)i-(b\times -b)\\
&=a^2+b^2+0i.
\end{aligned}$$

In fact, we use this same calculation to define the **absolute value** (sometimes called the **modulus**) of a complex number $z=a+bi$

$$|z|=|a+bi|=\sqrt{a^2+b^2}=\sqrt{z\bar{z}}.$$

Example 6.7.



Exercise 6.6. Find the value of

$$|3+6i| \quad \text{and} \quad |-3.5+2i|. \tag{6.2}$$

Consider two complex numbers $z = a + bi$ and $y = c + di$. Then, we have already seen that

$$zy = (a+bi) \times (c+di) = (ac-bd) + (ad+bc)i.$$

We can compute the square of the modulus of the product zy as

$$\begin{aligned} |zy|^2 &= (ac-bd)^2 + (ad+bc)^2 \\ &= a^2c^2 - 2abcd + b^2d^2 + a^2d^2 + 2abcd + b^2c^2 \\ &= a^2c^2 + b^2d^2 + a^2d^2 + b^2c^2 \\ &= (a^2+b^2)(c^2+d^2), \end{aligned}$$

and we have computed that $|zy| = |z||y|$.

In particular, if y has modulus 1, then $|zy| = |z|$. This means that zy and z are the same distance from the origin but 'point' in different directions. We can write the real and imaginary parts as $y = c + di = \cos(\theta) + i\sin(\theta)$, where θ is the angle between the positive real axis and the line between 0 and y. Then

$$\begin{aligned} zy &= (ac-bd) + (ad+bc)i \\ &= (a\cos(\theta) - b\sin(\theta)) + (a\sin(\theta) + b\cos(\theta))i. \end{aligned}$$

Recalling the example of a rotation matrix from Lecture ??, we see that multiplying by y is the same as rotating the complex point (z) by an angle of θ radians in the anticlockwise direction.

This leads us to thinking *polar coordinates* for the complex plane. Polar coordinates are a different form of coordinates that replace the usual x and y-directions (up and across) by two values which represent the distance to the origin (that we call radius) and angle to the positive x-axis (that we call the angle). When talking about a complex numbers z represented in the complex plane, we know that the modulus $|z|$ represents the radius. The idea of θ above represents the angle of a complex number that we know call the **argument**.

Definition 6.2. Let z be a complex number. The **polar form** of z is $R(cos\theta + i\sin\theta)$. We call R the modulus of z and θ is the argument of z.

The representation of the angle only unique up to adding integer multiples of 2π, since rotating a point by 2π about the origin leaves it unchanged.

Example 6.8. Let $z = 12 - 12i$. Then

$$|z| = |12 - 12i| = \sqrt{12^2 + 12^2} = \sqrt{2 \times 144} = 12\sqrt{2}.$$

We have $\arg z = -\pi/4$ since

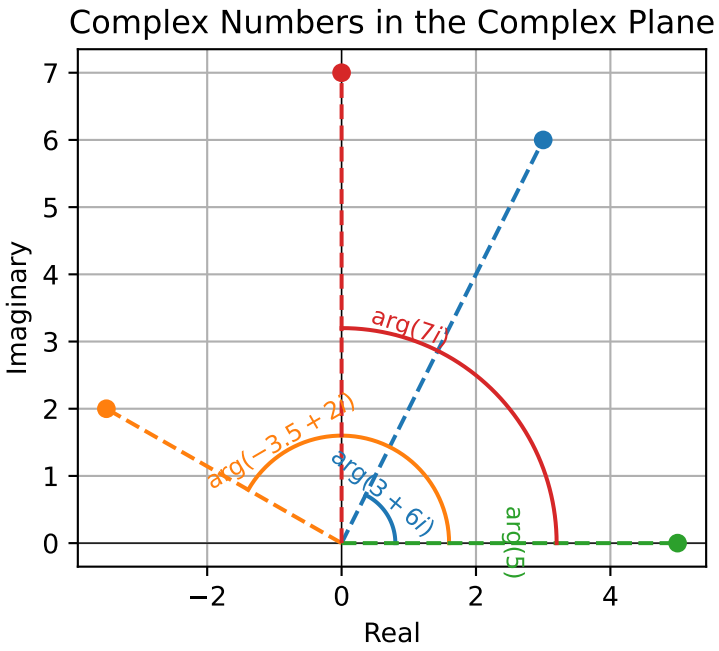
$$\cos(-\pi/4) = \frac{1}{\sqrt{2}}, \quad \text{and} \quad \sin(-\pi/4) = \frac{-1}{\sqrt{2}},$$

so

$$12\sqrt{2}(\cos(-\pi/4) + i\sin(-\pi/4)) = 12\sqrt{2}\left(\frac{1}{\sqrt{2}} + i\frac{-1}{\sqrt{2}}\right) = 12 - 12i.$$

Exercise 6.7. Compute the modulus and argument of 2, $3i$ and $4 + 4i$.

Example 6.9.



The polar representation of complex numbers then gives us a nice way to understand multiplication of complex numbers. If $y \neq 0$, then we can check that $\left|\frac{y}{|y|}\right| = 1$ and $\arg y = \arg \frac{y}{|y|}$. Then writing $zy = z\frac{y}{|y|}|y|$, we can use our calculations above to infer that multiplying by y corresponds to a anti-clockwise rotation by $\arg y$ then scaling by $|y|$.

Exercise 6.8. Check that for any non-zero complex number y, that $\left|\frac{y}{|y|}\right| = 1$ and $\arg y = \arg \frac{y}{|y|}$.

6.4 Solving polynomial equations

As we have mentioned above, we will be using complex numbers when solving polynomial equations to work out eigenvalues and eigenvectors of matrices later in the section. The reason complex numbers are useful here is this very important Theorem:

Theorem 6.1 (The Fundamental Theorem of Algebra)**.** *For any complex numbers $a_0, \dots, a_n$ not all zero, there is at least one complex number z which satisfies:*

$$a_n z^n + \cdots + a_1 z + a_0 = 0$$

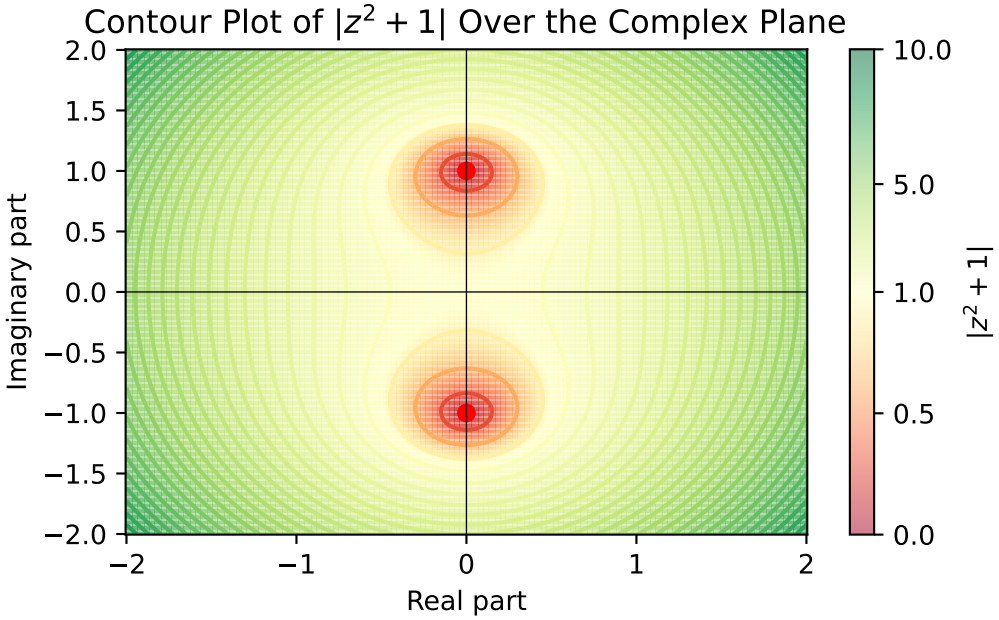
It is really important to note here that this is not true if we want z to be a real number. Let's revisit Example 6.1.

Example 6.10. Consider the polynomial equation

$$x^2 + 1 = 0. \tag{6.3}$$

We saw before that this equation has no solution over the real numbers, but the Fundamental Theorem of Algebra tells us there must be at least one solution which is a complex number. In fact it has two solutions - i and $-i$:

$$i^2 + 1 = 0$$
$$(-i)^2 + 1 = (-1)^2 i^2 + 1 = i^2 + 1 = 0.$$



Notice that along the real line (Imaginary part = 0), the value of the function is always above 1.

To find complex roots of other quadratic equations, we can apply the quadratic formula:

Example 6.11. To find the values of z which satisfy $z^2 - 2z + 2 = 0$, we see:

$$z = \frac{+2 \pm \sqrt{(-2)^2 - 4 \times 1 \times 2}}{2} = \frac{2 \pm \sqrt{-4}}{2} = \frac{2 \pm 2\sqrt{-1}}{2} = 1 \pm i.$$

Exercise 6.9. Find the value of z which satisfy $z^2 - 4z + 20$.

We will see in later sections that although this is one possible solution to compute the eigenvalues for 2×2 matrices this approach becomes impossible for larger size matrices and we need another approach!

7 Linear independence, span, basis

8 Eigenvectors and eigenvalues