# COMP2870 Theoretical Foundations: Linear Algebra

Thomas Ranner

16 September 2025

# Table of contents

# 1 Introduction

## 1.1 Contents of this submodule

This part of the module will deal with numerical algorithms that involve matrices. The study of this type of problem is called *linear algebra*. We will approach these problems using a combination of theoretical ideas and practical solutions, thinking through the lens of real-world applications. As a consequence, to succeed in linear algebra, you will do some programming (using Python) and some pen-and-paper theoretical work, too.

### 1.1.1 Topics

We will have 7 double lectures, 3 tutorials, and 3 labs. We break up the topics as follows:

Lectures

1. Introduction and motivation, key problem statements (week 4, Mon)
2. When can we solve systems of linear equations? (week 4, Wed)
3. Direct methods for systems of linear equations (week 5, Mon)
4. Iterative solution of linear equations (week 5, Wed)
5. Complex numbers (week 6, Mon)
6. Eigenvalues and eigenvectors (week 6, Wed)
7. Practical solutions for eigenvalues and eigenvectors / Summary (week 7, Mon)

Labs

1. Floating point numbers (week 4)
2. When can we solve systems of linear equations? (week 5)
3. Systems of linear equations (week 6)
4. Eigenvalues and eigenvectors (week 7)

Tutorials (how do the dates work out)

1. Linear independence, span, basis (week 5)
2. Solution of linear systems (week 6)
3. Eigenvalue and eigenvectors (week 7)

Later this term, week 11, you will complete a project based on the things you've learnt from this section of the module.

### 1.1.2 Learning outcomes

Candidates should be able to:

- explain practical challenges working with floating-point numbers;
- define and identify what it means for a set of vectors to be a basis, spanning set or linearly independent;
- apply direct and iterative solvers to solve systems of linear equations; implement methods using floating point numbers and investigate computational cost using computer experiments;
- apply algorithms to compute eigenvectors and eigenvalues of large matrices.

## 1.2 Textbooks and other resources

There are many textbooks and other external resources which could help your learning:

- Introduction to Linear Algebra (Fifth Edition), Gilbert Strang, Wellesley-Cambridge Press, 2016. with MIT course material. *Strongly recommended book and YouTube lecture series*
- Scientific Computing: An Introductory Survey, T.M. Heath, McGraw-Hill, 2002. Some lecture notes based on the book
- Engineering Mathematics, K.A. Stroud, Macmillan, 2001. *available online*
- Numerical Recipes in C++/C/FORTRAN: The Art of Scientific Computing, W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, Cambridge University Press, 2002/1993/1993. *A very practical view of the methods we develop in this module which could be used for library development*

Other links and resources are given at the end of each section of the notes.

## 1.3 Programming

This section of the notes links theoretical material with practical applications. We will have weekly lab sessions where you will see how the methods mentioned in the lecture notes work when implemented. More instructions on how we will do this will be covered in the lab sessions themselves.

An important theoretical aspect of translating the algorithms in linear algebra to computer implementations is the use of floating-point numbers. You will have already met floating-point numbers in your first year studies where you met how computer store numbers. You will already know that computer cannot store every possible real number exactly. The inexactness of floating-point numbers has real consequences when performing linear algebra computations.

Exploring and understanding the material in (Chapter 2) is really helpful for understanding many of the choices that we make when forming methods in linear algebra.

We will make extensive use of `Python` and `numpy` during this section of the module. It may help you to revise some of your notes from last year on these topics.

## 1.4 The big problems in linear algebra

We will cover two big linear algebra problems in this section of the module. Linear algebra can be defined as the study of problems involving matrices and vectors.

### 1.4.1 Reminder of matrices and vectors

There are two important objects we will work with that were defined in your first-year Theoretical Foundations module (COMP1870).

**Definition 1.1.** 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 1.1.** $A$ is a $4 \times 4$ matrix and $B$ is a $3 \times 4$ 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 1.1.**

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).

When considering systems of linear equations, the entries of the matrix will always be real numbers (later we will explore using complex numbers (Chapter 6) too)

**Definition 1.2.** 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 1.2.** $\vec{a}$ 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 1.2.**

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, $B\vec{b}$.

### 1.4.2 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:

$$A\vec{x} = \vec{b}. \tag{1.1}$$

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.

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

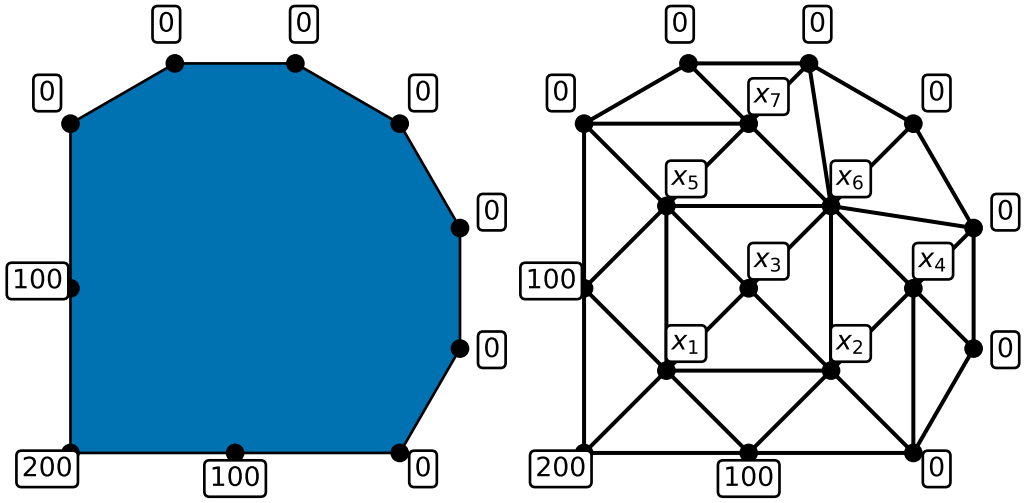



Figure 1.1: Image showing temperature sample points and relations in a room.

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

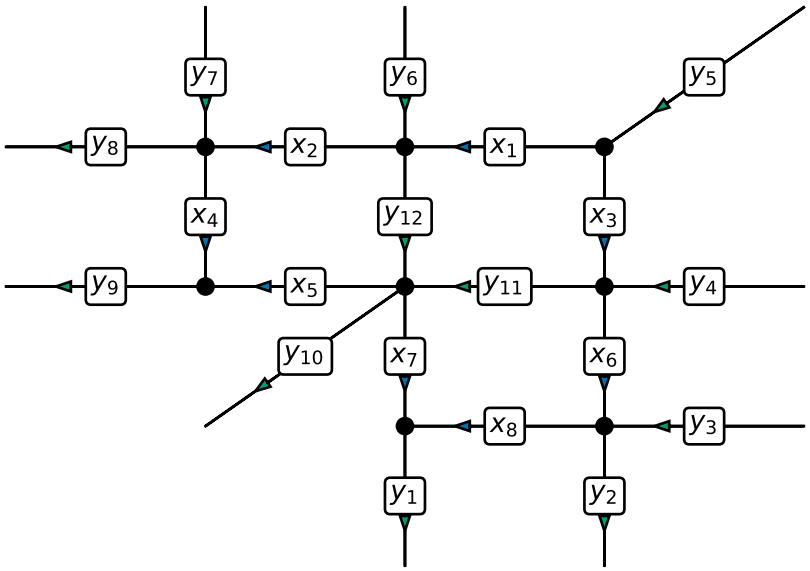



Figure 1.2: Example network showing traffic flow in a city

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

This example leads to the system:

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

### 1.4.3 Eigenvalues and eigenvectors

For this problem, we will think of a matrix $A$ acting on functions $\vec{x}$:

$$\vec{x} \mapsto A\vec{x}.$$

We are interested in when is the output vector $A\vec{x}$ is *parallel* to $\vec{x}$?

**Definition 1.3.** We say that any vector $\vec{x}$, where $A\vec{x}$ is parallel is $\vec{x}$, is called an *eigenvector* of $A$. Here by parallel, we mean that there exists a number $\lambda$ (can be positive, negative or zero) such that

$$A\vec{x} = \lambda\vec{x}. \tag{1.2}$$

We call the associated number $\lambda$ an *eigenvalue* of $A$.

We will later see that an $n \times n$ square matrix always has $n$ eigenvalues (which may not always be distinct).

To help with our intuition here, we start with some simple examples:

**Example 1.5.** Let $A$ be the $2 \times 2$ matrix that scales any input vector by $a$ in the $x$-direction and by $b$ in the $y$-direction. We can write this matrix as

$$A = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix},$$

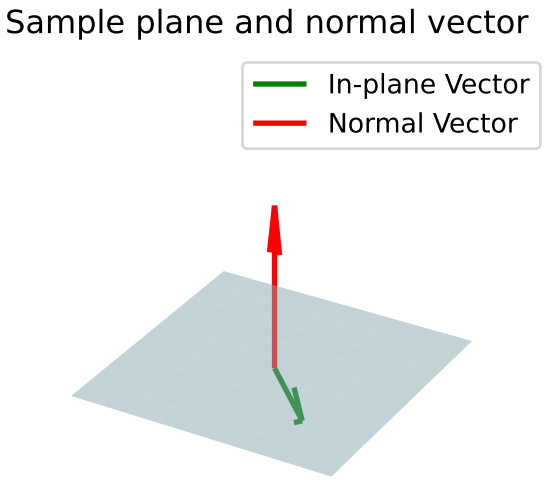
since then for a 2-vector $\vec{x} = (x, y)^T$, we have

$$A\vec{x} = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + 0y \\ 0x + by \end{pmatrix} = \begin{pmatrix} ax \\ by \end{pmatrix}.$$

Then, we infer we have two eigenvalues and two eigenvectors: One eigenvalue is $a$ with eigenvector $(1,0)^T$ and the other is $b$ with eigenvector $(0,1)^T$ since:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} a \\ 0 \end{pmatrix} = a \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$
$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix} = b \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

**Example 1.6.** Let $P$ be the 3x3 matrix that represents projection onto a plane $\pi$. What are the eigenvalues and eigenvectors of $p$?




- If $\vec{x}$ is in the plane $\Pi$, then $P\vec{x} = \vec{x}$. This means that $\vec{x}$ is an eigenvector and the associated eigenvalue is 1.
- If $\vec{y}$ is perpendicular to the plane $\Pi$, then $P\vec{y} = \vec{0}$. This means that $\vec{y}$ is an eigenvector and the associated eigenvalue is 0.

Let $\vec{y}$ be perpendicular to $\Pi$ (so that $P\vec{y} = \vec{0}$ and $\vec{y}$ is an eigenvector of $P$), then for any number $s$, we can compute

$$P(s\vec{y}) = sP\vec{y} = s\vec{0} = \vec{0}.$$

This means that $s\vec{y}$ is also an eigenvector of $P$ associated to the eigenvalue 0. As a consequence, when we compute eigenvectors, we need to take care to *normalise* the vector to ensure we get a unique answer.

We see we end up with a two-dimensional space of eigenvectors (i.e., the plane $\Pi$) associated to eigenvalue 1 and a one-dimensional space of eigenvectors (i.e., the line perpendicular to $\Pi$) eigenvalue 0. We use the term *eigenspace* the space of eigenvectors associated to a particular eigenvalue.

**Example 1.7.** Let $A$ be the permuatation matrix which takes an input two-vector and outputs a two-vector with the components swapped. The matrix is given by

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

What are the eigenvectors and eigenvalues of $A$?

- Let $\vec{x} = (1, 1)^T$, then swapping the components of $\vec{x}$ gives back the same vector $\vec{x}$. In equations, we can write $A\vec{x} = \vec{x}$. This means that $\vec{x}$ is an eigenvector and the eigenvalue is 1.
- Let $\vec{x} = (-1, 1)^T$, then swapping the components of $\vec{x}$ gives back $(1, -1)^T$, which we can see is $-\vec{x}$. In equations, we can write $A\vec{x} = -\vec{x}$. This means that $\vec{x}$ is an eigenvector of $A$ and the associated eigenvalue is $-1$.

Here we see that again we actually have two one-dimensional eigenspaces.

## 1.5 Comments on these notes

There are two versions of these notes available: as an online website and as a pdf. I expect minor adjustments to be made to both throughout the progress of delivering the materials.

You can always find the latest versions at:

- https://comp2870-2526.github.io/linear-algebra-notes/ (html version)
- https://comp2870-2526.github.io/linear-algebra-notes/COMP2870-Theoretical-Foundations--Linear-Algebra.pdf (pdf version)

Please either email me (T.Ranner@leeds.ac.uk) or use the github repository to report any corrections.

`This version of the notes is 7e259f8.`

# 2 Floating point number systems

> Module learning objective
>
> Explain practical challenges working with floating-point numbers.

This topic will not be presented explicitly in lectures; instead, you will study the material yourself in Lab Session 1 (**?@sec-lab1**).

## 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 learned how to store numbers as floating-point numbers last year in the module COMP1860 Building our Digital World.

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 fewer 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.
- $\beta$ is the **base** (always a positive integer).
- $e$ is the integer **exponent** and is bounded ($L \leq e \leq U$).

$(\beta, t, L, U)$ fully defines a finite precision number system.

**Normalised** finite precision systems will be considered here for which

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

**Example 2.1.**

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 represented 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'>
```

You can create double-precision floating point numbers without using `numpy` (i.e. 64 bit with the same $\beta, t, L, U$ as above) by calling the `float` function, or just giving the number in decimal format:

```
(np.double(5.1) == 5.1, np.double(5.1) == 5.1)
```

```
(np.True_, np.True_)
```

However, in some extreme edge cases, you are unlikely to meet, Python floats do not follow the IEEE double-precision standard in some extreme edge cases.

**Example 2.2.** 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 difference 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 2.3** (What about in Python)**.** We find that even with double-precision floating point numbers, we see some funniness when working with decimals:

```
a = np.double(0.0)

# add 0.1 ten times to get 1?
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 $\beta^{-d}$, then $fl(x)$ has $d$ correct significant digits.

In the number system given by $(\beta, t, L, U)$, 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 $y > x \geq .100 \ldots 00 \times \beta^e = \beta^{e-1}$ 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

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

```
2.220446049250313e-16
```

**Example 2.4.**

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 can store!

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

smaller = eps
for _ 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 approach means that arithmetic cannot be completely trusted and the usual rules do not necessarily apply!

**Example 2.5.** 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!)

**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

Module learning objective

Define and identify what it means for a set of vectors to be a basis, spanning set or linearly independent.

## 3.1 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.

## 3.2 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:
$$\vec{x} = A^{-1}\vec{b}.$$
However, in general, this is a *terrible* idea and we will see algorithms that are more efficient than finding the inverse of $A$.

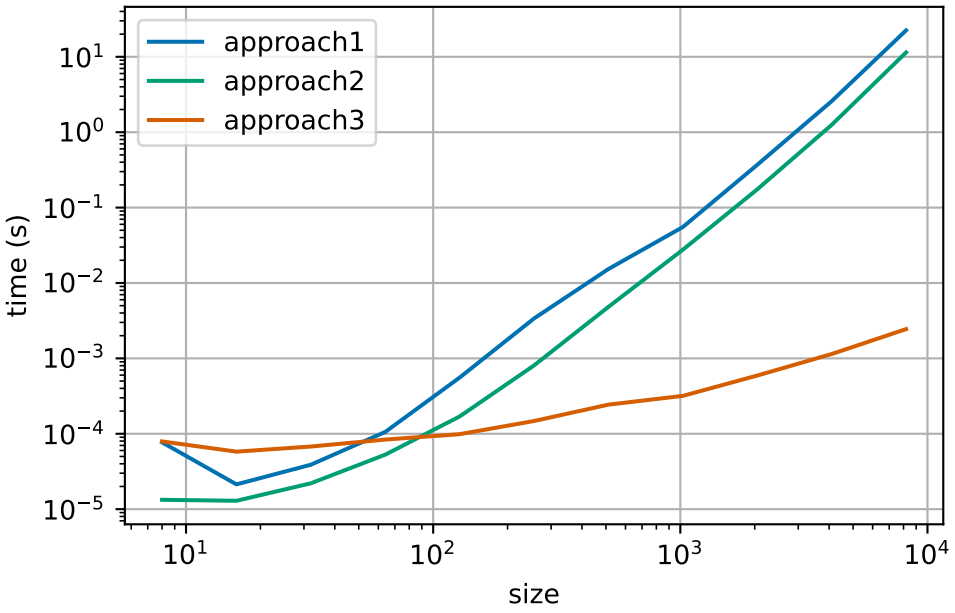
*Remark* 3.1. One way to solve a system of linear equations is to compute the inverse of $A$, $A^{-1}$, directly, then the solution is found through matrix multiplication: $\vec{x} = A^{-1}\vec{b}$. Here we compare two simple and one more specialised implementations:

```
def approach1(A, b):
    """
    Solve the system of linear equations Ax = b by applying the inverse of A
    """
    A_inv = np.linalg.inv(A)
    x = A_inv @ b
    return x


def approach2(A, b):
    """
    Solve the system of linear equations Ax = b through numpy's linear solver
    """
    x = np.linalg.solve(A, b)
    return x


def approach3(A_sparse, b):
    """
    Solve the system of linear equations Ax = b through scipy's sparse linear
```

```
    solver
    """
    x = sp.sparse.linalg.spsolve(A_sparse, b)
    return x
```




There are tools to help us determine when a matrix is invertible which arise naturally when considering 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 $3 \times 3$-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 that 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*.

### 3.2.1 The span of a set of vectors

**Definition 3.1.** Given a set of vectors of the same size, $S = \{\vec{v}_1, \ldots, \vec{v}_k\}$, we say the *span* of $S$ is the set of all vectors which are linear combinations of vectors in $S$:

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

**Example 3.1.** Consider three new vectors

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

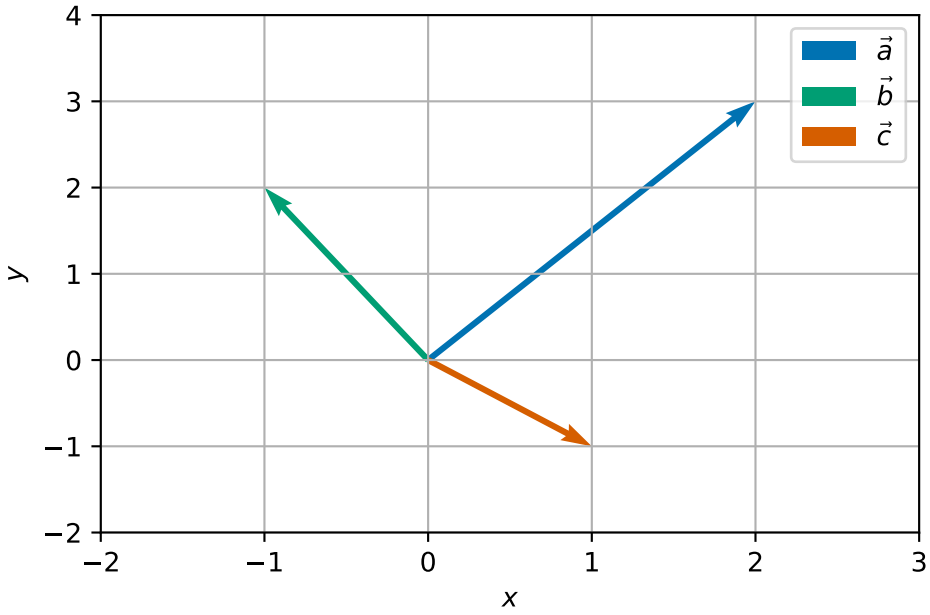



Figure 3.1: Three example vectors in a plane.

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

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

   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 from (3.6), 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.

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 3.1, Part 3, we are not guaranteed that the linear combination is unique! For this we need a further condition.

### 3.2.2 Linear independence

**Definition 3.2.** Given a set of vectors of the same size, $S = \{\vec{v}_1, \dots, \vec{v}_k\}$, we say that $S$ is *linearly dependent*, if there exist numbers $x_1, x_2, \dots x_k$, 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 3.1.** Can you write the definition of a linearly independent set of vectors explicitly?

**Example 3.2.** Continuing from Example 3.1.

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,

$$\begin{aligned} 2x_1 - x_2 &= 0 \qquad & 3x_1 + 2x_2 &= 0 \end{aligned}$$

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! The ideas of linear independence and spanning lead us to the final definition of this section.

### 3.2.3 When vectors form a basis

**Definition 3.3.** 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 3.3.**

1. From Example 3.1, 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. Later, when we work with at eigenvectors and eigenvalues we will see that there are other convenient 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 3.4.**

1. From the main examples TODO (@todo-label) 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 is 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 $\vec{y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \in \mathbb{R}^2$

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

*Proof of Theorem 3.1.* 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.

### 3.2.4 Another characterisation: the determinant

Let $\vec{e}_j$ be the (column) vector in $\mathbb{R}^n$ which has 0 for all coefficients apart from the $j$th place:

$$\vec{e}_j = (0, \dots, 0, \underbrace{1}_{j\text{th place}}, 0, \dots, 0).$$

Then, we can compute that for any $n \times n$ matrix $A$ that we have the $i$th component of $A\vec{e}_j$ is given by

$$(A\vec{e}_j)_i = \sum_{k=1}^{n} A_{ik}(\vec{e}_j)_k = A_{ij}.$$

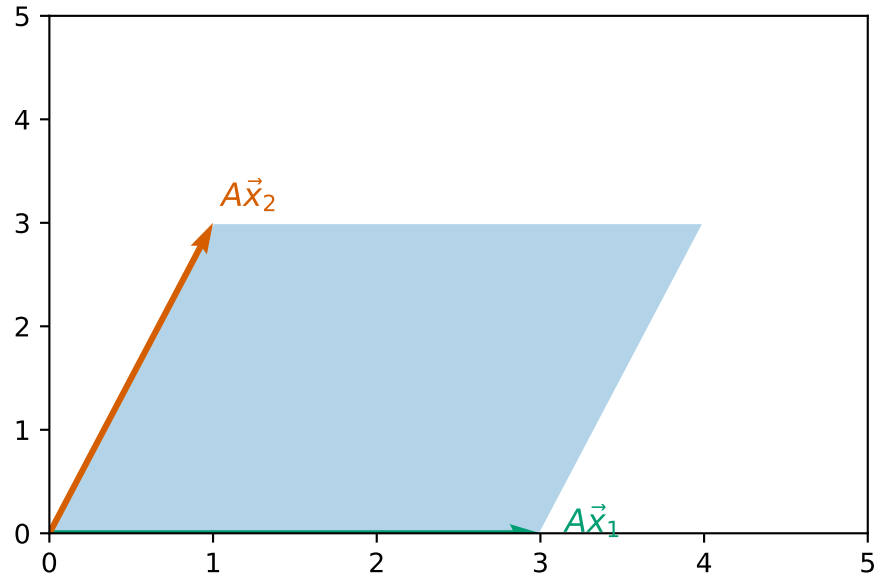
That is that $A\vec{e}_j$ gives the $j$th column of the matrix $A$.

One geometric way to see that the columns of $A$ form a basis is to explore the shape of polytope with edges away from the origin given by the columns of $A$ (hyper-parallelopiped).

**Example 3.5.** For $A$ given by

$$A = \begin{pmatrix} 3 & 1 \\ 0 & 3 \end{pmatrix}$$

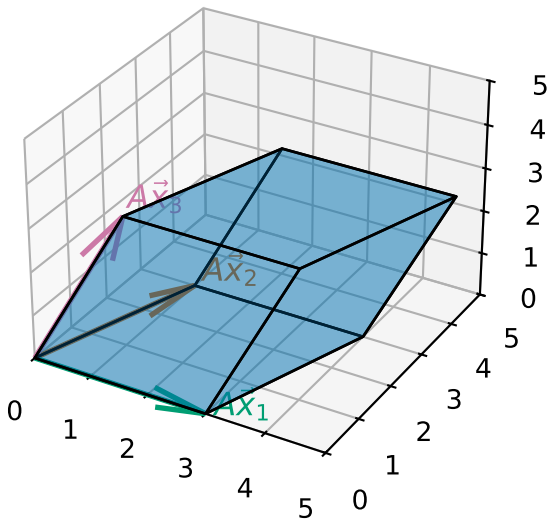
the shape is a parallelogram that looks like this:




For $A$ given by

$$A = \begin{pmatrix} 3 & 1 & 1 \\ 0 & 3 & 1 \\ 0 & 0 & 3 \end{pmatrix},$$

the shape is a parallelopiped given that looks like this:




If we are given an $n \times n$ matrix $A$, its $n$ columns form a basis of $\mathbb{R}^n$ if, and only if, the area/volume/hyper-volume of the associated hyper-parallelepiped is nonzero. Roughly speaking this follows since if the area/volume/hyper-volume is zero then two of the column vectors must point in the same direction. This property is so important that the area/volume/hyper-volume of the hyper-parallelepiped associated with the matrix $A$ has a special name: the *determinant* of $A$ and we write $\det A$.

**Definition 3.4.** Let $A$ be a square $n \times n$ matrix.

If $n = 2$,

$$\det A = \det \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = a_{11}a_{22} - a_{21}a_{12}. \tag{3.10}$$

If $n = 3$,

$$\begin{aligned} \det A &= \det \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \\ &= a_{11}(a_{22}a_{33} - a_{23}a_{32}) - a_{12}(a_{21}a_{33} - a_{23}a_{31}) + a_{13}(a_{21}a_{32} - a_{22} - a_{31}). \end{aligned}$$

For general $n$, the determinant can be found by, for example, Laplace expansions

$$\det A = \sum_{j=1}^{n} (-1)^{j+1} a_{1,j} m_{1,j},$$

where $a_{1,j}$ is the entry of the first row and $j$th column of $A$ and $m_{1,j}$ is the determinant of the submatrix obtained by removing the first row and the $j$th column from $A$.

**Example 3.6.**

1. Let $A$ be given by
$$A = \begin{pmatrix} 3 & 1 \\ 0 & 3 \end{pmatrix}.$$
Then we can compute that
$$\det A = 3 \times 3 - 1 \times 0 = 9 - 0 = 9.$$

2. Let $A$ be given by
$$A = \begin{pmatrix} 3 & 1 & 1 \\ 0 & 3 & 1 \\ 0 & 0 & 3 \end{pmatrix}.$$
Then we can compute that
$$\begin{aligned} \det A &= 3 \times (1 \times 3 - 0 \times 3) - 1 \times (0 \times 3 - 3 \times 0) + 1 \times (0 \times 0 - 3 \times 0) \\ &= 3 \times 3 - 1 \times 0 + 1 \times 0 = 9. \end{aligned}$$

**Exercise 3.2.** Compute the determinant of
$$\begin{pmatrix} 2 & -1 & 3 \\ 3 & 2 & -4 \\ 5 & 1 & 1 \end{pmatrix}.$$

We have been inaccurate in what we have said before. Strictly speaking the determinant we have defined here is the *signed* area/volume/hyper-volume of the associated hyper-parallelepiped, and we can recover the actual volume by taking $|\det A|$.

**Theorem 3.3.** *Let $A$ be an $n \times n$ matrix and $\vec{b}$ be an $n$-column vector. The following are equivalent:*

1. *$A\vec{x} = \vec{b}$ has a unique solution.*
2. *The columns of $A$ form a basis of $\mathbb{R}^n$.*
3. *$\det A \neq 0$.*

This proof is beyond the scope of this course.

We do have the following rule for manipulating determinants too:

- Let $A, B$ be square $n \times n$-matrices then $\det(AB) = \det A \det B$.

- Let $A$ be a square $n \times n$-matrix and $c$ a real number then $\det(cA) = c^n \det A$.
- Let $A$ be an invertible square matrix, then $\det(A^{-1}) = \frac{1}{\det A}$.

**Exercise 3.3.** Prove these three rules for manipulating determinants for $2 \times 2$ matrices by direct calculation.

*Remark* 3.2. Computing determinants numerically is an important topic in it's own right but we do not focus on this in this module.

You can use `numpy` to compute determinants

```
A = np.array([[3.0, 1.0], [0.0, 3.0]])
np.linalg.det(A)
```

```
np.float64(9.000000000000002)
```

## 3.3 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 3.7** (Triangular matrix)**.** One common (and important) structure takes the form

$$A = \begin{pmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots & 0 \\ a_{31} & a_{32} & a_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix}.$$

- 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.

- Note that the determinant of a triangular matrix is simply the product of diagonal coefficients:
$$\det A = a_{11}a_{22}\cdots a_{nn} = \prod_{i=1}^{n} a_{ii}.$$

**Example 3.8** (Sparse matrices)**. Sparse matrices** are prevalent in any application which relies on some form of *graph* structure (see both the temperature, Example 1.3, and traffic network examples, Example 1.4).

- 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 $\alpha n$ 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.

## 3.4 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 equations

Module learning objective

Apply direct and iterative solvers to solve systems of linear equations; implement methods using floating point numbers and investigate computational cost using computer experiments.

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.

It is important to note that Gaussian elimination should look very familiar to you - this looks a lot like solving simultaneous equations at high school. Here, we are trying to codify this approach into an algorithm that can computer can apply without any further human input in a way that always 'works' (or at least works when it should!).

## 4.1 Reminder of the problem

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:

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

**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.

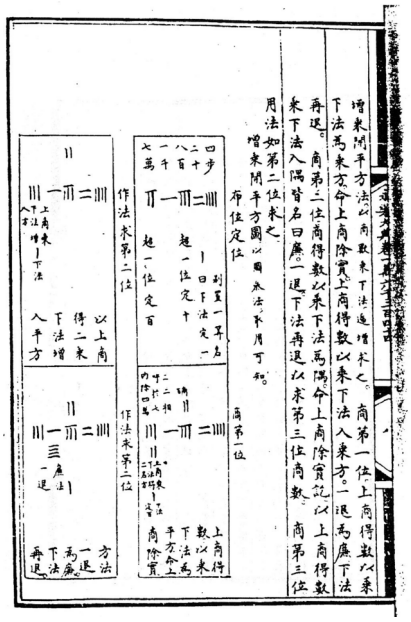


Figure 4.1: Image from Nine Chapter of the Mathematical art. By Yang Hui(1238-1298) - mybook, Public Domain, https://commons.wikimedia.org/w/index.php?curid=10317744

### 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.

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

The key idea is that:

- 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

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

### 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):
    """
    Validate the dimensions of a linear system and return its size.

    This function checks whether the given coefficient matrix `A` is square
    and whether its dimensions are compatible with the right-hand side vector
    `b`. If the dimensions are valid, it returns the size of the system.

    Parameters
    ----------
    A : numpy.ndarray
        A 2D array of shape ``(n, n)`` representing the coefficient matrix of
        the linear system.
    b : numpy.ndarray
        A array of shape ``(n, o)`` representing the right-hand side vector.

    Returns
    -------
    int
        The size of the system, i.e., the number of variables `n`.

    Raises
    ------
    ValueError
        If `A` is not square or if the size of `b` does not match the number of
```

```
        rows in `A`.
    """

    # Validate that A is a 2D square matrix
    if A.ndim != 2:
        raise ValueError(f"Matrix A must be 2D, but got {A.ndim}D array")

    n, m = A.shape
    if n != m:
        raise ValueError(f"Matrix A must be square, but got A.shape={A.shape}")

    if b.shape[0] != n:
        raise ValueError(
            f"System shapes are not compatible: A.shape={A.shape}, "
            f"b.shape={b.shape}"
        )

    return n
```

Then we can implement the elementary row operations

```
def row_swap(A, b, p, q):
    """
    Swap two rows in a linear system of equations in place.

    This function swaps the rows `p` and `q` of the coefficient matrix `A`
    and the right-hand side vector `b` for a linear system of equations
    of the form ``Ax = b``. The operation is performed **in place**, modifying
    the input arrays directly.

    Parameters
    ----------
    A : numpy.ndarray
        A 2D NumPy array of shape ``(n, n)`` representing the coefficient matrix
        of the linear system.
    b : numpy.ndarray
        A 2D NumPy array of shape ``(n, 1)`` representing the right-hand side
        vector of the system.
    p : int
        The index of the first row to swap. Must satisfy ``0 <= p < n``.
    q : int
        The index of the second row to swap. Must satisfy ``0 <= q < n``.
```

```
    Returns
    -------
    None
        This function modifies `A` and `b` directly and does not return
        anything.
    """
    # get system size
    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 a row of a linear system by a constant factor in place.

    This function multiplies all entries in row `p` of the coefficient matrix
    `A` and the corresponding entry in the right-hand side vector `b` by a
    scalar `k`. The operation is performed **in place**, modifying the input
    arrays directly.

    Parameters
    ----------
    A : numpy.ndarray
        A 2D NumPy array of shape ``(n, n)`` representing the coefficient matrix
        of the linear system.
    b : numpy.ndarray
        A 2D NumPy array of shape ``(n, 1)`` representing the right-hand side
        vector of the system.
    p : int
        The index of the row to scale. Must satisfy ``0 <= p < n``.
    k : float
        The scalar multiplier applied to the entire row.

    Returns
    -------
    None
        This function modifies `A` and `b` directly and does not return
        anything.
```

```
    """
    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):
    """
    Perform an in-place row addition operation on a linear system.

    This function applies the elementary row operation:

    ``row_p ← row_p + k * row_q``

    where `row_p` and `row_q` are rows in the coefficient matrix `A` and the
    right-hand side vector `b`. It updates the entries of `A` and `b`
    **in place**, directly modifying the original data.

    Parameters
    ----------
    A : numpy.ndarray
        A 2D NumPy array of shape ``(n, n)`` representing the coefficient matrix
        of the linear system.
    b : numpy.ndarray
        A 2D NumPy array of shape ``(n, 1)`` representing the right-hand side
        vector of the system.
    p : int
        The index of the row to be updated (destination row). Must satisfy
        ``0 <= p < n``.
    k : float
        The scalar multiplier applied to `row_q` before adding it to `row_p`.
    q : int
        The index of the source row to be scaled and added. Must satisfy
        ``0 <= q < n``.
    """
    n = system_size(A, b)

    # Perform the row operation
```

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

    # Update the corresponding value in b
    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.

```
def gaussian_elimination(A, b, verbose=False):
    """
    Perform Gaussian elimination to reduce a linear system to upper triangular
    form.

    This function performs **forward elimination** to transform the coefficient
    matrix `A` into an upper triangular matrix, while applying the same
    operations to the right-hand side vector `b`. This is the first step in
    solving a linear system of equations of the form ``Ax = b`` using Gaussian
    elimination.

    Parameters
    ----------
    A : numpy.ndarray
        A 2D NumPy array of shape ``(n, n)`` representing the coefficient matrix
```

```
        of the system.
    b : numpy.ndarray
        A 2D NumPy array of shape ``(n, 1)`` representing the right-hand side
        vector.
    verbose : bool, optional
        If ``True``, prints detailed information about each elimination step,
        including the row operations performed and the intermediate forms of
        `A` and `b`. Default is ``False``.

    Returns
    -------
    None
        This function modifies `A` and `b` **in place** and does not return
        anything.
    """
    # 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:

```
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 4.3.** 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 non-zero).
- 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 substitution.

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

```
def forward_substitution(A, b):
    """
    Solve a lower triangular system of linear equations using forward
    substitution.

    This function solves the system of equations:

    .. math::
        A x = b

    where `A` is a **lower triangular matrix** (all elements above the main
    diagonal are zero). The solution vector `x` is computed sequentially by
    solving each equation starting from the first row.

    Parameters
    ----------
    A : numpy.ndarray
        A 2D NumPy array of shape ``(n, n)`` representing the lower triangular
        coefficient matrix of the system.
    b : numpy.ndarray
        A 1D NumPy array of shape ``(n,)`` or a 2D NumPy array of shape
        ``(n, 1)`` representing the right-hand side vector.

    Returns
    -------
    x : numpy.ndarray
        A NumPy array of shape ``(n,)`` containing the solution vector.
    """
    """
    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
```

```
    if not np.allclose(A, np.tril(A)):
        raise ValueError("Matrix A is not lower triangular")

    # 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):
    """
    Solve an upper triangular system of linear equations using backward
    substitution.

    This function solves the system of equations:

    .. math::
        A x = b

    where `A` is an **upper triangular matrix** (all elements below the main
    diagonal are zero). The solution vector `x` is computed starting from the
    last equation and proceeding backward.

    Parameters
    ----------
    A : numpy.ndarray
        A 2D NumPy array of shape ``(n, n)`` representing the upper triangular
        coefficient matrix of the system.
    b : numpy.ndarray
        A 1D NumPy array of shape ``(n,)`` or a 2D NumPy array of shape
        ``(n, 1)`` representing the right-hand side vector.

    Returns
    -------
    x : numpy.ndarray
```

```
        A NumPy array of shape ``(n,)`` containing the solution vector.
    """
    # 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 grand 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
```

**Exercise 4.4.** Use Gaussian elimination followed by backwards elimination to 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 $\vec{x} = (1, 1, 1, 1)^T$.

## 4.6 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.

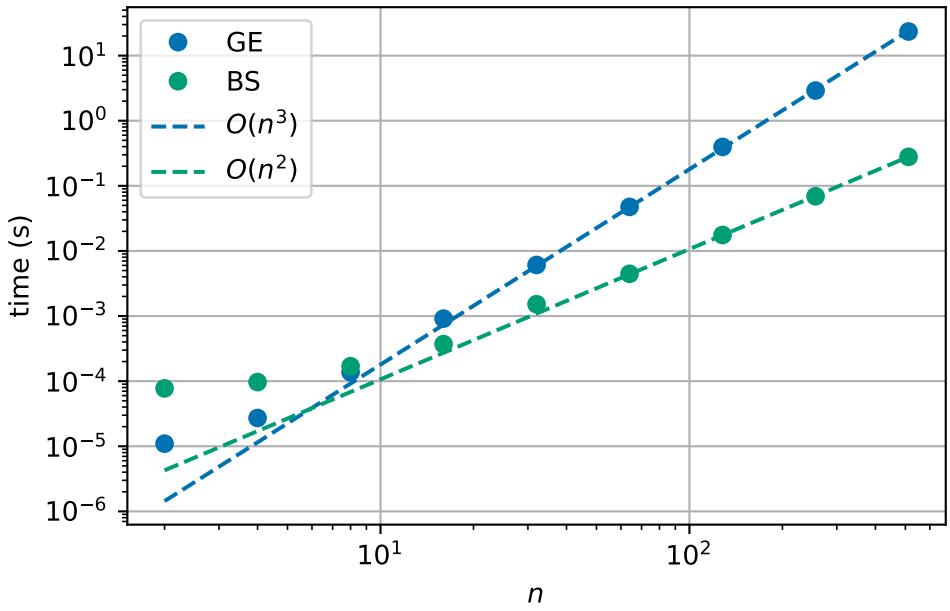



Figure 4.2: Runtime cost of applying Gaussian elimination (GE) and backwards solution (BS).

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 $\alpha$:

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

- $\alpha \times$ 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 $4 \times 4$ 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,

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

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

*Remark* 4.2.

- $L$ 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

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

Column 2 of $A$ gives

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

Column 3 of $A$ gives

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

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 for LU factorisation

We can implement computation of the LU factorisation:

```
def lu_factorisation(A):
    """
    Compute the LU factorisation of a square matrix A.

    The function decomposes a square matrix ``A`` into the product of a lower
    triangular matrix ``L`` and an upper triangular matrix ``U`` such that:

    .. math::
        A = L U

    where ``L`` has unit diagonal elements and ``U`` is upper triangular.

    Parameters
    ----------
    A : numpy.ndarray
        A 2D NumPy array of shape ``(n, n)`` representing the square matrix to
        factorise.

    Returns
    -------
    L : numpy.ndarray
        A lower triangular matrix with shape ``(n, n)`` and unit diagonal.
    U : numpy.ndarray
        An upper triangular matrix with shape ``(n, n)``.
    """
```

```
    n, m = A.shape
    if n != m:
        raise ValueError(f"Matrix A is not square {A.shape=}")

    # 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:

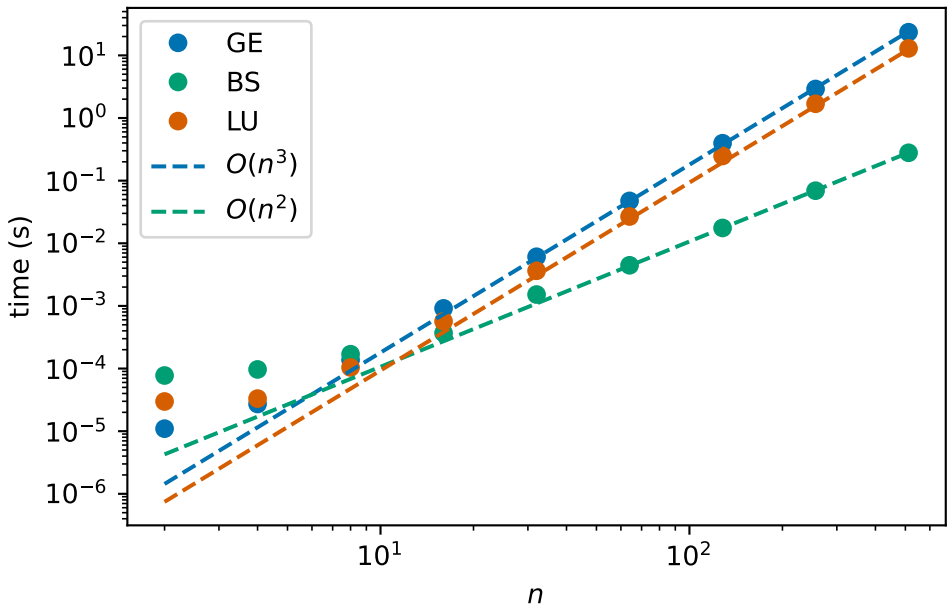



Figure 4.3: Runtime cost of applying Gaussian elimination (GE) and backwards solving (BS) equations as well as LU factorisation.

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 $\varepsilon$ 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* 4.3.

- 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 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 multiples 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 $\varepsilon$ 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 $\varepsilon$, so we will have problems if $\varepsilon$ 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 for Gaussian elimination with pivoting

```
def gaussian_elimination_with_pivoting(A, b, verbose=False):
    """
    perform Gaussian elimnation with pivoting to reduce the system of linear
    equations Ax=b to upper triangular form.
    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

Module learning objective

Apply direct and iterative solvers to solve systems of linear equations; implement methods using floating point numbers and investigate computational cost using computer experiments.

## 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!

*Remark* 5.1. We use a value in brackets in the superscript to denote the iteration number to avoid confusion between the iteration number and the component of the vector:

$$\vec{x}^{(k)} = (\vec{x}_1^{(k)}, \vec{x}_2^{(k)}, \dots, \vec{x}_n^{(k)}).$$

**Example 5.1** (Some very bad examples)**.** These are examples of potential iterative methods which would not work very well!

1. Consider

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

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

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

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

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 $\vec{r} \mapsto P\vec{r}$ 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 the above bad examples could be written in the form of (5.1) with $P = O$ (the zero matrix) or $P = A^{-1}$.

## 5.2 Jacobi iteration

One simple choice for $P$ in (5.1) 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 (Example 3.8).
- 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

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

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 5.2.** 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 5.3.** 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:

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

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 $\vec{x}^{(0)} = (2, 1)^T$.

**Exercise 5.1.** 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 when to stop later.
- 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):
    """
    Solve a linear system Ax = b using the Jacobi iterative method.

    The Jacobi method is an iterative algorithm for solving the linear system:

    .. math::
        Ax = b

    starting from an initial guess ``x0``. At each iteration, the solution is
    updated according to:

    .. math::
        x_i^{(k+1)} = x_i^{(k)} + \\frac{1}{A_{ii}} \\left( b_i -
        \\sum_{j=0}^{n-1} A_{ij} x_j^{(k)} \\right)

    Parameters
    ----------
    A : numpy.ndarray
        A 2D NumPy array of shape ``(n, n)`` representing the coefficient
        matrix.
    b : numpy.ndarray
        A 1D or 2D NumPy array of shape ``(n,)`` or ``(n, 1)`` representing the
        right-hand side vector.
    x0 : numpy.ndarray
        Initial guess for the solution, same shape as ``b``.
    max_iter : int
        Maximum number of iterations to perform.
    verbose : bool, optional
        If ``True``, prints the value of the solution vector at each iteration.
        Default is ``False``.

    Returns
    -------
    x : numpy.ndarray
        Approximated solution vector after ``max_iter`` iterations.
    """
    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):
    """
    Solve a linear system Ax = b using the Gauss-Seidel iterative method.

    The Gauss-Seidel method is an iterative algorithm for solving the linear
    system:

    .. math::
        Ax = b

    starting from an initial guess ``x0``. At each iteration, the solution is
    updated sequentially using the most recently computed values:

    .. math::
        x_i^{(k+1)} = x_i^{(k)} + \\frac{1}{A_{ii}} \\left( b_i -
        \\sum_{j=0}^{i-1} A_{ij} x_j^{(k+1)} -
        \\sum_{j=i}^{n-1} A_{ij} x_j^{(k)} \\right)

    Parameters
    ----------
    A : numpy.ndarray
        A 2D NumPy array of shape ``(n, n)`` representing the coefficient
        matrix.
```

```
    b : numpy.ndarray
        A 1D or 2D NumPy array of shape ``(n,)`` or ``(n, 1)`` representing the
        right-hand side vector.
    x0 : numpy.ndarray
        Initial guess for the solution, same shape as ``b``.
    max_iter : int
        Maximum number of iterations to perform.
    verbose : bool, optional
        If ``True``, prints the value of the solution vector at each iteration.
        Default is ``False``.

    Returns
    -------
    x : numpy.ndarray
        Approximated solution vector after ``max_iter`` iterations.

    """
    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 (Example 3.8). 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 $\alpha n$ non-zero entries (where $\alpha$ is independent of $n$). Typically this happens when we know there are at most $\alpha$ 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. $O(n)$. This is called the COO (coordinate) format.

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

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
):
    """
    Solve a sparse linear system Ax = b using the Jacobi iterative method.

    The system is represented in **sparse COO (coordinate) format** with:
    - `A_real`: non-zero values
    - `I_row`: row indices of non-zero entries
    - `I_col`: column indices of non-zero entries

    The Jacobi method updates the solution iteratively:

    .. math::
        x_i^{(k+1)} = x_i^{(k)} + \\frac{1}{A_{ii}} \\left( b_i -
```

```
        \\sum_{j=0}^{n-1} A_{ij} x_j^{(k)} \\right)

    Parameters
    ----------
    A_real : array-like
        Array of non-zero entries of the sparse matrix A.
    I_row : array-like
        Row indices corresponding to each entry in `A_real`.
    I_col : array-like
        Column indices corresponding to each entry in `A_real`.
    b : array-like
        Right-hand side vector of length `n`.
    x0 : numpy.ndarray
        Initial guess for the solution vector, same length as `b`.
    max_iter : int
        Maximum number of iterations to perform.
    verbose : bool, optional
        If ``True``, prints the solution vector after each iteration. Default is
        ``False``.

    Returns
    -------
    x : numpy.ndarray
        Approximated solution vector after `max_iter` iterations.
    """
    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
):
    """
    Solve a sparse linear system Ax = b using the Gauss-Seidel iterative method.

    The system is represented in **sparse COO (coordinate) format** with:
    - `A_real`: non-zero values
    - `I_row`: row indices of non-zero entries
    - `I_col`: column indices of non-zero entries

    The Gauss-Seidel method updates the solution sequentially using the most
    recently computed values:

    .. math::
        x_i^{(k+1)} = x_i^{(k)} + \\frac{1}{A_{ii}} \\left( b_i -
        \\sum_{j=0}^{i-1} A_{ij} x_j^{(k+1)} -
        \\sum_{j=i}^{n-1} A_{ij} x_j^{(k)} \\right)

    Parameters
    ----------
    A_real : array-like
        Array of non-zero entries of the sparse matrix A.
    I_row : array-like
        Row indices corresponding to each entry in `A_real`.
    I_col : array-like
```

```
    Column indices corresponding to each entry in `A_real`.
b : array-like
    Right-hand side vector of length `n`.
x0 : numpy.ndarray
    Initial guess for the solution vector, same length as `b`.
max_iter : int
    Maximum number of iterations to perform.
verbose : bool, optional
    If ``True``, prints the solution vector after each iteration. Default is
    ``False``.

Returns
-------
x : numpy.ndarray
    Approximated solution vector after `max_iter` iterations.
"""
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 = [-2.5   7.25 -1.75 12.25  1.5   1.25 -1.75 -2.5   3.25  1.5 ]
I_row =  [0 0 1 1 2 2 2 3 3 3]
I_col =  [3 0 2 1 3 2 1 0 3 2]

dense matrix:
A_dense = [  7.25,  0.00,  0.00, -2.50 ]
          [  0.00, 12.25, -1.75,  0.00 ]
          [  0.00, -1.75,  1.25,  1.50 ]
```

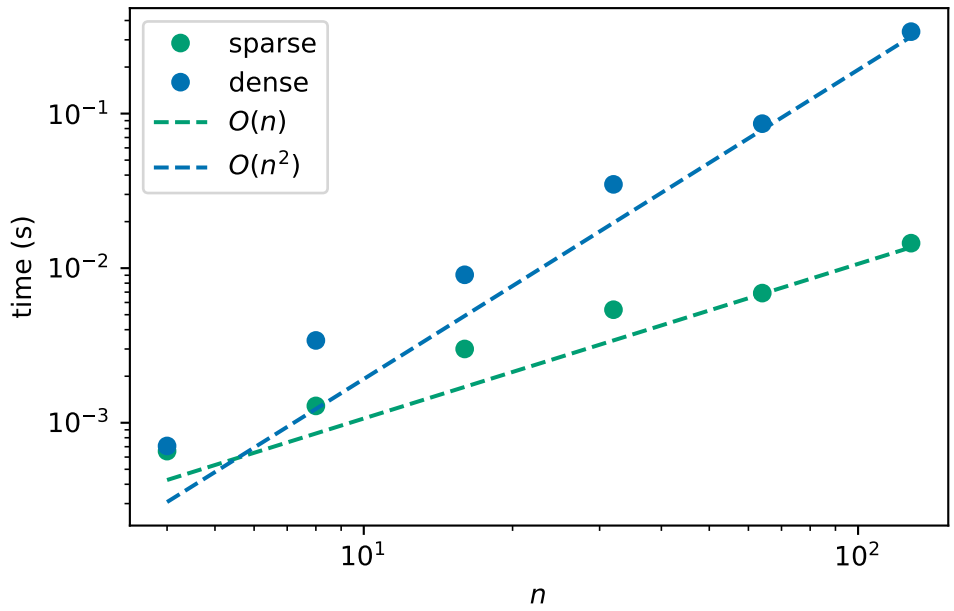
```
          [ -2.50,  0.00,  1.50,  3.25 ]

jacobi with sparse matrix
starting value: x.T = [  0.0,  0.0,  0.0,  0.0 ]
after iter=0: x.T = [  0.65517,  0.85714,  0.80000,  0.69231 ]
after iter=1: x.T = [  0.89390,  0.97143,  1.16923,  0.82706 ]
after iter=2: x.T = [  0.94036,  1.02418,  1.16753,  0.84028 ]
after iter=3: x.T = [  0.94492,  1.02393,  1.22551,  0.87680 ]
after iter=4: x.T = [  0.95752,  1.03222,  1.18134,  0.85355 ]

jacobi with dense matrix
starting value: x.T = [  0.0,  0.0,  0.0,  0.0 ]
after iter=0: x.T = [  0.65517,  0.85714,  0.80000,  0.69231 ]
after iter=1: x.T = [  0.89390,  0.97143,  1.16923,  0.82706 ]
after iter=2: x.T = [  0.94036,  1.02418,  1.16753,  0.84028 ]
after iter=3: x.T = [  0.94492,  1.02393,  1.22551,  0.87680 ]
after iter=4: x.T = [  0.95752,  1.03222,  1.18134,  0.85355 ]
```

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 both for 10 iterations. 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.

**Example 5.4.** Consider the following sequence $\vec{x}^{(k)}$:

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

- 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 the second and third cases, we call *tol* the **convergence tolerance** and the choice of *tol* will control the accuracy of the solution.

**Exercise 5.2** (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.:

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

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

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

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 Complex numbers

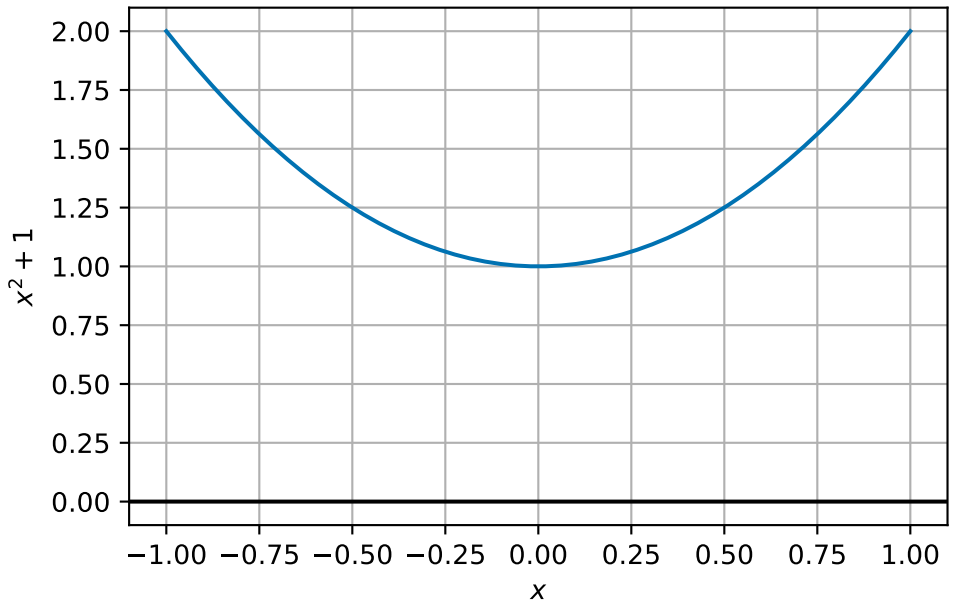
This section of the notes supports the final learning outcome around eigenvalues and eigenvectors.

## 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: There are no real values $x$ such that $x^2 + 1 = 0$.

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 simply 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

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

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:

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

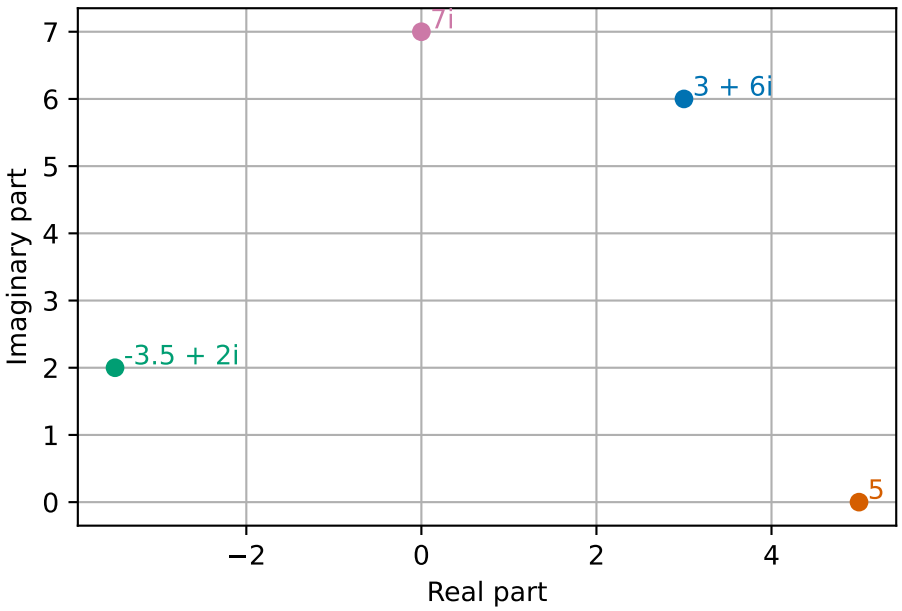
In fact, we have that $\sqrt{x}\sqrt{y} = \sqrt{xy}$ only if $x, y > 0$.

## 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 $-3.5 + 2i$.

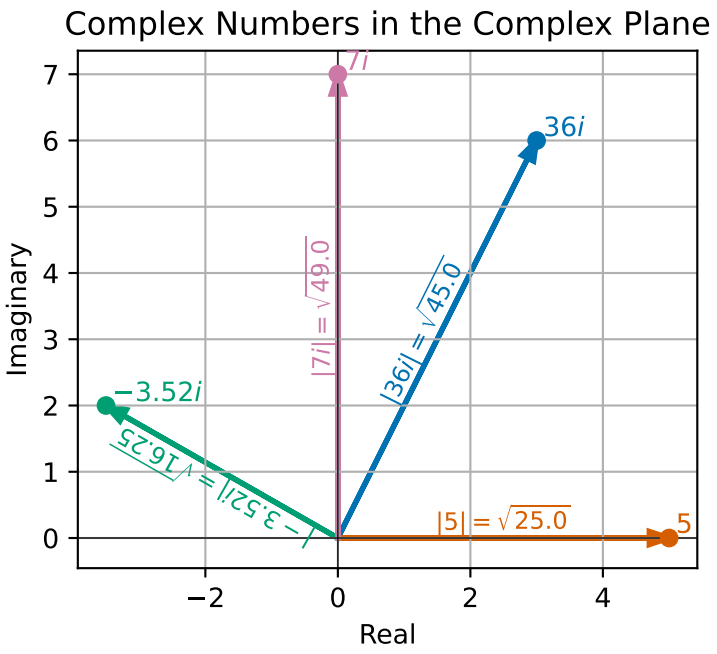
We have already seen that the complex conjugate of a complex number is helpful when performing division of complex numbers. The reason is that computing the product of a number and its conjugate always gives a real, positive number:

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

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

$$|z| = |a + bi| = \sqrt{a^2 + b^2} = \sqrt{z\bar{z}}. \tag{6.2}$$

**Example 6.7.**



**Exercise 6.6.** Find the value of

$$|3+6i| \quad \text{and} \quad |-3.5+2i|. \tag{6.3}$$

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 $\theta$ 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 (TODO is it there?), we see that multiplying by $y$ is the same as rotating the complex point ($z$) by an angle of $\theta$ 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 $\theta$ 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 $\theta$ is the argument of $z$.

The representation of the angle only unique up to adding integer multiples of $2\pi$, since rotating a point by $2\pi$ 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

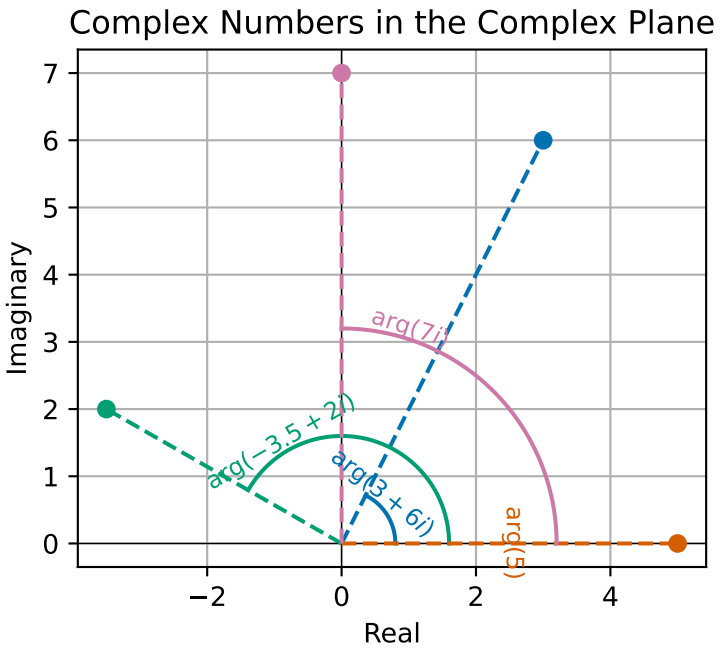
$$\cos(-\pi/4) = \frac{1}{\sqrt{2}}, \quad \text{and} \quad \sin(-\pi/4) = \frac{-1}{\sqrt{2}},$$

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:*

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

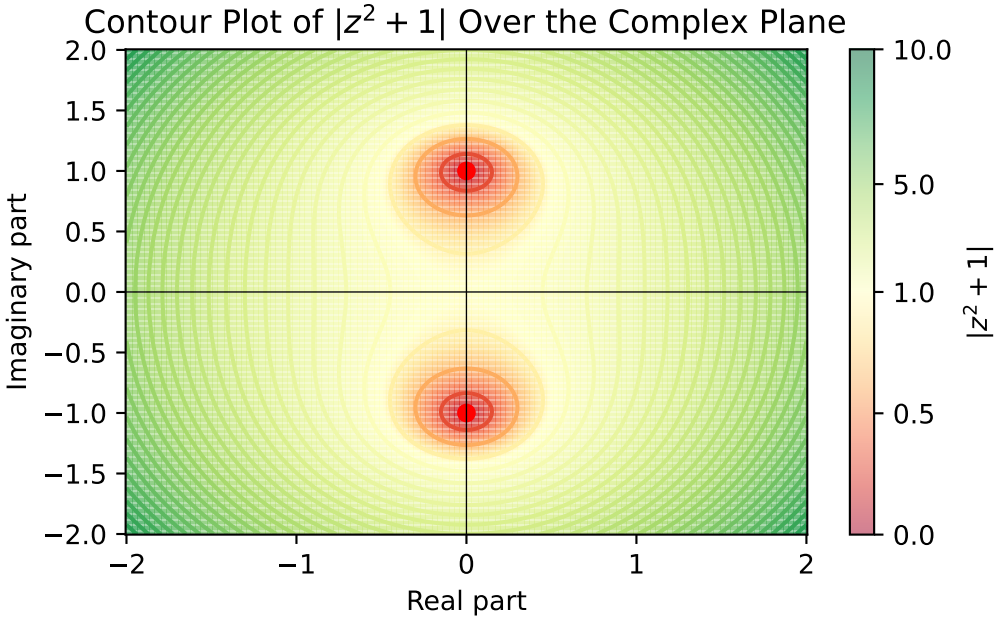
It is really important to note here that this is not true if we want $z$ to be a real number. Let's revisit Example 6.1.

**Example 6.10.** Consider the polynomial equation

$$x^2 + 1 = 0. \tag{6.4}$$

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 $-i$:

$$\begin{aligned} &i^2 + 1 = 0 \\ &(-i)^2 + 1 = (-1)^2 i^2 + 1 = i^2 + 1 = 0. \end{aligned}$$



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

In general, 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 $z^2 - 4z + 20 = 0$.

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

## 6.5 Complex vectors and matrices

Last year you were introduced to vectors of real numbers. In the remaining part of Linear Algebra, we will need to work with complex. The definitions are mostly what you would expect but with some slight subtleties.

**Definition 6.3.** A complex matrix is a matrix whose entries are complex numbers. A complex vector is a vector whose entries are complex numbers.

One thing we like to do with vectors is to take scalar products and to find their length. When working with complex vectors, we replace the scalar product with a *Hermitian product*. We write $\langle \vec{a}, \vec{b} \rangle$ for the Hermitian product of two complex $n$-vectors $\vec{a}$ and $\vec{b}$ and the value is given by

$$\langle \vec{a}, \vec{b} \rangle = \sum_{i=1}^{n} a_i \bar{b}_i. \tag{6.5}$$

We see that the Hermitian product is the same as taking the scalar product of $\vec{a}$ and the complex conjugate of $\vec{b}$. If the entries in $\vec{b}$ have no imaginary part (i.e. they are all real numbers), then the Hermitian product of $\vec{a}$ and $\vec{b}$ is equal to the scalar product of $\vec{a}$ and $\vec{b}$.

**Example 6.12.** Let $\vec{a} = (1 + 2i, 2 + i, 3 - 4i)^T$ and $\vec{b} = (6 + 2i, 3 + 4i, 1 + i)^T$ then:

$$\begin{aligned} \langle \vec{a}, \vec{b} \rangle &= (1 + 2i) \times (6 - 2i) + (2 + i) \times (3 - 4i) + (3 - 4i) \times (1 - i) \\ &= (10 + 10i) + (10 - 5i) + (-1 - 7i) = 19 - 2i. \end{aligned}$$

**Exercise 6.10.** Find the Hermitian product of $\vec{c} = (3+i, -2+2i, 4-i)^T$ and $\vec{d} = (-1+3i, 2+6i, 2i)^T$.

Recalling the definition of modulus of a complex number (6.2), we can also compute that for a complex $n$-vector $\vec{z}$ that

$$\langle \vec{z}, \vec{z} \rangle = \sum_{i=1}^{n} z_i \bar{z}_i = \sum_{i=1}^{n} |z_i|^2.$$

So for a complex vector, we define the Euclidean norm as

$$\|\vec{z}\| = \sqrt{\sum_{i=1}^{n} |z_i|^2} = \sqrt{\langle z, z \rangle}.$$

Recall that for a real *symmetric* $n \times n$ matrix $A$ and any $n$-vectors $\vec{x}$ and $\vec{y}$ that

$$\begin{aligned}
A\vec{x} \cdot \vec{y} &= \sum_{i=1}^{n} (A\vec{x})_i y_i && \text{(definition of scalar product)} \\
&= \sum_{i=1}^{n} \left( \sum_{j=1}^{n} A_{ij} x_j \right) y_i && \text{(definition of matrix-vector product)} \\
&= \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ji} x_j y_i && \text{(symmetry of } A\text{)} \\
&= \sum_{j=1}^{n} x_j \sum_{i=1}^{n} \left( A_{ji} y_i \right) && \text{(rearranging)} \\
&= \sum_{j=1}^{n} x_j (Ay)_j && \text{(definition of matrix-vector product)} \\
&= \vec{x} \cdot A\vec{y} && \text{(definition of scalar product).}
\end{aligned}$$

Replacing all terms with complex value and the scalar product with Hermitian product, we get

a similar calculation:

$$\begin{aligned}
\langle A\vec{x}, \vec{y}\rangle &= \sum_{i=1}^{n} (A\vec{x})_i \bar{y}_i && \text{(definition of Hermitian product)} \\
&= \sum_{i=1}^{n} \left( \sum_{j=1}^{n} A_{ij} x_j \right) \bar{y}_i && \text{(definition of matrix-vector product)} \\
&= \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ji} x_j \bar{y}_i && \text{(symmetry of } A\text{)} \\
&= \sum_{j=1}^{n} x_j \sum_{i=1}^{n} \left( A_{ji} \bar{y}_i \right) && \text{(rearranging)} \\
&= \sum_{j=1}^{n} x_j (\bar{A}y)_j && \text{(definition of matrix-vector product)} \\
&= \langle \vec{x}, \bar{A}\vec{y}\rangle && \text{(definition of Hermitian).}
\end{aligned}$$

This is quite unsatisfactory (note the extra bar on $A$ in the final equation) and we would like something the generalises the idea of a symmetric matrix to the Hermitian product case too.

The key idea is to generalise the definition of matrix transpose to the complex setting:

**Definition 6.4.** For a complex $n$-matrix $A$, we write $A^H$ for the **conjugate transpose** (also called the **Hermitian transpose**) given by

$$(A^H)_{ij} = \bar{A}_{ji} = \bar{A}^T.$$

If $A^H = A$, then we say that $A$ is **Hermitian**.

Repeating the above calculations, we can see that if $A$ is Hermitian then

$$\langle A\vec{x}, \vec{y}\rangle = \langle \vec{x}, A\vec{y}\rangle.$$

# 7 Eigenvectors and eigenvalues

TODO Add triangular matrix example.

> Module learning outcome:
>
> Apply algorithms to compute eigenvectors and eigenvalues of large matrices.

This section of the notes will introduce our second big linear algebra problem. Throughout, we will be considering a square $(n \times n)$ matrix.

## 7.1 Key definitions

For this problem, we will think of a matrix $A$ acting on functions $\vec{x}$:

$$\vec{x} \mapsto A\vec{x}.$$

We are interested in when is the output vector $A\vec{x}$ is *parallel* to $\vec{x}$.

**Definition 7.1.** We say that any vector $\vec{x}$ where $A\vec{x}$ is parallel is $\vec{x}$ is called an *eigenvector* of $A$. Here by parallel, we mean that there exists a number $\lambda$ (can be positive, negative or zero) such that

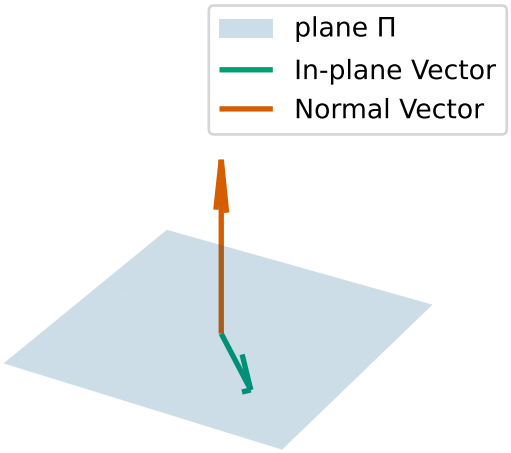
$$A\vec{x} = \lambda\vec{x}. \tag{7.1}$$

We call the associated number $\lambda$ an *eigenvalue* of $A$.

We will later see that an $n \times n$ square matrix always has $n$ eigenvalues (which may not always be distinct).

If $A\vec{x} = \vec{0}$, then $\vec{x}$ is an eigenvector associated to the eigenvalue 0. In fact, we know that 0 is an eigenvalue of $A$ if, and only if, $A$ is singular.

**Example 7.1.** Let $P$ be the 3x3 matrix that represents projection on to a plane $\Pi = \{\vec{x} \in \mathbb{R}^3 : \vec{a} \cdot \vec{x} = 0\}$ (i.e. $P$ maps any point in $\mathbb{R}^3$ to its nearest point in $\Pi$). What are the eigenvalues and eigenvectors of $P$?




- if $\vec{x}$ is in the plane $\Pi$, then $P\vec{x} = \vec{x}$. This means that $\vec{x}$ is an eigenvector and the associated eigenvalue is 1.
- if $\vec{y}$ is perpendicular to the plane $\Pi$, then $P\vec{y} = \vec{0}$. this means that $\vec{y}$ is an eigenvector and the associated eigenvalue is 0.

Let $\vec{y}$ be perpendicular to $\Pi$ (so that $P\vec{y} = \vec{0}$ and $\vec{y}$ is an eigenvector of $P$), then for any number $s$, we can compute

$$P(s\vec{y}) = sP\vec{y} = s\vec{0} = \vec{0}.$$

This means that $s\vec{y}$ is also an eigenvector of $P$ associated to the eigenvalue 0. As a consequence when we compute eigenvectors, we need to take care to *normalise* the vector to ensure we get a unique answer.

We see we end up with a two-dimensional space of eigenvectors (i.e., the plane $\Pi$) associated to eigenvalue 1 and a one-dimensional space of eigenvectors (i.e., the line perpendicular to $\Pi$) eigenvalue 0. We use the term *eigenspace* the space of eigenvectors associated to a particular eigenvalue.

**Example 7.2.** Let $A$ be the permuatation matrix which takes an input two-vector and outputs a two-vector with the components swapped. The matrix is given by

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

What are the eigenvectors and eigenvalues of $A$?

- Let $\vec{x} = (1, 1)^T$, the swapping the components of $\vec{x}$ gives back the same vector $\vec{x}$. In equations, we can write $A\vec{x} = \vec{x}$. This means that $\vec{x}$ is an eigenvector and the eigenvalue is 1.
- Let $\vec{x} = (-1, 1)^T$, the swapping the components of $\vec{x}$ gives back $(1, -1)^T$ which we can see is $-\vec{x}$. In equations, we can write $A\vec{x} = -\vec{x}$. This means that $\vec{x}$ is an eigenvector of $A$ and the associated eigenvalue is $-1$.

Here we see that again we actually have two one-dimensional eigenspaces.

## 7.2 Properties of eigenvalues and eigenvectors

TODO

Eigenvalues and eigenvectors can be used to completely describe the transformation described by $A$.

Let $A$ and $B$ be two $n \times n$ matrices. Then, we cannot use the eigenvalues of $A$ and $B$ to work out the eigenvalues of $A + B$ or the eigenvalues of $AB$, in general.

Sum of eigenvalues is trace Product of eigenvalues is determinant

*Remark* 7.1. If we add up all the eigenvalues of a $n \times n$ matrix, we get the *trace* of the matrix $A$. We can also find the trace by adding up the diagonal components of the matrix:

$$\lambda_1 + \cdots + \lambda_n = a_{11} + a_{22} + \cdots + a_{nn} = \text{trace}(A).$$

## 7.3 How to find eigenvalues and eigenvectors

To compute eigenvalues and eigenvectors, we start from (7.1) and move everything to the left-hand side and use the identity matrix ($I_n$):

$$(A - \lambda I_n)\vec{x} = \vec{0}.$$

This tells us that if we want to find an eigenvalue of $A$, then we need to find a number $\lambda$ such that $(A - \lambda I_n)$ can multiply a non-zero vector and give us back the zero-vector. This happens when $(A - \lambda I)$ is *singular* (Theorem 3.3).

One way to test if a matrix is singular, is if the determinant is 0. This gives us a test we can use to determine eigenvalues:

$$\det(A - \lambda I_n) = 0. \tag{7.2}$$

In fact, this equation no longer depends on the eigenvector $\vec{x}$, and if we can find solutions $\lambda$ to this equation then $\lambda$ is an eigenvalue of $A$.

We call (7.2) the *characteristic equation* or *eigenvalue equation*. We will see that (7.2) gives us a degree $n$ polynomial equation in $\lambda$.

Once we have found an eigenvalue by solving the characteristic equation for a value $\lambda^*$, we need to find a vector $\vec{x}$ such that

$$(A - \lambda^*)\vec{x} = \vec{0}.$$

In general, this is possible using a variation of Gaussian elimination with pivoting, but we do not explore this method in this module.

**Example 7.3.** Let $A$ be the matrix given by

$$A = \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}.$$

Then, we can compute that

$$\begin{aligned} \det(A - \lambda I_n) &= \det \begin{pmatrix} 3-\lambda & 1 \\ 1 & 3-\lambda \end{pmatrix} \\ &= (3-\lambda)(3-\lambda) - 1 \times 1 \\ &= \lambda^2 + 6\lambda + 8. \end{aligned}$$

So we want to find values $\lambda$ such that

$$\det(A - \lambda I_n) = \lambda^2 + 6\lambda + 8 = 0.$$

We can read off, by factorisation, that the values of $\lambda$ are 4 and 2.

We can now start computing the associated eigenvectors.

To find the eigenvector associated with the eigenvalue 4. We see that

$$A - 4I_n = \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}.$$

We can identify that $(A - 4I_n)(1,1)^T = \vec{0}$. So $(1,1)$ is an eigenvector associated with 4.

To find the eigenvector associated with the eigenvalue 2. We see that

$$A - 2I_n = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}.$$

We can identify that $(A - 2I_n)(-1,1)^T = \vec{0}$. So $(-1,1)$ is an eigenvector associated with 2.

TODO possibly remove if we don't do scaling/shifting methods. We note that this example is actually surprisingly similar to Example 7.2. We see that the eigenvectors are actually the same! We can see that the matrices are related too:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} - 3I = \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix}.$$

So we can compute that if $A\vec{x} = \lambda\vec{x}$ then

$$(A + 3I)\vec{x} = \lambda\vec{x} + 3\vec{x} = (\lambda + 3)\vec{x}.$$

So we see that $\vec{x}$ is also an eigenvector of $A$ and the associated eigenvector is $\lambda + 3$.

Although this procedure is gives us back eigenvalues there are cases where we have to be a bit careful. We have seen one example above with a two-dimensional eigenspace associated with one eigenvector (Example 7.1). Here are two other cases we must be careful:

**Example 7.4.** Let $Q$ denote the $2 \times 2$ matrix that rotates any vector by $\pi/2$ ($= 90^\circ$):

$$Q = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}.$$

Our intuition says that there can be no vectors that when rotated by $\pi/2$ give something parallel to the input vector, but we can still compute:

$$\det Q = \det \begin{pmatrix} -\lambda & -1 \\ 1 & -\lambda \end{pmatrix} = \lambda^2 + 1.$$

So we can find eigenvalues by finding the values $\lambda$ such that

$$\lambda^2 + 1 = 0.$$

We saw in the section on Complex Numbers (Chapter 6) that the solutions to this equation are $\pm i$. This means our algorithms for finding eigenvalues and eigenvectors need to handle complex numbers too.

**Example 7.5.** Let $A$ be the $2 \times 2$ matrix given by

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

If we follow our procedure above we get a single repeated eigenvalue 3.

Looking at the shifted matrix, $A - 3I_n$:

$$A - 3I_n = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}.$$

we can identify one eigenvector $(1,0)^T$, but there is no other eigenvector (in a different direction)! Indeed, we can compute that:

$$(A - 3I_n)\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ 0 \end{pmatrix}.$$

This tells us that if $(A - 3I_n)(x,y)^T = \vec{0}$ if, and only if, $y = 0$. Thus all eigenvector have the form $(x,0)^T$ and point in the same direction as $(1,0)^T$.

**Exercise 7.1.** Find the eigenvalues and eigenvectors for the matrices

$$A = \begin{pmatrix} 9 & -2 \\ -2 & 6 \end{pmatrix}.$$

TODO add another complex example

## 7.4 Important theory

We have established a way to identify eigenvalues and eigenvectors for an arbitrary square matrix. It turns out this method can be used to prove the existance of eigenvalues.

**Theorem 7.1.** *Any square $n \times n$ matrix has $n$ complex eigenvalues (possibly not distinct).*

*Proof.* For any matrix the characteristic equation (7.2) is a degree $n$ polynomial. The Fundamental Theorem of Algebra (Theorem 6.1) tells us that any degree $n$ polynomial has $n$ roots over the complex numbers. The $n$ roots of the characteristic equation are the $n$ eigenvalues. □

The Abel-Ruffini theorem states that there is no solution in the radicals for a general polynomial of degree 5 or higher with arbitrary coefficients. This implies that there is no 'nice' closed form for roots of polynomials of degree 5 or higher. So, if we want an algorithm to find eigenvalues and eigenvectors of larger matrices then we need to do something else!

Let's suppose that we have an $n \times n$ matrix $A$ and we have found $n$ eigenvectors and $n$ eigenvalues (all distinct). Let's call the eigenvectors by $\vec{x}_1, \ldots \vec{x}_n$ and the eigenvalues $\lambda_1, \ldots \lambda_n$ then we have the equation:

$$A\vec{x}_j = \lambda_j \vec{x}_j.$$

So if we form the matrices $S$ to have columns equal to each eigenvector in turn and $\Lambda$ (pronounced lambda) to be the diagonal matrix with the eigenvalues listed along the diagonal we see that we have:

$$AS = S\Lambda.$$

If $S$ is invertible, we can multiply on the right by $S^{-1}$ to see that we have

$$A = S\Lambda S^{-1}. \tag{7.3}$$

This formula shows another factorisation of the matrix $A$ into simpler matrices, very much like we had when we computed the LU-factorisation matrix (Section 4.7).

The equation (7.3) is an example of a more general idea of *similar matrices*. We say that two matrices $A$ and $B$ are similar if there exists an invertible $n \times n$ matrix $P$ such that

$$B = P^{-1}AP.$$

Since $P$ is invertible, we can pre-multiply this equation by $P$ and post-multiply by $P^{-1}$ and see that being similar is a symmetric property.

**Lemma 7.1.** *The matrix $A$ is similar to the diagonal matrix $\Lambda$ formed by the eigenvalues of $A$.*

*Proof.* From (7.3), we have that

$$\Lambda = S^{-1}AS.$$

□

This leads to a nice theorem which we will use to help compute eigenvectors and eigenvalues of larger matrices:

**Theorem 7.2.** *If $A$ and $B$ are similar matrices then $A$ and $B$ have the same eigenvalues.*

*Proof.* We start by writing $B = P^{-1}AP$. Then we can compute that

$$BP^{-1} = P^{-1}A. \tag{7.4}$$

Let $\lambda$ be an eigenvalue of $A$ with eigenvector $\vec{x}$ and write $\vec{y} = P^{-1}\vec{x}$. Then we have that

$$\begin{aligned}
B\vec{y} &= BP^{-1}\vec{x} && \text{(definition of } \vec{y}\text{)} \\
&= P^{-1}A\vec{x} && \text{(from (7.4))} \\
&= P^{-1}(\lambda\vec{x}) && \text{(since } \vec{x} \text{ is an eigenvector)} \\
&= \lambda P^{-1}\vec{x} && \text{(rearranging)} \\
&= \lambda\vec{y} && \text{(definition of } \vec{y}\text{).}
\end{aligned}$$

This shows that any eigenvalue of $A$ is an eigenvalue of $B$. It also gives a formula for how eigenvectors change between $A$ and $B$.

To show any eigenvalue of $B$ is an eigenvalue of $A$, we simply repeat the calculation with $A$ and $B$ swapped. □

The key idea of the methods we will use to compute eigenvalues to apply a sequence of matrices to convert a matrix $A$ into a form similar to $A$ for which reading off the eigenvalues is easier. However, the quality of the algorithms we apply depend heavily on properties of the matrix $A$.

## 7.5 Why symmetric matrices are nice

Whilst all our methods are developed for complex matrices, when we restrict to real-valued symmetric matrices, we have this nice result.

**Theorem 7.3.** *Let $A$ be a symmetric matrix ($A^T = A$) with real entries. Then $A$ has $n$ real eigenvalues (zero imaginary part) and its eigenvectors are distinct and orthogonal.*

*Proof.* Let $\lambda$ be an eigenvalue of $A$ with eigenvector $\vec{x}$. Recall that $\vec{x} \neq 0$. Then, since $A$ has real values, we can compute that:

$$(\bar{A\vec{x}})_i = \left(\sum_{j=1}^{n} \bar{A}_{ji} x_i\right) = \sum_{j=1}^{n} A_{ji} \bar{x_i} = (A\bar{\vec{x}})_i.$$

We also note that any real, symmetric matrix is automatically Hermitian.

Then we see that

$$\begin{aligned}
\lambda\langle\vec{x},\vec{x}\rangle &= \langle(\lambda\vec{x}),\vec{x}\rangle && \text{(from definition of Hermitian product)}\\
&= \langle(A\vec{x}),\vec{x}\rangle && \text{(definition of eigenvalue and eigenvector)}\\
&= \langle\vec{x},A\vec{x}\rangle && \text{(symmetry of } A\text{)}\\
&= \langle\vec{x},\lambda\vec{x}\rangle && \text{(definition of eigenvalue and eigenvector)}\\
&= \bar{\lambda}\langle\vec{x},\vec{x}\rangle && \text{(from definition of Hermitian product).}
\end{aligned}$$

Since, $\langle\vec{x},\vec{x}\rangle > 0$ (recall $\vec{x} \neq 0$), we can divide by $\langle\vec{x},\vec{x}\rangle$ so infer that

$$\lambda = \bar{\lambda}.$$

Next, let $\vec{x}$ and $\vec{y}$ be eigenvectors of $A$ with distinct, eigenvalues $\lambda$ and $\mu$, respectively. From the first part of the proof, we know that $\lambda$ and $\mu$ are real. We compute that

$$\begin{aligned}
\lambda\langle\vec{x},\vec{y}\rangle &= \langle\lambda\vec{x},\vec{y}\rangle\\
&= \langle A\vec{x},\vec{y}\rangle\\
&= \langle\vec{x},A^H\vec{y}\rangle\\
&= \langle\vec{x},A\vec{y}\rangle\\
&= \langle\vec{x},\mu\vec{y}\rangle\\
&= \bar{\mu}\langle\vec{x},\vec{y}\rangle\\
&= \mu\langle\vec{x},\vec{y}\rangle.
\end{aligned}$$

Subtracting the right-hand side from the left hand side we see that

$$(\lambda - \mu)\langle \vec{x}, \vec{y} \rangle = 0.$$

This implies that if $\lambda$ and $\mu$ are distinct, that $\langle \vec{x}, \vec{y} \rangle = 0$. □

**Corollary 7.1.** *Let $A$ be a symmetric $n \times n$ matrix with real entries. Then the eigenvectors of $A$ form a basis of $\mathbb{R}^n$.*

*Proof.* We can only given an incomplete proof of this result. We will show that the eigenvectors are linearly independent. The proof is completed by showing that if you have any $n$ linearly independent vectors in $\mathbb{R}^n$ then you must have a basis.

Denote by $\vec{x}^{(1)}, \ldots \vec{x}^{(n)}$ the $n$ eigenvectors of $A$. We want to show that the eigenvectors are linearly independent. Suppose that we have real numbers $\alpha_1, \alpha_2, \cdots, \alpha_n$ such that:

$$\sum_{i=1}^{n} \alpha_i \vec{x}^{(i)} = \vec{0}. \tag{7.5}$$

To show the eigenvectors are linearly independent, we need to show that all $\alpha_i = 0$. We can do this by taking the inner product of (7.5) with $\vec{x}^{(j)}$ for *any* $j$:

$$0 = \vec{0} \cdot \vec{x}^{(j)} = \left( \sum_{i=1}^{n} \alpha_i \vec{x}^{(i)} \right) \cdot \vec{x}^{(j)} = \sum_{i=1}^{n} \alpha_i \left( \vec{x^{(i)}} \cdot \vec{x}^{(j)} \right) = \alpha_j \vec{x}^{(j)} \cdot \vec{x}^{(j)}.$$

Since we have that $|\vec{x}^{(j)}| > 0$, we have $\alpha_j = 0$ and we have shown that the eigenvectors are linearly independent. □

# 8 Eigenvectors and eigenvalues: practical solutions

> Tip
>
> **Module learning outcome:** apply algorithms to compute eigenvectors and eigenvalues of large matrices.

TODO change this lecture to:

- QR with gram-schmidt
- power iteration and inverse power iteration
- orthogonal vs orthonormal

In the previous lecture, we defined the eigenvalue problem for a matrix $A$: Finding numbers $\lambda$ (eigenvalues) and vectors $\vec{x}$ (eigenvectors) which satisfy the equation:

$$A\vec{x} = \lambda\vec{x}. \tag{8.1}$$

We saw one starting point for finding eigenvalues is to find the roots of the characteristic equation: a polynomial of degree $n$ for an $n \times n$ matrix $A$. But we already have seen that this approach will be infeasible for large matrices. Instead, we will find a sequence of similar matrices to $A$ such that we can read off the eigenvalues from the final matrix.

In equations, we can say our "grand strategy" is to find a sequence of matrices $P_1, P_2, ...$ to form a sequence of matrices:

$$A, P_1^{-1}AP, P_2^{-1}P_1^{-1}AP_1P_2, P_3^{-1}P_2^{-1}P_1^{-1}AP_1P_2P_3, ... \tag{8.2}$$

Our aim is to get all the way to a simple matrix where we read off the eigenvalues and eigenvectors.

For example, if at level $m$, say, we have transformed $A$ into a diagonal matrix the eigenvalues are the diagonal of the matrix

$$P_m^{-1}P_{m-1}^{-1}\cdots P_2^{-1}P_1^{-1}AP_1P_2\cdots P_{m-1}P_m,$$

and the eigenvectors are the columns of the matrix

$$S_m = P_1P_2\cdots P_{m-1}P_m.$$

Sometimes, we only want to compute eigenvalues, and not eigenvectors, then it is sufficient to transform the matrix to be triangular (either upper or lower triangular). Then, we can read off that the eigenvalues are the diagonal entries (see **?@exm-eigenvalue-triangular**).

*Remark* 8.1. In the example below, and many others, we see that we typically want the matrices $P_j$ to be *orthogonal*. A (real-valued) matrix $Q$ is orthogonal if $Q^TQ = I_n$.

Orthogonal matrices have the nice property that $Q^{-1} = Q^T$ so we can very easily compute their inverse! They also always have determinant 1 and their columns are orthonormal too. Finally, we can apply them (or their inverses) without worrying about adding extra problems from finite precision.

Examples of orthogonal matrices include matrices which describe rotations and reflections.

## 8.1 QR algorithm

The QR algorithm is an iterative method for computing eigenvalues and eigenvectors. At each step a matrix is factored into a product in a similar fashion to LU factorisation (**?@sec-lu-factorisation**). In this case, we factor a matrix, $A$ into a product of an orthgonal matrix, $Q$, and an upper triangular matrix, $R$:

$$A = QR.$$

This is *QR factorisation*.

Given a matrix $A$, the algorithm repeatedly applies QR factorisation. First, we set $A^{(0)} = A$, then we successively perform for $k = 0, 1, 2, ...$:

1. Compute the QR factorisation of $A^{(k)}$ into an orthogonal part and upper triangular part

$$A^{(k)} = Q^{(k)}R^{(k)};$$

2. Update the matrix $A^{(k+1)}$ recombining $Q$ and $R$ in the reverse order:

$$A^{(k+1)} = R^{(k)}Q^{(k)}.$$

As we take more and more steps, we hope that $A^{(k)}$ converges to an upper triangular matrix whose diagonal entries are the eigenvalues of the original matrix.

Rearranging the first step within each iteration, we see that

$$R^{(k)} = (Q^{(k)})^{-1}A^{(k)} = (Q^{(k)})^TA^{(k)}.$$

Substituting this value of $R^{(k)}$ into the second step gives

$$A^{(k+1)} = (Q^{(k)})^{-1}A^{(k)}Q^{(k)},$$

and we see that at each step we are finding a sequence of similar matrices, all with the same eigenvalues ({**?@thm-XX**}). We can additionally find the eigenvectors of $A$ by forming the product

$$Q = Q^{(1)}Q^{(2)}\cdots Q^{(m)}.$$

The hard part of the method is compute the QR factorisation. One classical way to get a QR factorisation is to use the Gram-Schmidt process. In general, Gram-Schmidt is used for take a sequence of vectors and forming a new sequence which is orthogonal. We can apply this to the columns of $A$ to form an orthogonal matrix. It turns out that if we track this process as a matrix-matrix product we find that the other factor is upper triangular.

### 8.1.1 The Gram-Schmidt process

The key idea is shown in Figure 8.1. Given a vector $\vec{a}$ (blue) and a vector $\vec{q}$ (green) with length 1. We can compute the projection of $\vec{a}$ onto the direction $\vec{q}$ (orange) by

$$(\vec{a}\cdot\vec{q})\vec{q}.$$

If we subtract this term from $\vec{a}$. We end up with a vector $\vec{u}$ with $\vec{u}\cdot\vec{q} = 0$. The difference $\vec{u}$ is given by

$$\vec{u} = \vec{a} - (\vec{a}\cdot\vec{q})\vec{q},$$

and we can compute that

$$\begin{aligned}
\vec{u}\cdot\vec{q} &= (\vec{a} - (\vec{a}\cdot\vec{q})\vec{q})\cdot\vec{q} \\
&= (\vec{a}\cdot\vec{q}) - (\vec{a}\cdot\vec{q})(\vec{q}\cdot\vec{q}) && \text{(properties of scalar product)} \\
&= (\vec{a}\cdot\vec{q}) - (\vec{a}\cdot\vec{q}) && \text{(since } \|\vec{q}\| = 1\text{)} \\
&= 0.
\end{aligned}$$

**Example 8.1.** Consider the sequence of vectors

$$\vec{a}^{(1)} = \begin{pmatrix}1\\0\\1\end{pmatrix}, \vec{a}^{(2)} = \begin{pmatrix}2\\-1\\0\end{pmatrix}, \vec{a}^{(3)} = \begin{pmatrix}1\\2\\1\end{pmatrix}.$$

We will manipulate these vectors to form three vectors which are orthonormal.

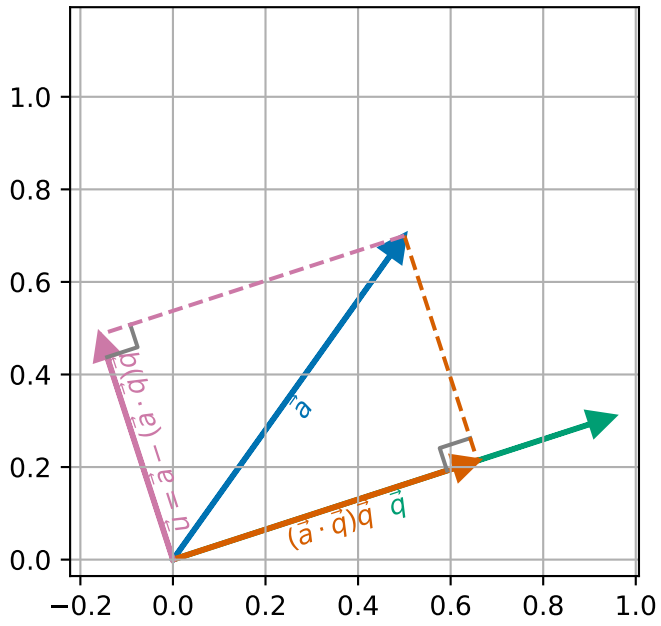
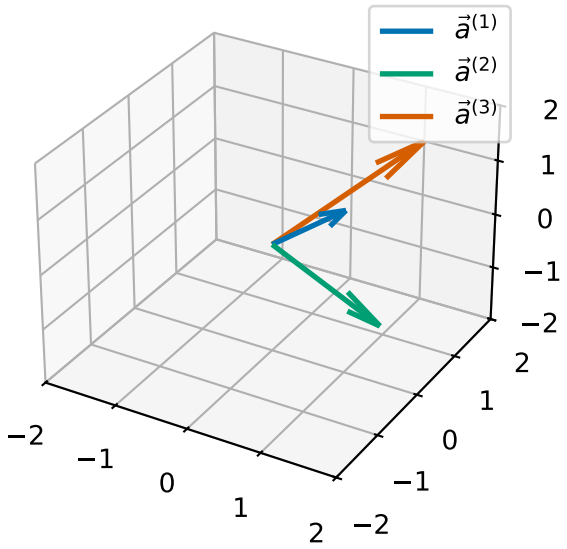



Figure 8.1: Projection of $\vec{a}$ away from $\vec{q}$ onto $\vec{u}$.




First, we set $\vec{q}^{(1)} = \vec{a}^{(1)}/\|\vec{a}^{(1)}\|$:

$$\|\vec{a}^{(1)}\| = \sqrt{1^2 + 0^2 + 1^2} = \sqrt{2},$$

so

$$\vec{q}^{(1)} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix}.$$

Second, we want to find $\vec{q}^{(2)}$ which must satisfy that $\vec{q}^{(2)} \cdot \vec{q}^{(1)} = 0$. We can do this by subtracting form $\vec{a}^{(2)}$ the portion of $\vec{a}^{(2)}$ which points in the direction $\vec{q}^{(1)}$. We call this $\vec{u}^{(2)}$:

$$\begin{aligned}
\vec{u}^{(2)} &= \vec{a}^{(2)} - \left(\vec{a}^{(2)} \cdot \vec{q}^{(1)}\right) \vec{q}^{(1)} \\
&= \begin{pmatrix} 2 \\ -1 \\ 0 \end{pmatrix} - \left( \begin{pmatrix} 2 \\ -1 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix} \right) \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix} \\
&= \begin{pmatrix} 2 \\ -1 \\ 0 \end{pmatrix} - \frac{2}{\sqrt{2}} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix} \\
&= \begin{pmatrix} 2 - \frac{2}{\sqrt{2}}\frac{1}{\sqrt{2}} \\ -1 - \frac{2}{\sqrt{2}}0 \\ 0 - \frac{2}{\sqrt{2}}\frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ -1 \end{pmatrix}.
\end{aligned}$$

We then normalise $\vec{u}^{(2)}$ to get a unit-length vector $q^{(2)}$:

$$\vec{q}^{(2)} = \vec{u}^{(2)} / \|\vec{u}^{(2)}\| = \begin{pmatrix} \frac{1}{\sqrt{3}} \\ \frac{-1}{\sqrt{3}} \\ \frac{-1}{\sqrt{3}} \end{pmatrix}.$$

Third, we will find $q^{(3)}$ which we need to check satisfies $\vec{q}^{(3)} \cdot \vec{q}^{(2)} = 0$ and $\vec{q}^{(3)} \cdot \vec{q}^{(1)} = 0$. We can do this by subtracting from $\vec{a}^{(3)}$ the portion of $\vec{a}^{(3)}$ which points in the direction $\vec{q}^{(1)}$ and the portion of $\vec{a}^{(3)}$ which points in the direction $\vec{q}^{(2)}$. We call this term $\vec{u}^{(3)}$

$$\begin{aligned}
\vec{u}^{(3)} &= \vec{a}^{(3)} - \left(\vec{a}^{(3)} \cdot \vec{q}^{(1)}\right) \vec{q}^{(1)} - \left(\vec{a}^{(3)} \cdot \vec{q}^{(2)}\right) \vec{q}^{(2)} \\
&= \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} - \left( \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix} \right) \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix} - \left( \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{\sqrt{3}} \\ \frac{-1}{\sqrt{3}} \\ \frac{-1}{\sqrt{3}} \end{pmatrix} \right) \begin{pmatrix} \frac{1}{\sqrt{3}} \\ \frac{-1}{\sqrt{3}} \\ \frac{-1}{\sqrt{3}} \end{pmatrix} \\
&= \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} - \left( \frac{2}{\sqrt{2}} \right) \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix} - \left( \frac{-2}{\sqrt{3}} \right) \begin{pmatrix} \frac{1}{\sqrt{3}} \\ \frac{-1}{\sqrt{3}} \\ \frac{-1}{\sqrt{3}} \end{pmatrix} \\
&= \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} - \begin{pmatrix} -2/3 \\ 2/3 \\ 2/3 \end{pmatrix} = \begin{pmatrix} 2/3 \\ 4/3 \\ -2/3. \end{pmatrix}.
\end{aligned}$$

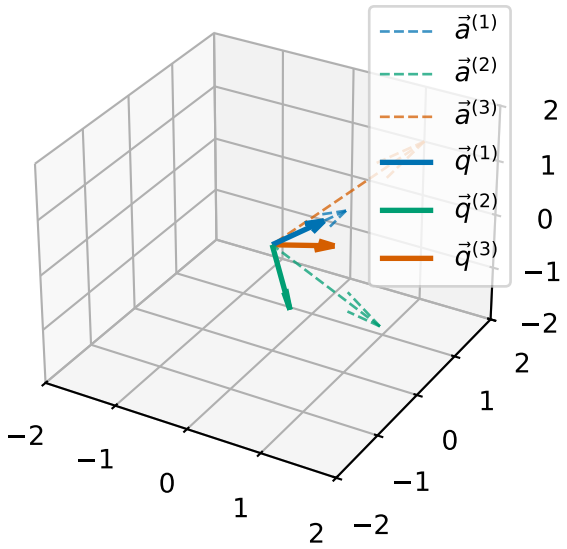
Again, we normalise $\vec{u}^{(3)}$ to get $\vec{q}^{(3)}$:

$$\begin{aligned}\|\vec{u}^{(3)}\| &= \sqrt{(2/3)^2 + (4/3)^2 + (-2/3)^2} = \sqrt{4/9 + 16/9 + 4/9} = \sqrt{24/9} \\ &= \frac{2}{3}\sqrt{6},\end{aligned}$$

so

$$\vec{q}^{(3)} = \begin{pmatrix} \frac{1}{\sqrt{6}} \\ \frac{2}{\sqrt{6}} \\ \frac{-1}{\sqrt{6}} \end{pmatrix}.$$

**Exercise 8.1.** Verify the orthonormality conditions for $\vec{q}^{(1)}, \vec{q}^{(2)}$ and $\vec{q}^{(3)}$.




Now we have applied the Gram-Schmidt process to convert from the vectors $\vec{a}^{(j)}$ to the vectors $\vec{q}^{(j)}$. We can consider the matrix $Q$ whose columns are the vectors $\vec{q}^{(j)}$ and the matrix $A$ whose columns are the vectors $\vec{a}^{(j)}$:

$$Q = \begin{pmatrix} \vec{q}^{(1)} & \vec{q}^{(2)} & \vec{q}^{(3)} \end{pmatrix} \quad \text{and} \quad A = \begin{pmatrix} \vec{a}^{(1)} & \vec{a}^{(2)} & \vec{a}^{(3)} \end{pmatrix}$$

Then we can compute that

$$(Q^T A)_{ij} = \vec{q}^{(j)} \cdot \vec{a}^{(j)}.$$

So

$$(Q^TA) = \begin{pmatrix} (1,0,1)^T \cdot (\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}})^T & (2,-1,0)^T \cdot (\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}})^T & (1,2,1)^T \cdot (\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}})^T \\ (1,0,1)^T \cdot (\frac{1}{\sqrt{3}}, \frac{-1}{\sqrt{3}}, \frac{-1}{\sqrt{3}})^T & (2,-1,0)^T \cdot (\frac{1}{\sqrt{3}}, \frac{-1}{\sqrt{3}}, \frac{-1}{\sqrt{3}})^T & (1,2,1)^T \cdot (\frac{1}{\sqrt{3}}, \frac{-1}{\sqrt{3}}, \frac{-1}{\sqrt{3}})^T \\ (1,0,1)^T \cdot (\frac{1}{\sqrt{6}}, \frac{2}{\sqrt{6}}, \frac{-1}{\sqrt{6}})^T & (2,-1,0)^T \cdot (\frac{1}{\sqrt{6}}, \frac{2}{\sqrt{6}}, \frac{-1}{\sqrt{6}})^T & (1,2,1)^T \cdot (\frac{1}{\sqrt{6}}, \frac{2}{\sqrt{6}}, \frac{-1}{\sqrt{6}})^T \end{pmatrix}$$
$$= \begin{pmatrix} \frac{2}{\sqrt{2}} & \frac{2}{\sqrt{2}} & \frac{2}{\sqrt{2}} \\ 0 & \frac{3}{\sqrt{3}} & \frac{-2}{\sqrt{3}} \\ 0 & 0 & \frac{4}{\sqrt{6}} \end{pmatrix}.$$

Hence we have found an upper triangular matrix $R = Q^TA$. Since $Q$ is orthogonal, we know $Q^{-1} = Q^T$ and we have a factorisation:

$$A = QR.$$

**Exercise 8.2.** Continue the QR-factorisation process by computing $B = RQ$ and apply the Gram-Schmidt process to the columns of $B$.

*Remark.* The Gram-Schmidt algorithm relies on the fact that after each projection there should be something left - i.e. $\vec{u}^{(j)}$ should be non-zero. If $\vec{a}^{(j)}$ is in the span of $\{\vec{q}^{(1)}, \dots, \vec{q}^{(j-1)}\}$, then the projection onto $\vec{u}^{(j)}$ will give $\vec{0}$. There are a few ways to test this, but the key idea is that if $A$ is non-singular then we will always have $\vec{u}^{(j)} \neq \vec{0}$ – at least in exact-precision calculations…

### 8.1.2 Python QR factorisation using Gram-Schmidt

```
def gram_schmidt_qr(A):
    """
    Compute the QR factorization of a square matrix using the classical
    Gram-Schmidt process.

    Parameters
    ----------
    A : numpy.ndarray
        A square 2D NumPy array of shape ``(n, n)`` representing the input
        matrix.

    Returns
    -------
    Q : numpy.ndarray
        Orthogonal matrix of shape ``(n, n)`` where the columns form an
        orthonormal basis for the column space of A.
```

```
    R : numpy.ndarray
        Upper triangular matrix of shape ``(n, n)``.
    """
    n, m = A.shape
    if n != m:
        raise ValueError(f"the matrix A is not square, {A.shape=}")

    Q = np.empty_like(A)
    R = np.zeros_like(A)

    for j in range(n):
        # Start with the j-th column of A
        u = A[:, j].copy()

        # Orthogonalize against previous q vectors
        for i in range(j):
            R[i, j] = np.dot(Q[:, i], A[:, j])  # projection coefficient
            u -= R[i, j] * Q[:, i]  # subtract the projection

        # Normalize u to get q_j
        R[j, j] = np.linalg.norm(u)
        Q[:, j] = u / R[j, j]

    return Q, R
```

Let's test it without our example above:

```
A = [  1.0,  2.0,  1.0 ]
    [  0.0, -1.0,  2.0 ]
    [  1.0,  0.0,  1.0 ]
QR factorisation:
Q = [  0.70711,  0.57735,  0.40825 ]
    [  0.00000, -0.57735,  0.81650 ]
    [  0.70711, -0.57735, -0.40825 ]
R = [  1.41421,  1.41421,  1.41421 ]
    [  0.00000,  1.73205, -1.15470 ]
    [  0.00000,  0.00000,  1.63299 ]
Have we computed a factorisation? (A == Q @ R?) True
```

## 8.2 Finding eigenvalues and eigenvectors

The algorithm given above says that we use the QR factorisation to iteratively find a sequence of matrices $A^{(j)}$ which *should* converge to an upper-triangular matrix.

We test this out in code first for the matrix from Example 7.3:

```
def gram_schmidt_eigen(A, maxiter=100, verbose=False):
    """
    Compute the eigenvalues and eigenvectors of a square matrix using the QR
    algorithm with classical Gram-Schmidt QR factorization.

    This function implements the basic QR algorithm:

    1. Factorize the matrix `A` into `Q` and `R` using Gram-Schmidt QR
       factorization.
    2. Update the matrix as:

       .. math::
           A_{k+1} = R_k Q_k

    3. Accumulate the orthogonal transformations in `V` to compute the
       eigenvectors.
    4. Iterate until `A` becomes approximately upper triangular or until the
       maximum number of iterations is reached.

    Once the iteration converges, the diagonal of `A` contains the eigenvalues,
    and the columns of `V` contain the corresponding eigenvectors.

    Parameters
    ----------
    A : numpy.ndarray
        A square 2D NumPy array of shape ``(n, n)`` representing the input
        matrix. This matrix will be **modified in place** during the
        computation.
    maxiter : int, optional
        Maximum number of QR iterations to perform. Default is 100.
    verbose : bool, optional
        If ``True``, prints intermediate matrices (`A`, `Q`, `R`, and `V`) at
        each iteration. Useful for debugging and understanding convergence.
        Default is ``False``.

    Returns
```

```
    -------
    eigenvalues : numpy.ndarray
        A 1D NumPy array of length ``n`` containing the eigenvalues of `A`.
        These are the diagonal elements of the final upper triangular matrix.
    V : numpy.ndarray
        A 2D NumPy array of shape ``(n, n)`` whose columns are the normalized
        eigenvectors corresponding to the eigenvalues.
    it : int
        The number of iterations taken by the algorithm.
    """
    # identity matrix to store eigenvectors
    V = np.eye(A.shape[0])

    if verbose:
        print_array(A)

    it = -1
    for it in range(maxiter):
        if verbose:
            print(f"\n\n{it=}")

        # perform factorisation
        Q, R = gram_schmidt_qr(A)
        if verbose:
            print_array(Q)
            print_array(R)

        # update A and V in place
        A[:] = R @ Q
        V[:] = V @ Q

        if verbose:
            print_array(A)
            print_array(V)

        # test for convergence: is A upper triangular up to tolerance 1.0e-8?
        if np.allclose(A, np.triu(A), atol=1.0e-8):
            break

    eigenvalues = np.diag(A)
    return eigenvalues, V, it
```

```
A = np.array([[3.0, 1.0], [1.0, 3.0]])
print_array(A)

eigenvalues, eigenvectors, it = gram_schmidt_eigen(A)
print_array(eigenvalues)
print_array(eigenvectors)
print("iterations required:", it)
```

```
A = [  3.0,  1.0 ]
    [  1.0,  3.0 ]
eigenvalues = [  4.0 ]
              [  2.0 ]
eigenvectors = [  0.7071, -0.7071 ]
               [  0.7071,  0.7071 ]
iterations required: 27
```

These values agree with those from Example 7.3. Note that this code normalises the eigenvectors to have length one, so we have slightly different values for the eigenvectors but still in the same directions.

## 8.3 Correctness and convergence

Let's see what happens when we try this same approach for a bigger symmetric matrix. We write a test that first checks how good the QR factorisation is for the initial matrix $A$ and then uses our approach to find eigenvalues and eigenvectors and tests how good that approximation is too.

```
# replicable seed
np.random.seed(42)

for n in [2, 4, 8, 16, 32]:
    print(f"\n\n matrix size {n=}")
    # generate a random matrix
    S = special_ortho_group.rvs(n)
    D = np.diag(np.random.randint(-5, 5, (n,)))
    A = S.T @ D @ S

    Q, R = gram_schmidt_qr(A)

    print(
```

```
        "- how accurate is Q R factorisation of A?",
        np.linalg.norm(A - Q @ R),
    )
    print("- is Q orthogonal?", np.linalg.norm(np.eye(n) - Q.T @ Q))

    print("- can we use this approach to find eigenvalues?")
    maxiter = 100_000
    D, V, k = gram_schmidt_eigen(A, maxiter=maxiter)
    if k == maxiter - 1:
        print(" -> too many iterations required")
    else:
        print(
            f" -> are the eigenvalues and eigenvectors accurate? {k=}",
            np.linalg.norm(A @ V - np.diag(D) @ V),
        )
```

```
 matrix size n=2
- how accurate is Q R factorisation of A? 1.1102230246251565e-16
- is Q orthogonal? 2.434890684976629e-16
- can we use this approach to find eigenvalues?
 -> are the eigenvalues and eigenvectors accurate? k=26 8.79893075620144e-09


 matrix size n=4
- how accurate is Q R factorisation of A? 4.965068306494546e-16
- is Q orthogonal? 3.316125075834646e-16
- can we use this approach to find eigenvalues?
 -> too many iterations required


 matrix size n=8
- how accurate is Q R factorisation of A? 1.0530671687875234e-15
- is Q orthogonal? 5.922760902330306e-16
- can we use this approach to find eigenvalues?
 -> are the eigenvalues and eigenvectors accurate? k=45 1.498289419704859e-08


 matrix size n=16
- how accurate is Q R factorisation of A? 1.5600199530047478e-15
- is Q orthogonal? 1.2863227488047944
```

```
- can we use this approach to find eigenvalues?
 -> too many iterations required


 matrix size n=32
- how accurate is Q R factorisation of A? 3.1345684296560498e-15
- is Q orthogonal? 5.464883404922812
- can we use this approach to find eigenvalues?
 -> too many iterations required
```

You will see in the lab session ways to improve our basic algorithm that allows faster, more robust convergence.