Numerical Solutions of Differential Equations

Waleed A A Ali

Table of contents

Introduction

This unit will cover some of the numerical techniques used for solving differential equations and using MATLAB to implement these numerical methods.

Material

All the material will be posted on the Microsoft Teams Page for the unit. Note that is document is regularly being updated so if you find any mistakes or parts missing then do let me know.

Unit Outline

All lectures will be held online on **Fridays**, **10.00 - 12.00** on Microsoft Teams, to which you should have received a link. The details for the link are as follows:

Meeting ID: 358 213 646 365 0

Passcode: Us2N6hB2

Lecture	Date	Topic
1	17/10	• Introduction to NSDE unit for TCC • Aims & Objectives of the unit • Floating point arithmetic • Computational complexity • Code timing and profiling • Applications for solving linear systems • Computational stability
2	24/10	• Solving linear systems using direct methods • Solving linear systems using iterative methods
3	31/10	• Euler method for IVPs
4	07/11	• Modified Euler method • Runge-Kutta method • Backwards Euler method • solving stiff IVPs
5	14/11	• Solving BVPs using the finite difference method
6	21/11	• Solving MVPs and symmetric BVPs
7	28/11	• Method of lines • Apply MoL for diffusion and/or convection
8	05/12	Stability of the method of lines

Aims & Objectives

The aim for this unit is to be able to understand and derive different numerical techniques for solving differential equations and being able to implement them on MATLAB.

Intended Learning Outcomes:

- Understand the internal working mechanisms of MATLAB,
- Solve linear systems using direct and iterative methods,
- Use different differencing schemes to assess their ability to solve ODEs and PDEs,
- Assess the stability of different numerical methods.

Questions

For any questions, queries or issues that you see in the material, do not hesitate to contact me on w.a.a.ali@bath.ac.uk.

1 Internal Workings of MATLAB

1.1 Floating-Point Arithmetic

Since computers have limited resources, only a finite strict subset $\mathcal{F}$ of the real numbers can be represented. This set of possible stored values is known as ***Floating-Point Numbers*** and these are characterised by properties that are different from those in $\mathbb{R}$, since any real number x is – in principle – truncated by the computer, giving rise to a new number denoted by $fl(x)$, which does not necessarily coincide with the original number x.

A computer represents a real number x as a floating-point number in $\mathcal{F}$ as

$$x = (-1)^s \times (a_1 a_2 \dots a_t) \times \beta^E \tag{1.1}$$

where:

- $s \in \{0, 1\}$ determines the sign of the number;
- $\beta \geq 2$ is the base;
- $E \in \mathbb{Z}$ is the exponent.
- $a_1 a_2 \dots a_t$ is the mantissa (or significand). The mantissa has length t which is the maximum number of digits that can be stored. Each term in the mantissa must satisfy $0 \leq a_i \leq \beta - 1$ for all $i = 1, 2, \dots, t$ and $a_1 \neq 0$ (to ensure that the same number cannot have different representations). The digits $a_1 a_2 \dots a_p$ (with $p \leq t$) are often called the p first significant digits of x.

The set $\mathcal{F}$ is therefore fully characterised by the basis β, the number of significant digits t and the range of values that E can take.

A computer typically uses binary representation, meaning that the base is $\beta = 2$ with the available digits $\{0, 1\}$ (also known as bits) and each digit is the coefficient of a power of 2. Available platforms (like MATLAB and Python) typically use the IEEE754 double precision format for $\mathcal{F}$, which uses 64-bits as follows:

- 1 bit for s (either 0 or 1) to determine the sign;
- 11 bits for E (which can be $0, 1, 2, \dots, 10$);
- 52 bits for $a_2 a_3 \dots a_{53}$ (since $a_1 \neq 0$, it has to be equal to 1).

For 32-bit storage, the exponent is at most 7 and the mantissa has 23 digits. Note that 0 does not belong to $\mathcal{F}$ since it cannot be represented in the form shown in Equation 1.1 and it is therefore handled separately.

The smallest and the largest positive real numbers that can be written in floating points can be found by using the `realmin` and `realmax` commands. A positive number smaller

than $x_{\min}$ yields underflow and a positive number greater than $x_{\max}$ yields overflow. The elements in $\mathcal{F}$ are more dense near $x_{\min}$, and less dense while approaching $x_{\max}$. However, the relative distance is small in both cases. Note that any number bigger than `realmax` or smaller than `-realmax` will be assigned the values ∞ and $-\infty$ respectively.

```
>> realmin
ans =
     2.2251e-308
>> realmax
ans =
     1.7977e308
```

If a non-zero real number x is replaced by its floating-point representation $fl(x) \in \mathcal{F}$, then there will inevitably be a round-off error, especially if the number is either too large or too small relative to the other numbers involved. For a floating point number x, there is a distance ε_x where any value in the interval $(x - \varepsilon_x, x + \varepsilon_x)$ cannot be written as a floating point and will therefore be assigned the value x. This interval width is called the ***Machine Epsilon*** and can be found for any floating point number x by using the command `eps(x)`.

```
>> ep1=eps(1)
ep1 =
     2.2204e-16
>> 1-(1+ep1/2)
ans =
     0
```

The larger the floating number is, the larger the machine epsilon will be, meaning that larger numbers will have much greater tolerances of error. The smaller the number is, the larger the relative size will be, rendering the numbers insginifciant overall.

```
>> eps(2^100)
ans =
     2.8147e+14
>> eps(2^-50)
ans =
     1.9722e-31
```

Since $\mathcal{F}$ is a strict subset of $\mathbb{R}$, elementary algebraic operations on floating-point numbers do not inherit all the properties of analogous operations on $\mathbb{R}$. Precisely, commutativity still holds for addition and multiplication, i.e. $fl(x + y) = fl(y + x)$ and $fl(xy) = fl(yx)$. Associativity is violated whenever a situation of overflow or underflow occurs or, similarly, whenever two numbers with opposite signs but similar absolute values are added, the result may be quite inexact and the situation is referred to as loss of significant digits.

Properly handling floating point computations can be tricky sometimes and, if not correctly done, may have serious consequences. There are many webpages (and books) collecting

examples of different disasters caused by a poor handling of computer arithmetic or a bad algorithmic implementation. See, for instance, Software Bugs and the Patriot Missile Fail among others.

1.2 Computational Complexity

The ***Computational Complexity*** of an algorithm can be defined as the relationship between the size of the input and the difficulty of running the algorithm to completion. The size (or at least, an attribute of the size) of the input is usually denoted n, for instance, for a 1-D array, n can be its length.

The difficulty of a problem can be measured in several ways. One suitable way to describe the difficulty of the problem is to count the number of ***Floating-Point Operations***, such as additions, subtractions, multiplications, divisions and assignments. Floating-point operations, also called ***flops***, usually measures the speed of a computer, measured as the maximum number of floating-point operations which the computer can execute in one second. Although each basic operation takes a different amount of time, the number of basic operations needed to complete a function is sufficiently related to the running time to be useful, and it is usually easy to count and less dependent on the specific machine (hardware) that is used to perform the computations.

A common notation for complexity is the ***Big-O*** notation (denoted $\mathcal{O}$), which establishes the relationship in the growth of the number of basic operations with respect to the size of the input as the input size becomes very large. In general, the basic operations grow in direct response to the increase in the size n of the input and, as n gets large, the highest power dominates. Therefore, only the highest power term is included in Big-O notation; moreover, coefficients are not required to characterise growth and are usually dropped (although this will also depend on the precision of the estimates).

Formally, a function f behaves as $f(x) \sim \mathcal{O}\left(p(x)\right)$ as x tends to infinity if

$$\lim_{x\to\infty} \frac{f(x)}{p(x)} = \text{constant}.$$

For example, the polynomial $f(x) = x^4 + 2x^2 + x + 5$ behaves like x^4 as x tends to infinity since this term will be the fastest to grow. This can be written as $f(x) \sim \mathcal{O}\left(x^4\right)$ as $x \to \infty$.

Couting flops

Let $f : \mathbb{N} \to \mathbb{N}$ be given by

$$f(n) = \left(\sum_{j=1}^{n} j\right)^2$$

This function f can be coded as `fun` in MATLAB as follows:

```
function [out]=fun(n)

out = 0;

for i=1:1:n

     for j=1:1:n

          out = out + i*j;

     end

end

end
```

For example, $f(3)$ should perform the overall calculation

$$(1 \times 1) + (1 \times 2) + (1 \times 3) + (2 \times 1) + (2 \times 2) + (2 \times 3) + (3 \times 1) + (3 \times 2) + (3 \times 3),$$

so `fun(3)` should output `out=36`.
This code requires the following operations:

- $1 + n + 2n^2$ assignments:
  - 1: `out=0`;
  - n: `i=1:1:n`;
  - n^2: for every `i`, `j=1:1:n`;
  - n^2: for every `i`, `out=out+i*j`;
- n^2 multiplications: `i*j`;
- n^2 additions: `out=out+i*j`.

Therefore, for any n, this code will need $4n^2+n+1$ flops, meaning that the computational complexity is $\mathcal{O}(n^2)$, i.e. the code runs in polynomial time. It is not uncommon to find algorithms that run in exponential time $\mathcal{O}(c^n)$, like some recursive algorithms, or in logarithmic time $\mathcal{O}(\log n)$.

For more complicated codes, it is important to see where most of the time is spent in a code and how execution can be improved. A rudimentary way of timing can be done by the `toc` `toc`:

```
>> tic;
>> Run code or code block
>> toc;
```

This will produce a simple time in seconds that MATLAB took from `tic` until `toc`, so if

`toc` has not been types, then the timer will continue.

For more advanced analysis, MATLAB uses a **Code Profiler** to analyse code which includes run times for each iteration, times a code has been called and a lot more.

Iterative vs Recursive

Suppose that a code needs to be written that finds the N^{th} Fibonacci number starting the sequence with (1,1). This can be done in two ways:

- **Iteratively** by having a self-contained code that generates all the terms of the sequence up to N and displays the last term.

```
function [F]=Fib_Iter(N)

S=ones(1,N);

for n=3:1:N

     S(n)=S(n-1)+S(n-2);

end

F=S(end);

end
```

- **Recursively** by have a self-referential code that keeps referring back to itself to generate the last term in the sequence from the previous terms.

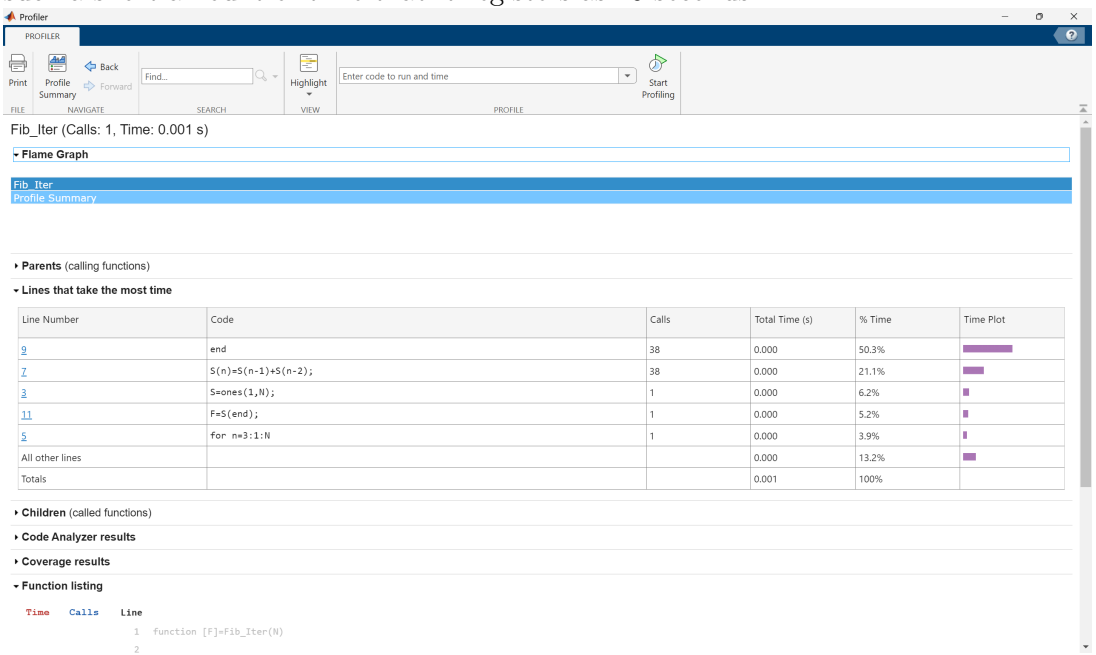
```
function [F]=Fib_Rec(N)

if N<3

     F=1;

else

     F=Fib_Rec(N-1)+Fib_Rec(N-2);

end

end
```

When running these codes for an input of $N = 10$, the times are very short, of the order of 10^{-5} seconds but as N gets larger, the recursive code starts to take much

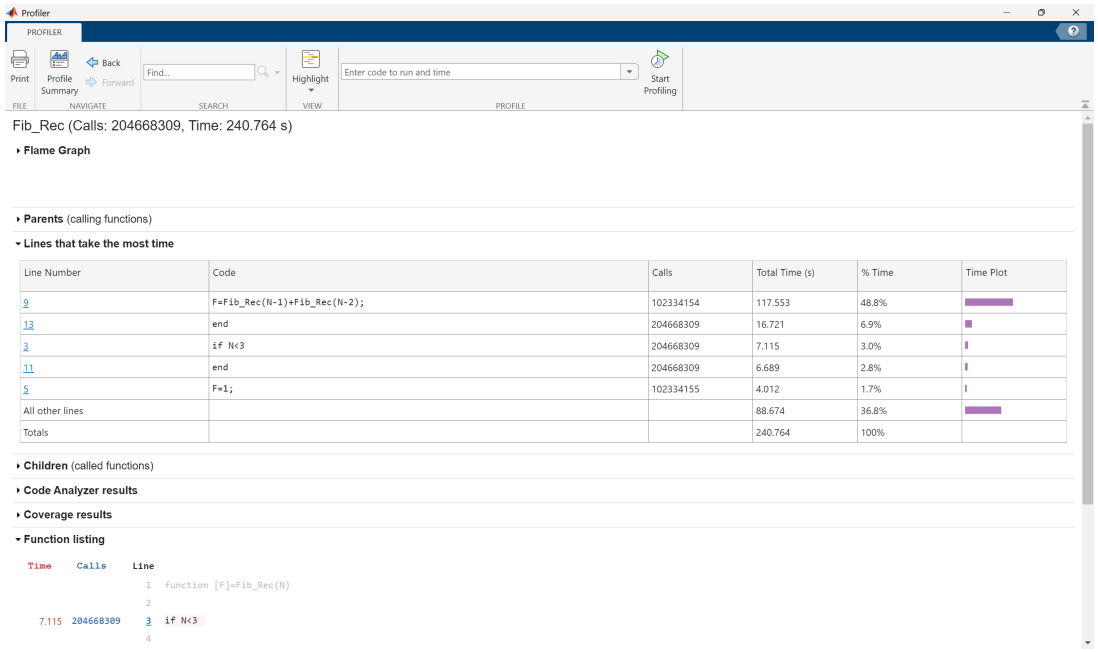
longer. Suppose the code efficiency is to be analysed for the input $N = 40$, this can be done using the profiler as follows:

```
>> profile on
>> Fib_Iter(40);
>> profile off
>> profile viewer
```

This will give a full breakdown of how many times every line was run and how much time it took. For `Fib_Iter(40)`, a total of 38 operations were performed, each taking such a short amount of time that it registers as "0 seconds".



However, performing the profiler for `Fib_Rec(40)` gives a *dramatically* different answer with the code taking nearly 247 seconds and having to call itself more than 102 million times.



This is why it is important to profile longer codes to see which parts take the longest time and which loops are the most time consuming.

Good Practice

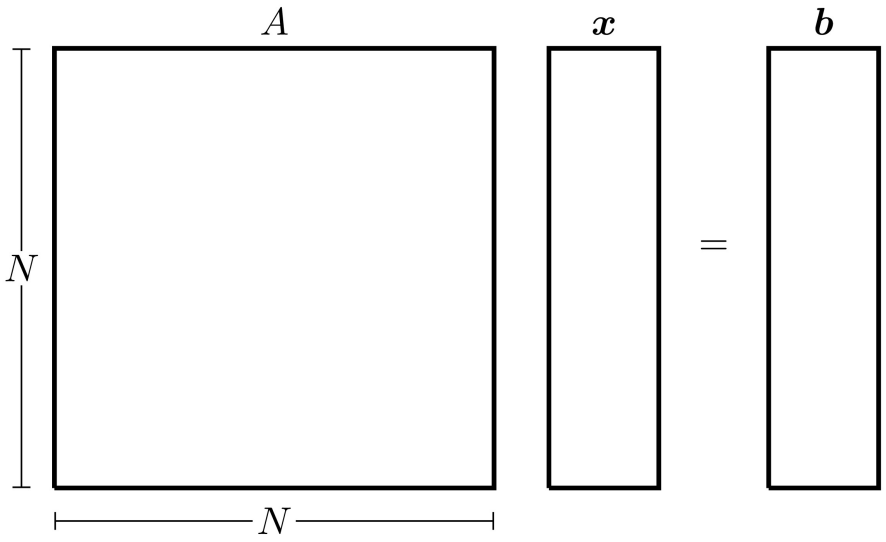
To reduce computational time in general, avoid self-referential codes because these tend to grow in usage exponentially. Another important practice is to use in-built MATLAB syntax, like using `sum` to add elements in a vector rather than manually hard coding it. This is where being familiar with a lot of the MATLAB syntax is important; MATLAB has a lot of built-in codes and syntaxes which can save a lot of time.

Part I

Linear Algebra

This section will cover some of the main methods that can be used to solve sets of linear equations of the form

$$A\boldsymbol{x} = \boldsymbol{b} \quad \text{where} \quad A \in \mathbb{R}^{N\times N}, \quad \boldsymbol{x} \in \mathbb{R}^{N}, \quad \boldsymbol{b} \in \mathbb{R}^{N}.$$



This can be done by using a *Direct Method* if the solution of the system can be obtained in a finite number of steps or an *Iterative Method* if the solution, in principle, requires an infinite number of steps.

The choice between direct and iterative methods may depend on several factors, primarily the predicted theoretical efficiency of the scheme, but also the particular type of matrix (such as systems that are sparse, diagonally dominant, tridiagonal and so forth) and the memory storage requirements.

2 Solving Linear Systems of Equations

Before embarking on the main purpose of the course, which is solving differential equations, first solving linear systems will be necessary. The linear systems will take the form

$$A\boldsymbol{x} = \boldsymbol{b} \quad \text{where} \quad A \in \mathbb{C}^{N\times N}, \boldsymbol{x} \in \mathbb{C}^N \quad \text{and} \quad \boldsymbol{b} \in \mathbb{C}^N.$$

This is a situation when the LHS forms a system of equations with a vector of unknowns $\boldsymbol{x}$ and the RHS is known.

Simple Example of a Linear System

Let a, b and c be integers such that:

- their sum is equal to 20
- a is twice as large as b
- b is bigger than c by 10.

These three relationships can be written in equation form as:

$$a + b + c = 20$$

$$a = 2b$$

$$b - c = 10$$

This can written in matrix form as:

$$\underbrace{\begin{pmatrix} 1 & 1 & 1 \\ 1 & -2 & 0 \\ 0 & 1 & -1 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} a \\ b \\ c \end{pmatrix}}_{\boldsymbol{x}} = \underbrace{\begin{pmatrix} 20 \\ 0 \\ 10 \end{pmatrix}}_{\boldsymbol{b}}.$$

There are two main ways in which this can be done, depending on the form of the matrix:

- Direct Methods:
  - Direct substitution for diagonal systems;
  - Forward substitution for lower triangular systems;
  - Backward substitution for upper triangular systems;
  - TDMA for tridiagonal systems;
  - Cramer's Rule and Gaussian Elimination for more general matrix systems.
- Iterative Methods

  - Jacobi;
  - Gauss-Seidel.
- In-built Methods:
  - Backsklash operator.

2.1 Computational Stability of Linear Systems

Before tackling any linear algebra techniques, it is important to understand ***Computational Stability***.

Consider the linear system

$$A\boldsymbol{x} = \boldsymbol{b} \quad \text{where} \quad A \in \mathbb{C}^{N\times N}, \boldsymbol{x} \in \mathbb{C}^N \quad \text{and} \quad \boldsymbol{b} \in \mathbb{C}^N.$$

In real-life applications, the matrix A is usually fully known and often invertible while the vector $\boldsymbol{b}$ may not be known exactly and its measurement may often include rounding errors. Suppose that the vector $\boldsymbol{b}$ has a small error $\delta\boldsymbol{b}$, then the solution $\boldsymbol{x}$ will also have a small error $\delta\boldsymbol{x}$, meaning that the system will in fact be

$$A(\boldsymbol{x} + \delta\boldsymbol{x}) = \boldsymbol{b} + \delta\boldsymbol{b}. \tag{2.1}$$

Subtracting $A\boldsymbol{x} = \boldsymbol{b}$ form Equation 2.1 gives $A\delta\boldsymbol{x} = \delta\boldsymbol{b}$, therefore $\delta\boldsymbol{x} = A^{-1}\delta\boldsymbol{b}$.

For $p \in \mathbb{N}$, consider the ratio between the p-norm of the error $\|\delta\boldsymbol{x}\|_p$ and the p-norm of the exact solution $\|\boldsymbol{x}\|_p$:

$$\begin{aligned}
\frac{\|\delta\boldsymbol{x}\|_p}{\|\boldsymbol{x}\|_p} &= \frac{\|A^{-1}\delta\boldsymbol{b}\|_p}{\|\boldsymbol{x}\|_p} && \text{since} \quad \delta\boldsymbol{x} = A^{-1}\delta\boldsymbol{b} \\
&\leq \frac{\|A^{-1}\|_p \|\delta\boldsymbol{b}\|_p}{\|\boldsymbol{x}\|_p} && \text{by the } \textit{Submultiplicative Property} \\
&= \frac{\|A^{-1}\|_p \|\delta\boldsymbol{b}\|_p}{\|\boldsymbol{x}\|_p} \times \frac{\|A\|_p}{\|A\|_p} && \text{multiplying by} \quad 1 = \frac{\|A\|_p}{\|A\|_p} \\
&= \|A\|_p \left\|A^{-1}\right\|_p \frac{\|\delta\boldsymbol{b}\|_p}{\|A\|_p \|\boldsymbol{x}\|_p} && \text{rearranging} \\
&\leq \|A\|_p \left\|A^{-1}\right\|_p \frac{\|\delta\boldsymbol{b}\|_p}{\|\boldsymbol{b}\|_p} && \text{since } \boldsymbol{b} = A\boldsymbol{x} \text{ then } \|\boldsymbol{b}\|_p \leq \|A\|_p \|\boldsymbol{x}\|_p \\
& && \text{by the Submultiplicative Property,} \\
& && \text{meaning that } \frac{1}{\|\boldsymbol{b}\|_p} \geq \frac{1}{\|A\|_p \|\boldsymbol{x}\|_p}
\end{aligned}$$

Note 1: Submultiplicative Property of Matrix Norms

For a matrix A and a vector $\boldsymbol{x}$,

$$\|A\boldsymbol{x}\| \leq \|A\|\|\boldsymbol{x}\|.$$

Let $\kappa_p(A) = \left\|A^{-1}\right\|_p \|A\|_p$, then

$$\frac{\|\delta\boldsymbol{x}\|_p}{\|\boldsymbol{x}\|_p} \leq \kappa_p(A)\frac{\|\delta\boldsymbol{b}\|_p}{\|\boldsymbol{b}\|_p}$$

The quantity $\kappa_p(A)$ is called the ***Condition Number***[1] and it can be regarded as a measure of how sensitive a matrix is to perturbations, in other words, it gives an indication as to the stability of the matrix system. A problem is ***Well-Conditioned*** if the condition number is small, and is ***Ill-Conditioned*** if the condition number is large (the terms "*small*" and "*large*" are somewhat subjective here and will depend on the context). Bear in mind that in practice, calculating the condition number may be computationally expensive since it requires inverting the matrix A.

The condition number derived above follows the assumption that the error only occurs in $\boldsymbol{b}$ which then results in an error in $\boldsymbol{x}$. If an error δA is also committed in A, then for sufficiently small δA, the error bound for the ratio is

$$\frac{\|\delta\boldsymbol{x}\|_p}{\|\boldsymbol{x}\|_p} \leq \frac{\kappa_p(A)}{1 - \kappa_p(A)\frac{\|\delta A\|_p}{\|A\|_p}}\left(\frac{\|\delta\boldsymbol{b}\|_p}{\|\boldsymbol{b}\|_p} + \frac{\|\delta A\|_p}{\|A\|_p}\right).$$

An example for which A is large is a discretisation matrix of a PDE, in this case, the condition number of A can be very large and increases rapidly as the number of mesh points increases. For example, for a PDE with N mesh points in 2-dimensions, the condition number $\kappa_2(A)$ is of order $\mathcal{O}(N)$ and it is not uncommon to have N between 10^6 and 10^8. In this case, errors in $\boldsymbol{b}$ may be amplified enormously in the solution process. Thus, if $\kappa_p(A)$ is large, there may be difficulties in solving the system reliably, a problem which plagues calculations with partial differential equations.

Moreover, if A is large, then the system $A\boldsymbol{x} = \boldsymbol{b}$ may be solved using an *iterative method* which generate a sequence of approximations $\boldsymbol{x}_n$ to $\boldsymbol{x}$ while ensuring that each iteration is easy to perform and that $\boldsymbol{x}_n$ rapidly tends to $\boldsymbol{x}$, within a certain tolerance, as n tends to infinity. If $\kappa_p(A)$ is large, then the number of iterations to reach this tolerance increases rapidly as the size of A increases, often being proportional to $\kappa_p(A)$ or even to $\kappa_p(A)^2$. Thus not only do errors in $\boldsymbol{x}$ accumulate for large $\kappa_p(A)$, but the number of computation required to find $\boldsymbol{x}$ increases as well.

In MATLAB, the condition number can be calculated using the `cond(A,p)` command where `A` is the square matrix in question and `p` is the chosen norm which can only be equal to `1`, `2`,

[1] Note that A^{-1} exists only if A is non-singular, meaning that the condition number number only exists if A is non-singular.

`inf` or `'Fro'` (when using the Frobenius norm). Also note that `cond(A)` without the second argument `p` produces the condition number with the 2-norm by default.

Properties of the Condition Number

Let A and B be invertible matrices, $p \in \mathbb{N}$ and $\lambda \in \mathbb{R}$. The condition number κ_p has the following properties:

- $\kappa_p(A) \geq 1$;
- $\kappa_p(A) = 1$ if and only if A is an orthogonal matrix, i.e. $A^{-1} = A^{\mathrm{T}}$;
- $\kappa_p(A^{\mathrm{T}}) = \kappa_p(A^{-1}) = \kappa_p(A)$;
- $\kappa_p(\lambda A) = \kappa_p(A)$;
- $\kappa_p(AB) \leq \kappa_p(A)\kappa_p(B)$.

2.2 Direct Methods

Direct methods can be used to solve matrix systems in a finite number of steps, although these steps could possibly be computationally expensive.

2.2.1 Direct Substitution

Direct substitution is the simplest direct method and requires the matrix A to be a diagonal with none of the diagonal terms being 0 (otherwise the matrix will not be invertible).

Consider the matrix system $A\boldsymbol{x} = \boldsymbol{b}$ where

$$A = \begin{pmatrix} a_1 & & & & \\ & a_2 & & & \\ & & \ddots & & \\ & & & a_{N-1} & \\ & & & & a_N \end{pmatrix}, \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{N-1} \\ b_N \end{pmatrix}$$

and $a_1, a_2, \ldots, a_N \neq 0$. Direct substitution involves simple multiplication and division:

$$A\boldsymbol{x} = \boldsymbol{b} \quad \Longrightarrow \quad \begin{pmatrix} a_1 & & & & \\ & a_2 & & & \\ & & \ddots & & \\ & & & a_{N-1} & \\ & & & & a_N \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{N-1} \\ b_N \end{pmatrix}$$

$$\Longrightarrow \quad \begin{matrix} a_1x_1 = b_1 \\ a_2x_2 = b_2 \\ \vdots \\ a_{N-1}x_{N-1} = b_{N-1} \\ a_Nx_N = b_N \end{matrix} \quad \Longrightarrow \quad \begin{matrix} x_1 = \frac{b_1}{a_1} \\ x_2 = \frac{b_2}{a_2} \\ \vdots \\ x_{N-1} = \frac{b_{N-1}}{a_{N-1}} \\ x_N = \frac{b_N}{a_N}. \end{matrix}$$

The solution can be written explicitly as $x_n = \frac{b_n}{a_n}$ for all $n = 1, 2, \ldots, N$. Every step can done independently, meaning that direct substitution lends itself well to parallel computing. In total, direct substitution requires exactly N computations (all being division).

Example of Direct Substituion

Consider the system $A\boldsymbol{x} = \boldsymbol{b}$ where

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -1 \end{pmatrix}, \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{pmatrix} 4 \\ 2 \\ 4 \end{pmatrix}.$$

Solving the system using direct substitution:

$$A\boldsymbol{x} = \boldsymbol{b} \quad \Longrightarrow \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \\ 4 \end{pmatrix}$$

$$\Longrightarrow \quad \begin{pmatrix} x_1 \\ 2x_2 \\ -x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \\ 4 \end{pmatrix} \quad \Longrightarrow \quad \begin{aligned} x_1 &= 4 \\ x_2 &= 1 \\ x_3 &= -4. \end{aligned}$$

2.2.2 Forward/Backward Substitution

Forward/backward substitution require that the matrix A be lower/upper triangular.

Consider the matrix system $A\boldsymbol{x} = \boldsymbol{b}$ where

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1,N-1} & a_{1N} \\ & a_{22} & \ldots & a_{2,N-1} & a_{2N} \\ & & \ddots & \vdots & \\ & & & a_{N-1,N-1} & a_{N-1,N} \\ & & & & a_{NN} \end{pmatrix},$$

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{N-1} \\ b_N \end{pmatrix}$$

and $a_{11}, a_{22}, \ldots, a_{NN} \neq 0$ (so that the determinant is non-zero). The matrix A is upper

triangular in this case and will require backwards substitution:

$$A\boldsymbol{x} = \boldsymbol{b} \quad \Longrightarrow \quad \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1,N-1} & a_{1N} \\ & a_{22} & \dots & a_{2,N-1} & a_{2N} \\ & & \ddots & \vdots & \\ & & & a_{N-1,N-1} & a_{N-1,N} \\ & & & & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{N-1} \\ b_N. \end{pmatrix}$$

$$\begin{aligned} & a_{11}x_1 + a_{12}x_2 + \dots + a_{1,N-1}x_{N-1} + a_{1N}x_N = b_1 \\ & \qquad a_{22}x_2 + \dots + a_{2,N-1}x_{N-1} + a_{2N}x_N = b_2 \\ \Longrightarrow \qquad & \qquad\qquad \vdots \\ & \qquad\qquad\qquad a_{N-1,N-1}x_{N-1} + a_{N-1,N}x_N = b_{N-1} \\ & \qquad\qquad\qquad\qquad\quad a_{NN}x_N = b_N \end{aligned}$$

Backward substitution involves using the solutions from the later equations to solve the earlier ones, this gives:

$$\begin{aligned} x_N &= \frac{b_N}{a_{NN}} \\ x_{N-1} &= \frac{b_{N-1} - a_{N-1,N}x_N}{a_{N-1,N-1}} \\ &\vdots \\ x_2 &= \frac{b_2 - a_{2N}x_N - a_{2,N-1}x_{N-1} - \dots - a_{23}x_3}{a_{22}} \\ x_1 &= \frac{b_1 - a_{1N}x_N - a_{1,N-1}x_{N-1} - \dots - a_{12}x_2}{a_{11}}. \end{aligned}$$

This can be written more explicitly as:

$$x_n = \begin{cases} \frac{b_N}{a_{NN}} & \text{for} \quad n = N \\ \frac{1}{a_{nn}} \left(b_n - \sum_{i=n+1}^{N} a_{ni}x_i \right) & \text{for} \quad n = N-1, \dots, 2, 1. \end{cases}$$

A similar version can be obtained for the forward substitution for lower triangular matrices as follows:

$$x_n = \begin{cases} \frac{b_1}{a_{11}} & \text{for} \quad n = 1 \\ \frac{1}{a_{nn}} \left(b_n - \sum_{i=1}^{n-1} a_{ni}x_i \right) & \text{for} \quad n = 2, 3, \dots, N-1. \end{cases}$$

For any $n = 1, 2, \dots, N-1$, calculating it requires 1 division, $N - n$ multiplications and $N-n$ subtractions. Therefore cumulatively, $x_1, x_2, \dots, x_{N-1}$ require N divisions, $\frac{1}{2}\left(N^2 - N\right)$ multiplications and $\frac{1}{2}\left(N^2 - N\right)$ additions with one more division required for x_N, meaning that in total, backward (and forward) substitution requires $N^2 + 1$ computations.

Example of Backward Substitution

Consider the system $A\boldsymbol{x} = \boldsymbol{b}$ where

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 0 & -1 & 4 \\ 0 & 0 & -1 \end{pmatrix}, \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}.$$

This problem can be solved by suing backward substitution:

$$A\boldsymbol{x} = \boldsymbol{b} \implies \begin{pmatrix} 1 & 2 & 1 \\ 0 & -1 & 4 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \implies \begin{matrix} x_1 + 2x_2 + x_3 = 1 \\ -x_2 + 4x_3 = 0 \\ -x_3 = 1 \end{matrix}$$

$$-x_2 + 4x_3 = 0 \underset{x_3=-1}{\Rightarrow} -x_2 - 4 = 0 \implies x_2 = -4$$

$$x_1 + 2x_2 + x_3 = 1 \underset{x_2=-4,\, x_3=-1}{\Rightarrow} x_1 - 8 - 1 = 1 \implies x_1 = 10.$$

2.2.3 TDMA Algorithm

The ***TriDiagonal Matrix Algorithm***, abbreviated as ***TDMA*** (also called the ***Thomas Algorithm***) was developed by Llewellyn Thomas which solves tridiagonal matrix systems.

Consider the matrix system $A\boldsymbol{x} = \boldsymbol{b}$ where

$$A = \begin{pmatrix} m_1 & r_1 & & & \\ l_2 & m_2 & r_2 & & \\ & \ddots & \ddots & \ddots & \\ & & l_{N-1} & m_{N-1} & r_{N-1} \\ & & & l_N & m_N \end{pmatrix},$$

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{N-1} \\ b_N \end{pmatrix}.$$

The m terms denote the diagonal elements, l denote subdiagonal elements (left of the diagonal terms) and r denote the superdiagonal elements (right of the diagonal terms). The TDMA algorithm works in two steps: first, TDMA performs a forward sweep to eliminate all the subdiagonal terms and rescale the matrix to have 1 as the diagonal (the same can

also be done to eliminate the superdiagonal instead). This give the matrix system

$$\begin{pmatrix} 1 & R_1 & & & \\ & 1 & R_2 & & \\ & & \ddots & \ddots & \\ & & & 1 & R_{N-1} \\ & & & & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_{N-1} \\ B_N \end{pmatrix}$$

where

$$R_n = \begin{cases} \frac{r_1}{m_1} & n = 1 \\ \frac{r_n}{m_n - l_n R_{n-1}} & n = 2, 3, \ldots, N-1 \end{cases}$$

$$B_n = \begin{cases} \frac{b_1}{m_1} & n = 1 \\ \frac{b_n - l_n B_{n-1}}{m_n - l_n R_{n-1}} & n = 2, 3, \ldots, N. \end{cases}$$

This can now be solved with backward substitution:

$$x_n = \begin{cases} B_N & n = N \\ B_n - R_n x_{n+1} & n = N-1, N-2, \ldots, 2, 1. \end{cases}$$

The computational complexity can be calculated as follows:

Term	$\times$	$+$	$\div$
R_1	0	0	1
R_2	1	1	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
R_{N-1}	1	1	1
B_1	0	0	1
B_2	2	2	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
B_{N-1}	2	2	1
B_N	2	2	1
x_1	1	1	0
x_2	1	1	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$
x_{N-1}	1	1	0

This gives a total of $3N - 5$ computations for R, $5N - 4$ computations for B and $2N - 2$ computations for x giving a total of $10N - 11$ computations.

There are similar ways of performing eliminations that be done for pentadiagonal systems as well as tridiagonal systems with a full first row.

2.2.4 Cramer's Rule

Cramer's Rule is a method that can be used to solve *any* system $A\boldsymbol{x} = \boldsymbol{b}$ (of course provided that A is non-singular).

Cramer's rule states that the elements of the vector $\boldsymbol{x}$ are given by

$$x_n = \frac{\det(A_n)}{\det(A)} \quad \text{for all} \quad n = 1, 2, \ldots, N$$

where A_n is the matrix obtained from A by replacing the n^{th} column by $\boldsymbol{b}$. This method seems very simple to execute thanks to its very simple formula, but in practice, it can be very computationally expensive.

Example of Cramer's Rule

Consider the system $A\boldsymbol{x} = \boldsymbol{b}$ where

$$A = \begin{pmatrix} 0 & 4 & 7 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{pmatrix} 14 \\ 1 \\ 7 \end{pmatrix}.$$

The determinant of A is equal to 7. Using Cramer's rule, the solution $\boldsymbol{x} = (x_1,\ x_2,\ x_3)^{\mathrm{T}}$ can be calculated as:

$$x_1 = \frac{\det(A_1)}{\det(A)} = \frac{\det\begin{pmatrix} 14 & 4 & 7 \\ 1 & 0 & 1 \\ 7 & 1 & 0 \end{pmatrix}}{7} = \frac{21}{7} = 3.$$

$$x_2 = \frac{\det(A_1)}{\det(A)} = \frac{\det\begin{pmatrix} 0 & 14 & 7 \\ 1 & 1 & 1 \\ 0 & 7 & 0 \end{pmatrix}}{7} = \frac{49}{7} = 7.$$

$$x_3 = \frac{\det(A_1)}{\det(A)} = \frac{\det\begin{pmatrix} 0 & 4 & 14 \\ 1 & 0 & 1 \\ 0 & 1 & 7 \end{pmatrix}}{7} = \frac{-14}{7} = -2.$$

Generally, for a matrix of size $N \times N$, the determinant will require $\mathcal{O}(N!)$ computations (other matrix forms or methods may require fewer, of $\mathcal{O}(N^3)$ at least). Cramer's rule requires calculating the determinants of $N+1$ matrices each is size $N \times N$ and performing N divisions, therefore the computational complexity of Cramer's rule is $\mathcal{O}(N + (N+1) \times N!) = \mathcal{O}(N + (N+1)!)$. This means that if a machine runs at 1 Gigaflops per second (10^9 flops), then a matrix system of size 20×20 will require 1620 years to compute.

2.2.5 Other Direct Methods

There are many other direct methods with more involved calculations like *Gaussian Elimination*, *LU factorisation*, *QR decomposition*, *Singular Value Decomposition* amongst others. All these methods will be placed in the appendix.

2.3 Iterative Methods

For a large matrix A, solving the system $A\boldsymbol{x} = \boldsymbol{b}$ directly can be computationally restrictive as seen in the different methods shown in Section 2.2. An alternative would be to use *iterative* methods which generate a sequence of approximations $\boldsymbol{x}^{(k)}$ to the exact solution $\boldsymbol{x}$. The hope is that the iterative method converges to the exact solution, i.e.

$$\lim_{k\to\infty} \boldsymbol{x}^{(k)} = \boldsymbol{x}.$$

A possible strategy to realise this process is to consider the following recursive definition

$$\boldsymbol{x}^{(k)} = B\boldsymbol{x}^{(k-1)} + \boldsymbol{g} \quad \text{for} \quad k \geq 1,$$

where B is a suitable matrix called the ***Iteration Matrix*** (which would generally depend on A) and $\boldsymbol{g}$ is a suitable vector (depending on A and $\boldsymbol{b}$). Since the iterations $\boldsymbol{x}^{(k)}$ must tend to $\boldsymbol{x}$ as k tends to infinity, then

$$\boldsymbol{x}^{(k)} = B\boldsymbol{x}^{(k-1)} + \boldsymbol{g} \tag{2.2}$$

$$\underset{k\to\infty}{\Rightarrow} \quad \boldsymbol{x} = B\boldsymbol{x} + \boldsymbol{g}. \tag{2.3}$$

Next, a sufficient condition needs to be derived; define $\boldsymbol{e}^{(k)}$ as the error incurred from iteration k, i.e. $\boldsymbol{e}^{(k)} := \boldsymbol{x} - \boldsymbol{x}^{(k)}$ and consider the linear systems

$$\boldsymbol{x} = B\boldsymbol{x} + \boldsymbol{g} \quad \text{and} \quad \boldsymbol{x}^{(k)} = B\boldsymbol{x}^{(k-1)} + \boldsymbol{g}.$$

Subtracting these gives

$$\begin{aligned} & \boldsymbol{x} - \boldsymbol{x}^{(k)} = (B\boldsymbol{x} + \boldsymbol{g}) - \left(B\boldsymbol{x}^{(k-1)} + \boldsymbol{g}\right) \\ \implies \quad & \boldsymbol{x} - \boldsymbol{x}^{(k)} = B\left(\boldsymbol{x} - \boldsymbol{x}^{(k-1)}\right) \\ \implies \quad & \boldsymbol{e}^{(k)} = B\boldsymbol{e}^{(k-1)}. \end{aligned}$$

In order to find a bound for the error, take the 2-norm of the error equation

$$\boldsymbol{e}^{(k)} = B\boldsymbol{e}^{(k-1)} \quad \underset{\|\cdot\|_2}{\Rightarrow} \quad \|\boldsymbol{e}^{(k)}\|_2 = \|B\boldsymbol{e}^{(k-1)}\|_2.$$

By the submultiplicative property of matrix norms given in Note 1, the error $\|\boldsymbol{e}^{(k)}\|$ can be bounded above as

$$\left\|\boldsymbol{e}^{(k)}\right\|_2 = \left\|B\boldsymbol{e}^{(k-1)}\right\|_2 \leq \|B\|_2 \left\|\boldsymbol{e}^{(k-1)}\right\|_2.$$

This can be iterated backwards, so for $k \geq 1$,

$$\|\boldsymbol{e}^{(k)}\|_2 \leq \|B\|_2 \|\boldsymbol{e}^{(k-1)}\|_2 \leq \|B\|_2^2 \|\boldsymbol{e}^{(k-2)}\|_2 \leq \cdots \leq \|B\|_2^k \|\boldsymbol{e}^{(0)}\|_2.$$

Generally, this means that the error at any iteration k can be bounded above by the error at the initial iteration $\boldsymbol{e}^{(0)}$. Therefore, since $\boldsymbol{e}^{(0)}$ is arbitrary, if $\|B\|_2 < 1$ then the set of vectors $\left\{\boldsymbol{x}^{(k)}\right\}_{k\in\mathbb{N}}$ generated by the iterative scheme $\boldsymbol{x}^{(k)} = B\boldsymbol{x}^{(k-1)} + \boldsymbol{g}$ will converge to the exact solution $\boldsymbol{x}$ which solves $A\boldsymbol{x} = \boldsymbol{b}$, hence giving a sufficient condition for convergence.

2.3.1 Constructing an Iterative Method

A general technique to devise an iterative method to solve $A\boldsymbol{x} = \boldsymbol{b}$ is based on a "splitting" of the matrix A. First, write the matrix A as $A = P - (P - A)$ where P is a suitable non-singular matrix (somehow linked to A and "easy" to invert). Then

$$\begin{aligned} P\boldsymbol{x} &= [A + (P - A)]\,\boldsymbol{x} && \text{since } P = A + P - A \\ &= (P - A)\boldsymbol{x} + A\boldsymbol{x} && \text{expanding} \\ &= (P - A)\boldsymbol{x} + \boldsymbol{b} && \text{since } A\boldsymbol{x} = \boldsymbol{b} \end{aligned}$$

Therefore, the vector $\boldsymbol{x}$ can be written implicitly as

$$\boldsymbol{x} = P^{-1}(P - A)\boldsymbol{x} + P^{-1}\boldsymbol{b}$$

which is of the form given in Equation 2.3 where $B = P^{-1}(P - A) = I - P^{-1}A$ and $\boldsymbol{g} = P^{-1}\boldsymbol{b}$. It would then stand to reason that if the iterative procedure was of the form

$$\boldsymbol{x}^{(k)} = P^{-1}(P - A)\boldsymbol{x}^{(k-1)} + P^{-1}\boldsymbol{b}$$

(as in Equation 2.2), then the method should converge to the exact solution (provided a suitable choice for P). Of course, for the iterative procedure, the iteration needs an initial vector to start which will be

$$\boldsymbol{x}^{(0)} = \begin{pmatrix} x_1^{(0)} \\ x_2^{(0)} \\ \vdots \\ x_N^{(0)} \end{pmatrix}.$$

The choice of the matrix P should depend on A in some way. So suppose that the matrix A is broken down into three parts, $A = D + L + U$ where D is the matrix of the diagonal entries of A, L is the strictly lower triangular part or A (i.e. not including the diagonal) and U is the strictly upper triangular part of A.

Note

For example

$$\underbrace{\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} a & 0 & 0 \\ 0 & e & 0 \\ 0 & 0 & i \end{pmatrix}}_{D} + \underbrace{\begin{pmatrix} 0 & 0 & 0 \\ d & 0 & 0 \\ g & h & 0 \end{pmatrix}}_{L} + \underbrace{\begin{pmatrix} 0 & b & c \\ 0 & 0 & f \\ 0 & 0 & 0 \end{pmatrix}}_{U}.$$

- ***Jacobi Method*: $P = D$**

The matrix P is chosen to be equal to the diagonal part of A, then the splitting procedure gives the iteration matrix $B = I - D^{-1}A$ and the iteration itself is $\boldsymbol{x}^{(k+1)} = B\boldsymbol{x}^{(k)} + D^{-1}\boldsymbol{b}$ for $k \geq 0$, which can be written component-wise as

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{N} a_{ij} x_j^{(k)} \right) \quad \text{for all} \quad i = 1, \ldots, N. \tag{2.4}$$

If A is strictly diagonally dominant by rows[2], then the Jacobi method converges (i.e. $\rho(B) < 1$, where $B = \mathcal{I} - D^{-1}A$). Note that each component $x_i^{(k+1)}$ of the new vector $\boldsymbol{x}^{(k+1)}$ is computed independently of the others, meaning that the update is simultaneous which makes this method suitable for parallel programming.

- ***Gauss-Seidel Method*: $P = D + L$**

The matrix P is chosen to be equal to the lower triangular part of A, therefore the iteration matrix is given by $B = (D + L)^{-1}(D + L - A)$ and the iteration itself is $\boldsymbol{x}^{(k+1)} = B\boldsymbol{x}^{(k)} + (D + L)^{-1}\boldsymbol{b}$ which can be written component-wise as

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{N} a_{ij} x_j^{(k)} \right) \quad \text{for all} \quad i = 1, \ldots, N. \tag{2.5}$$

Contrary to Jacobi method, Gauss-Seidel method updates the components in sequential mode.

There are many other methods that use splitting like:

- Richardson method: $P = \frac{1}{\omega}\mathcal{I}$ where $\mathcal{I}$ is the identity matrix and $\omega \neq 0$

[2] A matrix $A \in \mathbb{R}^{N \times N}$ is ***Diagonally Dominant*** if every diagonal entry is larger in absolute value than the sum of the absolute value of all the other terms in that row. More formally

$$|a_{ii}| \geq \sum_{\substack{j=1 \\ j \neq i}}^{N} |a_{ij}| \quad \text{for all} \quad i = 1, \ldots, N.$$

The matrix is ***Strictly Diagonally Dominant*** if the inequality is strict.

- Damped Jacobi method: $P = \frac{1}{\omega} D$ for some $\omega \neq 0$
- Successive over-relaxation method: $P = \frac{1}{\omega} D + L$ for some $\omega \neq 0$
- Symmetric successive over-relaxation method: $P = \frac{1}{\omega(2-\omega)}(D + \omega L)D^{-1}(D + \omega U)$ for some $\omega \neq 0, 2$.

2.3.2 Computational Cost & Stopping Criteria

There are essentially two factors contributing to the effectiveness of an iterative method for $A\boldsymbol{x} = \boldsymbol{b}$: the computational cost per iteration and the number of performed iterations. The computational cost per iteration depends on the structure and sparsity of the original matrix A and on the choice of the splitting. For both Jacobi and Gauss-Seidel methods, without further assumptions on A, the computational cost per iteration is $\mathcal{O}\left(N^2\right)$. Iterations should be stopped when one or more stopping criteria are satisfied, as will be discussed below. For both Jacobi and Gauss-Seidel methods, the cost of performing k iterations is $\mathcal{O}\left(kN^2\right)$; so as long as $k \ll N$, these methods are much cheaper than Gaussian elimination.

In theory, iterative methods require an infinite number of iterations to converge to the exact solution of a linear system but in practice, aiming at the exact solution is neither reasonable nor necessary. Indeed, what is actually needed is an approximation $\boldsymbol{x}^{(k)}$ for which the error is guaranteed to be lower than a desired tolerance $\tau > 0$. On the other hand, since the error is itself unknown (as it depends on the exact solution), a suitable *a posteriori* error estimator is needed which predicts the error starting from quantities that have already been computed. There are two natural estimators one may consider:

- The residual at the k^{th} iteration, i.e. $\boldsymbol{r}^{(k)} = \boldsymbol{b} - A\boldsymbol{x}^{(k)}$. More precisely, an iterative method can be stopped at the first iteration step $k = k_{\min}$ for which

  $$\|\boldsymbol{r}^{(k)}\| \leq \tau \|\boldsymbol{b}\|.$$

  When the above estimate is satisfied, it is guaranteed that

  $$\frac{\|\boldsymbol{e}^{(k)}\|}{\|\boldsymbol{x}\|} \leq \tau \kappa(A),$$

  i.e. the control on the residual is meaningful only for those matrices whose condition number is reasonably small; in this way the relative error will be of the same size as the relative residual.
- The increment at the $(k+1)^{\text{st}}$ iteration, i.e. $\boldsymbol{\delta}^{(k)} = \boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)}$. More precisely, iterative method would stop after the first iteration step $k = k_{\min}$ for which

  $$\|\boldsymbol{\delta}^{(k)}\| \leq \tau.$$

  If B is symmetric and positive definite, then

  $$\|\boldsymbol{e}^{(k+1)}\| = \|\boldsymbol{e}^{(k)} + \boldsymbol{\delta}^{(k)}\| \leq \rho(B)\|\boldsymbol{e}^{(k)}\| + \|\boldsymbol{\delta}^{(k)}\|.$$

Recalling that $\rho(B)$ should be less than 1 in order for the iterative method to converge, we deduce

$$\|\boldsymbol{e}^{(k)}\| \leq \frac{1}{1-\rho(B)}\|\boldsymbol{\delta}^{(k)}\|,$$

i.e. the control on the increment is meaningful only if $\rho(B) \ll 1$ since in that case the error will be of the same size as the increment.

2.4 In-Built MATLAB Procedures

Given that MATLAB is well-suited to dealing with matrices, it has a very powerful method of solving linear systems and it is using the ***Backslash Operator***. This is a powerful in-built method that can solve any square linear system regardless of its form. MATLAB does this by first determining the general form of the matrix (sparse, triangular, Hermitian, etc.) before applying the appropriate optimised method.

For the linear system

$$A\boldsymbol{x} = \boldsymbol{b} \quad \text{where} \quad A \in \mathbb{R}^{N\times N}, \quad \boldsymbol{x} \in \mathbb{R}^{N}, \quad \boldsymbol{b} \in \mathbb{R}^{N}$$

MATLAB can solve this using the syntax `x=A\b`.

Starting Example

Returning to the example in the beginning of this section, the matrix system was

$$\underbrace{\begin{pmatrix} 1 & 1 & 1 \\ 1 & -2 & 0 \\ 0 & 1 & -1 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} a \\ b \\ c \end{pmatrix}}_{\boldsymbol{x}} = \underbrace{\begin{pmatrix} 20 \\ 0 \\ 10 \end{pmatrix}}_{\boldsymbol{b}}.$$

This can be solved as follows:

```
>> A=[1,1,1;1,-2,0;0,1,-1];
>> b=[20;0;10];
>> A\b
ans =
     15.0000
      7.5000
     -2.5000
```

Note

The MATLAB website shows the following flowcharts for how `A\b` classifies the problem before solving it.

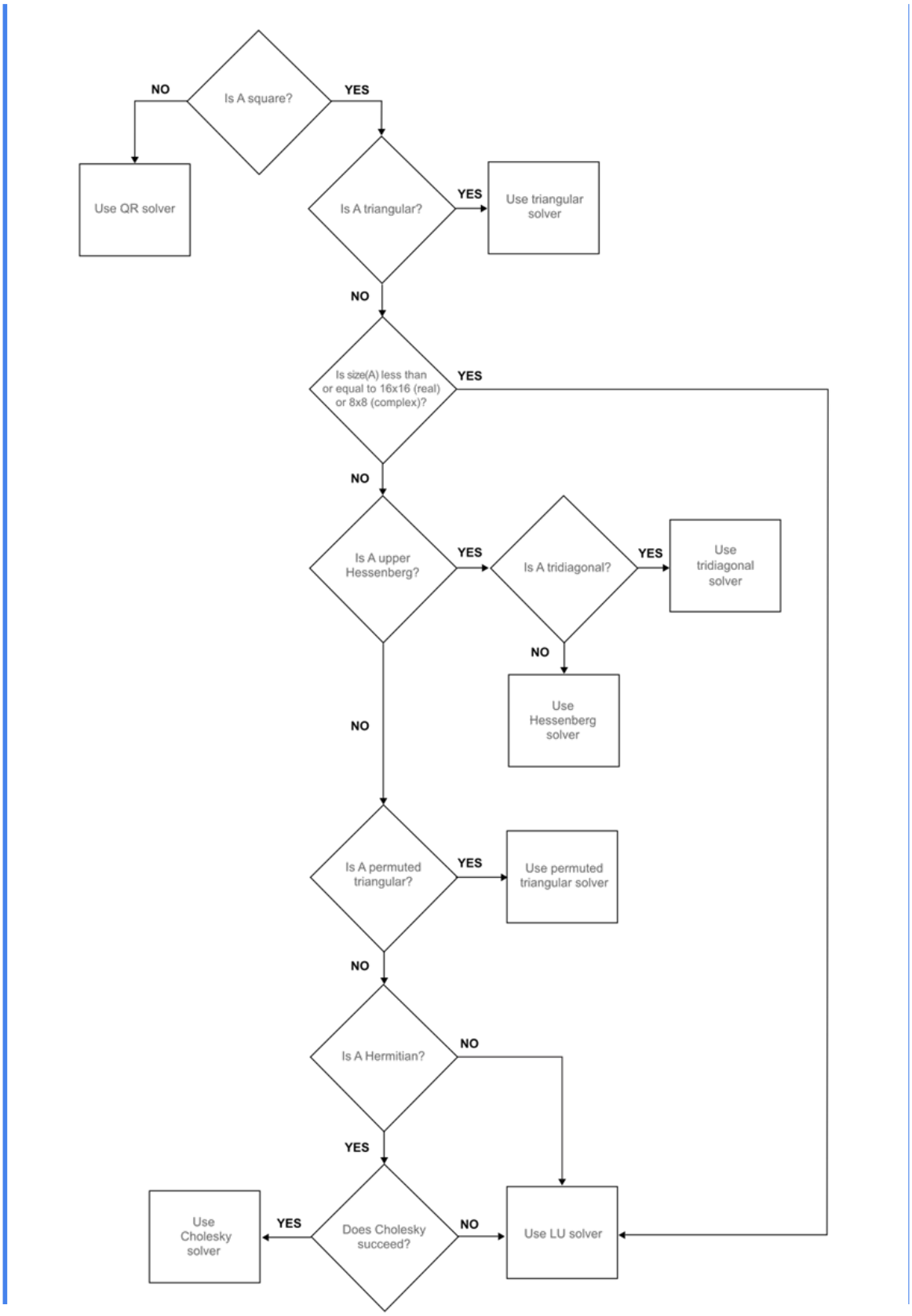


Figure 2.1: If the matrix A is full.

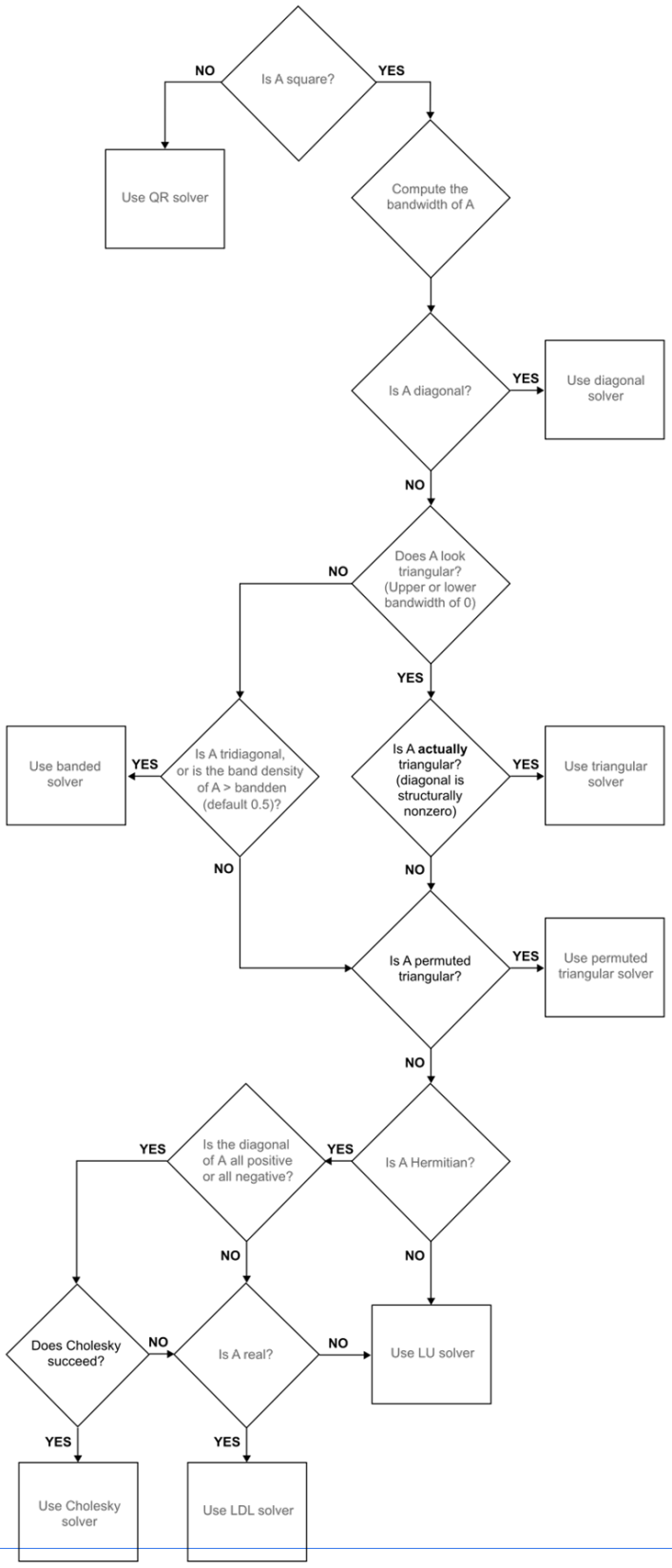


Figure 2.2: If the matrix A is sparse.

2.5 Exersises

Exersise 2.1

Let A and B be invertible matrices, $p \in \mathbb{N}$ and $\lambda \in \mathbb{R}$. The condition number κ_p has the following properties:

i. $\kappa_p(A) \geq 1$;
ii. $\kappa_p(A) = 1$ if and only if A is an orthogonal matrix, i.e. $A^{-1} = A^{\mathrm{T}}$;
iii. $\kappa_p(A^{\mathrm{T}}) = \kappa_p(A^{-1}) = \kappa_p(A)$;
iv. $\kappa_p(\lambda A) = \kappa_p(A)$;
v. $\kappa_p(AB) \leq \kappa_p(A)\kappa_p(B)$.

Solution 2.1

i.
ii.
iii.
iv.
v.

Exersise 2.2

Solve the following linear systems of the form $A\boldsymbol{x} = \boldsymbol{b}$ using the following direct methods:

i. Direct substitution

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 5 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 3 \\ 9 \\ 0 \end{pmatrix}$$

ii. Backward substitution

$$A = \begin{pmatrix} 7 & 2 & 1 \\ 0 & 2 & -1 \\ 0 & 0 & 3 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

iii. Direct substitution

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 1 & 1 & -1 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 3 \\ 0 \\ 5 \end{pmatrix}$$

iv. TDMA

$$A = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

v. Cramer's Rule

$$A = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 2 & 0 \\ 1 & -1 & 2 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Exersise 2.3

Using the formulas derived, write MATLAB codes that can perform:

- Direct substitution
- Backward substitution
- Forward substitution
- TDMA

Use the examples in Exersise 2.2 as test cases.

Part II

Solving Initial Value Problems

3 The Euler Method

Consider the first order ordinary differential equation (ODE)

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y), \quad t \in [t_0, t_f]$$

where f is a known function, t_0 is an initial time and t_f is the final time. An *initial condition* can be prescribed to this differential equation which will assign a "starting value" for the unknown function y at the starting time as $y(t_0) = y_0$. The combination of the first order ODE and the initial value gives the ***Initial Value Problem*** (or ***IVP***)

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y) \quad \text{with} \quad y(t_0) = y_0, \quad t \in [t_0, t_f].$$

There are many analytic methods for solving first order ordinary differential equations, but they all hold restrictions, like linearity or homogeneity. This chapter will develop the simplest numerical technique for solving any first order ordinary differential equation, this method is called the ***Euler Method***.

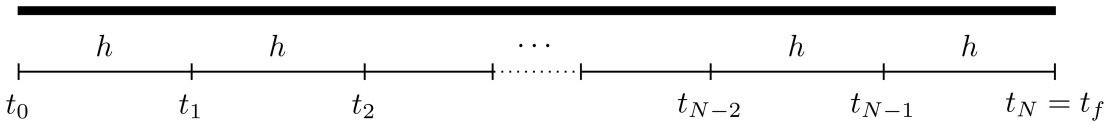
Consider the following first order IVP

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y), \quad \text{with} \quad y(t_0) = y_0 \quad t \in [t_0, t_f].$$

The function f is known and in most cases, is assumed to be "well-behaved" (does not have discontinuities or sharp corners). The term y_0 is known as the ***Initial Value*** of the function y at the starting time t_0. Solving this initial value problem is essentially finding an unknown curve $y(t)$ that starts at the point (t_0, y_0) and ends at time t_f.

The first step in the Euler method (as is the case in most numerical techniques) is to discretise the domain. This changes the domain from the continuous interval $[t_0, t_f]$ to N subintervals, each with constant[1] width h (sometimes also denoted δt), which is known as the ***Stepsize***. The discretised interval will be the set of points

$$\{t_0, t_0 + h, t_0 + 2h, \ldots, t_0 + Nh\}.$$



[1] In most cases, the interval width h is constant but more advanced numerical techniques have different subinterval widths.

The aim of the numerical procedure is to start from the starting point (t_0, y_0) and progressively find consequent points until the final time t_f is reached.

The Euler method uses the gradient, namely $\frac{\mathrm{d}y}{\mathrm{d}t}$, at the starting point (t_0, y_0) in order to find the value of y at the subsequent point which will be labelled (t_1, y_1). This will, in turn, determine the new gradient at (t_1, y_1) and this process is then continued until the final time is reached. The smaller the value of h is, the more points there will be between t_0 and t_f resulting in a more accurate final solution to the initial value problem.

The accuracy of the Euler method is usually characterised by how small h is or how large N is. Since the stepsize may not always give an appropriate subdivision (like dividing the interval $[0, 1]$ into subintervals of width 0.4), then the number of subdivisions N can be used to find an appropriate h by using

$$h = \frac{t_f - t_0}{N}.$$

3.1 Steps of the Euler Method

Consider the IVP

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y), \quad \text{with} \quad y(t_0) = y_0 \quad t \in [t_0, t_f].$$

Parallel Example

The steps of the Euler method will be explained theoretically and applied to this IVP in parallel to demonstrate the steps:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = 6 - 2y \quad \text{with} \quad y(0) = 0, \quad t \in [0, 2].$$

In this case, the function on the RHS is $f(t, y) = 6 - 2y$. Note that this IVP has the exact solution

$$y(t) = 3 - 3\mathrm{e}^{-2t}.$$

1. Discretise the interval $[t_0, t_f]$ with stepsize h to form the set of points

$$\{t_0, t_0 + h, t_0 + 2h, \ldots, t_0 + Nh\}.$$

Inverval Discretisation

Suppose that the interval $[0, 2]$ is to be split into 5 subintervals, then $N = 5$ and

$$h = \frac{t_f - t_0}{N} = \frac{2 - 0}{5} = 0.4.$$

Therefore the discretised points are

$$\{0.0, 0.4, 0.8, 1.2, 1.6, 2.0\}.$$

Note that N denotes the number of subintervals and **not** the number of points, that would be $N+1$ points since the starting point is 0.

2. At the starting point (t_0, y_0), the gradient is known since
$$y'(t_0) = f(t_0, y_0).$$

Gradient at (t_0, y_0)

At the initial point,
$$y'(t_0) = f(t_0, y_0) \quad \Longrightarrow \quad y'(0) = f(0,0) = 6 - 2(0) = 6.$$
So the starting gradient is 6.

3. The next step is to find the the value of y at the subsequent time $t_1 = t_0 + h$. For this purpose, consider the Taylor series expansion of y at $t = t_1$,
$$y(t_1) = y(t_0 + h) = y(t_0) + hy'(t_0) + \frac{h^2}{2!}y''(t_0) + \mathcal{O}\left(h^3\right).$$

Note

The term $\mathcal{O}\left(h^3\right)$ simply means that the terms after this point have a common factor of h^3 and these terms are regarded as *higher order terms* and can be neglected since they are far smaller than the first terms provided h is small.

Since h is assumed to be sufficiently small, then all terms higher order terms, in this case h^2 or higher, can be neglected (i.e. $h^n \approx 0$ for $n \geq 2$). Therefore
$$y(t_1) \approx y(t_0) + hy'(t_0).$$

Let Y_1 denote the *approximated value* of the solution at the point t_1, i.e. $Y_1 \approx y(t_1)$, so in this case,
$$Y_1 = y_0 + hy'(t_0). \tag{3.1}$$
This determines the value of Y_1 which is an approximation to $y(t_1)$.

Calcuating Y_1

The point Y_1 can be calculated as follows:
$$Y_1 = y_0 + hy'(t_0) = 0 + (0.4)(6) = 2.4.$$
This means that the next point is $(t_1, Y_1) = (0.4, 2.4)$.

4. This iteration can be continued to find Y_{n+1} (which is the approximate value of $y(t_{n+1})$) for all $n = 1, 2, \ldots, N-1$
$$Y_{n+1} = Y_n + hy'(t_n) \quad \text{where} \quad y'(t_n) = f(t_n, Y_n).$$

Calculating $\boldsymbol{Y_n}$

The values of Y_2, Y_3, Y_4 and Y_5 can be calculated as follows:

$$Y_2: \quad y'(t_1) = f(t_1, Y_1) \quad \Longrightarrow \quad y'(0.4) = f(0.4, 2.4) = 6 - 2(2.4) = 1.2$$
$$\Longrightarrow \quad Y_2 = Y_1 + hy'(t_1) = 2.4 + (0.4)(1.2) = 2.88$$

$$Y_3: \quad y'(t_2) = f(t_2, Y_2) \quad \Longrightarrow \quad y'(0.8) = f(0.8, 2.88) = 6 - 2(2.88) = 0.24$$
$$\Longrightarrow \quad Y_3 = Y_2 + hy'(t_2) = 2.88 + (0.4)(0.24) = 2.976$$

$$Y_4: \quad y'(t_3) = f(t_3, Y_3) \quad \Longrightarrow \quad y'(1.2) = f(1.2, 2.976) = 6 - 2(2.976) = 0.048$$
$$\Longrightarrow \quad Y_4 = Y_3 + hy'(t_3) = 2.976 + (0.4)(0.048) = 2.9952$$

$$Y_5: \quad y'(t_4) = f(t_4, Y_4) \quad \Longrightarrow \quad y'(1.6) = f(1.6, 2.9952) = 6 - 2(2.9952) = 0.0096$$
$$\Longrightarrow \quad Y_5 = Y_4 + hy'(t_4) = 2.9952 + (0.4)(0.0096) = 2.99904$$

5. The solution to the IVP can now be approximated by the function that passes through the points
$$(t_0, Y_0), \quad (t_1, Y_1), \quad \ldots \quad (t_N, Y_N).$$

Solution to the IVP

The approximate solution to the IVP

$$\frac{\mathrm{d}y}{\mathrm{d}t} = 6 - 2y \quad \text{with} \quad y(0) = 0, \quad t \in [0, 2]$$

is the function that passes through the points:

$$(0, 0), \quad (0.4, 2.4), \quad (0.8, 2.88), \quad (1.2, 2.976), \quad (1.6, 2.9952), \quad (2, 2.99904).$$

This is a good approximation since the *exact* locations, as per the exact solution are, (to 4 decimal places):

$$(0, 0), \quad (0.4, 1.6520), \quad (0.8, 2.3943), \quad (1.2, 2.7278), \quad (1.6, 2.8777), \quad (2, 2.9451)$$

which is not bad for such a coarse interval breakdown.

The Euler method needs N steps to complete and every step $n \in \{1, 2, \ldots, N\}$ requires finding $y'(t_{n-1}) = f(t_{n-1}, y_{n-1})$ and $Y_n = Y_{n-1} + hy'(t_{n-1})$. Of course, the larger N is, the smaller h becomes, meaning that more steps will be required but the solution will be closer

to the exact solution

Notice that the terms on the right hand side of Equation 3.1 are all known and for this reason, the Euler method is known as an ***Explicit Method***.

3.2 Accuracy

Consider the Taylor series expansion for the function y at the point $t_1 = t_0 + h$,

$$y(t_1) = y(t_0 + h) = y(t_0) + hy'(t_0) + \frac{h^2}{2!}y''(t_0) + \mathcal{O}\left(h^3\right).$$

Using ***Taylor's Theorem***[2], this can be written as

$$y(t_1) = y(t_0 + h) = y(t_0) + hy'(t_0) + \frac{h^2}{2!}y''(\tau_1)$$

for some point τ_1 between t_0 and t_1. The Euler method determines the approximation Y_1 to the function y at the point t_1, particularly,

$$Y_1 = y(t_0) + hy'(t_0) \approx y(t_1).$$

The ***Local Truncation Error*** at the first step, denoted e_1, is defined as the absolute difference between the exact and approximated values at the first step, and this is given by

$$e_1 = |y(t_1) - Y_1| = \frac{h^2}{2!}\left|y''(\tau_1)\right|.$$

This can be done for all the locations to give a list of local truncation errors $e_1, e_2, e_3, \ldots, e_N$. Note that *technically*, these errors are hypothetical since the *exact solution* y, and thus $y(t_n)$, are not known but these are put as placeholders to establish the full accuracy of the method. In this case, the local truncation error e is said to be of *second order* since $e = \mathcal{O}\left(h^2\right)$.

As the iteration progresses, the errors will accumulate to result in a ***Global Integration Error*** denoted E. In this case, the global integration error is

$$E = |y(t_f) - Y_N|.$$

[2] ***Taylor's Theorem*** states that for a function f that is at least $N + 1$ times differentiable in the open interval (x, x_0) (or (x_0, x)), then

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2!}f''(x_0)(x - x_0)^2 + \frac{1}{3!}f'''(x_0)(x - x_0)$$
$$+ \cdots + \frac{1}{N!}f^{(N)}(x_0)(x - x_0)^N + \frac{1}{(N+1)!}f^{(N+1)}(\xi)(x - x_0)^{N+1}$$

for some point ξ between x and x_0.

The global integration error has to be at most the accumulation of all the local truncation errors, namely

$$E = |y(t_f) - Y_N| \leq \underbrace{\sum_{n=1}^{N} e_n}_{\substack{\text{sum of all} \\ \text{local truncation} \\ \text{errors}}} = \sum_{n=1}^{N} \frac{h^2}{2!} |y''(\tau_n)| = h^2 \sum_{n=1}^{N} \frac{1}{2} |y''(\tau_n)|.$$

$$\implies \quad E \leq h^2 \sum_{n=1}^{N} \frac{1}{2} |y''(\tau_n)| \tag{3.2}$$

A bound for the sum needs to be found in order bound the global integration error. To this end, consider the set of the second derivatives in the sum above, i.e.

$$\left\{ \frac{1}{2} |y''(\tau_1)|, \frac{1}{2} |y''(\tau_2)|, \ldots, \frac{1}{2} |y''(\tau_n)| \right\}.$$

Since all these terms take a finite value, then at least one of these terms must be larger than all the rest, this is denoted M and can be written as

$$M = \max \left\{ \frac{1}{2} |y''(\tau_1)|, \frac{1}{2} |y''(\tau_2)|, \ldots, \frac{1}{2} |y''(\tau_n)| \right\}.$$

This can also be expressed differently as

$$M = \max_{\tau \in [t_0, t_f]} \left\{ \frac{1}{2} |y''(\tau)| \right\}.$$

Therefore, since

$$\frac{1}{2} |y''(\tau_n)| \leq M \quad \text{for all} \quad n = 1, 2, \ldots, N$$

then

$$\sum_{n=1}^{N} \frac{1}{2} |y''(\tau_n)| \leq \sum_{n=1}^{N} M = NM.$$

Thus, returning back to the expression for E in Equation 3.2

$$E \leq h^2 \sum_{n=1}^{N} \frac{1}{2} |y''(\tau_n)| \leq NMh^2 = Mh \cdot (Nh) = Mh(t_f - t_0) = \mathcal{O}(h).$$

Hence, the global integration error $E = \mathcal{O}(h)$, this means that the Euler method is a ***First Order Method***. This means that both h and the global integration error behave linearly to one another, so if h is halved, then the global integration error is halved as well.

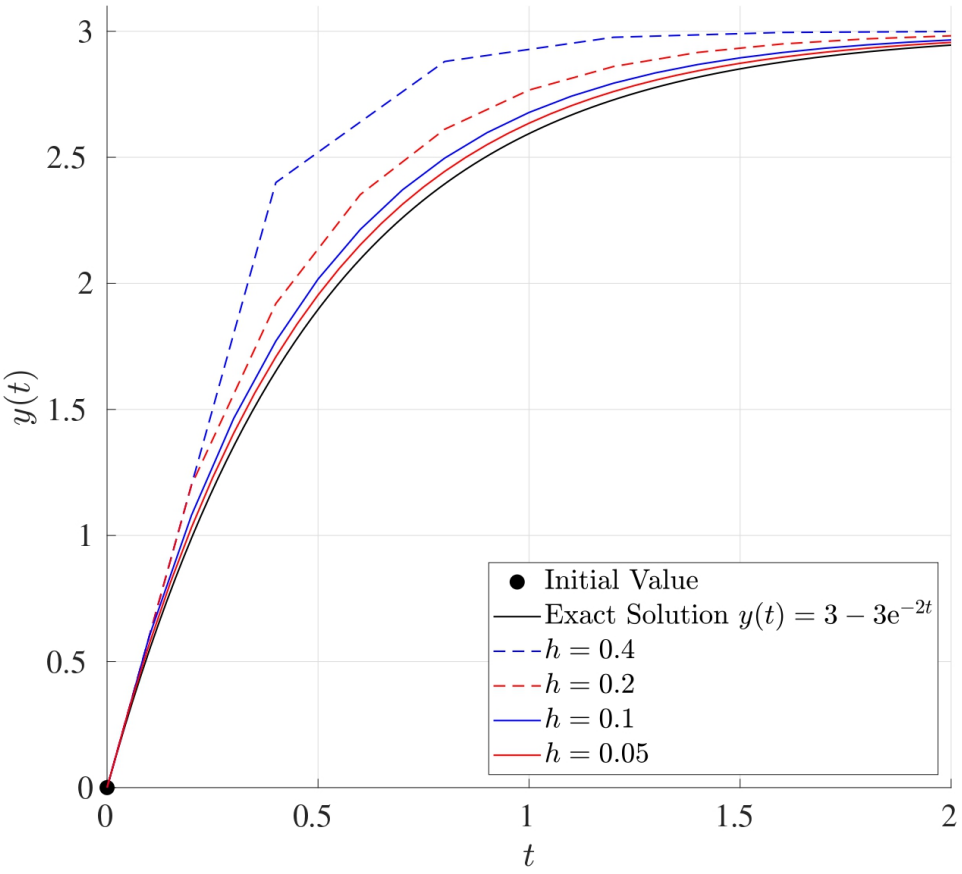
In conclusion, the local truncation error of the Euler method is $e = \mathcal{O}(h^2)$ while the global integration error $E = \mathcal{O}(h)$ when h is small.

Different Stepsizes

Returning to the IVP

$$\frac{\mathrm{d}y}{\mathrm{d}t} = 6 - 2y \quad \text{with} \quad y(0) = 0, \quad t \in [0, 2].$$

The Euler method can be repeated for different values of h and these can be seen in the figure below.



The table below shows the global integration error for the different values of h:

h	E
0.4	0.05399
0.2	0.03681
0.1	0.02036
0.05	0.01060

When the value of h is halved, the global integration error is approximately halved as well.

3.3 Set of IVPs

SO far, the Euler Method has been used to solve a single IVP, however this can be extended to solving a set of linear IVPs.

Consider the set of K linear IVPs defined on the interval $[t_0, t_f]$:

$$\begin{aligned}
\frac{\mathrm{d}y_1}{\mathrm{d}t} &= a_{11}y_1 + a_{12}y_2 + \cdots + a_{1K}y_K + b_1, & y_1(t_0) &= \tilde{y}_1 \\
\frac{\mathrm{d}y_2}{\mathrm{d}t} &= a_{21}y_1 + a_{22}y_2 + \cdots + a_{2K}y_K + b_2, & y_2(t_0) &= \tilde{y}_2 \\
&\vdots \\
\frac{\mathrm{d}y_K}{\mathrm{d}t} &= a_{K1}y_1 + a_{K2}y_2 + \cdots + a_{KK}y_K + b_K, & y_K(t_0) &= \tilde{y}_K
\end{aligned}$$

where, for $i, j = 1, 2, \ldots, K$, the functions $y_i = y_i(t)$ are unknown, a_{ij} are known constant coefficients and b_i are all known (these can generally depend on t).

This set of initial value problems need to be written in matrix form as

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = A\boldsymbol{y} + \boldsymbol{b} \quad \text{with} \quad \boldsymbol{y}(t_0) = \boldsymbol{y}_0, \quad t \in [t_0, t_f]$$

$$\text{where} \quad \boldsymbol{y}(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_K(t) \end{pmatrix}, \quad A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1K} \\ a_{21} & a_{22} & \ldots & a_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ a_{K1} & a_{K2} & \ldots & a_{KK} \end{pmatrix},$$

$$\boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_K \end{pmatrix}, \quad \boldsymbol{y}_0 = \begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \\ \vdots \\ \tilde{y}_K \end{pmatrix}.$$

In this case, $\boldsymbol{y}(t)$ is the unknown solution vector, A is a matrix of constants, $\boldsymbol{y}_0$ is the vector of initial values and $\boldsymbol{b}$ is a vector of known terms (possibly depending on t) and is referred to as the ***Inhomogeneity*** or ***Forcing Term***.

The Euler iteration would be performed in a similar way as before. First, the interval $[t_0, t_f]$ needs to be discretised into N equally spaced subintervals, each of width h to give the set of discrete times $(t_0, t_1, \ldots, t_N)$ where $t_n = t_0 + nh$ for $n = 0, 1, \ldots, N$. Let $\boldsymbol{Y}_n$ be the approximation to the function vector $\boldsymbol{y}$ at the time $t = t_n$, then

$$\boldsymbol{Y}_{n+1} = \boldsymbol{Y}_n + h\boldsymbol{y}'(t_n) \quad \text{where} \quad \boldsymbol{y}'(t_n) = A\boldsymbol{Y}_n + \boldsymbol{b}_n \quad \text{for} \quad n = 0, 1, 2, \ldots, N-1$$

subject to the initial values $\boldsymbol{Y}_0 = \boldsymbol{y}_0$. (Note that if the vector $\boldsymbol{b}$ depends on t, then $\boldsymbol{b}_n = \boldsymbol{b}(t_n)$.)

Sets of IVPs

Consider the two coupled IVPs on the interval $[0, 1]$:

$$\begin{aligned} \frac{\mathrm{d}y}{\mathrm{d}t} &= y + 2z, \quad y(0) = 1 \\ \frac{\mathrm{d}z}{\mathrm{d}t} &= \frac{3}{2}y - z, \quad z(0) = 0 \end{aligned}$$

Before attempting to solve this set of IVPs, it needs to be written in matrix form as

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = A\boldsymbol{y} + \boldsymbol{b} \quad \text{with} \quad \boldsymbol{y}(0) = \boldsymbol{y}_0.$$

In this case,

$$\boldsymbol{y}(t) = \begin{pmatrix} y(t) \\ z(t) \end{pmatrix}, \quad A = \begin{pmatrix} 1 & 2 \\ \frac{3}{2} & -1 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \boldsymbol{y}_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

Let $N = 5$, so

$$h = \frac{t_f - t_0}{N} = \frac{1 - 0}{5} = 0.2.$$

The Euler iteration will be

$$\boldsymbol{Y}_{n+1} = \boldsymbol{Y}_n + h\boldsymbol{y}'(t_n) \quad \text{where} \quad \boldsymbol{y}'(t_n) = A\boldsymbol{Y}_n + \boldsymbol{b}_n \quad \text{for} \quad n = 0, 1, 2, 3, 4.$$

This can be written as

$$\boldsymbol{Y}_{n+1} = \boldsymbol{Y}_n + h\left[A\boldsymbol{Y}_n + \boldsymbol{b}_n\right] \quad \text{for} \quad n = 0, 1, 2, 3, 4$$

keeping in mind that $t_n = hn = 0.2n$ the vector $\boldsymbol{b}_n = \boldsymbol{b}(t_n) = \mathbf{0}$ and $\boldsymbol{Y}_0 = \boldsymbol{y}_0$:

$$\boldsymbol{Y}_1 = \boldsymbol{Y}_0 + 0.2\,[A\boldsymbol{Y}_0 + \boldsymbol{b}_0] = \begin{pmatrix}1\\0\end{pmatrix} + 0.2\left[\begin{pmatrix}1 & 2\\ \frac{3}{2} & -1\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} + \begin{pmatrix}0\\0\end{pmatrix}\right] = \begin{pmatrix}1.2\\0.3\end{pmatrix}$$

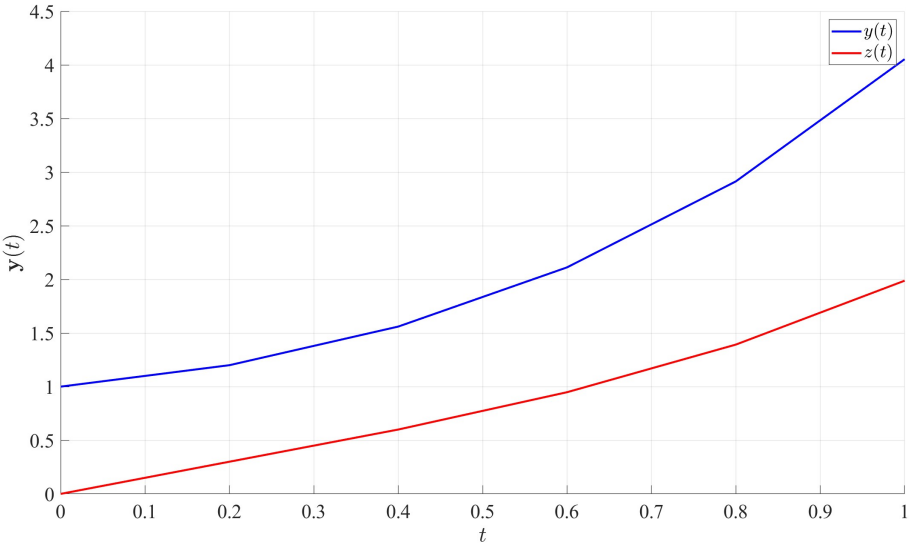
$$\boldsymbol{Y}_2 = \boldsymbol{Y}_1 + 0.2\,[A\boldsymbol{Y}_1 + \boldsymbol{b}_1] = \begin{pmatrix}1.2\\0.3\end{pmatrix} + 0.2\left[\begin{pmatrix}1 & 2\\ \frac{3}{2} & -1\end{pmatrix}\begin{pmatrix}1.2\\0.3\end{pmatrix} + \begin{pmatrix}0\\0\end{pmatrix}\right] = \begin{pmatrix}1.56\\0.6\end{pmatrix}$$

$$\boldsymbol{Y}_3 = \boldsymbol{Y}_2 + 0.2\,[A\boldsymbol{Y}_2 + \boldsymbol{b}_2] = \begin{pmatrix}1.56\\0.6\end{pmatrix} + 0.2\left[\begin{pmatrix}1 & 2\\ \frac{3}{2} & -1\end{pmatrix}\begin{pmatrix}1.56\\0.6\end{pmatrix} + \begin{pmatrix}0\\0\end{pmatrix}\right] = \begin{pmatrix}2.112\\0.948\end{pmatrix}$$

$$\boldsymbol{Y}_4 = \boldsymbol{Y}_3 + 0.2\,[A\boldsymbol{Y}_3 + \boldsymbol{b}_3] = \begin{pmatrix}2.112\\0.948\end{pmatrix} + 0.2\left[\begin{pmatrix}1 & 2\\ \frac{3}{2} & -1\end{pmatrix}\begin{pmatrix}2.112\\0.948\end{pmatrix} + \begin{pmatrix}0\\0\end{pmatrix}\right] = \begin{pmatrix}2.9136\\1.3920\end{pmatrix}$$

$$\boldsymbol{Y}_5 = \boldsymbol{Y}_4 + 0.2\,[A\boldsymbol{Y}_4 + \boldsymbol{b}_4] = \begin{pmatrix}2.9136\\1.3920\end{pmatrix} + 0.2\left[\begin{pmatrix}1 & 2\\ \frac{3}{2} & -1\end{pmatrix}\begin{pmatrix}2.9136\\1.3920\end{pmatrix} + \begin{pmatrix}0\\0\end{pmatrix}\right] = \begin{pmatrix}4.0531\\1.9877\end{pmatrix}$$

therefore $\quad y(1) = 4.0531, \quad z(1) = 1.9877.$



3.4 Higher Order IVPs

The previous sections solved one first order IVP and a set of first order IVPs. What happens if a higher order IVP is to be solved? Or a set of higher order IVPs? The difference will be minimal, subject to a few manipulations first.

Consider the K^{th} order linear IVP on the interval $[t_0, t_f]$

$$\frac{\mathrm{d}^K y}{\mathrm{d}t^K} + a_{K-1}\frac{\mathrm{d}^{K-1} y}{\mathrm{d}t^{K-1}} + \cdots + a_2\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} + a_1\frac{\mathrm{d}y}{\mathrm{d}t} + a_0 y = f(t) \tag{3.3}$$

where $a_k \in \mathbb{R}$ and f is a known function. This IVP is to be solved subject to the initial conditions

$$y(t_0) = \eta_0, \quad \frac{\mathrm{d}y}{\mathrm{d}t}(t_0) = \eta_1 \quad \dots \quad \frac{\mathrm{d}^{K-1}y}{\mathrm{d}t^{K-1}}(t_0) = \eta_{K-1}.$$

This K^{th} order IVP can be written as a set of K first order IVPs. Indeed, let the functions y_k be given by

$$\begin{aligned}
y_1(t) &= \frac{\mathrm{d}y}{\mathrm{d}t} \\
y_2(t) = y_1'(t) &= \frac{\mathrm{d}^2y}{\mathrm{d}t^2} \\
y_3(t) = y_2'(t) &= \frac{\mathrm{d}^3y}{\mathrm{d}t^3} \\
&\vdots \\
y_{K-3}(t) = y_{K-4}'(t) &= \frac{\mathrm{d}^{K-3}y}{\mathrm{d}t^{K-3}} \\
y_{K-2}(t) = y_{K-3}'(t) &= \frac{\mathrm{d}^{K-2}y}{\mathrm{d}t^{K-2}} \\
y_{K-1}(t) = y_{K-2}'(t) &= \frac{\mathrm{d}^{K-1}y}{\mathrm{d}t^{K-1}}
\end{aligned}$$

Notice that

$$\begin{aligned}
\frac{\mathrm{d}y_{K-1}}{\mathrm{d}t} = \frac{\mathrm{d}^K y}{\mathrm{d}t^K} &= -a_{K-1}\frac{\mathrm{d}^{K-1}y}{\mathrm{d}t^{K-1}} - \cdots - a_2\frac{\mathrm{d}^2y}{\mathrm{d}t^2} - a_1\frac{\mathrm{d}y}{\mathrm{d}t} - a_0 y + f(t) \\
&= -a_{K-1}y_{K-1} - \cdots - a_2y_2 - a_1y_1 - a_0y + f(t)
\end{aligned}$$

Let $\boldsymbol{y}$ be the vector of the unknown functions $y, y_1, y_2, \dots, y_{K-1}$. This means that the IVP in Equation 3.3 can be written in matrix form $\boldsymbol{y}' = A\boldsymbol{y} + \boldsymbol{b}$ as follows:

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} y \\ y_1 \\ y_2 \\ \vdots \\ y_{K-3} \\ y_{K-2} \\ y_{K-1} \end{pmatrix} = \begin{pmatrix} y' \\ y_1' \\ y_2' \\ \vdots \\ y_{K-3}' \\ y_{K-2}' \\ y_{K-1}' \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{K-2} \\ y_{K-1} \\ \frac{\mathrm{d}^K y}{\mathrm{d}t^K} \end{pmatrix}$$

$$= \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{K-2} \\ y_{K-1} \\ -a_{K-1}y_{K-1} - \cdots - a_2y_2 - a_1y_1 - a_0y + f(t) \end{pmatrix}$$

$$= \underbrace{\begin{pmatrix} 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 1 \\ -a_0 & -a_1 & -a_2 & \dots & -a_{K-3} & -a_{K-2} & -a_{K-1} \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} y \\ y_1 \\ y_2 \\ \vdots \\ y_{K-3} \\ y_{K-2} \\ y_{K-1} \end{pmatrix}}_{\boldsymbol{y}} + \underbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ f(t) \end{pmatrix}}_{\boldsymbol{b}} = A\boldsymbol{y} + \boldsymbol{b}.$$

The initial condition vector will be

$$\boldsymbol{y}_0 = \begin{pmatrix} y(0) \\ y_1(0) \\ y_2(0) \\ \vdots \\ y_{K-3}(0) \\ y_{K-2}(0) \\ y_{K-1}(0) \end{pmatrix} = \begin{pmatrix} y(0) \\ \frac{\mathrm{d}y}{\mathrm{d}t}(0) \\ \frac{\mathrm{d}^2y}{\mathrm{d}t^2}(0) \\ \vdots \\ \frac{\mathrm{d}^{K-3}y}{\mathrm{d}t^{K-3}}(0) \\ \frac{\mathrm{d}^{K-2}y}{\mathrm{d}t^{K-2}}(0) \\ \frac{\mathrm{d}^{K-1}y}{\mathrm{d}t^{K-1}}(0) \end{pmatrix} = \begin{pmatrix} \eta_0 \\ \eta_1 \\ \eta_2 \\ \vdots \\ \eta_{K-3} \\ \eta_{K-2} \\ \eta_{K-1} \end{pmatrix}.$$

The matrix A is called the ***Companion Matrix*** and is a matrix with 1 on the super diagonal and the last row is the minus of the coefficients in the higher order IVP, and zeros otherwise. Now that the K^{th} order IVP has been converted into a set of K linear IVPs, it can be solved just as in Section 3.3. Note that *any* linear K^{th} order IVP can always be converted into a set of K first order IVPs but the converse is not always possible.

Higher Order IVPs

Consider the following higher order IVP

$$\frac{\mathrm{d}^4y}{\mathrm{d}t^4} - 8\frac{\mathrm{d}^3y}{\mathrm{d}t^3} + 7\frac{\mathrm{d}^2y}{\mathrm{d}t^2} - \frac{\mathrm{d}y}{\mathrm{d}t} + 2y = \cos(t) \quad \text{for} \quad t \in \mathbb{R}_{\geq 0}$$

$$\text{with} \quad y(0) = 4, \quad \frac{\mathrm{d}y}{\mathrm{d}t}(0) = 1, \quad \frac{\mathrm{d}^2y}{\mathrm{d}t^2}(0) = 3, \quad \frac{\mathrm{d}^3y}{\mathrm{d}t^3}(0) = 0.$$

Let $u = \frac{\mathrm{d}y}{\mathrm{d}t}, v = u' = \frac{\mathrm{d}^2y}{\mathrm{d}t^2}$ and $w = v' = \frac{\mathrm{d}^3y}{\mathrm{d}t^3}$. The derivatives of u, v and w are:

$$u' = v$$
$$v' = w$$
$$w' = \frac{\mathrm{d}^4y}{\mathrm{d}t^4} = 8\frac{\mathrm{d}^3y}{\mathrm{d}t^3} - 7\frac{\mathrm{d}^2y}{\mathrm{d}t^2} + \frac{\mathrm{d}y}{\mathrm{d}t} - 2y + \cos(t) = 8w - 7v + u + 2y + \cos(t)$$

Define the vector $\boldsymbol{y} = (y, u, v, w)^{\mathrm{T}}$

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} y \\ u \\ v \\ w \end{pmatrix} = \begin{pmatrix} u \\ v \\ w \\ \cos(t) + 8w - 7v + u - 2y \end{pmatrix}$$

$$= \underbrace{\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -2 & 1 & -7 & 8 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} y \\ u \\ v \\ w \end{pmatrix}}_{\boldsymbol{y}} + \underbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \\ \cos(t) \end{pmatrix}}_{\boldsymbol{b}(t)} = A\boldsymbol{y} + \boldsymbol{b}(t).$$

The initial condition vector will be

$$\boldsymbol{y}_0 = \begin{pmatrix} y(0) \\ u(0) \\ v(0) \\ w(0) \end{pmatrix} = \begin{pmatrix} y(0) \\ \frac{\mathrm{d}y}{\mathrm{d}t}(0) \\ \frac{\mathrm{d}^2y}{\mathrm{d}t^2}(0) \\ \frac{\mathrm{d}^3y}{\mathrm{d}t^3}(0) \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ 3 \\ 0 \end{pmatrix}.$$

Now the IVP can be solved using the Euler method as before but only the first function is the most relevant, all others have been used as placeholders.

3.4.1 Sets of Higher Order IVPs

The method above can be extended into a set of higher order IVPs.

Set of Higher Order IVPs

Consider the following coupled system of higher order IVPs

$$y'' + 6y' + y = \sin(t), \quad z''' - 8z'' = 5y - 2y' + \mathrm{e}^{2t}$$

$$\text{with} \quad y(0) = 1, \quad \frac{\mathrm{d}y}{\mathrm{d}t}(0) = 2, \quad z(0) = 4, \quad \frac{\mathrm{d}z}{\mathrm{d}t}(0) = 1, \quad \frac{\mathrm{d}^2z}{\mathrm{d}t^2}(0) = 2$$

In the case of a coupled system, the vector function $\boldsymbol{y}$ should consist of all the unknown functions and their derivatives up to but not including their highest order derivative. In other words,

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} y \\ y' \\ z \\ z' \\ z'' \end{pmatrix} = \begin{pmatrix} y' \\ y'' \\ z' \\ z'' \\ z''' \end{pmatrix} = \begin{pmatrix} y' \\ -y - 6y' + \sin(t) \\ z' \\ z'' \\ 5y - 2y' + 8z'' + \mathrm{e}^{2t} \end{pmatrix}$$

$$= \underbrace{\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ -1 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 5 & -2 & 0 & 0 & 8 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} y \\ y' \\ z \\ z' \\ z'' \end{pmatrix}}_{\boldsymbol{y}} + \underbrace{\begin{pmatrix} 0 \\ \sin(t) \\ 0 \\ 0 \\ \mathrm{e}^{2t} \end{pmatrix}}_{\boldsymbol{b}}.$$

The vector of initial values would be

$$\boldsymbol{y}(0) = \begin{pmatrix} y(0) \\ y'(0) \\ z(0) \\ z'(0) \\ z''(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 4 \\ 1 \\ 2 \end{pmatrix}.$$

Now this can be solved just as before with the most relevant terms being the first and third (since those are y and z).

3.4.2 Stability of a Set of ODEs

Consider the set of K homogeneous ODEs

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = A\boldsymbol{y}.$$

Let $\lambda_1, \lambda_2, \ldots, \lambda_K$ be the eigenvalues of the matrix A and $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_K$ be their *distinct* corresponding eigenvectors (distinct for the sake argument). Analytically, the set of differential equations $\boldsymbol{y}' = A\boldsymbol{y}$ has the general solution

$$\boldsymbol{y}(t) = C_1\boldsymbol{v}_1\mathrm{e}^{\lambda_1 t} + C_2\boldsymbol{v}_2\mathrm{e}^{\lambda_2 t} + \cdots + C_K\boldsymbol{v}_K\mathrm{e}^{\lambda_K t}$$

where $C_1, C_2, \ldots, C_n$ are constants that can be determined from the initial values.

Definition 3.1. The initial value problem

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = A\boldsymbol{y} + \boldsymbol{b} \quad \text{with} \quad \boldsymbol{y}(0) = \boldsymbol{y}_0$$

is said to be ***Asymptotically Stable*** if $\boldsymbol{y} \to \boldsymbol{0}$ as $t \to \infty$, in other words, all functions in $\boldsymbol{y}$ tend to 0 as t tends to infinity.

This definition will be important when looking at the long term behaviour of solutions from the eigenvalues to then determine stepsize bounds.

Theorem 3.1. *The initial value problem*

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = A\boldsymbol{y} + \boldsymbol{b}$$

is asymptotically stable if ***all*** *the eigenvalues of the matrix A have negative real parts. If A has at least one eigenvalue with a non-negative real part, then the system is not asymptomatically stable.*

Notice that the stability of a set of ODEs does *not* depend on the forcing term $\boldsymbol{b}$ nor does it depend on the initial condition $\boldsymbol{y}(0)$.

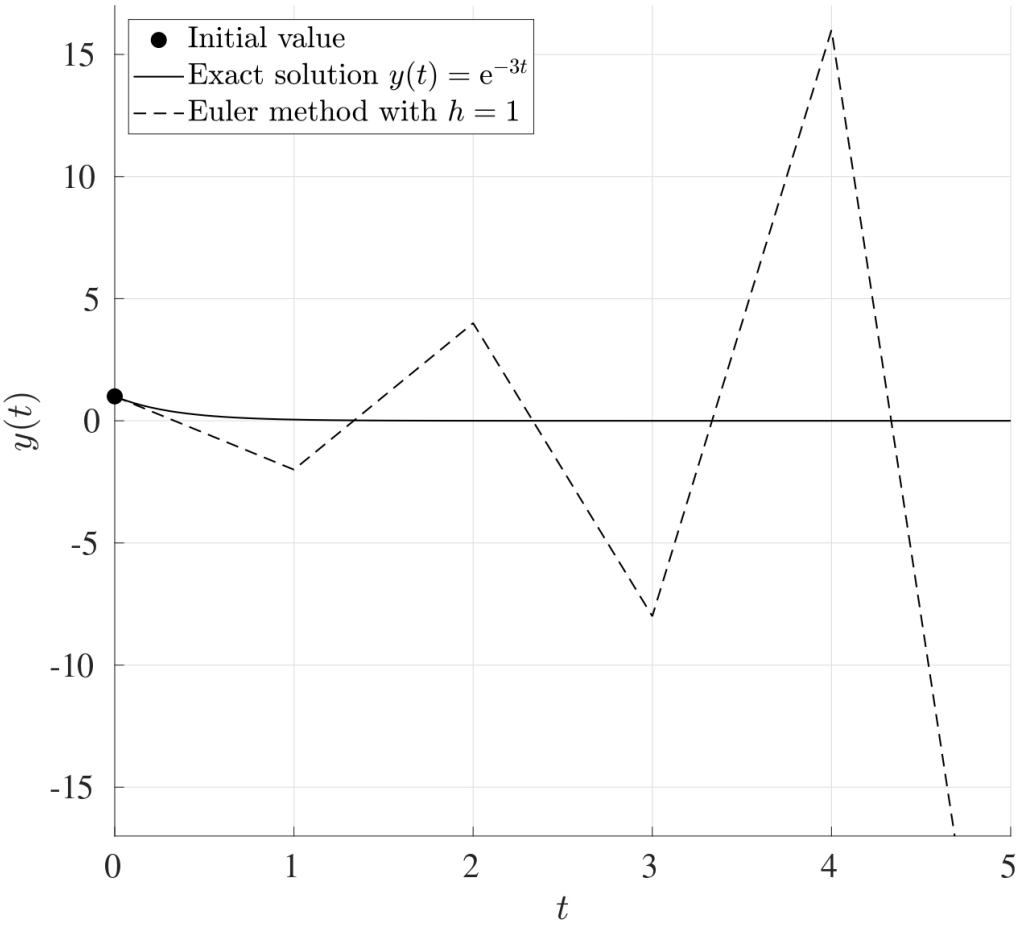
3.5 Limitations of the Euler Method

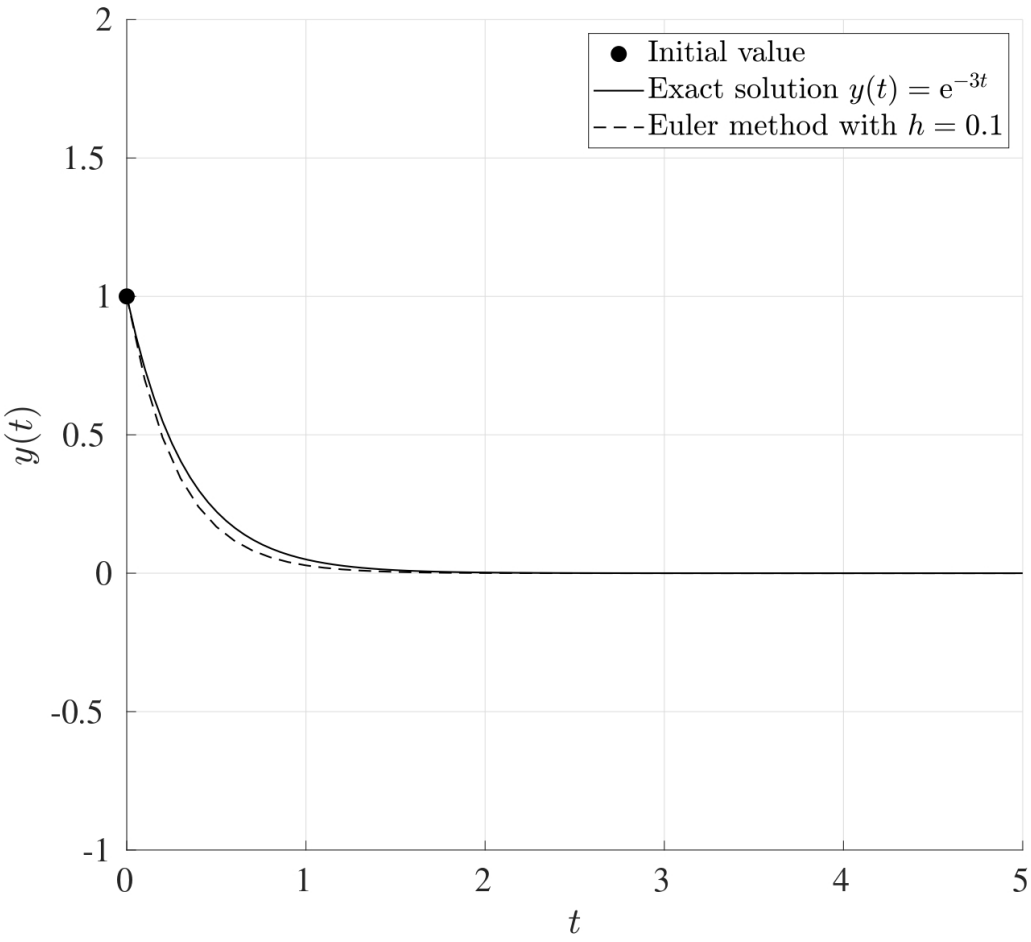
In some cases, if the stepsize h is taken to be too large, then the Euler method can give misleading results.

For example, consider the initial value problem:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -3y \quad \text{with} \quad y(0) = 1, \quad t \in [0, 5].$$

Choosing a large stepsize h can render the method ineffective. Case in point, when $h = 1$, the approximate solution oscillates and grows quite rapidly, however choosing a smaller value of h, say $h = 0.1$, gives a very good approximation to the exact solution. These are illustrated in the figures below.





Another situation when the Euler method fails is when the IVP does not have a unique solution. For example, consider the IVP:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = y^{\frac{1}{3}} \quad \text{with} \quad y(0) = 0, \quad t \in [0, 2].$$

This has the exact solution $y(t) = \left(\frac{2}{3}t\right)^{\frac{3}{2}}$ however this is not unique since $y(t) = 0$ is also a perfectly valid solution. The Euler method in this case will not be able to capture the first non-trivial solution but will only capture the second trivial solution giving a straight line at 0[3].

[3] In general, according to the ***Picard-Lindelöf Theorem***, an IVP of the form $y' = f(t, y)$ with $y(0) = y_0$ has a *unique solution* if the function f is continuous in t and uniformly Lipschitz continuous in y. In this example shown above, the function $f(t, y) = y^{\frac{1}{3}}$ does not satisfy the aforementioned conditions and therefore the initial value problem does not have a unique solution. These concepts of continuity are far beyond the realms of this course and no further mention of them will be made.

3.5.1 Bounds on the Stepsize

Consider the initial value problem

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = A\boldsymbol{y} + \boldsymbol{b} \quad \text{with} \quad \boldsymbol{y}(0) = \boldsymbol{y}_0.$$

If A is asymptotically stable, then a maximum bound h_0 for the stepsize can be found to ensure that the iterations converge. (This means that asymptotic stability of A is a necessary and sufficient condition for the existence of an upper bound h_0 such that if $h < h_0$, then the Euler iteration converges.)

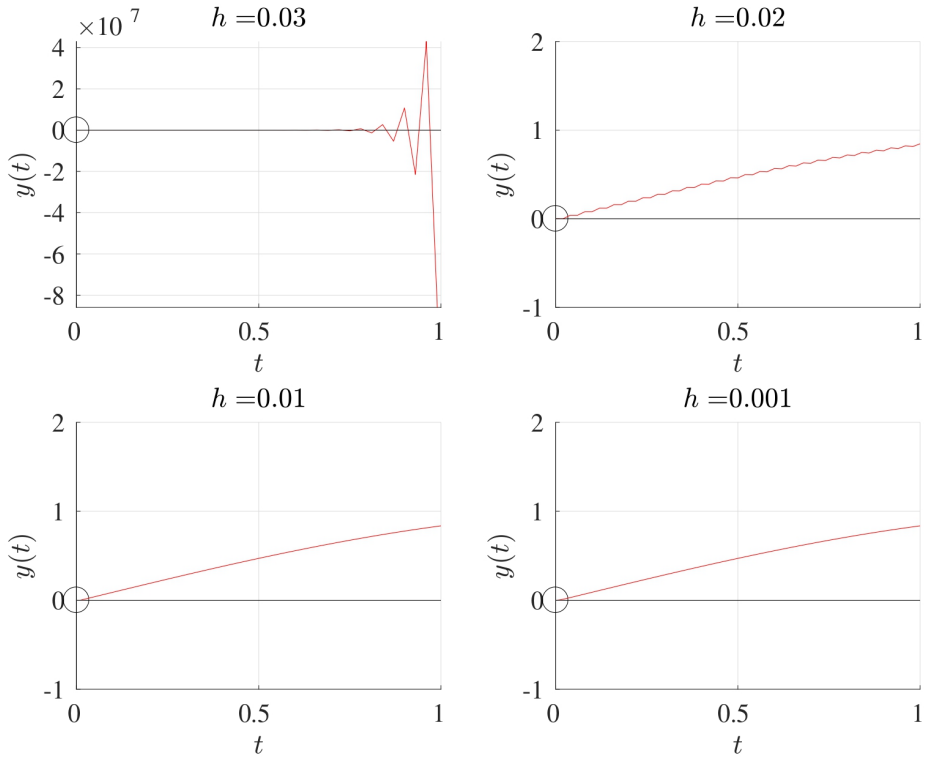
If the stepsize is too large, then the method may not converge but on the other hand if it is too low, then the iteration will take a considerable amount of time to perform. Therefore an "optimal" stepsize is needed to obtain sufficiently accurate solutions.

Different Stepsizes

Consider the following initial value problem

$$\frac{\mathrm{d}y}{\mathrm{d}t} = 100(\sin(t) - y) \quad \text{with} \quad y(0) = 0.$$

The figure below shows the Euler method being used to solve the initial value problem in the interval $[0, 1]$ for the stepsizes $h = 0.03, 0.02, 0.01, 0.001$.



When $h = 0.03$, the Euler method does not converge. At $h = 0.02$, the Euler method

converges but there clearly is a distinct artefact in the solution that shows a slight oscillation. For h less than 0.02, this oscillation is no longer observed and the Euler method is convergent.

3.5.2 Exact Bound

Consider the IVP

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = A\boldsymbol{y} + \boldsymbol{b} \quad \text{with} \quad \boldsymbol{y}(0) = \boldsymbol{y}_0.$$

Let $\lambda_1, \lambda_2, \ldots, \lambda_K$ be the eigenvalues of A. Suppose that the matrix A is asymptotically stable (i.e. $\Re(\lambda_k) < 0$ for all $k = 1, 2, \ldots, K$). In order for the Euler iterations to converge, the stepsize h needs be less than the threshold stepsize h_0 where

$$h_0 = 2 \min_{k=1,2,\ldots,K} \left\{ \frac{|\Re(\lambda_k)|}{|\lambda_k|^2} \right\} \tag{3.4}$$

$$\text{or} \quad h_0 = 2 \min_{k=1,2,\ldots,K} \left\{ \frac{1}{|\lambda_k|} \right\} \quad \text{if all the eigenvalues are real.}$$

In other words, if the initial value problem is asymptotically stable, then the Euler method is stable if an only if $h < h_0$. This means that the convergence of the Euler is characterised by the eigenvalue that is furthest away from the origin, also called the ***Dominant Eigenvalue***.

!!!!!DO!!!!!

Euler Upper Bound

Consider the system of differential equations $\boldsymbol{y}' = A\boldsymbol{y}$ with $\boldsymbol{y}(0) = \boldsymbol{y}_0$ where

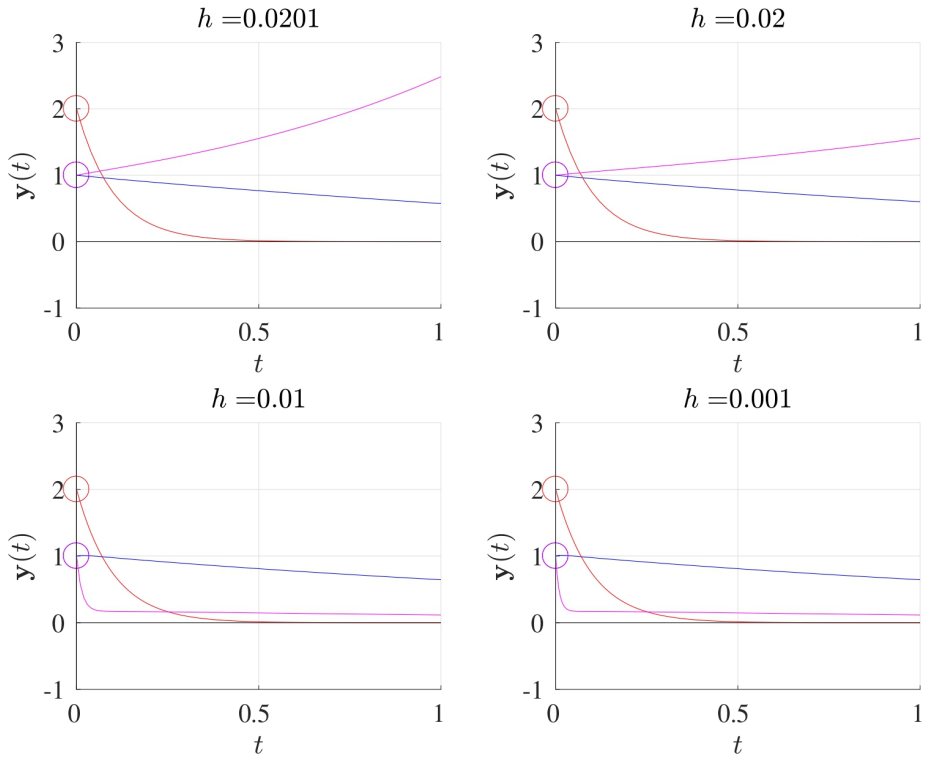
$$A = \begin{pmatrix} -1 & 0 & 3 \\ 0 & -10 & 0 \\ 18 & -1 & -100 \end{pmatrix}.$$

The eigenvalues of the matrix A are $-0.4575, -100.5425, -10$. Since all the eigenvalues are negative, this system is asymptotically stable. Since all the eigenvalues are real, then the threshold stepsize for a convergent Euler method is

$$h_0 = 2 \min \left\{ \frac{1}{|\lambda_k|} \right\} = 2 \min \left\{ \frac{1}{|-0.4575|}, \frac{1}{|-100.5425|}, \frac{1}{|-10|} \right\}$$

$$= 2 \min \{2.0858, 0.0099, 0.1\} = 2 \times 0.0099 = 0.0199.$$

Solutions for different stepsizes are as shown below with the initial values $y_1(0) = 1$ (blue), $y_2(0) = 2$ (red) and $y_3(0) = 1$ (magenta). It can be seen that if $h \geq h_0$, then at least one solution will diverge but if $h < h_0$, then all solutions converge to 0.



3.5.3 Estimated Bound

One drawback in attempting to determine the value of h_0 using Equation 3.4 is that *all* the eigenvalues of the matrix A have to be determined before h_0 can be found. This can be computationally expensive for especially for very large matrices.

An estimate for the threshold stepsize h_0 can be found with far fewer computations using the *sup-norm* $\|\cdot\|_\infty$ (also known as the *infinity norm* or the *Chebyshev norm*). Recall that for a vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, the **sup-norm** of $\boldsymbol{x}$ is the maximum absolute value in the vector, i.e.

$$\|\boldsymbol{x}\|_\infty = \max |x_n|.$$

Whereas for a matrix A, the **sup-norm** of A is the maximal absolute row sum. In other words, for a given matrix A, take the absolute value of all the terms, take the sum of each row and the sup-norm will be the largest out of these.

Sup-Norm of Vectors & Matrices

Consider the vector $\boldsymbol{x}$ and matrix M given by

$$\boldsymbol{x} = \begin{pmatrix} 1 \\ -4 \\ -9 \\ 7 \end{pmatrix}, \quad M = \begin{pmatrix} 5 & 2 & 4 & 1 \\ -9 & 5 & 3 & -7 \\ 6 & 0 & -1 & 4 \\ 9 & 5 & -2 & 4 \end{pmatrix}.$$

The sup-norm of $\boldsymbol{x}$ is simply the largest absolute element which is 9, therefore $\|\boldsymbol{x}\|_\infty = 9$. As for M, to find the sup-norm, first take the absolute value of all the terms, then add the rows. The sup-norm is the maximum element that results:

$$\begin{pmatrix} 5 & 2 & 4 & 1 \\ -9 & 5 & 3 & -7 \\ 6 & 0 & -1 & 4 \\ 9 & 5 & -2 & 4 \end{pmatrix} \xrightarrow[|\bullet|]{} \left.\begin{pmatrix} 5 & 2 & 4 & 1 \\ 9 & 5 & 3 & 7 \\ 6 & 0 & 1 & 4 \\ 9 & 5 & 2 & 4 \end{pmatrix} \begin{matrix} \to 12 \\ \to 24 \\ \to 11 \\ \to 20 \end{matrix}\right\} \text{maximum is 24.}$$

Therefore $\|M\|_\infty = 24$.
Both of these can be found in MATLAB using `norm(x,Inf)` and `norm(M,Inf)`.

Theorem 3.2. *Consider the set of linear IVPs*

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = A\boldsymbol{y} + \boldsymbol{b} \quad \textit{with} \quad \boldsymbol{y}(0) = \boldsymbol{y}_0$$

where A is asymptotically stable. Then the Euler method is numerically convergent for any choice of h which satisfies

$$\|\mathcal{I} + hA\|_\infty \leq 1.$$

Computing all the eigenvalues of the matrix A can be computationally expensive but obtaining the sup-norm is takes far fewer computations, however as a drawback, the resulting value of h_0 would be an estimate.

Stepsize Bound Estimate 1 (Tridiagonal)

Consider the differential equation $\boldsymbol{y}' = A\boldsymbol{y}$ where

$$A = \begin{pmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{pmatrix}.$$

To find the upper bound for the stepsize for which the Euler method converges, first

evaluate $\mathcal{I} + hA$:

$$\mathcal{I} + hA = \begin{pmatrix} 1-2h & h & 0 & 0 & 0 \\ h & 1-2h & h & 0 & 0 \\ 0 & h & 1-2h & h & 0 \\ 0 & 0 & h & 1-2h & h \\ 0 & 0 & 0 & h & 1-2h \end{pmatrix}$$

To find the sup-norm, take the absolute value of all the terms and find the maximal row sum:

$$\xrightarrow[|\bullet|]{} \begin{pmatrix} |1-2h| & h & 0 & 0 & 0 \\ h & |1-2h| & h & 0 & 0 \\ 0 & h & |1-2h| & h & 0 \\ 0 & 0 & h & |1-2h| & h \\ 0 & 0 & 0 & h & |1-2h| \end{pmatrix} \begin{matrix} \to |1-2h| + h \\ \to |1-2h| + 2h \\ \to |1-2h| + 2h \\ \to |1-2h| + 2h \\ \to |1-2h| + h. \end{matrix}$$

Let $a = |1-2h| + 2h$ and $b = |1-2h| + h$. Since $h > 0$, then $a > b$, therefore

$$\|\mathcal{I} + hA\|_\infty = |1-2h| + 2h.$$

In order to satisfy the inequality $\|\mathcal{I} + hA\|_\infty \leq 1$, consider the cases when $1-2h \geq 0$ and $1-2h < 0$ separately:

1. If $1-2h \geq 0$, then $h \leq \frac{1}{2}$:

   $$\|\mathcal{I} + hA\|_\infty = |1-2h| + 2h = 1 - 2h + 2h = 1.$$

   Therefore $\|\mathcal{I} + hA\|_\infty = 1 \leq 1$ is indeed true.

2. If $1-2h < 0$, then $h > \frac{1}{2}$:

   $$\|\mathcal{I} + hA\|_\infty = |1-2h| + 2h = 2h - 1 + 2h = 4h - 1.$$

   If $\|\mathcal{I} + hA\|_\infty \leq 1$, then $4h - 1 \leq 1$. Simplifying this would result in $h \leq \frac{1}{2}$ which contradicts with the assumption that $h > \frac{1}{2}$.

From these two cases, it is clear that $h \not> \frac{1}{2}$ (since that case leads to a contradiction), therefore $h \leq \frac{1}{2}$. Thus for a convergent Euler method, the stepsize h must be less than the threshold stepsize $h_0 = \frac{1}{2}$.
This can be compared to the exact bound; the eigenvalues of the matrix A are

$$-3.7321, \quad -3, \quad -2, \quad -1, \quad -0.2679.$$

Therefore

$$h_0 = 2 \min \left\{ \frac{1}{|\lambda_k|} \right\} = 0.5359$$

which is a larger bound compared to the one obtained using the sup-norm method. Observe that if the size of the matrix was larger but followed the same theme (i.e. 2 on the main diagonal and -1 and the sub and super diagonals), then no further calculations are required for the sup-norm method, the outcome will still be $h_0 = \frac{1}{2}$. As for the eigenvalue method, all the eigenvalues have to be recalculated again.

Stepsize Bound Estimate 2 (Bidiagonal)

Consider the differential equation $\boldsymbol{y}' = A\boldsymbol{y}$ where

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

To find the upper bound for the stepsize for which the Euler method converges, first evaluate $\mathcal{I} + hA$:

$$\mathcal{I} + hA = \begin{pmatrix} 1-h & 0 & 0 & 0 & 0 \\ h & 1-h & 0 & 0 & 0 \\ 0 & h & 1-h & 0 & 0 \\ 0 & 0 & h & 1-h & 0 \\ 0 & 0 & 0 & h & 1-h \end{pmatrix}$$

To find the sup-norm, take the absolute value of all the terms and find the maximal row sum:

$$\xrightarrow[|\bullet|]{} \begin{pmatrix} |1-h| & 0 & 0 & 0 & 0 \\ h & |1-h| & 0 & 0 & 0 \\ 0 & h & |1-h| & 0 & 0 \\ 0 & 0 & h & |1-h| & 0 \\ 0 & 0 & 0 & h & |1-h| \end{pmatrix} \begin{matrix} \to \quad |1-h| \\ \to |1-h| + h \\ \to |1-h| + h \\ \to |1-h| + h \\ \to |1-h| + h. \end{matrix}$$

Let $a = |1-h| + h$ and $b = |1-h|$. Clearly $a > b$ since $h > 0$, therefore

$$\|\mathcal{I} + hA\|_\infty = |1-h| + h.$$

In order to satisfy the inequality, $\|\mathcal{I} + hA\|_\infty \leq 1$, consider the cases when $1-h \geq 0$ and $1-h < 0$:

1. If $1-h \geq 0$, then $h \leq 1$:

   $$\|\mathcal{I} + hA\|_\infty = |1-h| + h = 1 - h + h = 1,$$

   therefore $\|\mathcal{I} + hA\|_\infty \leq 1$ is indeed true.

2. If $1-h < 0$, then $h > 1$:

   $$\|\mathcal{I} + hA\|_\infty = |1-h| + h = h - 1 + h = 2h - 1.$$

   If $\|\mathcal{I} + hA\|_\infty \leq 1$, then $2h - 1 \leq 1$, meaning that $h \leq 1$ which contradicts with the assumption that $h > 1$.

This means that for a convergent Euler method, the stepsize h must be less than $h_0 = 1$. This can be compared to the exact upper bound. The eigenvalues of the matrix A are just -1 five times, therefore

$$h_0 = 2 \min \left\{ \frac{1}{|\lambda_k|} \right\} = 2,$$

this shows that the sup-norm method gives a tighter than using eigenvalues.

The sup-norm method works well when the matrix in question has a diagonal, bidiagonal or tridiagonal structure where the diagonal terms are the same. In general, the sup-norm method might *not* be suitable for any matrix.

Stepsize Bound Estimate 3 (General)

Consider the differential equation $\boldsymbol{y}' = A\boldsymbol{y}$ where

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

Find the sup-norm:

$$\mathcal{I} + hA = \begin{pmatrix} 1-h & -2h \\ 4h & 1-3h \end{pmatrix} \xrightarrow[|\bullet|]{} \begin{pmatrix} |1-h| & 2h \\ 4h & |1-3h| \end{pmatrix} \begin{matrix} \to |1-h| + 2h \\ \to |1-3h| + 4h \end{matrix}$$

Let $a = |1-h| + 2h$ and $b = |1-3h| + 4h$. Here, it is not obvious which is larger, a or b. Therefore, consider the three cases $0 < h < \frac{1}{3}$, $\frac{1}{3} < h < 1$ and $h > 1$.

1. $0 < h < \frac{1}{3}$: In this case, $1-h > 0$ and $1-3h > 0$, therefore $a = |1-h|+2h = 1+h$ and $b = |1-3h| + 4h = 1+h$, hence $\|\mathcal{I} + hA\|_\infty = 1+h$. In order to satisfy $\|\mathcal{I} + hA\|_\infty \leq 1$, this would mean that $h < 0$ which contradicts with the fact that $h > 0$. Therefore $h \notin \left(0, \frac{1}{3}\right)$.

2. $\frac{1}{3} < h < 1$: In this case, $1-h > 0$ and $1-3h < 0$, therefore $a = |1-h|+2h = 1+h$ and $b = |1-3h| + 4h = 7h-1$. This should now be split into two subcases to check which one will lead to a contradiction:

   i. Suppose that $a > b$, then

   $$1 + h > 7h - 1 \quad \Longrightarrow \quad h < \frac{1}{3}$$

   which contradicts with $h > \frac{1}{3}$

   ii. Suppose that $a < b$, then

   $$1 + h < 7h - 1 \quad \Longrightarrow \quad h > \frac{1}{3}$$

   not leading to any contradiction. therefore since $b > a$, then $\|\mathcal{I} + hA\|_\infty = b = 7h - 1$.

In order to satisfy $\|\mathcal{I} + hA\|_\infty \leq 1$ then $h < \frac{2}{7}$ which contradicts with the fact that $\frac{1}{3} < h$. Therefore $h \notin \left(\frac{1}{3}, 1\right)$.

3. $h > 1$: In this case, $1-h < 0$ and $1-3h < 0$, therefore $a = |1-h| + 2h = 3h-1$ and $b = |1-3h|+4h = 7h-1$. Clearly $b > a$ since $h > 0$, so $\|\mathcal{I} + hA\|_\infty = 7h-1$. In order to satisfy $\|\mathcal{I} + hA\|_\infty \leq 1$ then $h < \frac{2}{7}$ which contradicts with the fact that $h > 1$. This means that $h \not> 1$.

So in every possible case, there will be a contradiction when using the sup-norm method. This *does not* mean that the system is asymptotically unstable, in fact, the eigenvalues of the matrix A are $-2 \pm 2.65\mathrm{i}$ meaning that the system is asymptotically stable and the threshold stepsize is in fact $h_0 = 0.0992$.
This example shows that the sup-norm method cannot be used for any matrix system, but if a matrix has a banded structure, then it would be appropriate and would require fewer computations compared to finding all the eigenvalues.

3.6 MATLAB Code

The following MATLAB code performs the Euler iteration for the following set of IVPs on the interval $[0, 1]$:

$$\begin{aligned} \frac{\mathrm{d}u}{\mathrm{d}t} &= 2u + v + w + \cos(t), & u(0) &= 0 \\ \frac{\mathrm{d}v}{\mathrm{d}t} &= \sin(u) + \mathrm{e}^{-v+w}, & v(0) &= 1 \\ \frac{\mathrm{d}w}{\mathrm{d}t} &= uv - w, & w(0) &= 0. \end{aligned}$$

Linearity

Note that this code is built for a general case that *does not* have to be linear even though the entire derivation process was built on the fact that the system is linear.

```
function IVP_Euler

%% Solve a set of first order IVPs using Euler

% This code solves a set of IVP when written explicitly
% on the interval [t0,tf] subject to the initial conditions
% y(0)=y0.  The output will be the graph of the solution(s)
% and the vector value at the final point tf.  Note that the
% IVPs do not need to be linear or homogeneous.

%% Lines to change:

% Line 28   : t0 - Start time
% Line 31   : tf - End time
% Line 34   : N  - Number of subdivisions
% Line 37   : y0 - Vector of initial values
% Line 105+ : Which functions to plot, remembering to assign
%             a colour, texture and legend label
```

```
% Line 125+ : Set of differential equations written
%              explicitly.  These can also be non-linear and
%              include forcing terms.  These equations can
%              also be written in matrix form if the
%              equations are linear.

%% Set up input values

% Start time
t0=0;

% End time
tf=1;

% Number of subdivisions
N=50;

% Column vector initial values y0=y(t0)
y0=[0;1;0];

%% Set up IVP solver parameters

% T = Vector of times t0,t1,...,tN.
% This is generated using linspace which splits the
% interval [t0,tf] into N+1 points (or N subintervals)
T=linspace(t0,tf,N+1);

% Stepsize
h=(tf-t0)/N;

% Number of differential equations
K=length(y0);

%% Perform the Euler iteration

% Y = Solution matrix
% The matrix Y will contain K rows and N+1 columns.  Every
% row corresponds to a different IVP and every column
% corresponds to a different time.  So the matrix Y will
% take the following form:
% y_1(t_0)  y_1(t_1)  y_1(t_2)  ...  y_1(t_N)
% y_2(t_0)  y_2(t_1)  y_2(t_2)  ...  y_2(t_N)
% ...
% y_K(t_0)  y_K(t_1)  y_K(t_2)  ...  y_K(t_N)
Y=zeros(K,N+1);
```

```
64
65  % The first column of the vector Y is the initial vector y0
66  Y(:,1)=y0;
67
68  % Set the current time t to be the starting time t0 and the
69  % current value of the vector y to be the strtaing values y0
70  t=t0;
71  y=y0;
72
73  for n=2:1:N+1
74
75      dydt=DYDT(t,y,K);  % Find gradient at the current step
76
77      y=y+h*dydt;        % Find y at the current step
78
79      t=T(n);            % Update the new time
80
81      Y(:,n)=y;          % Replace row n in Y with y
82
83  end
84
85  %% Setting plot parameters
86
87  % Clear figure
88  clf
89
90  % Hold so more than one line can be drawn
91  hold on
92
93  % Turn on grid
94  grid on
95
96  % Setting font size and style
97  set(gca,'FontSize',20,'FontName','Times')
98
99  % Label the axes
100 xlabel('$t$','Interpreter','Latex')
101 ylabel('$\mathbf{y}(t)$','Interpreter','Latex')
102
103 % Plot the desried solutions.  If all the solutions are
104 % needed, then consider using a for loop in that case
105 plot(T,Y(1,:),'-b','LineWidth',2)
106 plot(T,Y(2,:),'-r','LineWidth',2)
107 plot(T,Y(3,:),'-k','LineWidth',2)
108
```

```
% Legend labels
legend('$y_1(t)$','$y_2(t)$','$y_3(t)$')
set(legend,'Interpreter','Latex')

% Display the values of the vector y at tf
disp(strcat('The vector y at t=',num2str(tf),' is:'))
disp(Y(:,end))

end

function [dydt]=DYDT(t,y,K)

% When the equation are written in explicit form

dydt=zeros(K,1);

dydt(1)=2*y(1)+y(2)+y(3)+cos(t);

dydt(2)=sin(y(1))+exp(-y(2)+y(3));

dydt(3)=y(1)*y(2)-y(3);

% If the set of equations is linear, then these can be
% written in matrix form as dydt=A*y+b(t).  For example, if
% the set of equations is:
% dudt = 7u - 2v +  w + exp(t)
% dvdt = 2u + 3v - 9w + cos(t)
% dwdt =      2v + 5w + 2
% Then:
% A=[7,-2,1;2,3,-9;0,2,5];
% b=@(t) [exp(t);cos(t);2];
% dydt=A*y+b(t)

end
```

4 The Modified Euler Method

The Euler method can be effective when it comes to solving differential equations numerically but on occasions, the global error of $\mathcal{O}(h)$ is rather poor. The Euler method can modified and improved to give **Modified** or **Improved Euler Method** (also known as the *Heun Method*, named after Karl Heun).

4.1 Steps of the Modified Euler Method

The Modified Euler Method utilises the **Fundamental Theorem of Calculus** which states that for a differentiable function y defined on the interval $[t_0, t_1]$ (where $t_1 = t_0 + h$ for some stepsize h),

$$y(t_1) - y(t_0) = \int_{t_0}^{t_1} y'(t)\ \mathrm{d}t.$$

In the interval $[t_0, t_1]$, the derivative $y'(t)$ may be approximated by the derivative at the leftmost point $y'(t_0)$, this approximation forms the basis of the standard Euler method;

$$\begin{aligned} y(t_1) - y(t_0) &= \int_{t_0}^{t_1} y'(t)\ \mathrm{d}t \\ &= \int_{t_0}^{t_1} y'(t_0)\ \mathrm{d}t \\ &= hy'(t_0) \\ \implies \quad y(t_1) &= y(t_0) + hy'(t_0). \end{aligned}$$

However, if $y'(t)$ varies substantially then this approximation can lead to some poor predictions. This can be modified so rather than approximating $y'(t)$ by $y'(t_0)$ only, it can be approximated by taking an average between $y'(t_0)$ and $y'(y_1)$, namely

$$y'(t) \approx \frac{1}{2}\left(y'(t_0) + y'(t_1)\right).$$

Thus

$$\begin{aligned} y(t_1) - y(t_0) &= \int_{t_0}^{t_1} y'(t)\ \mathrm{d}t \\ &= \int_{t_0}^{t_1} \frac{1}{2}\left(y'(t_0) + y'(t_1)\right)\ \mathrm{d}t \\ &= \frac{h}{2}\left(y'(t_0) + y'(t_1)\right) \end{aligned}$$

$$\implies \quad y(t_1) = y(t_0) + \frac{h}{2}\left(y'(t_0) + y'(t_1)\right).$$

Initially, one might suspect that the derivative $y'(t_1)$ can be found from the differential equation itself, namely, $y'(t_1) = f(t_1, y(t_1))$ but to do that, a **Prediction-Correction** procedure needs to be employed where the Euler method can be used to predict a value of $y(t_1)$ and this is then corrected afterwards. This is done as follows:

- Predictor: $\tilde{Y}_{n+1} = Y_n + hf(t_n, Y_n)$
- Corrector: $Y_{n+1} = Y_n + \frac{h}{2}\left[f(t_n, Y_n) + f(t_{n+1}, \tilde{Y}_{n+1})\right].$

Modified Euler Method

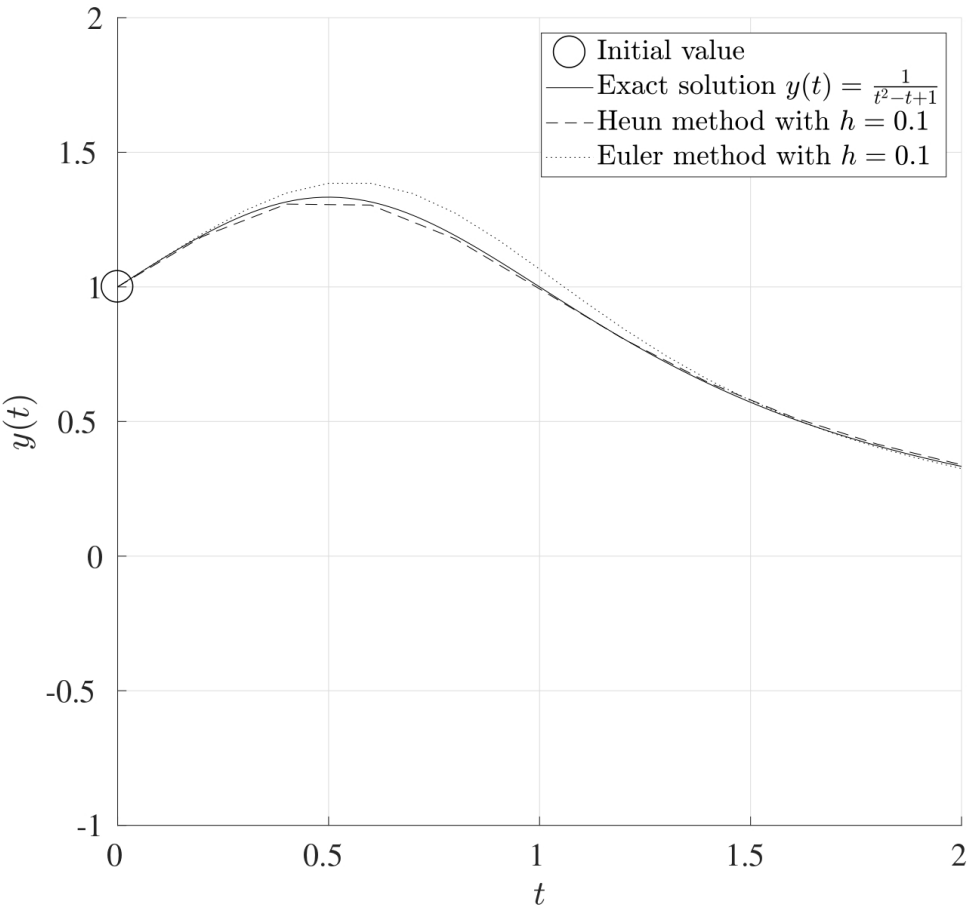
Consider the differential equation

$$\frac{\mathrm{d}y}{\mathrm{d}t} = (1-2t)y^2 \quad \text{with} \quad y(0) = 1, \quad t \in [0, 2].$$

This differential equation is non-linear but has a known particular solution which is

$$y(t) = \frac{1}{t^2 - t + 1}$$

and this will be compared to the approximate solutions obtained from the standard and Modified Euler methods.
The figure below shows how the standard and modified Euler methods compare to the exact solution for the same stepsize $h = 0.1$. This suggests that the Modified Euler method has improved accuracy compared to the Euler method for the same stepsize, however as a consequence, the function f on the right hand side of the differential equation has to be calculated twice for every step; once in the prediction stage and once for the correction. However even with this in mind, doubling the number of calculations to improve accuracy can also warrant for a coarser choice of the stepsize to allow for a more efficient use of computational time.



4.2 Accuracy of the Modified Euler Method

In order to asses the accuracy of the Modified Euler method, consider the Taylor series expansion of y at the points t_0 and t_1 about $t_{0.5} = t_0 + \frac{1}{2}h$:

$$y(t_1) = y\left(t_{0.5} + \frac{h}{2}\right) = y(t_{0.5}) + \frac{h}{2}y'(t_{0.5}) + \left(\frac{h}{2}\right)^2 \frac{1}{2!}y''(t_{0.5}) + \mathcal{O}\left(h^3\right),$$

$$y(t_0) = y\left(t_{0.5} - \frac{h}{2}\right) = y(t_{0.5}) - \frac{h}{2}y'(t_{0.5}) + \left(\frac{h}{2}\right)^2 \frac{1}{2!}y''(t_{0.5}) + \mathcal{O}\left(h^3\right).$$

Subtracting $y(t_0)$ from $y(t_1)$ gives

$$y(t_1) - y(t_0) = hy'(t_{0.5}) + \mathcal{O}\left(h^3\right). \tag{4.1}$$

The Taylor series expansion can also be done for the derivative of y at the points t_0 and t_1 about $t_{0.5} = t_0 + \frac{1}{2}h$ in a similar way as above, i.e.

$$y'(t_1) = y'\left(t_{0.5} + \frac{h}{2}\right) = y'(t_{0.5}) + \frac{h}{2}y''(t_{0.5}) + \mathcal{O}\left(h^2\right),$$

$$y'(t_0) = y'\left(t_{0.5} - \frac{h}{2}\right) = y'(t_{0.5}) - \frac{h}{2}y''(t_{0.5}) + \mathcal{O}\left(h^2\right).$$

Adding $y'(t_0)$ to $y'(t_1)$ gives

$$y'(t_1) + y'(t_0) = 2y'(t_{0.5}) + \mathcal{O}\left(h^2\right),$$

thus multiplying by $\frac{h}{2}$ and using equation Equation 4.1 yields

$$\frac{h}{2}\left[y'(t_1) + y'(t_0)\right] = y(t_1) - y(t_0) + \mathcal{O}\left(h^3\right). \tag{4.2}$$

The first step of the Modified Euler method is to predict the value of $y'(t_1)$ using the Euler iteration;

$$\tilde{Y}_1 = \underbrace{y(t_0) + hy'(t_0)}_{\approx y(t_1)} + \mathcal{O}\left(h^2\right).$$

Hence

$$y'(t_1) = f(t_1, y(t_1)) \approx f(t_1, \tilde{Y}_1) + \mathcal{O}\left(h^2\right).$$

All this information can now be used to obtain the improved update Y_1 which is the corrected form of $\tilde{Y}_1$. Thus from equation Equation 4.2,

$$\underbrace{y(t_1)}_{\approx Y_1} = \underbrace{y(t_0)}_{=Y_0} + \frac{h}{2}\Big[\underbrace{y'(t_1)}_{=f(t_1,\tilde{Y}_1)} + \underbrace{y'(t_0)}_{=f(t_0,Y_0)}\Big] + \mathcal{O}\left(h^3\right)$$

$$\implies \quad Y_1 = Y_0 + \frac{h}{2}\left[f(t_1, \tilde{Y}_1) + f(t_0, Y_0)\right]. \tag{4.3}$$

Equations Equation 4.3 and Equation 4.2 can be used to find the local truncation error for the Modified Euler method at the first time step which is

$$e = |y(t_1) - Y_1| = \left|y(t_1) - \left[y(t_0) + \frac{h}{2}\left(y'(t_1) + y'(t_0)\right)\right]\right| + \mathcal{O}\left(h^3\right) = \mathcal{O}\left(h^3\right).$$

Therefore the *local truncation error* $e = \mathcal{O}\left(h^3\right)$ meaning that the Modified Euler method is third order accurate which is an improvement over the Euler method.

The global integration error can be obtained just as before to show that the global integration error of the Modified Euler method is $E = \mathcal{O}\left(h^2\right)$ meaning that this is a second order method. In particular, if the stepsize h is halved, the global integration error will be reduced by a factor of four while the local truncation error will reduce by a factor of eight.

4.3 MATLAB Code

The following MATLAB code performs the Modified Euler iteration for the following set of IVPs on the interval $[0, 1]$:

$$\begin{aligned} \frac{\mathrm{d}u}{\mathrm{d}t} &= 2u + v + w + \cos(t), & u(0) &= 0 \\ \frac{\mathrm{d}v}{\mathrm{d}t} &= \sin(u) + \mathrm{e}^{-v+w}, & v(0) &= 1 \\ \frac{\mathrm{d}w}{\mathrm{d}t} &= uv - w, & w(0) &= 0. \end{aligned}$$

> **Linearity**
>
> Note that this code is built for a general case that *does not* have to be linear even though the entire derivation process was built on the fact that the system is linear.

```
function IVP_Mod_Euler

%% Solve a set of first order IVPs using Modified Euler

% This code solves a set of IVP when written explicitly
% on the interval [t0,tf] subject to the initial conditions
% y(0)=y0.  The output will be the graph of the solution(s)
% and the vector value at the final point tf.  Note that the
% IVPs do not need to be linear or homogeneous.

%% Lines to change:

% Line 28   : t0 - Start time
% Line 31   : tf - End time
% Line 34   : N  - Number of subdivisions
% Line 37   : y0 - Vector of initial values
% Line 115+ : Which functions to plot, remembering to assign
%             a colour, texture and legend label
% Line 135+ : Set of differential equations written
%             explicitly.  These can also be non-linear and
%             include forcing terms.  These equations can
%             also be written in matrix form if the
%             equations are linear.

%% Set up input values

% Start time
t0=0;
```

```
29
30 % End time
31 tf=1;
32
33 % Number of subdivisions
34 N=50;
35
36 % Column vector initial values y0=y(t0)
37 y0=[0;1;0];
38
39 %% Set up IVP solver parameters
40
41 % T = Vector of times t0,t1,...,tN.
42 % This is generated using linspace which splits the
43 % interval [t0,tf] into N+1 points (or N subintervals)
44 T=linspace(t0,tf,N+1);
45
46 % Stepsize
47 h=(tf-t0)/N;
48
49 % Number of differential equations
50 K=length(y0);
51
52 %% Perform the Modified Euler iteration
53
54 % Y = Solution matrix
55 % The matrix Y will contain K rows and N+1 columns.  Every
56 % row corresponds to a different IVP and every column
57 % corresponds to a different time.  So the matrix Y will
58 % take the following form:
59 % y_1(t_0)  y_1(t_1)  y_1(t_2)  ...  y_1(t_N)
60 % y_2(t_0)  y_2(t_1)  y_2(t_2)  ...  y_2(t_N)
61 % ...
62 % y_K(t_0)  y_K(t_1)  y_K(t_2)  ...  y_K(t_N)
63 Y=zeros(K,N+1);
64
65 % The first column of the vector Y is the initial vector y0
66 Y(:,1)=y0;
67
68 % Set the current time t to be the starting time t0 and the
69 % current value of the vector y to be the strtaing values y0
70 t=t0;
71 y=y0;
72
73 for n=2:1:N+1
```

```
74
75      % Prediction Step:
76      % Use the Euler iteration to obtain an appromxation for
77      % the derivatives at the current time step
78
79      dydt=DYDT(t,y,K);      % Find gradient at the current step
80      y_pred=y+h*dydt;    % Predict y at current time step
81
82      % Corrector Step:
83      % Use the Modified Euler to correct y_pred
84
85      dydt_pred=DYDT(t,y_pred,K);     % Predict the gradient
86                                      % from the predicted y
87      y=y+0.5*h*(dydt+dydt_pred); % Find y at the current step
88
89      t=T(n);             % Update the new time
90
91      Y(:,n)=y;           % Replace row n in Y with y
92
93  end
94
95  %% Setting plot parameters
96
97  % Clear figure
98  clf
99
100 % Hold so more than one line can be drawn
101 hold on
102
103 % Turn on grid
104 grid on
105
106 % Setting font size and style
107 set(gca,'FontSize',20,'FontName','Times')
108
109 % Label the axes
110 xlabel('$t$','Interpreter','Latex')
111 ylabel('$\mathbf{y}(t)$','Interpreter','Latex')
112
113 % Plot the desried solutions.  If all the solutions are
114 % needed, then consider using a for loop in that case
115 plot(T,Y(1,:),'-b','LineWidth',2)
116 plot(T,Y(2,:),'-r','LineWidth',2)
117 plot(T,Y(3,:),'-k','LineWidth',2)
118
```

```
% Legend labels
legend('$y_1(t)$','$y_2(t)$','$y_3(t)$')
set(legend,'Interpreter','Latex')

% Display the values of the vector y at tf
disp(strcat('The vector y at t=',num2str(tf),' is:'))
disp(Y(:,end))

end

function [dydt]=DYDT(t,y,K)

% When the equation are written in explicit form

dydt=zeros(K,1);

dydt(1)=2*y(1)+y(2)+y(3)+cos(t);

dydt(2)=sin(y(1))+exp(-y(2)+y(3));

dydt(3)=y(1)*y(2)-y(3);

% If the set of equations is linear, then these can be
% written in matrix form as dydt=A*y+b(t).  For example, if
% the set of equations is:
% dudt = 7u - 2v +  w + exp(t)
% dvdt = 2u + 3v - 9w + cos(t)
% dwdt =      2v + 5w + 2
% Then:
% A=[7,-2,1;2,3,-9;0,2,5];
% b=@(t) [exp(t);cos(t);2];
% dydt=A*y+b(t)

end
```

5 The Runge-Kutta Method

The Modified Euler method extended the Euler method to a two-stage procedure with a global integration error of $\mathcal{O}\left(h^2\right)$. This can be extended further to a ***Multi-Stage Method***, also called a ***Runge-Kutta Method*** with p stages and a global error integration error of $\mathcal{O}\left(h^p\right)$ for any arbitrarily large p (in this case, the Modified Euler method is known as a second order Runge-Kutta method since it has two stages). For instance, the *fourth order Runge-Kutta method* requires four calculations for every step and has a global integration error of $\mathcal{O}\left(h^4\right)$, this is formulated as follows:

$$\begin{aligned} K_1 &= f(t_n, Y_N), \\ K_2 &= f\left(t_n + \frac{h}{2}, Y_n + \frac{h}{2}K_1\right), \\ K_3 &= f\left(t_n + \frac{h}{2}, Y_n + \frac{h}{2}K_2\right), \\ K_4 &= f\left(t_{n+1}, Y_n + hK_3\right) \\ Y_{n+1} &= Y_n + \frac{h}{6}\left[K_1 + 2K_2 + 2K_3 + K_4\right]. \end{aligned}$$

Runge-Kutta methods like this are quite versatile and are generally the most used methods for their accuracy since the stepsize h does not need to be too small to achieve good results. Even though every step requires four calculations, the value of h can be made larger in order to reduce the cost but retain considerable accuracy.

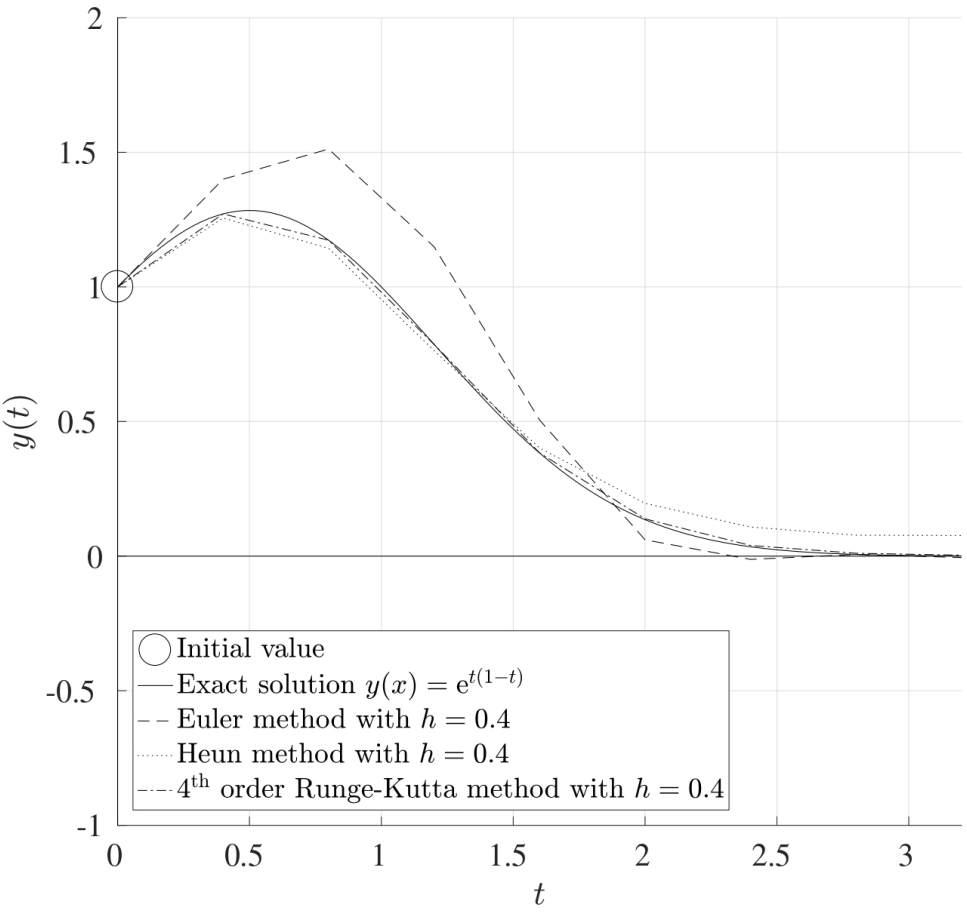
Runge-Kutta Method

Consider the differential equation

$$\frac{\mathrm{d}y}{\mathrm{d}t} = y(1-2t) \quad \text{where} \quad y(0) = 1, \quad t \in [0, 3.2].$$

The exact solution to this differential equation is known to be

$$y(t) = \mathrm{e}^{t(1-t)}.$$



The figure above shows the exact solution to the differential equation (solid line) with the three different methods used to approximate the solution all at the same resolution of $h = 0.4$. The stepsize h is quite coarse but this is merely for the purposes of demonstration. The Euler method is the least accurate for this coarse grid, the Heun method improves the accuracy while the fourth order Runge-Kutta method is the most accurate out of the three even for the same stepsize.

5.1 MATLAB Code

The following MATLAB code performs the fourth order Runge-Kutta iteration for the following set of IVPs on the interval $[0, 1]$:

$$\begin{aligned} \frac{\mathrm{d}u}{\mathrm{d}t} &= 2u + v + w + \cos(t), & u(0) &= 0 \\ \frac{\mathrm{d}v}{\mathrm{d}t} &= \sin(u) + \mathrm{e}^{-v+w}, & v(0) &= 1 \\ \frac{\mathrm{d}w}{\mathrm{d}t} &= uv - w, & w(0) &= 0. \end{aligned}$$

> **Linearity**
>
> Note that this code is built for a general case that *does not* have to be linear even though the entire derivation process was built on the fact that the system is linear.

```
function IVP_RK4

%% Solve a set of first order IVPs using RK4

% This code solves a set of IVP when written explicitly
% on the interval [t0,tf] subject to the initial conditions
% y(0)=y0.  The output will be the graph of the solution(s)
% and the vector value at the final point tf.  Note that the
% IVPs do not need to be linear or homogeneous.

%% Lines to change:

% Line 28   : t0 - Start time
% Line 31   : tf - End time
% Line 34   : N  - Number of subdivisions
% Line 37   : y0 - Vector of initial values
% Line 109+ : Which functions to plot, remembering to assign
%             a colour, texture and legend label
% Line 129+ : Set of differential equations written
%             explicitly.  These can also be non-linear and
%             include forcing terms.  These equations can
%             also be written in matrix form if the
%             equations are linear.

%% Set up input values

% Start time
t0=0;
```

```
29
30 % End time
31 tf=1;
32
33 % Number of subdivisions
34 N=50;
35
36 % Column vector initial values y0=y(t0)
37 y0=[0;1;0];
38
39 %% Set up IVP solver parameters
40
41 % T = Vector of times t0,t1,...,tN.
42 % This is generated using linspace which splits the
43 % interval [t0,tf] into N+1 points (or N subintervals)
44 T=linspace(t0,tf,N+1);
45
46 % Stepsize
47 h=(tf-t0)/N;
48
49 % Number of differential equations
50 K=length(y0);
51
52 %% Perform the RK4 iteration
53
54 % Y = Solution matrix
55 % The matrix Y will contain K rows and N+1 columns.  Every
56 % row corresponds to a different IVP and every column
57 % corresponds to a different time.  So the matrix Y will
58 % take the following form:
59 % y_1(t_0)  y_1(t_1)  y_1(t_2)  ...  y_1(t_N)
60 % y_2(t_0)  y_2(t_1)  y_2(t_2)  ...  y_2(t_N)
61 % ...
62 % y_K(t_0)  y_K(t_1)  y_K(t_2)  ...  y_K(t_N)
63 Y=zeros(K,N+1);
64
65 % The first column of the vector Y is the initial vector y0
66 Y(:,1)=y0;
67
68 % Set the current time t to be the starting time t0 and the
69 % current value of the vector y to be the strtaing values y0
70 t=t0;
71 y=y0;
72
73 for n=2:1:N+1
```

```
74
75      % Determine the coefficients of RK4
76
77      K1=DYDT(t,y,K);
78      K2=DYDT(t+h/2,y+h*K1/2,K);
79      K3=DYDT(t+h/2,y+h*K2/2,K);
80      K4=DYDT(t+h,y+h*K3,K);
81      y=y+(h/6)*(K1+2*K2+2*K3+K4);
82
83      t=T(n);             % Update the new time
84
85      Y(:,n)=y;           % Replace row n in Y with y
86
87  end
88
89  %% Setting plot parameters
90
91  % Clear figure
92  clf
93
94  % Hold so more than one line can be drawn
95  hold on
96
97  % Turn on grid
98  grid on
99
100 % Setting font size and style
101 set(gca,'FontSize',20,'FontName','Times')
102
103 % Label the axes
104 xlabel('$t$','Interpreter','Latex')
105 ylabel('$\mathbf{y}(t)$','Interpreter','Latex')
106
107 % Plot the desried solutions.  If all the solutions are
108 % needed, then consider using a for loop in that case
109 plot(T,Y(1,:),'-b','LineWidth',2)
110 plot(T,Y(2,:),'-r','LineWidth',2)
111 plot(T,Y(3,:),'-k','LineWidth',2)
112
113 % Legend labels
114 legend('$y_1(t)$','$y_2(t)$','$y_3(t)$')
115 set(legend,'Interpreter','Latex')
116
117 % Display the values of the vector y at tf
118 disp(strcat('The vector y at t=',num2str(tf),' is:'))
```

```
disp(Y(:,end))

end

function [dydt]=DYDT(t,y,K)

% When the equation are written in explicit form

dydt=zeros(K,1);

dydt(1)=2*y(1)+y(2)+y(3)+cos(t);

dydt(2)=sin(y(1))+exp(-y(2)+y(3));

dydt(3)=y(1)*y(2)-y(3);

% If the set of equations is linear, then these can be
% written in matrix form as dydt=A*y+b(t).  For example, if
% the set of equations is:
% dudt = 7u - 2v +  w + exp(t)
% dvdt = 2u + 3v - 9w + cos(t)
% dwdt =      2v + 5w + 2
% Then:
% A=[7,-2,1;2,3,-9;0,2,5];
% b=@(t) [exp(t);cos(t);2];
% dydt=A*y+b(t)

end
```

6 MATLAB's In-Built Procedures

So far, the three main iterative methods have been developed that solve IVPs numerically. MATLAB, however, has its own built-in procedures that can solve IVPs with a combination of several methods. The two main ones are `ode23` (which uses a combination of a second and third order RK methods) and `ode45` (which uses a combination of a fourth and fifth order RK methods).

Both `ode45` and `ode23` are hybrid methods and use adaptive meshing, this means that the time span grid is not necessarily uniform, but it changes depending on the gradients; if the gradient is large at some point, then the stepsize will be small to capture these drastic changes.

The following MATLAB code solves the following set of IVPs on the interval $[0, 1]$ using `ode45`:

$$\begin{aligned} \frac{\mathrm{d}u}{\mathrm{d}t} &= 2u + v + w + \cos(t), & u(0) &= 0 \\ \frac{\mathrm{d}v}{\mathrm{d}t} &= \sin(u) + \mathrm{e}^{-v+w}, & v(0) &= 1 \\ \frac{\mathrm{d}w}{\mathrm{d}t} &= uv - w, & w(0) &= 0. \end{aligned}$$

```
function IVP_InBuilt

%% Solve a set of first order IVPs using In-Built codes

% This code solves a set of IVP when written explicitly
% on the interval [t0,tf] subject to the initial conditions
% y(0)=y0.  The output will be the graph of the solution(s)
% and the vector value at the final point tf.  Note that the
% IVPs do not need to be linear or homogeneous.

%% Lines to change:

% Line 28   : t0 - Start time
% Line 31   : tf - End time
% Line 43   : T_Span - Time span for evaluation
% Line 46   : y0 - Vector of initial values
% Line 85+  : Which functions to plot, remembering to assign
%             a colour, texture and legend label
% Line 105+ : Set of differential equations written
```

```
%             explicitly.  These can also be non-linear and
%             include forcing terms.  These equations can
%             also be written in matrix form if the
%             equations are linear.

%% Set up input values

% Start time
t0=0;

% End time
tf=1;

% Time span
% In-built methods tend to use adaptive meshing; decreasing
% the stepsize near locations with drastic derivative
% changes and increasing near small derivative changes.
% Sometimes this is not desired but a uniform meshing is
% requiredfrom the start time t0 to the end time tf being
% split into N equal sub intervals.  This can be changed
% here:
% Adaptive meshing:  T_Span=[t0 tf]
% Specific meshing:  T_Span=linspace(t0,tf,N)
T_Span=[t0 tf];

% Column vector initial values y0=y(t0)
y0=[0;1;0];

%% Set up IVP solver parameters

% Number of differential equations
K=length(y0);

%% Use solver

% Set the solver tolerance
tol=odeset('RelTol',1e-6);

% Solve the IVP using ode45 or ode23
[T,Y]=ode45(@(t,y) DYDT(t,y,K),T_Span,y0,tol);

% Convert T and Y to columns for consistency
T=T';
Y=Y';

```

```
%% Setting plot parameters

% Clear figure
clf

% Hold so more than one line can be drawn
hold on

% Turn on grid
grid on

% Setting font size and style
set(gca,'FontSize',20,'FontName','Times')

% Label the axes
xlabel('$t$','Interpreter','Latex')
ylabel('$\mathbf{y}(t)$','Interpreter','Latex')

% Plot the desried solutions.  If all the solutions are
% needed, then consider using a for loop in that case
plot(T,Y(1,:),'-b','LineWidth',2)
plot(T,Y(2,:),'-r','LineWidth',2)
plot(T,Y(3,:),'-k','LineWidth',2)

% Legend labels
legend('$y_1(t)$')
set(legend,'Interpreter','Latex')

% Display the values of the vector y at tf
disp(strcat('The vector y at t=',num2str(tf),' is:'))
disp(Y(:,end))

end

function [dydt]=DYDT(t,y,K)

% When the equation are written in explicit form

dydt=zeros(K,1);

dydt(1)=2*y(1)+y(2)+y(3)+cos(t);

dydt(2)=sin(y(1))+exp(-y(2)+y(3));

dydt(3)=y(1)*y(2)-y(3);
```

```
110
111 % If the set of equations is linear, then these can be
112 % written in matrix form as dydt=A*y+b(t).  For example, if
113 % the set of equations is:
114 % dudt = 7u - 2v +  w + exp(t)
115 % dvdt = 2u + 3v - 9w + cos(t)
116 % dwdt =      2v + 5w + 2
117 % Then:
118 % A=[7,-2,1;2,3,-9;0,2,5];
119 % b=@(t) [exp(t);cos(t);2];
120 % dydt=A*y+b(t)
121
122 end
```

7 Implicit IVP Solvers

In some cases, IVPs can be difficult to solve because of the non-linearily of its terms, this is where ***Implicit Methods*** should be used to accomodate for these issues.

7.1 Backwards Euler Method

Consider the Euler method at the starting time $t = t_0$. The value of the function y at $t_1 = t_0 + h$ is approximated by

$$y(t_1) \approx Y_1 = Y_0 + hy'(t_0)$$

and this gives an upper bound for a stable stepsize of

$$h_0 = 2 \min \left(\frac{|\Re(\lambda_k)|}{|\lambda_k|^2} \right)$$

in order to ensure that the Euler method is computationally stable. However, suppose that this modified slightly by using the gradient at $y(t_1)$ rather than at $y(t_0)$, in other words, suppose that the value of y at t_1 is approximated by

$$y(t_1) \approx Y_1 = Y_0 + h\underline{\underline{y'(t_1)}}.$$

This approach is known as the ***Backwards Euler Method*** and is an implicit procedure since the value of $y'(t_1)$ is not known to begin with.

The general formulation is as follows: Consider the system of differential equations

$$\boldsymbol{y}' = A\boldsymbol{y} + \boldsymbol{b}(t) \quad \text{with} \quad \boldsymbol{y}(0) = \boldsymbol{y}_0, \quad x \in [t_0, t_f].$$

Discretise the interval $[t_0, t_f]$ into N equal subintervals, each with width $h = \frac{t_f - t_0}{N}$. At the time step $t = t_n = t_0 + nh$, the backwards Euler method is

$$\boldsymbol{Y}_{n+1} = \boldsymbol{Y}_n + h\boldsymbol{y}'(t_{n+1}) = \boldsymbol{Y}_n + h\left[A\boldsymbol{Y}_{n+1} + \boldsymbol{b}(t_{n+1})\right].$$

This can be rearranged to give

$$(I - hA)\boldsymbol{Y}_{n+1} = \boldsymbol{Y}_n + h\boldsymbol{b}(t_{n+1}).$$

Rearranging further fives the basis for the Backwards Euler iteration which is

$$\boldsymbol{Y}_{n+1} = (I - hA)^{-1}\left[\boldsymbol{Y}_n + h\boldsymbol{b}(t_{n+1})\right]$$

whereas the standard Euler method in matrix form is

$$\boldsymbol{Y}_{n+1} = (I + hA)\boldsymbol{Y}_n + h\boldsymbol{g}(t_n).$$

The Euler method requires *explicit calculations* using matrix multiplications but the backwards Euler method requires matrix inversion instead.

7.2 Stability of the Backwards Euler Method

Consider the initial value problem in its scalar form

$$\frac{\mathrm{d}y}{\mathrm{d}t} = \lambda y + b(t) \quad \text{with} \quad y(0) = y_0.$$

The backwards Euler method at the time $t = t_{n+1} = t_0 + (n+1)h$ gives

$$Y_{n+1} = (1 - h\lambda)^{-1} \left[Y_n + hg(t_{n+1}) \right].$$

This initial condition can be perturbed by adding a small parameter $\varepsilon \neq 0$ to give the perturbed differential equation

$$\frac{\mathrm{d}z}{\mathrm{d}t} = \lambda z + g(t) \quad \text{with} \quad z(0) = y_0 + \varepsilon.$$

The backwards Euler then yields

$$Z_{n+1} = (1 - h\lambda)^{-1} \left[Z_n + hg(t_{n+1}) \right]$$

The differential equations in Y and Z can be subtracted to give a perturbation term E where

$$E_{n+1} = Z_{n+1} - Y_{n+1} = (1 - h\lambda)^{-1} \left[Z_n - Y_n \right] = (1 - h\lambda)^{-1} E_n.$$

Notice that once again, the forcing function $g(t)$ has been eliminated and therefore does not affect the stability of the backwards Euler method. The differential equation for E will have the initial condition $E_0 = Z_0 - Y_0 = \varepsilon$. This expression can be used to represent E_n in terms of ε recursively as:

$$\begin{aligned} E_n = (1 - h\lambda)^{-1} E_{n-1} &= (1 - h\lambda)^{-2} E_{n-2} \\ &= \cdots = (1 - h\lambda)^{-(n-1)} E_1 = (1 - h\lambda)^{-n} E_0 = (1 - h\lambda)^{-n} \varepsilon. \\ \implies \quad E_n &= (1 - h\lambda)^{-n} \varepsilon. \end{aligned}$$

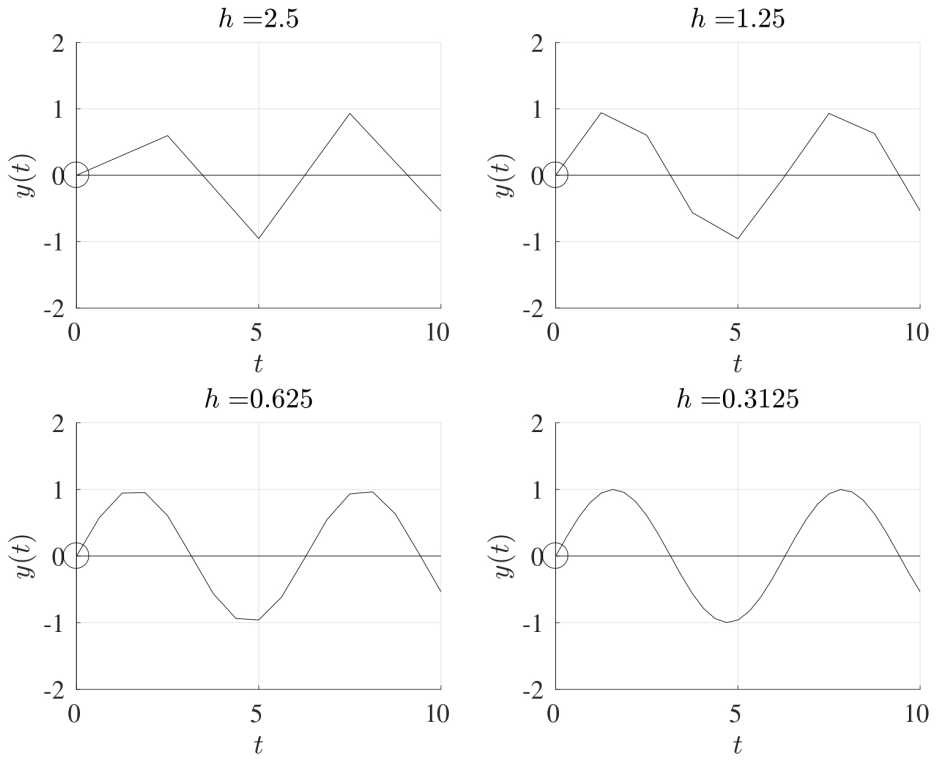
This means that the method is stable for stepsizes h that satisfy $|1 - h\lambda| > 1$ and since $\lambda < 0$ for an asymptotically stable system, then this inequality is *always* satisfied. This means that the backwards Euler method is stable *for all* stepsizes $h > 0$, no matter how large.

Backwards Euler Method

Consider the differential equation

$$y' = -100y + 100 \sin(t) \quad \text{with} \quad y(0) = 1.$$

In this case, $\lambda < 0$ meaning that this differential equation is asymptotic stable. The maximum allowable stepsize for the Euler method is $h_0 = \frac{2}{|-100|} = 0.02$. However, the backwards Euler method is stable for any stepsize h as seen below (very large stepsizes will still converge but they will not give any useful information).



The formulation presented above also holds for sets of differential equations in the same way with one difference. Instead of having $(1 - h\lambda)^{-1} = \frac{1}{1-h\lambda}$, the procedure for systems will require the matrix inverse $(1 - \lambda A)^{-1}$ or the MATLAB backslash operator can be used instead.

7.3 Order of Accuracy

The backwards Euler method is numerically stable for all values the stepsize h and has the same order of accuracy as the Euler method, i.e. the local truncation error is of $\mathcal{O}(h^2)$ while the global integration error is of $\mathcal{O}(h)$. However, this increased stability comes at a cost, the backwards Euler methods requires double the computational cost compared to the Euler method.

7.4 Stiff Differential Equations

Stiff sets of differential equations with a large value of the total computational cost N_0 can be very difficult to solve numerically using explicit methods but implicit methods can work very well. MATLAB has its very own built-in stiff differential equation solver under

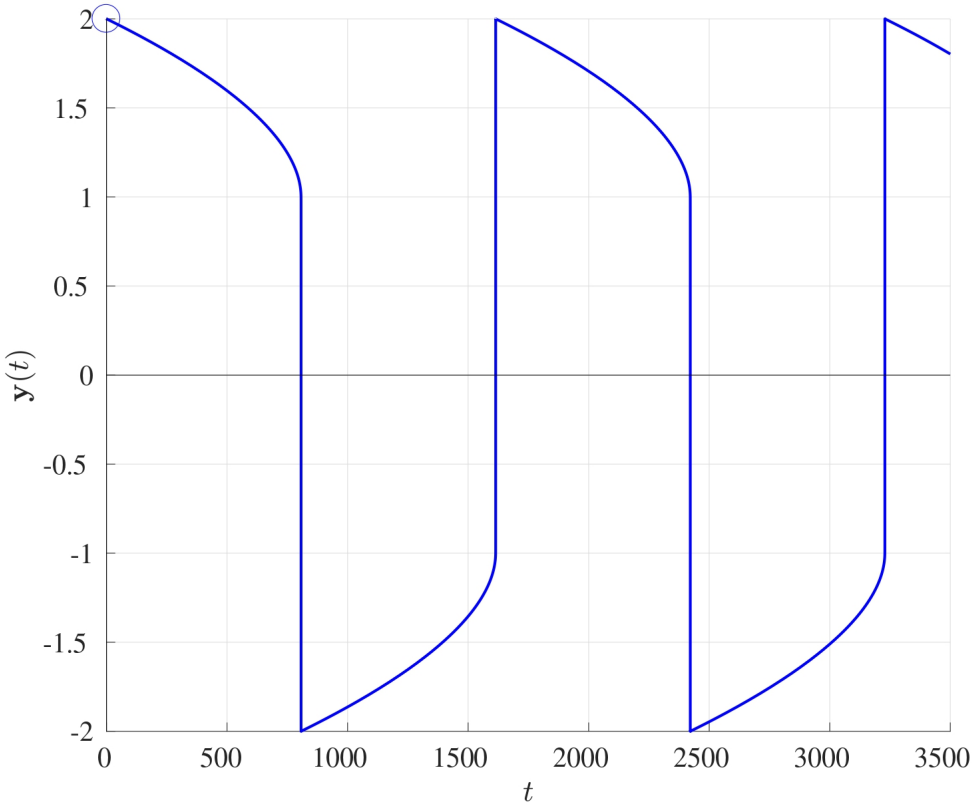
the command `ode15s` and this can be implemented exactly as `ode45`. This solves sets of differential equations implicitly using numerical differentiation of orders 1 to 5.

Stiff IVPs

Consider the set of differential equations on the interval $[0, 3500]$

$$\begin{aligned} \frac{\mathrm{d}y_1}{\mathrm{d}t} &= y_2 & y_1(0) &= 2 \\ \frac{\mathrm{d}y_2}{\mathrm{d}t} &= 1000(1 - y_1^2)y_2 - y_1 & y_2(0) &= 0. \end{aligned}$$

This is a very stiff set of differential equations, solving this using `ode45` takes upwards of 92 seconds while solving using the stiff solver `ode15s` requires a mere 0.233 seconds (depending on you machine). The result of solving this differential equation is shown below for $y_1(t)$ only since $y_2(t)$ takes very large values and this distorts the graphical interpretation.



Using the stiff solver optimises the stepsizes for stiff regions. Particularly, if a region is deemed to be considerably "stiff", the `ode15s` will use smaller stepsizes to solve the problem but if there is a region where the differential is not "stiff", then larger stepsizes will be used. Therefore, `ode15s` usually requires fewer grid points overall, for instance to solve the above set of differential equations, `ode15s` only requires 1,836 grid points

while `ode45` requires 7,820,485 grid points, that is over 4,200 times more grid points than `ode15s`. This just goes to show that stiff differential need implicit methods, even though the cost for every step is greater than that of an explicit method, fewer steps are required in total.
An alternative stiff differential equation solver is `ode23s` which achieves that same outcome as `ode15s` but with a lower accuracy and more grid points using only second and third order methods.

Part III

Solving Boundary Value Problems

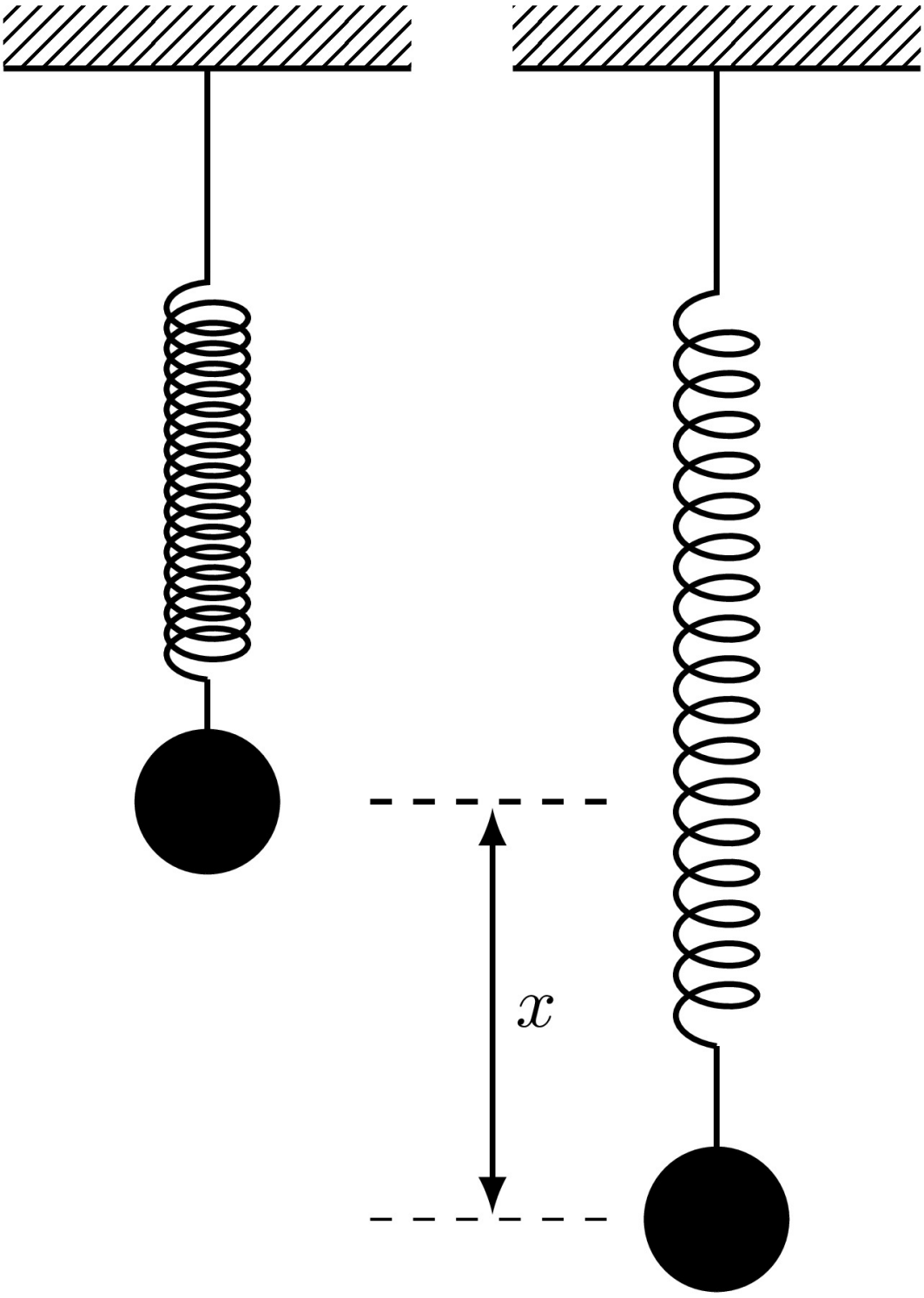
8 Boundary Value Problems

Boundary Value Problems (BVPs) are similar in many ways to initial value problems in the sense that a set of differential equations are given that are to be solved subject to certain conditions. In initial value problems, these conditions are imposed at the starting time but in boundary value problems, they are imposed at particular locations.

One of the most important differences when it comes to solving BVPs versus IVPs is the existence of solutions. Solutions to initial value problems always *exist* and are *unique* (subject to certain restriction on the right hand side), this is as a consequence of the *Picard-Lindelöf* theorem. The same cannot be said for boundary value problems; the solution to BVPs could exist and be unique, exist and not be unique or not exist at all.

8.1 Example of Boundary Value Problems

Consider a mass m hanging from a spring with spring constant K. Suppose that the spring is extended (by pulling the mass) by a distance x as seen below.



Then by *Hooke's Law*, the force pulling the mass back to its equilibrium position is given

by

$$F = -Kx.$$

As the mass is released, it will accelerate upwards with an acceleration a and the force responsible for this acceleration is given by *Newton's Second Law of Motion*

$$F = ma.$$

The acceleration a is the second derivative of the displacement x with respect to time and since it acts in a direction opposite to the extension, then

$$a = -\frac{\mathrm{d}^2x}{\mathrm{d}t^2} \quad \Longrightarrow \quad F = -m\frac{\mathrm{d}^2x}{\mathrm{d}t^2}$$

Equating the two expressions for the force from Newton's Second Law and Hooke's Law will give

$$-Kx = -m\frac{\mathrm{d}^2x}{\mathrm{d}t^2} \quad \Longrightarrow \quad \frac{\mathrm{d}^2x}{\mathrm{d}t^2} + \omega^2 x = 0 \quad \text{where} \quad \omega = \sqrt{\frac{K}{m}}.$$

This differential equation represents the simple harmonic motion of a mass hanging on a frictionless massless spring which oscillates with a frequency ω. Since this is a second order differential equation, two conditions need to be imposed:

- *Initial conditions* can be imposed at the starting time, specifically $x(0)$ and $x'(0)$ which prescribe the initial position and initial speed,
- *Boundary conditions* can be imposed at different times, say $x(0)$ and $x(10)$ which prescribe the location at time $t = 0$ and time $t = 10$.

8.2 Finite Difference Method for Boundary Value Problems

Consider the general second order boundary value problem

$$a(x)\frac{\mathrm{d}^2u}{\mathrm{d}x^2} + b(x)\frac{\mathrm{d}u}{\mathrm{d}x} + c(x)u = f(x) \quad \text{with} \quad 0 < x < L$$

$$\text{and} \quad u(0) = u_l, \quad u(L) = u_r$$

where the functions a, b, c and f are known functions of x. Boundary value problems like this are solved using an incredibly versatile method known as the ***Finite Difference Method***. This procedure essentially changes a differential equation into a set of difference equations by using approximations to the derivatives.

8.3 Existence & Uniqueness of Solutions to BVPs

Consider the differential equation for the undamped simple harmonic oscillator with frequency 1, namely

$$\frac{\mathrm{d}^2u}{\mathrm{d}t^2} = -u.$$

This differential equation has the general analytic solution

$$u(t) = C_1 \cos(t) + C_2 \sin(t)$$

where C_1 and C_2 are constants of integration which will be determined form the boundary conditions.

Three qualitatively different sets of boundary conditions will be investigated:

- $u(0) = 1$ and $u(\frac{5\pi}{2}) = -1$: The constants C_1 and C_2 can be found as:

$$1 = u(0) = C_1 \cos(0) + C_2 \sin(0) = C_1 \quad \Longrightarrow \quad C_1 = 1$$

$$-1 = u\left(\frac{5\pi}{2}\right) = C_1 \cos\left(\frac{5\pi}{2}\right) + C_2 \sin\left(\frac{5\pi}{2}\right) = C_2 \quad \Longrightarrow \quad C_2 = -1.$$

  Therefore the analytic solution to the boundary value problem subject to these conditions is

$$u(t) = \cos(t) - \sin(t)$$

  and this is captured by the finite difference approximation. In this case, the solution to the boundary value problem *exists* and is *unique*.

- $u(0) = 0$ and $u(2\pi) = 0$: The constants C_1 and C_2 can be found as:

$$0 = u(0) = C_1 \cos(0) + C_2 \sin(0) = C_1 \quad \Longrightarrow \quad C_1 = 0$$

$$0 = u(2\pi) = C_1 \cos(2\pi) + C_2 \sin(2\pi) = C_1 \quad \Longrightarrow \quad C_1 = 0.$$

  These two conditions provide an expressions for the constant C_1 only and not C_2, therefore the particular solution will be

$$u(t) = C_2 \sin(t)$$

  which is valid for *any* value of C_2. Therefore in this case, the solution *exists* but is *not unique*.

- $u(0) = 1$ and $u(2\pi) = -1$: The constants C_1 and C_2 can be found as:

$$1 = u(0) = C_1 \cos(0) + C_2 \sin(0) = C_1 \quad \Longrightarrow \quad C_1 = 1$$

$$-1 = u(\pi) = C_1 \cos(2\pi) + C_2 \sin(2\pi) = C_1 \quad \Longrightarrow \quad C_1 = -1.$$

  In this case, the boundary values have resulted in a contradiction and therefore the solution *does not exist* when subject to these boundary conditions.

This final case is when the solution to a boundary value problem does *not exist*.

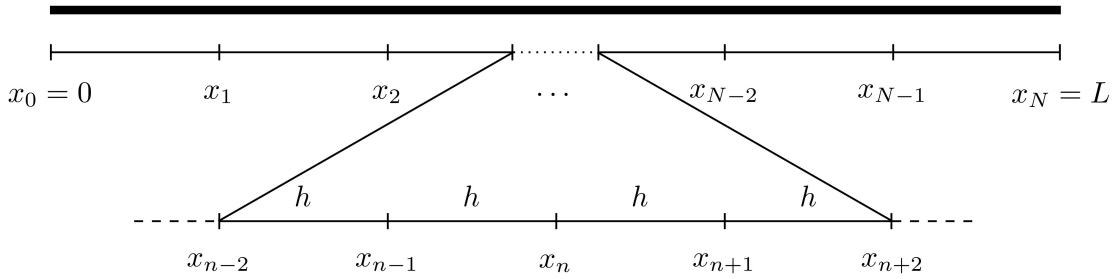
8.3.1 Finite Difference Approximations to the Derivatives

The term *finite difference approximation* refers to how derivatives can be approximated using linear expressions. For instance, the derivative of some function f at a given point X can be approximated as the gradient of f between two points around X, for example

$$\frac{\mathrm{d}f}{\mathrm{d}x}(X) \approx \frac{f(X+h) - f(X-h)}{2h}.$$

There are many other ways in which these approximations can be made depending on the way in which the grid has been set up or on the context of the problem.

Consider a general unknown function $u(x)$ defined on $[0, L]$ where $u(0)$ and $u(L)$ are given (as boundary conditions). First, split the interval into N equally sized sections, each of width h, and label the points as $x_0, x_1, \ldots, x_N$ where $x_n = nh$.



For first and second derivatives, there are three main approximations that are most widely used:

- **Forward Difference**:

$$\frac{\mathrm{d}u}{\mathrm{d}x}(x_n) \approx \frac{u(x_{n+1}) - u(x_n)}{h}$$

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2}(x_n) \approx \frac{u(x_{n+2}) - 2u(x_{n+1}) + u(x_n)}{h^2}$$

- **Backward Difference**:

$$\frac{\mathrm{d}u}{\mathrm{d}x}(x_n) \approx \frac{u(x_n) - u(x_{n-1})}{h}$$

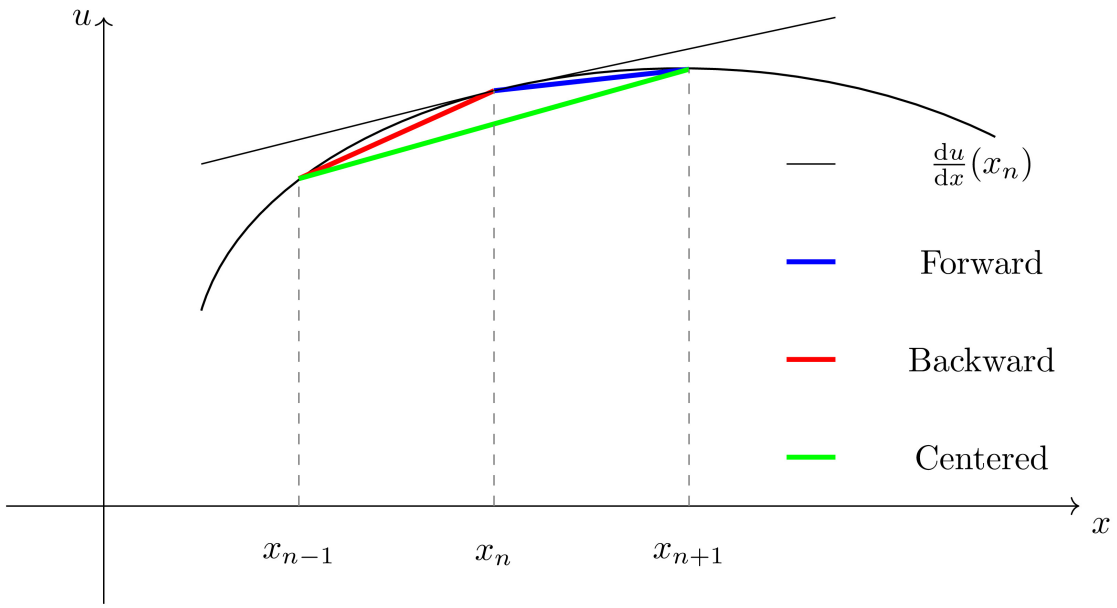
$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2}(x_n) \approx \frac{u(x_n) - 2u(x_{n-1}) + u(x_{n-2})}{h^2}$$

- **Centred Difference**:

$$\frac{\mathrm{d}u}{\mathrm{d}x}(x_n) \approx \frac{u(x_{n+1}) - u(x_{n-1})}{2h}$$

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2}(x_n) \approx \frac{u(x_{n+1}) - 2u(x_n) + u(x_{n-1})}{h^2}$$

The graphical interpretation of the approximations to the first derivatives are shown below.



Second Derivative Expression

To show how the second derivative expressions are obtained, consider the centred difference approximation

$$\frac{\mathrm{d}u}{\mathrm{d}x}(x_n) \approx \frac{u(x_{n+1}) - u(x_{n-1})}{2h}.$$

To derive the expression for the second derivative, introduce two fictitious points $x_{n-0.5}$ (which is half-way between x_{n-1} and x_n) and $x_{n+0.5}$ (which is half-way between x_n and x_{n+1}). Then

$$\begin{aligned}\frac{\mathrm{d}^2u}{\mathrm{d}x^2}(x_n) = \frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{\mathrm{d}u}{\mathrm{d}x}(x_n)\right) &\approx \frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{u(x_{n+0.5}) - u(x_{n-0.5})}{h}\right) \\ &\approx \frac{u'(x_{n+0.5}) - u'(x_{n-0.5})}{h} \\ &\approx \frac{\frac{u(x_{n+1})-u(x_n)}{h} - \frac{u(x_n)-u(x_{n-1})}{h}}{h} \\ &= \frac{u(x_{n+1}) - 2u(x_n) + u(x_{n-1})}{h^2}.\end{aligned}$$

The derivation of the second derivative approximations for the forward and backward differences can be done in a very similar way but without the need for half steps.

Any of these three approximations can be used to approximate the derivatives of the function u at the point x_n. Denote the approximation of u at the point x_n by U_n, i.e. $U_n \approx u(x_n)$,

then

- **Forward Difference**:

$$\frac{\mathrm{d}u}{\mathrm{d}x}(x_n) \approx \frac{U_{n+1} - U_n}{h} \quad ; \quad \frac{\mathrm{d}^2u}{\mathrm{d}x^2}(x_n) \approx \frac{U_{n+2} - 2U_{n+1} + U_n}{h^2}$$

- **Backward Difference**:

$$\frac{\mathrm{d}u}{\mathrm{d}x}(x_n) \approx \frac{U_n - U_{n-1}}{h} \quad ; \quad \frac{\mathrm{d}^2u}{\mathrm{d}x^2}(x_n) \approx \frac{U_n - 2U_{n-1} + U_{n-2}}{h^2}$$

- **Centred Difference**:

$$\frac{\mathrm{d}u}{\mathrm{d}x}(x_n) \approx \frac{U_{n+1} - U_{n-1}}{2h} \quad ; \quad \frac{\mathrm{d}^2u}{\mathrm{d}x^2}(x_n) \approx \frac{U_{n+1} - 2U_n + U_{n-1}}{h^2}.$$

These approximations will form the basis for solving the BVP.

8.3.2 Discretisation of the Differential Equation

Returning to the differential equation

$$a(x)\frac{\mathrm{d}^2u}{\mathrm{d}x^2} + b(x)\frac{\mathrm{d}u}{\mathrm{d}x} + c(x)u = f(x).$$

Evaluate this equation at $x = x_n$ for some n, then

$$a(x_n)\frac{\mathrm{d}^2u}{\mathrm{d}x^2}(x_n) + b(x_n)\frac{\mathrm{d}u}{\mathrm{d}x}(x_n) + c(x_n)u(x_n) = f(x_n).$$

For now, suppose the *centred differencing approximation* is used to approximate the derivatives. Replacing the approximations of the derivatives of u at x_n gives

$$a(x_n)\frac{U_{n+1} - 2U_n + U_{n-1}}{h^2} + b(x_n)\frac{U_{n+1} - U_{n-1}}{2h} + c(x_n)U_n = f(x_n).$$

This can be simplified by collecting the U terms resulting in:

$$\alpha_n U_{n-1} + \beta_n U_n + \gamma_n U_{n+1} = f(x_n)$$

$$\text{where} \quad \alpha_n = \frac{a(x_n)}{h^2} - \frac{b(x_n)}{2h}, \quad \beta_n = -\frac{2a(x_n)}{h^2} + c(x_n), \quad \gamma_n = \frac{a(x_n)}{h^2} + \frac{b(x_n)}{2h}.$$

This expression will hold for all the values of $n = 1, 2, \ldots, N-1$ (otherwise there will be points x_{-1} and x_{N+1} which are outside the domain $[0, L]$). Therefore, this mean that there will be $N-1$ equations in $N+1$ unknowns which are $U_0, U_1, U_2, \ldots, U_N$.

This system may seem to be undetermined however, there are two boundary conditions that have not been taken into consideration yet, namely $u(x_0) = u_l$ and $u(x_N) = u(L) = u_r$. Since these are known, the approximations U_0 and U_N have defined values, i.e. $U_0 \approx u(x_0) = u_l$ and $U_N = u(x_N) = u_r$. This eliminates two of the unknowns giving $N-1$ equations in $N-1$ unknowns.

At $n = 1$, the approximation to the differential equation is

$$\alpha_1 U_0 + \beta_1 U_1 + \gamma_1 U_2 = f(x_1)$$

and since U_0 is already known, then it can be taken to the right hand side to give

$$\beta_1 U_1 + \gamma_1 U_2 = f(x_1) - \alpha_1 u_0.$$

Similarly, at $n = N - 1$, the approximation is

$$\alpha_{N-1} U_{N-2} + \beta_{N-1} U_{N-1} + \gamma_{N-1} U_N = f(x_{N-1})$$

and since U_N is known, this can be rewritten as

$$\alpha_{N-1} U_{N-2} + \beta_{N-1} U_{N-1} = f(x_{N-1}) - \gamma_{N-1} u_L.$$

For $n = 2, 3, \ldots, N - 2$, the approximation is

$$\alpha_n U_{n-1} + \beta_n U_n + \gamma_n U_{n+1} = f(x_n)$$

where U_{n-1}, U_n and U_{n+1} are al unknown. In summary, all of these $N - 1$ equations are:

$$\begin{aligned}
n = 1: &\quad \beta_1 U_1 + \gamma_1 U_2 = f(x_1) - \alpha_1 u_0 \\
n = 2: &\quad \alpha_2 U_1 + \beta_2 U_2 + \gamma_2 U_3 = f(x_2) \\
n = 3: &\quad \alpha_3 U_2 + \beta_3 U_3 + \gamma_3 U_4 = f(x_3) \\
&\quad \vdots \\
n = N-3: &\quad \alpha_{N-3} U_{N-4} + \beta_{N-3} U_{N-3} + \gamma_{N-3} U_{N-2} = f(x_{N-3}) \\
n = N-2: &\quad \alpha_{N-2} U_{N-3} + \beta_{N-2} U_{N-2} + \gamma_{N-2} U_{N-1} = f(x_{N-2}) \\
n = N-1: &\quad \alpha_{N-1} U_{N-2} + \beta_{N-1} U_{N-1} = f(x_{N-1}) - \gamma_{N-1} u_L
\end{aligned}$$

These can be written in matrix form as $A\boldsymbol{U} = \boldsymbol{g}$, namely

$$\underbrace{\begin{pmatrix} \beta_1 & \gamma_1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ \alpha_2 & \beta_2 & \gamma_2 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & \alpha_3 & \beta_3 & \gamma_3 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & \alpha_4 & \beta_4 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \beta_{N-4} & \gamma_{N-4} & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & \alpha_{N-3} & \beta_{N-3} & \gamma_{N-3} & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & \alpha_{N-2} & \beta_{N-2} & \gamma_{N-2} \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & \alpha_{N-1} & \beta_{N-1} \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ \vdots \\ U_{N-4} \\ U_{N-3} \\ U_{N-2} \\ U_{N-1} \end{pmatrix}}_{\boldsymbol{U}}$$

$$= \underbrace{\begin{pmatrix} f(x_1) - \alpha_1 u_l \\ f(x_2) \\ f(x_3) \\ f(x_4) \\ \vdots \\ f(x_{N-4}) \\ f(x_{N-3}) \\ f(x_{N-2}) \\ f(x_{N-1}) - \gamma_{N-1} u_r \end{pmatrix}}_{\boldsymbol{g}}.$$

The matrix A is of size $(N-1) \times (N-1)$ all of whose terms are known, the vector $\boldsymbol{g}$ of size $(N-1) \times 1$ also has terms that are all known. The unknown vector here is $\boldsymbol{U}$ and it can be found by inverting A to give $\boldsymbol{U} = A^{-1}\boldsymbol{g}$.

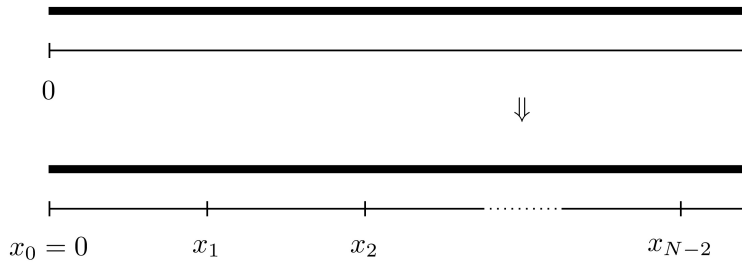
Carrying out matrix inversions by hand can become increasingly cumbersome if A is larger than 2×2 and therefore this process should be done computationally. This can be solved in MATLAB by using either `U=inv(A)*g` or `U=A\g`. The *backslash* method is faster than explicit matrix inversion if the matrix is of a large size.

The same process can be done for the forward and backward differencing approximations as well.

8.3.3 Steps of The Finite Difference Method

In summary, these are the steps of the finite difference method:

1. Divide the interval $[0, L]$ into N equally sized sections, each of width $h = \frac{L}{N}$ and label the



points as $x_0, x_1, x_2, \ldots, x_N$ where $x_n = nh$.

2. The values of the function u are to be found at all the locations x_n. Denote the approximation to the function u at the point x_n by U_n, i.e. $U_n \approx u(x_n)$ for all $n = 0, 1, 2, \ldots, N$.
3. Evaluate the differential equation at all the points x_n where the derivatives are replaced by their *finite difference approximations*.
4. This will result in a set of $N - 1$ linear equations in $N + 1$ unknowns, namely, $U_0, U_1, U_2, \ldots, U_N$.
5. The values for U_0 and U_N are known from the boundary conditions, since $U_0 = u(0) = u_l$ and $U_N = u(L) = u_r$ and no approximation is needed since the exact values are known.
6. Write the whole system of equations in the matrix form $\boldsymbol{AU} = \boldsymbol{g}$ and solve using MATLAB's backlash operator.

BVP Example

Consider the boundary value problem

$$\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} = x^3, \quad x \in [0, 2] \quad \text{with} \quad u(0) = 0 \quad \text{and} \quad u(2) = 1.$$

The differential equation itself can be solved analytically to give

$$u(x) = \frac{1}{20}x^5 - \frac{3}{10}x.$$

This example will be used for the purposes of demonstration and comparison between the numerically obtained solution and the exact solution.
Suppose the interval $[0, 2]$ is to be divided into 5 equally sized sections, therefore $N = 5$ and $h = \frac{L}{N} = \frac{2}{5} = 0.4$.

The functions $a(x), b(x), c(x)$ and $f(x)$ in this interval are:

$$a(x) = 1, \quad b(x) = 0, \quad c(x) = 0, \quad f(x) = x^3.$$

The matrix values are

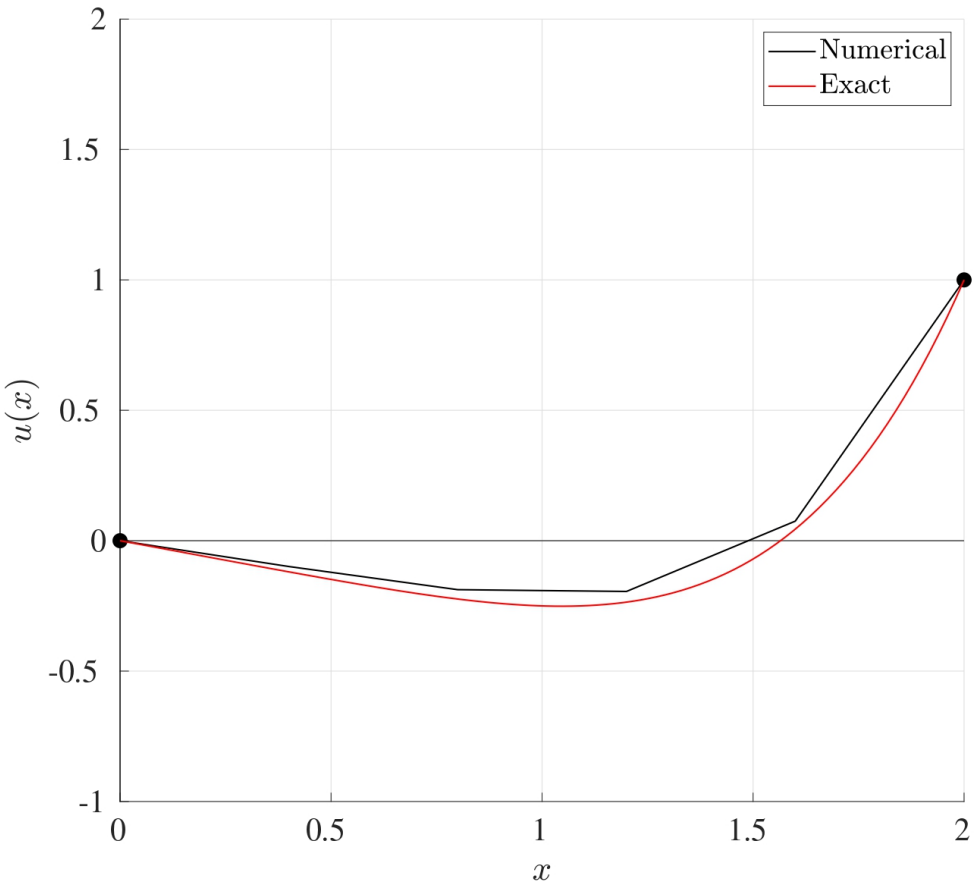
$$\alpha_n = \frac{a(x_n)}{h^2} - \frac{b(x_n)}{2h} = 6.25$$

$$\beta_n = -\frac{2a(x_n)}{h^2} + c(x_n) = -12.5$$
$$\gamma_n = \frac{a(x_n)}{h^2} + \frac{b(x_n)}{2h} = 6.25.$$

These can be used to obtain expressions for the matrix A and the vector $\boldsymbol{g}$ as

$$A = \begin{pmatrix} -12.5 & 6.25 & 0 & 0 \\ 6.25 & -12.5 & 6.25 & 0 \\ 0 & 6.25 & -12.5 & 6.25 \\ 0 & 0 & 6.25 & -12.5 \end{pmatrix}, \quad \boldsymbol{g} = \begin{pmatrix} 0.064 \\ 0.512 \\ 1.728 \\ 4.096 \end{pmatrix}.$$

This system can be solved using `U=inv(A)*g` or `U=A\g`. The numerical solution is compared to exact solution below.



The advantage of using this boundary value solver is that the computations are in no way taxing on MATLAB. The system that results is composed entirely of linear equations and this system is solvable (provided the boundary value problem does indeed have a solution which may not always be possible). MATLAB's backslash operator is very effective in dealing with matrices, especially owing to the fact that the matrix A is a tridiagonal matrix.

8.4 MATLAB Code

Below is the MATLAB code that solves the BVP

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} + 2\frac{\mathrm{d}u}{\mathrm{d}t} + \mathrm{e}^{-x}u = \sin(x), \quad x \in [0, 10] \quad \text{with} \quad u(0) = 1 \quad \text{and} \quad u(10) = -1$$

using the centred differencing method with $N = 50$.

```
function BVP_CD

%% Solve BVPs using centered differences

% The bvp is written in the form
% a(x) u'' + b(x) u' + c(x) u = f(x) on x in [x0,L]
% with the boundary conditions u(x0)=ul and u(L)=ur.
% After the centered difference approximation is
% used, the system will be written in the form AU=g.

%% Lines to change:

% Line 26  : x0 - Start point
% Line 29  : L  - End point
% Line 32  : N  - Number of subdivisions
% Line 35  : xl - Left boundary value
% Line 38  : xr - Right boundary value
% Line 119 : Expression for the function a(x)
% Line 127 : Expression for the function b(x)
% Line 135 : Expression for the function c(x)
% Line 143 : Expression for the function f(x)

%% Set up input values

% Start point
x0=0;

% End point
L=10;

% Number of subdivisions
N=50;

% Boundary value at x=x0
ul=1;

% Boundary value at x=L
```

```
ur=-1;

%% Set up BVP solver parameters

% Interval width
h=(L-x0)/N;

% X = Vector of locations
% (x1, x2, x3, ..., xN) (notice the start is x1 NOT x0)
X=x0+h:h:L;

% Evaluate the functions a(x), b(x), c(x) and f(x) at X
aX=a(X);
bX=b(X);
cX=c(X);
fX=f(X);

% Find the expressions for alpha, beta and gamma at X
alpha=aX/(h^2)-bX/(2*h);
beta=-2*aX/(h^2)+cX;
gamma=aX/(h^2)+bX/(2*h);

% Set up the vector g on the right hand side
g=zeros(N-1,1);
g(1)=fX(1)-alpha(1)*ul;
g(N-1)=fX(N-1)-gamma(N-1)*ur;
for j=2:1:N-2
    g(j)=fX(j);
end

% Set up the matrix A on the left hand side (LHS_A is
% to avoid confusion with the function a(x))
A=zeros(N-1,N-1);
A(1,1)=beta(1);
A(1,2)=gamma(1);
A(N-1,N-1)=beta(N-1);
A(N-1,N-2)=alpha(N-1);
for j=2:1:N-2
    A(j,j-1)=alpha(j);
    A(j,j)=beta(j);
    A(j,j+1)=gamma(j);
end

% Solve for the unknown vector U (it is then readjusted
% from a column vector to a row vector for plotting)
```

```
U=A\g;
U=U';

% Add the missing term x0 to the start of the vector x
X=[x0,X];

% Add the left and right boundary values to the vector U
U=[ul,U,ur];

%% Setting plot parameters

% Clear figure
clf

% Hold so more than one line can be drawn
hold on

% Turn on grid
grid on

% Setting font size and style
set(gca,'FontSize',20,'FontName','Times')

% Label the axes
xlabel('$t$','Interpreter','Latex')
ylabel('$u(t)$','Interpreter','Latex')

% Plot solution
plot(X,U,'-k','LineWidth',2)

end

function [A]=a(X)
A=zeros(size(X));
for i=1:1:length(X)
    x=X(i);
    A(i)=1;
end
end

function [B]=b(X)
B=zeros(size(X));
for i=1:1:length(X)
    x=X(i);
    B(i)=2;
```

```
end
end

function [C]=c(X)
C=zeros(size(X));
for i=1:1:length(X)
    x=X(i);
    C(i)=exp(-x);
end
end

function [F]=f(X)
F=zeros(size(X));
for i=1:1:length(X)
    x=X(i);
    F(i)=sin(x);
end
end
```

8.5 Comparison Between Forward, Backward & Centred Difference Approximations

The main difference between the different differencing schemes if the order of accuracy. Indeed, the error of the forward and backward differencing methods are $\mathcal{O}(h)$ whereas the error for the centred differencing is $\mathcal{O}(h^2)$. This means that if the stepsize h was reduced by a factor of 10, then the error for the forward and backward finite difference approximations would also reduce by a factor of 10 while the centred would reduce by a factor of 100.

Comparison

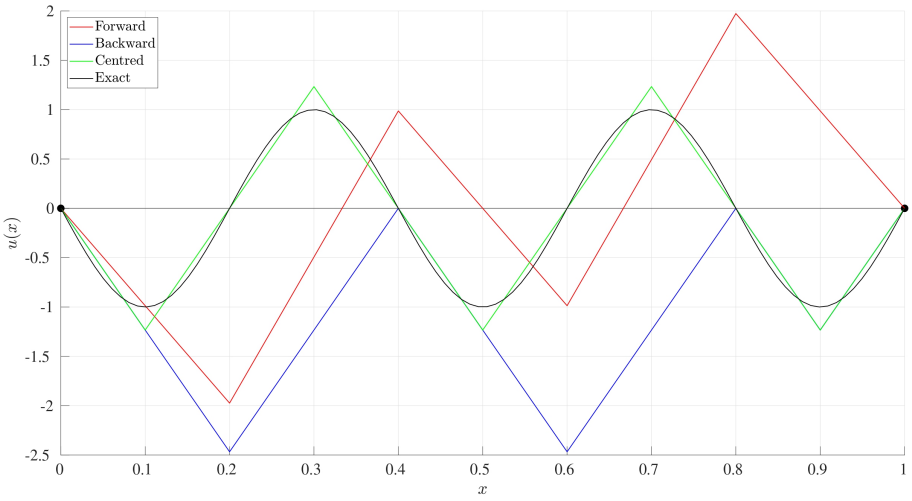
Consider the BVP

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} = 25\pi^2 \sin(5\pi x), \quad x \in [0,1] \quad \text{with} \quad u(0) = 0 \quad \text{and} \quad u(1) = 0.$$

This has the exact solution

$$u(x) = -\sin(5\pi x).$$

Below are the plots for the numerical solution to this boundary value problem using the forward (red), backward (blue) and centred (green) difference approximations compared to the exact solution when $N = 10$.



It can be seen that even for this relatively crude interval subdivision of $N = 10$, the centred approximation has yielded a far more favourable result compared to the other two methods. The following table shows the 2-norm error between the exact solution and the approximation for different values of N:

Method	$N = 10$	$N = 20$	$N = 50$	$N = 100$
Forward	4.1444	3.0875	1.9823	1.4048
Backward	4.7243	3.8535	2.0939	1.4251
Centred	0.5226	0.1677	0.0413	0.0146

It can be seen that even when $N = 100$, the 2-norm error has still not reduced below 1 for the forward and backward difference approximations but the centred has already achieved that even at $N = 10$. This is just a demonstration to show that how a simple change in the way in which derivatives are approximated can have such a drastic effect on the final solution.

8.6 MATLAB's In-Built Procedures

MATLAB has an in-built mechanism that can also solve second (or even higher order) BVPs, this is done using the `bvp4c` command.

Below is the MATLAB code that solves the BVP

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2} + 2\frac{\mathrm{d}u}{\mathrm{d}t} + \mathrm{e}^{-x}u = \sin(x), \quad x \in [0, 10] \quad \text{with} \quad u(0) = 1 \quad \text{and} \quad u(10) = -1$$

using `bvp4c`.

```
function BVP_InBuilt

%% Solve BVPs using bvp4c

% The bvp is written in the form
% a(x) u'' + b(x) u' + c(x) u = f(x) on x in [x0,L]
% with the boundary conditions u(x0)=ul and u(L)=ur.

%% Lines to change:

% Line 24 : x0 - Start point
% Line 27 : L  - End point
% Line 30 : N  - Number of spatial points
% Line 33 : xl - Left boundary value
% Line 36 : xr - Right boundary value
% Line 44 : Expression for the function a(x)
% Line 45 : Expression for the function b(x)
% Line 46 : Expression for the function c(x)
% Line 47 : Expression for the function f(x)

%% Set up input values

% Start point
x0=0;

% End point
L=10;

% Number of spatial points
N=50;

% Boundary value at x=x0
ul=1;

% Boundary value at x=L
ur=-1;

%% Set up BVP solver parameters

% Set up solving space
X=linspace(x0,L,N);

% Define the functions in the BVP
a= @(x) 1;
b= @(x) 2;
```

```
c= @(x) exp(-x);
f= @(x) sin(x);

%% Set up BVP solving parameters

% First, write the second order ODE as a set of first order
% ODEs:
% U'=V
% V'=(-b(x)*V-c(x)*U+f(x))/a(x)

% Second order BVPs can have more than one solution
% and vector v is the initialising vector of solutions.
% It can be kept as a vector of zeros
v=[0 0];

% Initialise vectors for space and v
init=bvpinit(X,v);

% Solve the bvp subject to the boundary values and
% inital guesses
sol=bvp4c(@(x,u) DUDT(x,u,a,b,c,f),@(x0,L) BCs(x0,L,ul,ur),init);

% Evaluate the solution at the grid points
U=deval(sol,X);

% Convert U to columns for consistency
U=U';

%% Setting plot parameters

% Clear figure
clf

% Hold so more than one line can be drawn
hold on

% Turn on grid
grid on

% Setting font size and style
set(gca,'FontSize',20,'FontName','Times')

% Label the axes
xlabel('$t$','Interpreter','Latex')
ylabel('$u(t)$','Interpreter','Latex')
```

```
91
92  % Plot solution
93  plot(X,U(:,1),'-k','LineWidth',2)
94
95  end
96
97  function [dudx]=DUDT(x,u,a,b,c,f)
98
99  dudx(1)=u(2);
100
101 dudx(2)=(-b(x)*u(2)-c(x)*u(1)+f(x))/(a(x));
102
103 end
104
105 function res=BCs(x0,L,ul,ur)
106 % The boundary conditions are written as
107 % u(x0)=ul
108 % x(L)=ur
109
110 res=[x0(1)-ul;L(1)-ur];
111
112 end
```

9 Mixed Value Problems

Initial and boundary value problems are not the only two ways in which conditions can be expressed. Sometimes these conditions can be presented in a *mixed form* where the condition on one or both boundaries may depend on the derivative of the solution function. For instance, consider the steady-state convection-diffusion equation on a bar on length 5 with density ρ, convective velocity v, specific heat capacity C_p, thermal conductivity k_f and heat source f:

$$-k_f \frac{\mathrm{d}^2T}{\mathrm{d}x^2} + \rho v C_p \frac{\mathrm{d}T}{\mathrm{d}x} = f(x) \quad \text{on} \quad x \in [0,5] \quad \text{with} \quad T(0) = 100 \quad \text{and} \quad \frac{\mathrm{d}T}{\mathrm{d}x}(5) = 0$$

where $T(x)$ is the temperature at x. This set of conditions are known as ***Mixed Conditions***: the first $T(0) = 100$ means that the temperature at the location $x = 0$ is 100, the second $\frac{\mathrm{d}T}{\mathrm{d}x}(5) = 0$ means that at the location $x = 5$, there is no heat *flux*. This can be quite useful if say, a metal rod is being heated to 100°C on one side an insulated on the other.

The method to solving MVPs is the same as boundary value problems subject to a few modifications.

9.1 Finite Difference Method for MVPs

Consider the differential equation

$$a(x)\frac{\mathrm{d}^2u}{\mathrm{d}x^2} + b(x)\frac{\mathrm{d}u}{\mathrm{d}x} + c(x)u = f(x) \quad \text{with} \quad 0 < x < L$$

as before. The interval $[0, L]$ will be split into N equally sized sections each of width $h = \frac{L}{N}$ and the grid points are labelled $x_n = nh$ for $n = 0, 1, 2, \ldots, N$. This differential equation can be discretised using the centred difference approximation just as before to give

$$\alpha_n U_{n-1} + \beta_n U_n + \gamma_n U_{n+1} = f(x_n) \quad \text{for} \quad n = 1, 2, \ldots, N-1$$

$$\text{where} \quad \alpha_n = \frac{a(x_n)}{h^2} - \frac{b(x_n)}{2h}, \quad \beta_n = -\frac{2a(x_n)}{h^2} + c(x_n), \quad \gamma_n = \frac{a(x_n)}{h^2} + \frac{b(x_n)}{2h}.$$

This gives a set of $N - 1$ equations in $N + 1$ unknowns, namely $U_0, U_1, U_2, \ldots, U_N$ (recall that $U_n \approx u(x_n)$ for $n = 0, 1, 2, \ldots, N$).

When the differential equation is subjected to two boundary conditions, say

$$u(0) = u_l \quad \text{and} \quad u(L) = u_r,$$

then expressions for U_0 and U_L are provided which gives $N-1$ equations in $N-1$ unknowns, hence resulting in a well-defined system which can be solved as before.

However, suppose that a set of mixed conditions is given as

$$\frac{\mathrm{d}u}{\mathrm{d}x}(0) = \tilde{u}_l \quad \text{and} \quad u(L) = u_r.$$

In this case, only $U_N \approx u(L) = u_r$ is explicitly known, meaning that there will be $N-1$ equations in N unknowns since $U_0 \approx u(x_0)$ is not known giving an under-determined system (a system with more unknowns than equations). So either one more equation is needed or one more unknown needs to be removed. All the unknowns are certainly needed, otherwise the solution will be incomplete, so the alternative is to find another equation to add to the set of equations.

The set of $N-1$ equations is:

$$\begin{aligned}
n = 1: &\quad \alpha_1 U_0 + \beta_1 U_1 + \gamma_1 U_2 = f(x_1) \\
n = 2: &\quad \alpha_2 U_1 + \beta_2 U_2 + \gamma_2 U_3 = f(x_2) \\
&\quad \vdots \\
n = N-1: &\quad \alpha_{N-1} U_{N-2} + \beta_{N-1} U_{N-1} = f(x_{N-1}) - \gamma_{N-1} u_L.
\end{aligned}$$

All these come from the discretisation

$$\alpha_n U_{n-1} + \beta_n U_n + \gamma_n U_{n+1} = f(x_n).$$

Evaluating this at $n = 0$ gives

$$\alpha_0 U_{-1} + \beta_0 U_0 + \gamma_0 U_1 = f(x_0). \tag{9.1}$$

Initially, this may seem to be quite strange since there is a point U_{-1} which is the approximation to the solution u at the point $x = x_{-1} = -h$ which is certainly out of the range of consideration. This point is considered to be *an artificial grid point* that will act as a placeholder in meantime.

Consider the condition at the start point

$$\frac{\mathrm{d}u}{\mathrm{d}x}(0) = \tilde{u}_l.$$

Using the centred finite difference approximation on the derivative gives

$$\tilde{u}_l = \frac{\mathrm{d}u}{\mathrm{d}x}(0) = \frac{\mathrm{d}u}{\mathrm{d}x}(x_0) \approx \frac{u(x_1) - u(x_{-1})}{2h} \approx \frac{U_1 - U_{-1}}{2h} \quad \Longrightarrow \quad \frac{U_1 - U_{-1}}{2h} \approx \tilde{u}_l$$

This approximation can be manipulated to provide an expression for the artificial point U_{-1} as

$$U_{-1} = U_1 - 2h\tilde{u}_l.$$

Replacing this into the equation Equation 9.1 will eliminate U_{-1} completely giving an equation in terms of U_0 and U_1 only, namely

$$\beta_0 U_0 + (\gamma_0 + \alpha_0) U_1 = f(x_0) + 2h\tilde{u}_l \alpha_0.$$

Therefore, another equation has been found which now completes the system of N equations in N unknowns. Thus the system of equations is:

$$\begin{aligned}
n = 0: &\quad \beta_0 U_0 + (\gamma_0 + \alpha_0) U_1 = f(x_0) + 2h\tilde{u}_l \alpha_0 \\
n = 1: &\quad \alpha_1 U_0 + \beta_1 U_1 + \gamma_1 U_2 = f(x_1) \\
n = 2: &\quad \alpha_2 U_1 + \beta_2 U_2 + \gamma_2 U_3 = f(x_2) \\
&\quad \vdots \\
n = N-1: &\quad \alpha_{N-1} U_{N-2} + \beta_{N-1} U_{N-1} = f(x_{N-1}) - \gamma_{N-1} u_L.
\end{aligned}$$

This can be written in matrix form as $A\boldsymbol{U} = \boldsymbol{g}$ where

$$\underbrace{\begin{pmatrix}
\beta_0 & \gamma_0 + \alpha_0 & 0 & \dots & 0 & 0 & 0 \\
\alpha_1 & \beta_1 & \gamma_1 & \dots & 0 & 0 & 0 \\
0 & \alpha_2 & \beta_2 & \dots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \dots & \beta_{N-3} & \gamma_{N-3} & 0 \\
0 & 0 & 0 & \dots & \alpha_{N-2} & \beta_{N-2} & \gamma_{N-2} \\
0 & 0 & 0 & \dots & 0 & \alpha_{N-1} & \beta_{N-1}
\end{pmatrix}}_{A}
\underbrace{\begin{pmatrix} U_0 \\ U_1 \\ U_2 \\ \vdots \\ U_{N-3} \\ U_{N-2} \\ U_{N-1} \end{pmatrix}}_{\boldsymbol{U}} =$$

$$\underbrace{\begin{pmatrix} f(x_0) + 2h\alpha_0 \tilde{u}_l \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{N-3}) \\ f(x_{N-2}) \\ f(x_{N-1}) - \gamma_{N-1} u_r \end{pmatrix}}_{\boldsymbol{g}}.$$

This can once again be solved on MATLAB using `U=inv(A)*g` or `U=A\g`.

If, on the other hand, the mixed conditions were instead

$$u(0) = u_l \quad \text{and} \quad \frac{\mathrm{d}u}{\mathrm{d}x}(L) = \tilde{u}_r,$$

then the artificial point will be located at $x = x_{N+1}$ but the same procedure can be done

give the matrix system $\boldsymbol{AU} = \boldsymbol{g}$ where

$$\underbrace{\begin{pmatrix} \beta_1 & \gamma_1 & 0 & \dots & 0 & 0 & 0 \\ \alpha_2 & \beta_2 & \gamma_2 & \dots & 0 & 0 & 0 \\ 0 & \alpha_3 & \beta_3 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \beta_{N-2} & \gamma_{N-2} & 0 \\ 0 & 0 & 0 & \dots & \alpha_{N-1} & \beta_{N-1} & \gamma_{N-1} \\ 0 & 0 & 0 & \dots & 0 & \alpha_N + \gamma_N & \beta_N \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ \vdots \\ U_{N-2} \\ U_{N-1} \\ U_N \end{pmatrix}}_{\boldsymbol{U}} =$$

$$\underbrace{\begin{pmatrix} f(x_1) - \alpha_1 u_l \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_{N-2}) \\ f(x_{N-1}) \\ f(x_N) - 2h\gamma_N \tilde{u}_r \end{pmatrix}}_{\boldsymbol{g}}.$$

Mixed Value Problem

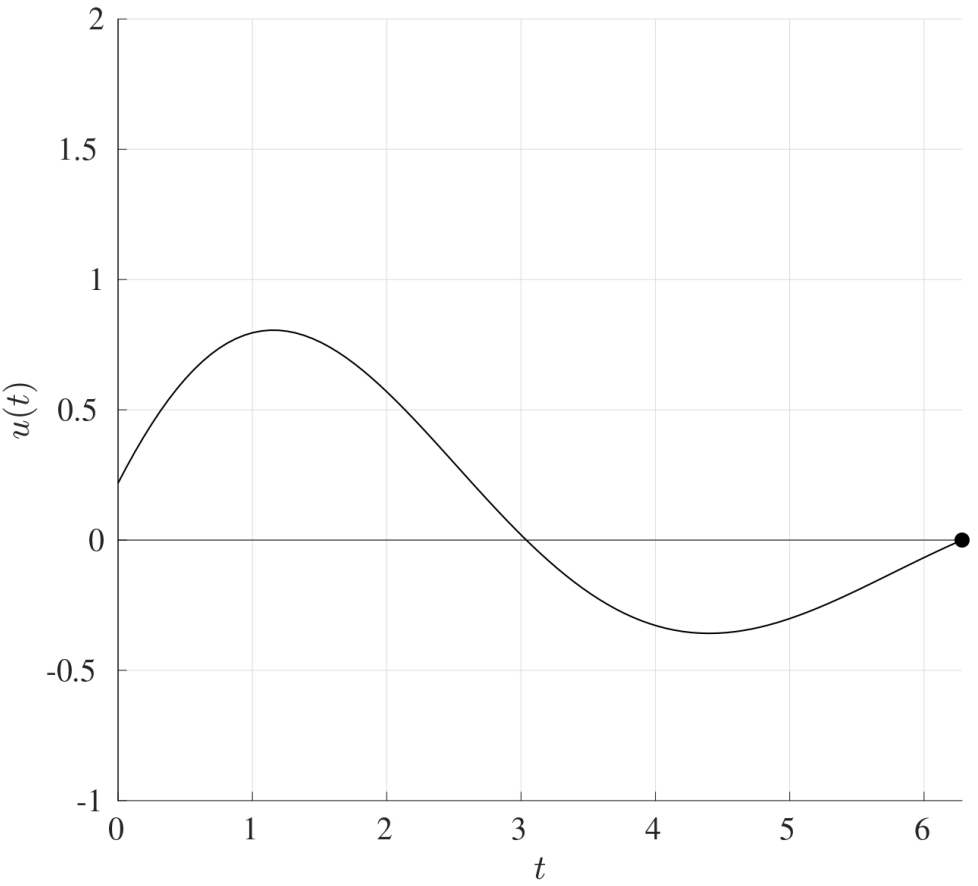
Consider the differential equation for a damped harmonic oscillator

$$\frac{\mathrm{d}^2 u}{\mathrm{d}t^2} + 0.5\frac{\mathrm{d}u}{\mathrm{d}t} + u = 0 \quad \text{for} \quad 0 < t < 2\pi$$

with the mixed conditions

$$\frac{\mathrm{d}u}{\mathrm{d}t}(0) = 1 \quad \text{and} \quad u(2\pi) = 0.$$

This MVP is to determine the trajectory of the mass if the launching speed at the start is 1, which is $\frac{\mathrm{d}u}{\mathrm{d}t}(0) = 1$, and after 2π seconds, the mass reaches its equilibrium state, which is $u(2\pi) = 0$. Notice that there is no restriction on the starting location, only the starting speed, so the mass can start anywhere as long as it is launched with a velocity 1.



The starting location here happens to be at 0.2188 but that is no restricted by the mixed conditions as long as the gradient at the start is 1.

10 Symmetric Boundary Conditions

The use of symmetric boundary conditions arises in many cases where conditions at the ends are not known explicitly but they are related. For instance, consider the ODE representing the conduction problem

$$-k_f \frac{\mathrm{d}^2 T}{\mathrm{d}x^2} = q_{gen}(x) \quad \text{in} \quad -L < x < L$$

where k_f is the material's conductivity and q_{gen} is the heat transfer. Symmetric boundary conditions can be imposed as

$$T(-L) = T(L) \quad \text{and} \quad k_f \frac{\mathrm{d}T}{\mathrm{d}x}(L) = \alpha(T(L) - T_{air})$$

for some constant α. This problem can be interpreted as an insulated metal rod of length $2L$ that has been heated all the way through and then as it cools, it loses heat equally from both ends (which is the condition $T(-L) = T(L)$), and that this heat loss at L is proportional to the temperature gradient between the rod and the air (which is the second condition $k_f \frac{\mathrm{d}T}{\mathrm{d}x}(L) = \alpha(T(L) - T_{air})$). The issue with this type of problems is that the temperature at both boundaries are not explicitly known, but it is known that they are the same.

10.1 Finite Difference Method for Symmetric Boundary Value Problems

This problem can be tackled in a very similar way to BVPs and MVPs. Consider the differential equation

$$a(x)\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} + b(x)\frac{\mathrm{d}u}{\mathrm{d}x} + c(x)u = f(x) \quad \text{with} \quad -L < x < L.$$

The interval $[-L, L]$ will be split into N equally sized sections each of width $h = \frac{2L}{N}$ and the grid points are labelled $x_n = -L + nh$ for $n = 0, 1, 2, \ldots, N$. This differential equation can be discretised using the centred difference approximation (just as in Section 8.2) to give

$$\alpha_n U_{n-1} + \beta_n U_n + \gamma_n U_{n+1} = f(x_n) \quad \text{for} \quad n = 1, 2, \ldots, N-1$$

$$\text{where} \quad \alpha_n = \frac{a(x_n)}{h^2} - \frac{b(x_n)}{2h}, \quad \beta_n = -\frac{2a(x_n)}{h^2} + c(x_n), \quad \gamma_n = \frac{a(x_n)}{h^2} + \frac{b(x_n)}{2h}.$$

This gives a set of $N-1$ equations in $N+1$ unknowns, namely $U_0, U_1, U_2, \ldots, U_N$. In this case, neither U_0 nor U_N are explicitly known, therefore none of the unknowns can be eliminated from the boundary conditions *per se*.

Suppose the given conditions are

$$u(-L) = u(L) \quad \text{and} \quad \frac{\mathrm{d}u}{\mathrm{d}x}(L) = pu(L) + q$$

where p and q are some constants. The first condition is the symmetric boundary condition which represents the fact that the value of the unknown solution u at both ends is the same, then $U_0 = U_N$, even though neither is explicitly known. The term U_0 can be eliminated since determining U_N automatically determines U_0, this reduces the number of unknowns to N.

Consider the discretisation at $n = 1$, namely

$$\alpha_1 U_0 + \beta_1 U_1 + \gamma_1 U_2 = f(x_1),$$

since $U_0 = U_N$, this can be rewritten in terms of U_N instead as

$$\beta_1 U_1 + \gamma_1 U_2 + \alpha_1 U_N = f(x_1).$$

The discretised form of the differential equation at $n = N$ is

$$\alpha_N U_{N-1} + \beta_N U_N + \gamma_N U_{N+1} = f(x_N). \tag{10.1}$$

Just as in the case with the MVPs, an artificial point U_{N+1} is introduced which is the solution approximated at the point $x = x_{N+1} = L + h$ which is beyond the computational domain.

To find an expression for U_{N+1}, first consider the second condition

$$\frac{\mathrm{d}u}{\mathrm{d}x}(x_N) = \frac{\mathrm{d}u}{\mathrm{d}x}(L) pu(L) + q \approx pU_N + q.$$

The LHS can be rewritten in terms of its centred differencing approximation as

$$\frac{\mathrm{d}u}{\mathrm{d}x}(x_N) \approx \frac{u(x_{N+1}) - u(x_{N-1})}{2h} \approx \frac{U_{N+1} - U_{N-1}}{2h}.$$

Combining these two can give an expression for U_{N+1} as:

$$\frac{U_{N+1} - U_{N-1}}{2h} \approx pU_N + q \quad \Longrightarrow \quad U_{N+1} = U_{N-1} + 2hpU_N + 2hq.$$

Replacing this into Equation 10.1 gives

$$(\alpha_N + \gamma_N)U_{N-1} + (\beta_N + 2hp\gamma_N)U_N = f(x_N) - 2hq\gamma_N,$$

thus providing the last equation to complete the set. Finally, this system can be written in matrix form as $\boldsymbol{AU} = \boldsymbol{g}$ where

$$\underbrace{\begin{pmatrix} \beta_1 & \gamma_1 & 0 & \dots & 0 & 0 & \alpha_1 \\ \alpha_2 & \beta_2 & \gamma_2 & \dots & 0 & 0 & 0 \\ 0 & \alpha_3 & \beta_3 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \beta_{N-2} & \gamma_{N-2} & 0 \\ 0 & 0 & 0 & \dots & \alpha_{N-1} & \beta_{N-1} & \gamma_{N-1} \\ 0 & 0 & 0 & \dots & 0 & \alpha_N + \gamma_N & \beta_N + 2hp\gamma_N \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ \vdots \\ U_{N-2} \\ U_{N-1} \\ U_N \end{pmatrix}}_{\boldsymbol{U}} = \underbrace{\begin{pmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_{N-2}) \\ f(x_{N-1}) \\ f(x_N) - 2hq\gamma_N \end{pmatrix}}_{\boldsymbol{g}}.$$

This can then be solved in MATLAB but bearing in mind that $U_0 = U_N$ which determines the function U at $-L$ and L.

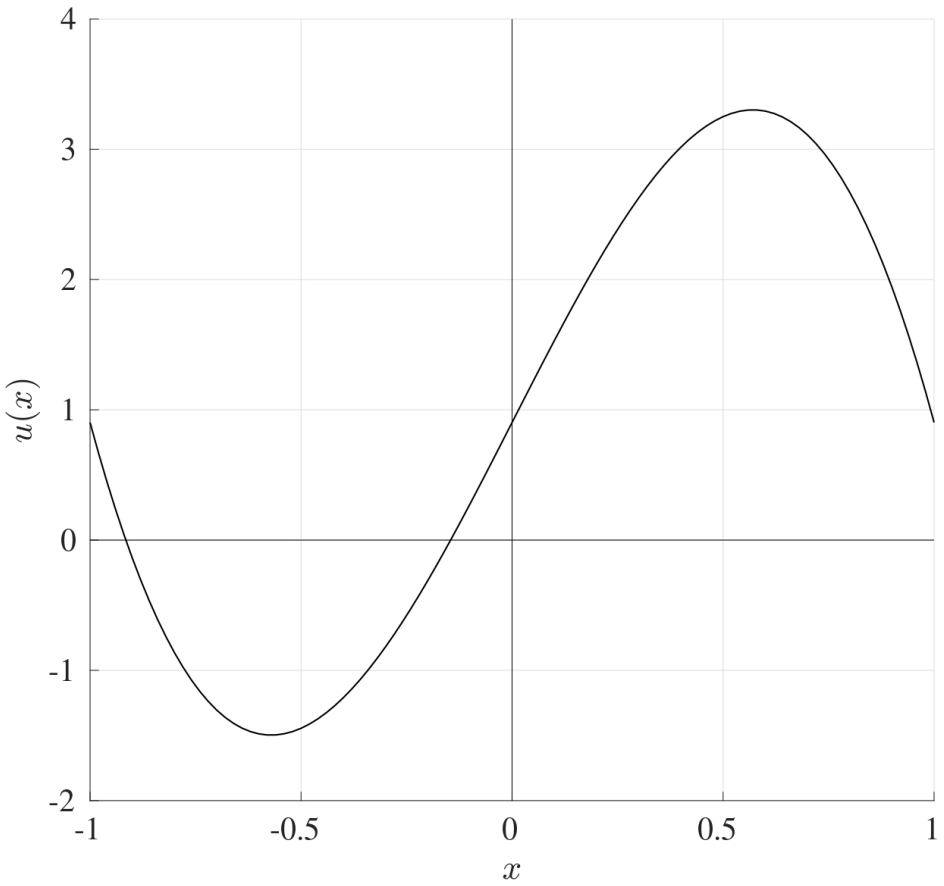
Symmetric Boundary Value Problem

Consider the conduction problem

$$-\frac{\mathrm{d}^2T}{\mathrm{d}x^2} = 40\sin(x) \quad \text{in} \quad -1 < x < 1$$

with the conditions

$$T(-1) = T(1) \quad \text{and} \quad \frac{\mathrm{d}T}{\mathrm{d}x}(1) = \frac{1}{2}(T(1) - 25).$$


4
3
2
1
0
-1
-2
$u(x)$
-1
-0.5
0
0.5
1
x

Part IV

Solving Partial Differential Equations

11 Heat Equation

Ordinary differential equations have been the main focus of this course so far but this will now be extended to *partial differential equations*. The differential equations that will be studied here are the *1-Dimensional Heat (or Diffusion) Equation* and the *1-Dimensional Advection (or Convection) Equation*.

The 1-dimensional *heat* (or *diffusion*) *equation* is a partial differential equation that represents the heat transfer across a rod and is given by

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \quad \text{with} \quad 0 < x < L \quad \text{and} \quad t > 0$$

where $u = u(x,t)$ is the temperature at location x at time t and α is the thermal diffusivity[1]. This equation represents the flow of heat along the length of a rod of length L.

This partial differential equation has three derivatives in total, two derivatives in x and one derivative in t, this means that three conditions are needed, two on x and one on t:

- $u(x,0) = u_{init}(x)$ for $x \in [0,L]$: Initial heat distribution across the rod;
- $u(0,t) = u_l(t)$ for $t > 0$: The temperature at the left end of the rod;
- $u(L,t) = u_r(t)$ for $t > 0$: The temperature at the right end of the rod.

This set of conditions along with the differential equation are known collectively as an ***Initial-Boundary Value Problem*** and can be solved using the ***Method of Lines***.

11.1 The Method of Lines for the Heat Equation

The outline of the method of lines for the heat equation is as follows:

1. Divide the spatial interval $[0,L]$ into N_x equally sized sections and label the points as $x_0, x_1, x_2, \ldots, x_N$ where $x_n = nh$ and the spatial interval width is $h_x = \frac{L}{N}$.

[1]The thermal diffusivity will always be regarded as a constant and usually takes the form $\alpha = \frac{k}{\rho Cp}$ where k is the thermal conductivity, ρ is the density of the material and Cp is the specific heat capacity.

2. **Left Hand Side**: For each point x_n, define the approximation $U_n(t) \approx u(x_n, t)$. Therefore the left hand side of the heat equation can be written as

$$\frac{\partial u}{\partial t}(x_n, t) \approx \frac{\mathrm{d}U_n}{\mathrm{d}t}(t)$$

and this holds for $n = 1, 2, \ldots, N-1$ since $U_0(t) \approx u(0,t) = u_l(t)$ and $U_N(t) \approx u(L,t) = u_r(t)$ are already known from the boundary conditions. Notice that the derivative of U_n is an ordinary derivative since U_n is a function of t only.

3. **Right Hand Side**: Use the finite difference approximation to approximate the *spatial* derivative in the differential equation. Here, the centred difference approximation for the second derivative will be used, namely

$$\frac{\partial^2 u}{\partial x^2}(x_n, t) \approx \frac{U_{n+1}(t) - 2U_n(t) + U_{n-1}(t)}{h_x^2}.$$

Therefore the right hand side of the heat equation will become

$$\alpha \frac{\partial^2 u}{\partial x^2}(x_n, t) \approx \frac{\alpha}{h_x^2}\left[U_{n-1}(t) - 2U_n(t) + U_{n+1}(t)\right].$$

This holds for $n = 1, 2, \ldots, N-1$ bearing in mind, once again, that $U_0(t) \approx u(0,t) = u_l(t)$ and $U_N(t) \approx u(L,t) = u_r(t)$ are known beforehand.

4. These can be combined to give the discretised form of the heat equation

$$\frac{\mathrm{d}U_n}{\mathrm{d}t} = \frac{\alpha}{h_x^2}\left[U_{n-1} - 2U_n + U_{n+1}\right]$$

for all $n = 1, 2, \ldots, N-1$ where $U_n = U_n(t)$. This means that the partial differential equation has been split into $N-1$ ordinary differential equations.

5. This entire system of $N-1$ equations can now be written in matrix form as $\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = A\boldsymbol{U} + \boldsymbol{b}$ where

$$\underbrace{\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} U_1(t) \\ U_2(t) \\ U_3(t) \\ \vdots \\ U_{N-3}(t) \\ U_{N-2}(t) \\ U_{N-1}(t) \end{pmatrix}}_{\boldsymbol{U}} = \underbrace{\frac{\alpha}{h_x^2}\begin{pmatrix} -2 & 1 & 0 & \ldots & 0 & 0 & 0 \\ 1 & -2 & 1 & \ldots & 0 & 0 & 0 \\ 0 & 1 & -2 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & -2 & 1 & 0 \\ 0 & 0 & 0 & \ldots & 1 & -2 & 1 \\ 0 & 0 & 0 & \ldots & 0 & 1 & -2 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} U_1(t) \\ U_2(t) \\ U_3(t) \\ \vdots \\ U_{N-3}(t) \\ U_{N-2}(t) \\ U_{N-1}(t) \end{pmatrix}}_{\boldsymbol{U}}$$

$$+ \underbrace{\frac{\alpha}{h_x^2}\begin{pmatrix} u_l(t) \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ u_r(t) \end{pmatrix}}_{\boldsymbol{b}}$$

subject to the *initial condition*

$$\boldsymbol{U}_0 = \begin{pmatrix} U_1(0) \\ U_2(0) \\ U_3(0) \\ \vdots \\ U_{N-3}(0) \\ U_{N-2}(0) \\ U_{N-1}(0) \end{pmatrix} \approx \begin{pmatrix} u(x_1, 0) \\ u(x_2, 0) \\ u(x_3, 0) \\ \vdots \\ u(x_{N-3}, 0) \\ u(x_{N-2}, 0) \\ u(x_{N-1}, 0) \end{pmatrix} = \begin{pmatrix} u_{init}(x_1) \\ u_{init}(x_2) \\ u_{init}(x_3) \\ \vdots \\ u_{init}(x_{N-3}) \\ u_{init}(x_{N-2}) \\ u_{init}(x_{N-1}) \end{pmatrix}.$$

This system can now be solved using any of the IVP solvers with a temporal stepsize h_t.

In essence, the *Method of Lines* has converted a PDE into a set of ODEs using the same techniques as BVPs and will be solved in the same way as IVPs.

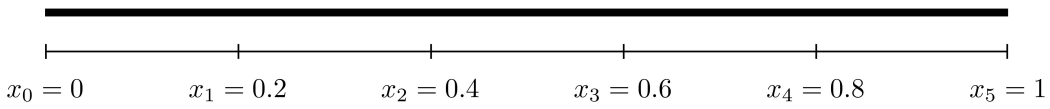
Heat Equation

Consider an iron rod (of thermal diffusivity $\alpha = 2.3 \times 10^{-5}$) of length 1 where the middle section of length 0.2 has been heated to a temperature of 1 while the rest is at 0. The ends of the rod have been kept at a constant temperature of 2. This system can be represented by the IBVP

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial t^2}, \quad x \in [0, 1], \quad t > 0$$

$$u(x, 0) = u_{init}(x) = \begin{cases} 0 & 0 \leq x < 0.4 \\ 1 & 0.4 \leq x < 0.6 \\ 0 & 0.6 \leq x \leq 1 \end{cases}, \quad u(0, t) = u_l(t) = 2, \quad u(L, t) = u_r(t) = 2.$$

First, divide the interval $[0, 1]$ into five equal sections (which will be of width $h_x = \frac{1-0}{5} = 0.2$).



This system can be discretised using the centred difference method and written in matrix form as $\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = A\boldsymbol{U} + \boldsymbol{b}$ where

$$\underbrace{\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} U_1(t) \\ U_2(t) \\ U_3(t) \\ U_4(t) \\ U_5(t) \end{pmatrix}}_{\boldsymbol{U}} = \underbrace{\frac{\alpha}{h^2}\begin{pmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} U_1(t) \\ U_2(t) \\ U_3(t) \\ U_4(t) \\ U_5(t) \end{pmatrix}}_{\boldsymbol{U}} + \underbrace{\frac{\alpha}{h^2}\begin{pmatrix} u_l(t) \\ 0 \\ 0 \\ 0 \\ u_r(t) \end{pmatrix}}_{\boldsymbol{b}}$$

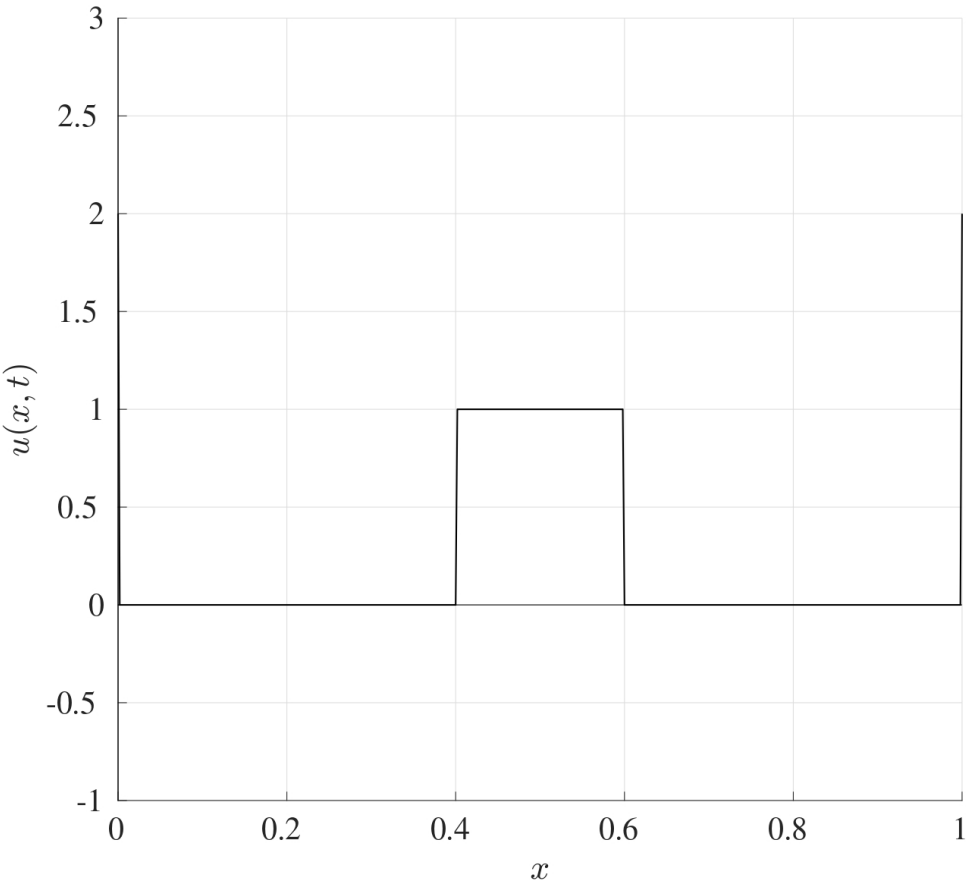
The differential equation

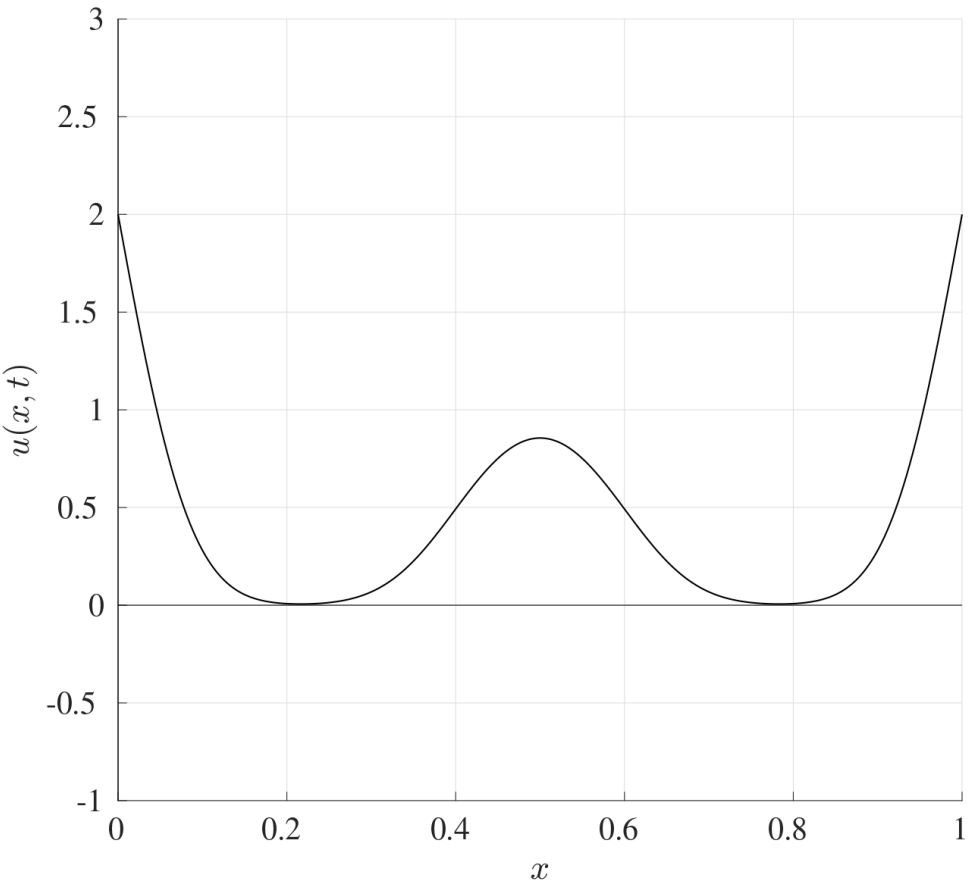
$$\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = A\boldsymbol{U} + \boldsymbol{b}$$

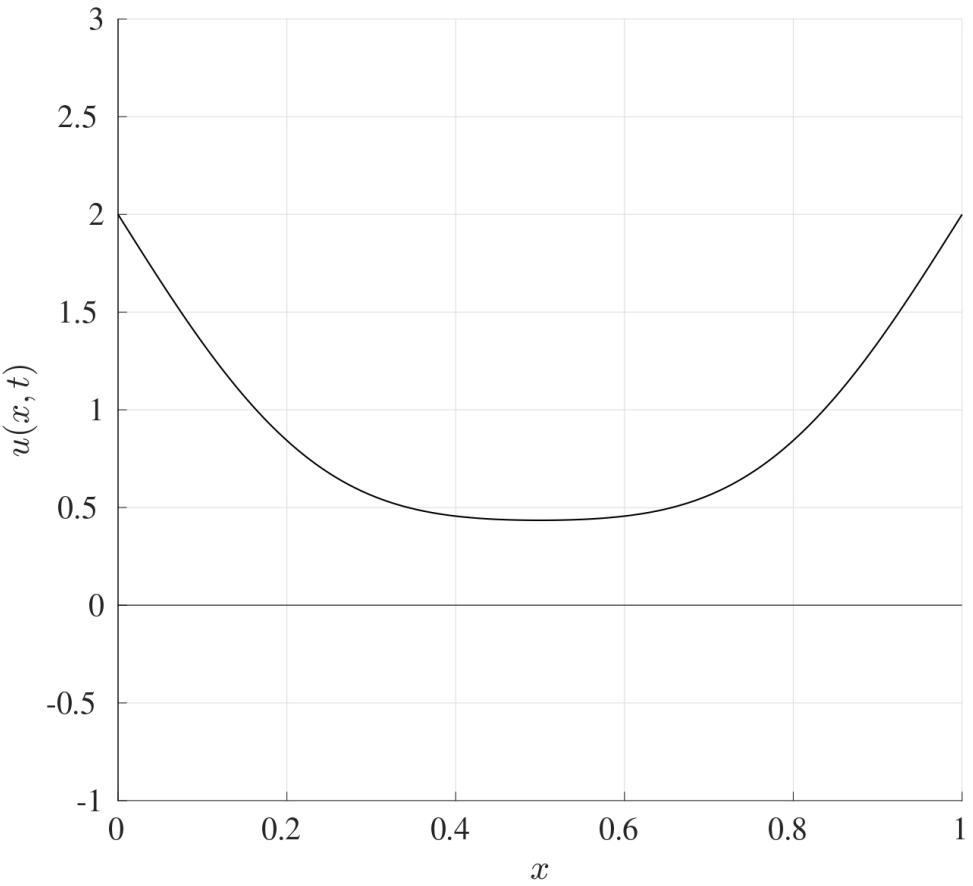
can be solved using the Euler method with the initial condition

$$\boldsymbol{U}(0) = \begin{pmatrix} u_{init}(x_1) \\ u_{init}(x_2) \\ u_{init}(x_3) \\ u_{init}(x_4) \\ u_{init}(x_5) \end{pmatrix}$$

subject to a time stepsize h_t. Below are the plots of the heat distribution at $t = 0, 100, 1000$ for $N_x = 500$ ($h_x = 0.002$) and $h_t = 0.02$ ($N_t = 50000$).




3
2.5
2
1.5
1
0.5
0
-0.5
-1
0
0.2
0.4
0.6
0.8
1
$u(x,t)$
x



At the beginning, the temperature at the ends is 2 and the middle section is at a temperature of 1. As time progresses, the heat evens out across the iron bar until eventually, the whole bar will be the same temperature.

11.2 Linear Advection Equation

The heat equation deals with heat transfer through diffusion throughout a material. Another way in which heat transfer can be achieved by advection (or convection) and this is given by

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} \quad \text{with} \quad 0 < x < L \quad \text{and} \quad t > 0$$

where $u = u(x,t)$ is the temperature at location x at time t and v is the flow speed.

This partial differential equation has two derivatives in total, one in x and one in t, this means that two conditions are needed, one spatial and one temporal:

- $u(x,0) = u_{init}(x)$ for $x \in [0,L]$: Initial heat distribution across the rod;

- $u(0,t) = u_l(t)$ for $t > 0$: The temperature at the left end of the rod.

Consider the PDE along with the initial condition only, namely $u(x,0) = u_{init}(x)$ for $x \in [0, L]$. The exact solution to this differential equation is given by

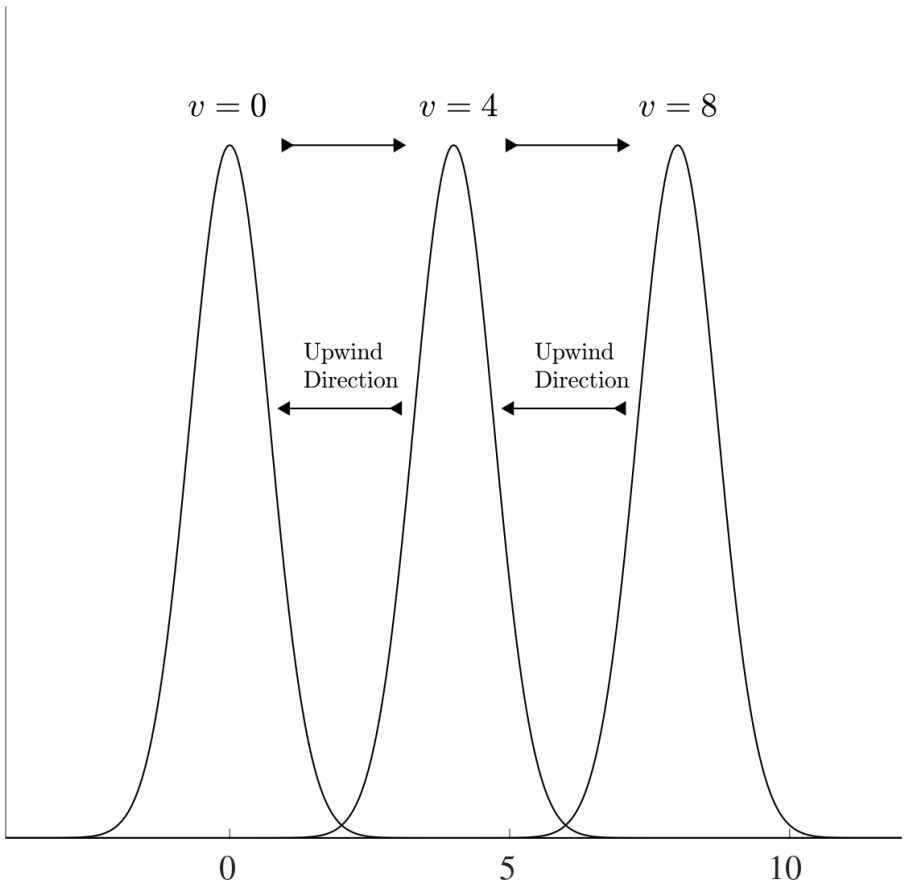
$$u(x,t) = u_{init}(x - vt),$$

this can be verified from the partial differential equation as follows:

$$\frac{\partial u}{\partial t} = -v\frac{\partial u}{\partial x} \quad \text{at} \quad u(x,t) = u_{init}(x - vt)$$

$$\text{LHS} = \frac{\partial}{\partial t}u(x,t) = \frac{\partial}{\partial t}\left(u_{init}(x - vt)\right) = -vu'_{init}(x - vt)$$
$$\text{RHS} = \frac{\partial}{\partial x}u(x,t) = \frac{\partial}{\partial x}\left(u_{init}(x - vt)\right) = -vu'_{init}(x - vt).$$

This means that if the initial heat profile takes the form of $u_{init}(x)$, then after time t, the profile will look exactly the same but shifted to the right by a distance vt.



The "information" moves from left to right so if the finite differences are to be used, the centred differencing approach would not be suitable since the information on the right is not known yet. Therefore the backwards differencing approximation will be the most suitable. This is known as an *upwind/upstream scheme* (i.e. against the direction of the wind/stream) if $v > 0$. Therefore using the convention $U_n(t) \approx u(x_n, t)$ where $x = x_n$ is the discretisation of the spatial points for $n = 0, 1, 2, \ldots, N$, the backward differencing approximation to the spatial derivative is

$$\frac{\partial u}{\partial x}(x_n, t) \approx \frac{\partial U_n}{\partial x} = \frac{U_n - U_{n-1}}{h_x}.$$

Therefore is discretised advection equation is

$$\frac{\mathrm{d}U_n}{\mathrm{d}t} = \frac{v}{h_x}\left(U_{n-1} - U_n\right) \quad \text{for} \quad n = 1, 2, \ldots, N$$

and this can be solved subject to the initial condition

$$u(x, 0) = u_{init}(x)$$

and boundary condition

$$u(0, t) = u_l(t)$$

to give the discretised set of equations in the form $\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = A\boldsymbol{U} + \boldsymbol{b}$ where

$$\underbrace{\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} U_1(t) \\ U_2(t) \\ U_3(t) \\ \vdots \\ U_{N-2}(t) \\ U_{N-1}(t) \\ U_N(t) \end{pmatrix}}_{\boldsymbol{U}} = \underbrace{\frac{v}{h_x}\begin{pmatrix} -1 & 0 & 0 & \ldots & 0 & 0 & 0 \\ 1 & -1 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 1 & -1 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & -1 & 0 & 0 \\ 0 & 0 & 0 & \ldots & 1 & -1 & 0 \\ 0 & 0 & 0 & \ldots & 0 & 1 & -1 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} U_1(t) \\ U_2(t) \\ U_3(t) \\ \vdots \\ U_{N-2}(t) \\ U_{N-1}(t) \\ U_N(t) \end{pmatrix}}_{\boldsymbol{U}}$$

$$+ \underbrace{\frac{v}{h_x}\begin{pmatrix} u_l(t) \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \end{pmatrix}}_{\boldsymbol{b}}$$

and the initial condition is

$$\boldsymbol{U}_0 = \begin{pmatrix} U_1(0) \\ U_2(0) \\ U_3(0) \\ \vdots \\ U_{N-3}(0) \\ U_{N-2}(0) \\ U_{N-1}(0) \end{pmatrix} \approx \begin{pmatrix} u(x_1, 0) \\ u(x_2, 0) \\ u(x_3, 0) \\ \vdots \\ u(x_{N-3}, 0) \\ u(x_{N-2}, 0) \\ u(x_{N-1}, 0) \end{pmatrix} = \begin{pmatrix} u_{init}(x_1) \\ u_{init}(x_2) \\ u_{init}(x_3) \\ \vdots \\ u_{init}(x_{N-3}) \\ u_{init}(x_{N-2}) \\ u_{init}(x_{N-1}) \end{pmatrix}$$

.

11.3 Convection-Diffusion Equation

The heat (or diffusion) equation dictates the spread of heat across a length of material while on the other hand, the advection (or convection) equation dictates the flow of heat in a certain direction. The combination of these two effects gives rise to the ***Convection-Diffusion Equation*** which takes the form

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} - v \frac{\partial u}{\partial x} \quad \text{with} \quad 0 < x < L, \quad t > 0.$$

Just as in the heat equation, this partial differential equation has three derivatives in total, two derivatives in x and one derivative in t, this means that three conditions are needed, two on x and one on t, these will be as follows:

- $u(x, 0) = u_{init}(x)$ for $x \in [0, L]$: Initial heat distribution across the rod;
- $u(0, t) = u_l(t)$ for $t > 0$: The temperature at the left end of the rod;
- $u(L, t) = u_r(t)$ for $t > 0$: The temperature at the right end of the rod.

In order to discretise this system, a finite difference approximation needs to be chosen first. The centred difference approximation was used for the heat equation and the backwards difference approximation for the advection. Here, the combination of both will be used. Even though this might initially seem like an inconsistency, but in fact, this will allow the system to present a distinct stable advantage as will be seen in the next section.

This system can be discretised in exactly the same way as before

$$\frac{\mathrm{d}U_n}{\mathrm{d}t}(t) = \frac{\alpha}{h_x^2}\left[U_{n-1}(t) - 2U_n(t) + U_{n+1}(t)\right] - \frac{v}{h_x}\left[U_n(t) + U_{n-1}(t)\right] \quad \text{for} \quad n = 1, 2, \ldots, N-1.$$

This system can be written in the form $\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = A\boldsymbol{U} + \boldsymbol{b}$ where

$$A = \frac{\alpha}{h_x^2}\begin{pmatrix} -2 & 1 & 0 & \ldots & 0 & 0 & 0 \\ 1 & -2 & 1 & \ldots & 0 & 0 & 0 \\ 0 & 1 & -2 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & -2 & 1 & 0 \\ 0 & 0 & 0 & \ldots & 1 & -2 & 1 \\ 0 & 0 & 0 & \ldots & 0 & 1 & -2 \end{pmatrix} + \frac{v}{h_x}\begin{pmatrix} -1 & 0 & 0 & \ldots & 0 & 0 & 0 \\ 1 & -1 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 1 & -1 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & -1 & 0 & 0 \\ 0 & 0 & 0 & \ldots & 1 & -1 & 0 \\ 0 & 0 & 0 & \ldots & 0 & 1 & -1 \end{pmatrix}$$

$$\boldsymbol{U}\begin{pmatrix} U_1(t) \\ U_2(t) \\ \vdots \\ U_{N-2}(t) \\ U_{N-1}(t) \end{pmatrix}, \quad \boldsymbol{b} = \frac{\alpha}{h_x^2}\begin{pmatrix} u_l(t) \\ 0 \\ \vdots \\ 0 \\ u_r(t) \end{pmatrix} + \frac{v}{h_x}\begin{pmatrix} u_l(t) \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}.$$

and this system can be solved using an Euler iteration subject to the initial condition $u(x, 0) = u_{init}(x)$.

11.4 Asymptotic Stability

The method of lines is essentially a hybrid method that makes use of a combination between a finite difference approximation and the Euler method and is very effective at solving partial differential equations, as seen from solving the heat, advection and convection-diffusion equations. The derivation of the method of lines for the different methods builds on the very same principle and the codes can be adapted quite easily. One main issue that arises here is the choice for the stepsizes for both the spatial and temporal discretisations, i.e. the choice of h_t and h_x respectively. When both methods are combined, there needs to be a restriction on both stepsizes.

The first issue that needs to be addressed is the asymptotic stability of the heat equation and the advection equation. For arbitrarily large matrices, it may not be simple to determine if all the eigenvalues are negative since it may be computationally restrictive to do so. However, a result can be used to see if all the eigenvalues are negative without explicitly calculating them.

Theorem 11.1 (Gershgorin Circle Theorem)**.** *Let A be an $N \times N$ given by*

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1N} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2N} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & a_{N3} & \dots & a_{NN} \end{pmatrix}.$$

On the complex plane, consider N closed discs, each centred at the locations a_{ii} for $i = 1, 2, \dots, n$ (the diagonal terms) where the disc centred at a_{ii} has a radius R_i where

$$R_i = \sum_{j \neq i} |a_{ij}|.$$

Then all the eigenvalues of the matrix A will have to lie in at least one of these discs. In other words, every eigenvalues of A satisfies

$$|\lambda - a_{ii}| \leq R_i \quad \text{for at least one} \quad i = 1, 2, \dots, n.$$

Gershgorin Circle Theorem Exapmple

Consider the matrix

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

Following the steps of the theorem:

1. Indicate the locations of the diagonal terms (namely $-1, 5, 0, -6, 7$) on the complex

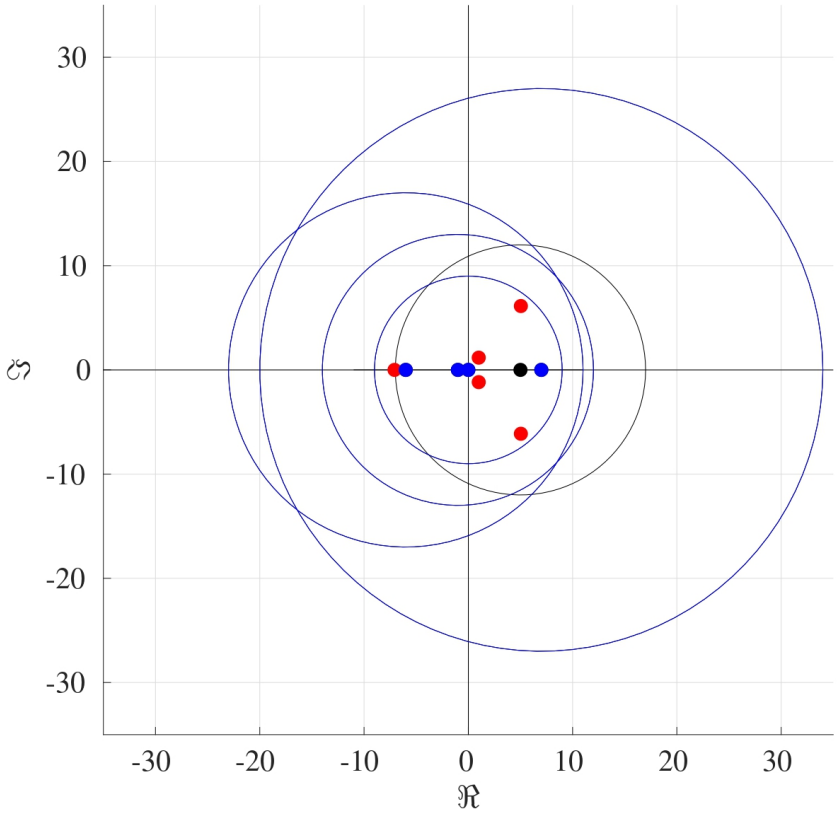
plane.

2. Find the radii R_i which are equal to the row sum of the absolute terms *without* the diagonal terms, in other words,

$$\text{abs}(A) = \begin{pmatrix} 1 & 3 & 4 & 2 & 4 \\ 0 & 5 & 4 & 7 & 1 \\ 4 & 2 & 0 & 3 & 0 \\ 6 & 6 & 4 & 6 & 1 \\ 7 & 4 & 7 & 9 & 7 \end{pmatrix} \begin{matrix} \rightarrow 3+4+2+4=13 \rightarrow R_1 \\ \rightarrow 0+4+7+1=12 \rightarrow R_2 \\ \rightarrow 4+2+3+0=9 \rightarrow R_3 \\ \rightarrow 6+6+4+1=17 \rightarrow R_4 \\ \rightarrow 7+4+7+9=27 \rightarrow R_5 \end{matrix}$$

3. Draw a circle around $a_{11} = -1$ with radius $R_1 = 13$, a circle around $a_{22} = 5$ with radius $R_2 = 12$ and so on.

4. All the eigenvalues of the matrix A must lie in at least one of the circles indicated. Indeed, the following figure shows the diagonal terms each with circles around them with the appropriate radius. The eigenvalues are given in red and the blue circles are those which contain all said eigenvalues.

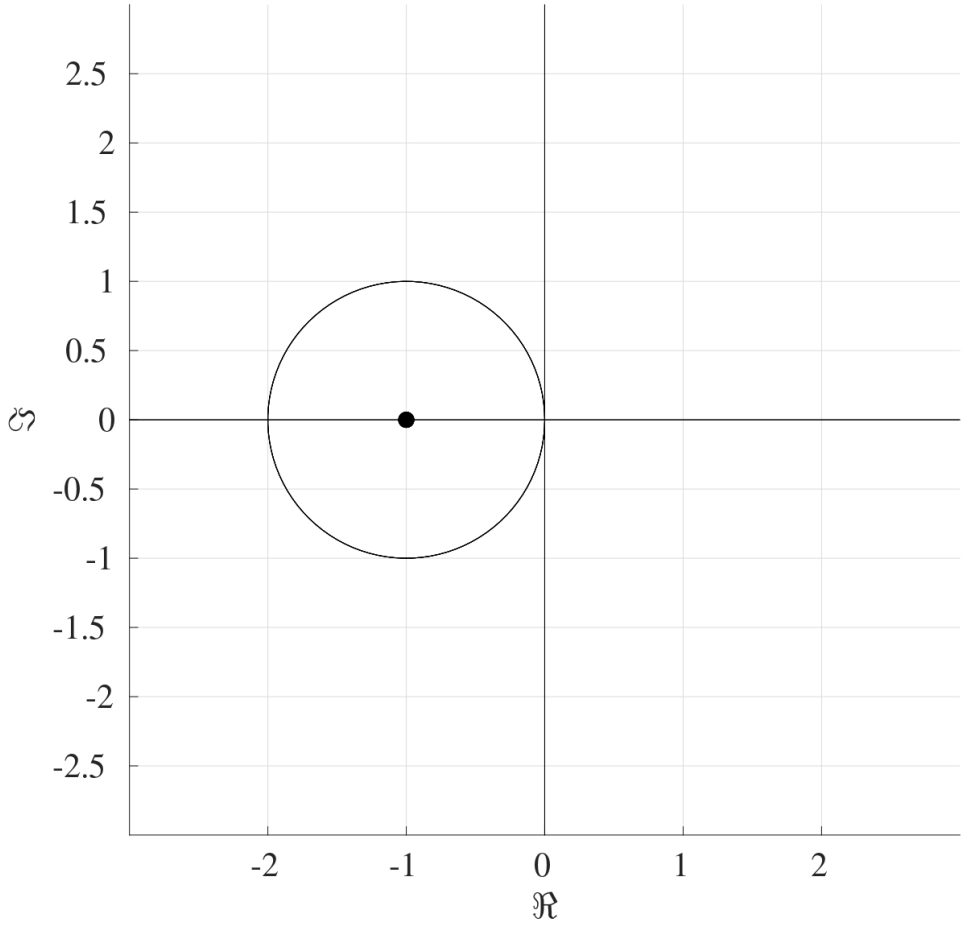


11.4.1 Stability of the Euler Method for the Advection Equation

Consider the matrix A_2 of size $N \times N$ from the advection equation

$$A_2 = \begin{pmatrix} -1 & 0 & \dots & 0 & 0 \\ 1 & -1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & -1 & 0 \\ 0 & 0 & \dots & 1 & -1 \end{pmatrix}.$$

Following the steps of the Gershgorin theorem, the centres of all the circles on the complex plane will be located at the diagonal terms, all of which are -1. The radii of these circles are the row sums of the matrix A_2 without the diagonal terms, which means that all the radii will be 1. The figure below shows the circle that results on the complex plane. Therefore regardless of what the eigenvalues might be, it is known that they will always have negative real parts and therefore the advection matrix forms an asymptotically stable system.



Since the advection equation is asymptotically stable, a bound for the temporal stepsize needs to be found. Consider the advection equation after the discretisation $\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = \boldsymbol{AU} + \boldsymbol{b}$

where $A = \frac{v}{h_x} A_2$. The Euler method is numerically stable if the time step h_t satisfies

$$\|\mathcal{I} + h_t A\|_\infty \leq 1.$$

First calculate $\mathcal{I} + h_t A$:

$$\mathcal{I} + h_t A = \mathcal{I} + \frac{v h_t}{h_x} A_2 = \begin{pmatrix} 1-\tilde{v} & 0 & 0 & \dots & 0 & 0 & 0 \\ \tilde{v} & 1-\tilde{v} & 0 & \dots & 0 & 0 & 0 \\ 0 & \tilde{v} & 1-\tilde{v} & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & \tilde{v} & 1-\tilde{v} \end{pmatrix}.$$

where $\tilde{v} = \frac{v h_t}{h_x}$. Now taking the absolute value of all the terms and taking the row sums gives:

$$\text{abs}\left(\mathcal{I} + \frac{v h_t}{h_x} A_2\right) = \begin{pmatrix} |1-\tilde{v}| & 0 & 0 & \dots & 0 & 0 & 0 \\ \tilde{v} & |1-\tilde{v}| & 0 & \dots & 0 & 0 & 0 \\ 0 & \tilde{v} & |1-\tilde{v}| & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & \tilde{v} & |1-\tilde{v}| \end{pmatrix} \begin{matrix} \rightarrow \ |1-\tilde{v}| \\ \rightarrow \tilde{v} + |1-\tilde{v}| \\ \rightarrow \tilde{v} + |1-\tilde{v}| \\ \vdots \qquad \vdots \\ \rightarrow \tilde{v} + |1-\tilde{v}| \end{matrix}.$$

The row sums of the absolute terms of this matrix are

$$a = |1-\tilde{v}| \quad \text{and} \quad b = |1-\tilde{v}| + \tilde{v}.$$

Since it is assumed that $v > 0$, then $b > a$ therefore, $\|\mathcal{I} + h_t A\|_\infty = b = |1-\tilde{v}| + \tilde{v}$. Consider the two cases when $1 - \tilde{v} > 0$ and $1 - \tilde{v} < 0.\$\ 1$. If $1 - \tilde{v} > 0$, then $0 < \tilde{v} < 1$:

$$\|\mathcal{I} + h_t A\|_\infty = |1-\tilde{v}| + \tilde{v} = 1 - \tilde{v} + \tilde{v} = 1.$$

Therefore if $1 - \tilde{v} > 0$, then $\|\mathcal{I} + h_t A\|_\infty \leq 1$.

2. If $1 - \tilde{v} < 0$, then $\tilde{v} > 1$:

$$\|\mathcal{I} + h_t A\|_\infty = |1-\tilde{v}| + \tilde{v} = \tilde{v} - 1 + \tilde{v} = 2\tilde{v} - 1,$$

therefore in this case, if $\|\mathcal{I} + h_t A\|_\infty$ needs to be less than or equal to 1, then

$$\|\mathcal{I} + h_t A\|_\infty \leq 1 \quad \Longrightarrow \quad 2\tilde{v} - 1 \leq 1 \quad \Longrightarrow \quad \tilde{v} \leq 1$$

which contradicts with the assumption that $\tilde{v} > 1$.

Therefore, the Euler method will produce a convergent solution if

$$\tilde{v} < 1 \quad \Longrightarrow \quad v\frac{h_t}{h_x} < 1.$$

In terms of number of spatial and temporal points N_x and N_t respectively, this restriction would be

$$v\frac{t_f - t_0}{L - x_0}\frac{N_x}{N_t} < 1$$

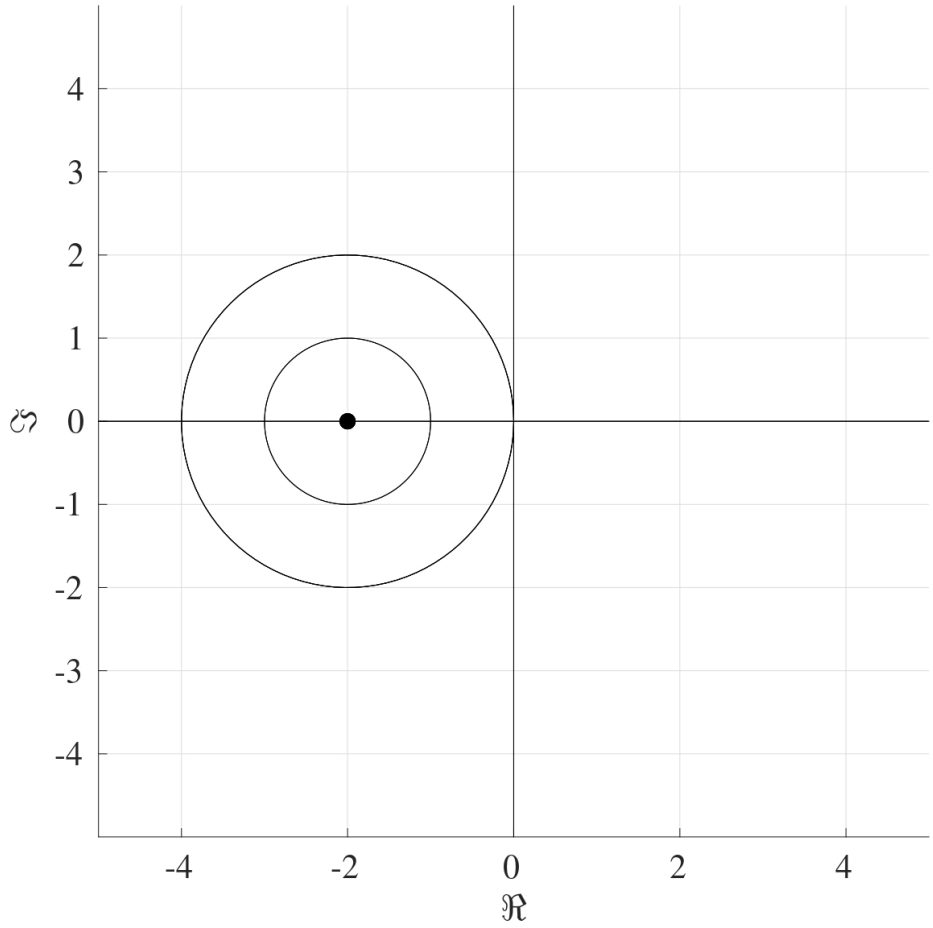
So for a fixed velocity v, if the time step h_t is to be halved, then the spatial step would also need to be halved as well.

11.4.2 Stability of the Euler Method for the Heat Equation

Consider the matrix A_1 of size $N \times N$ from the heat equation

$$A_1 = \begin{pmatrix} -2 & 1 & \dots & 0 & 0 \\ 1 & -2 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & -2 & 1 \\ 0 & 0 & \dots & 1 & -2 \end{pmatrix}.$$

The steps of the Gershgorin theorem can be followed to produce the following figure on the



complex plane.
Once again, this shows that all the eigenvalues will have negative real parts even though their explicit values are not known.

To determine the bound on the stepsize, consider the heat equation after the discretisation, which is $\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = A\boldsymbol{U} + \boldsymbol{b}$ where $A = \frac{\alpha}{h_x^2} A_1$. The Euler method is numerically stable if the time step h_t satisfies

$$\|\mathcal{I} + h_t A\|_\infty \leq 1.$$

First calculate $\mathcal{I} + h_t A$:

$$\mathcal{I} + h_t A = \mathcal{I} + \frac{\alpha h_t}{h_x^2} A_1 = \begin{pmatrix} 1-2\tilde{\alpha} & \tilde{\alpha} & 0 & \dots & 0 & 0 & 0 \\ \tilde{\alpha} & 1-2\tilde{\alpha} & \tilde{\alpha} & \dots & 0 & 0 & 0 \\ 0 & \tilde{\alpha} & 1-2\tilde{\alpha} & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & \tilde{\alpha} & 1-2\tilde{\alpha} \end{pmatrix}$$

where $\tilde{\alpha} = \frac{\alpha h_t}{h_x^2}$. Now taking the absolute value of all the terms and taking the row sums gives:

$$\text{abs}\left(\mathcal{I} + \frac{\alpha h_t}{h_x^2} A_1\right) = \begin{pmatrix} |1-2\tilde{\alpha}| & \tilde{\alpha} & 0 & \dots & 0 & 0 & 0 \\ \tilde{\alpha} & |1-2\tilde{\alpha}| & \tilde{\alpha} & \dots & 0 & 0 & 0 \\ 0 & \tilde{\alpha} & |1-2\tilde{\alpha}| & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & \tilde{\alpha} & |1-2\tilde{\alpha}| \end{pmatrix} \begin{matrix} \to \ \tilde{\alpha} + |1-2\tilde{\alpha}| \\ \to 2\tilde{\alpha} + |1-2\tilde{\alpha}| \\ \to 2\tilde{\alpha} + |1-2\tilde{\alpha}| \\ \vdots \\ \to \ \tilde{\alpha} + |1-2\tilde{\alpha}|. \end{matrix}$$

The row sums of the absolute terms of this matrix are

$$a = \tilde{\alpha} + |1-2\tilde{\alpha}| \quad \text{and} \quad b = 2\tilde{\alpha} + |1-2\tilde{\alpha}|.$$

Since $tilde\alpha > 0$, then $b > a$ and therefore, $\|\mathcal{I} + h_t A\|_\infty = b = 2\tilde{\alpha} + |1-2\tilde{\alpha}|$. Consider the two cases $1 - 2\tilde{\alpha} > 0$ and $1 - 2\tilde{\alpha} < 0$.

1. If $1 - 2\tilde{\alpha} > 0$, then $0 < \tilde{\alpha} < \frac{1}{2}$:

   $$\|\mathcal{I} + h_t A\|_\infty = |1-2\tilde{\alpha}| + 2\tilde{\alpha} = 1 - 2\tilde{\alpha} + 2\tilde{\alpha} = 1,$$

   therefore $\|\mathcal{I} + h_t A\|_\infty \leq 1$.

2. If $1 - 2\tilde{\alpha} < 0$, then $\tilde{\alpha} > \frac{1}{2}$:

   $$\|\mathcal{I} + h_t A\|_\infty = |1-2\tilde{\alpha}| + 2\tilde{\alpha} = 2\tilde{\alpha} - 1 + 2\tilde{\alpha} = 4\tilde{\alpha} - 1,$$

   therefore in this case, if $\|\mathcal{I} + h_t A\|_\infty$ needs to be less than or equal to 1, then

   $$\|\mathcal{I} + h_t A\|_\infty \leq 1 \quad \Longrightarrow \quad 4\tilde{\alpha} - 1 \leq 1 \quad \Longrightarrow \quad \tilde{\alpha} \leq \frac{1}{2}$$

   which contradicts with the assumption that $\tilde{\alpha} > \frac{1}{2}$.

This means that the Euler method produces a stable convergent solution if

$$\tilde{\alpha} < \frac{1}{2} \quad \Longrightarrow \quad \alpha \frac{h_t}{h_x^2} < \frac{1}{2}.$$

In terms of number of spatial and temporal points N_x and N_t respectively, this restriction would be

$$2\alpha \frac{t_f - t_0}{(L - x_0)^2} \frac{N_x^2}{N_t} < 1$$

So for a fixed diffusivity α, if the time step h_t is to be halved, then the spatial step would should be quartered.

11.5 Stability of the Convection-Diffusion Equation

Now that it has been established that both the heat and advection equations are asymptotically stable and the stepsize bounds have been found, it is time to combine both cases to tackle the convection-diffusion equation.

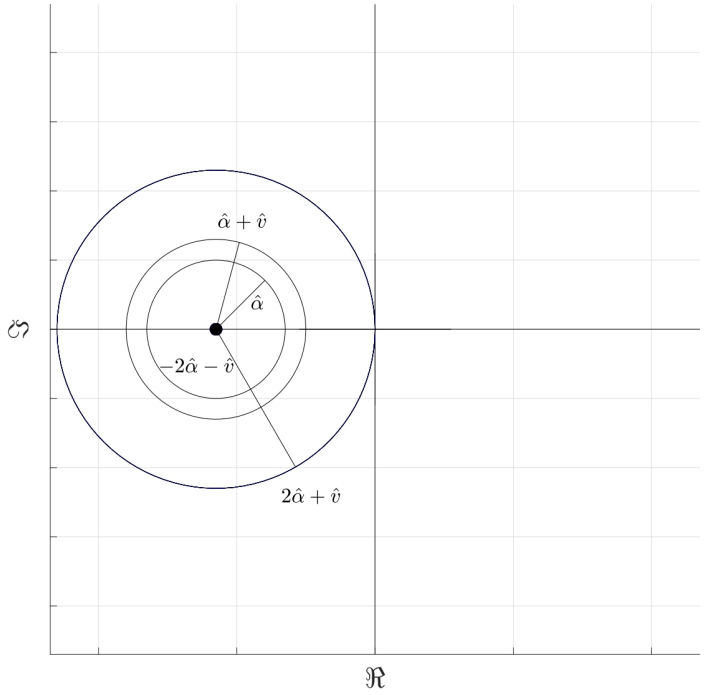
When discretised, the convection-diffusion equation can be written as $\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = \boldsymbol{AU} + \boldsymbol{b}$ where the matrix A is given by

$$A = \frac{\alpha}{h_x^2}\begin{pmatrix} -2 & 1 & \dots & 0 & 0 \\ 1 & -2 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & -2 & 1 \\ 0 & 0 & \dots & 1 & -2 \end{pmatrix} + \frac{v}{h_x}\begin{pmatrix} -1 & 0 & \dots & 0 & 0 \\ 1 & -1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & -1 & 0 \\ 0 & 0 & \dots & 1 & -1 \end{pmatrix}.$$

The Gershgorin theorem can be applied to the matrix A to show that all the eigenvectors have negative real parts. Indeed,

$$A = \begin{pmatrix} -2\hat{\alpha} - \hat{v} & \hat{\alpha} & 0 & \dots & 0 & 0 & 0 \\ \hat{\alpha} + \hat{v} & -2\hat{\alpha} - \hat{v} & \hat{\alpha} & \dots & 0 & 0 & 0 \\ 0 & \hat{\alpha} + \hat{v} & -2\hat{\alpha} - \hat{v} & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & \hat{\alpha} + \hat{v} & -2\hat{\alpha} - \hat{v} \end{pmatrix}.$$

where $\hat{\alpha} = \frac{\alpha}{h_x^2}$ and $\hat{v} = \frac{v}{h_x}$. By the Gershgorin theorem, the centres of the circles will be located at the diagonal terms, namely at $-2\hat{\alpha} - \hat{v}$ with the radii $\hat{\alpha}$, $\hat{\alpha} + \hat{v}$ and $2\hat{\alpha} + \hat{v}$. The largest radius is $2\hat{\alpha} + \hat{v}$ which means that all the eigenvalues will be negative as shown below. Therefore the convection-diffusion equation is asymptotically stable.



To find the bound for the stepsizes, consider the convection-diffusion equation after the discretisation $\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = A\boldsymbol{U} + \boldsymbol{b}$ where $A = \frac{\alpha}{h_x^2}A_1 + \frac{v}{h_x}A_2$. The Euler method is numerically stable if the time step h_t satisfies

$$\|\mathcal{I} + h_t A\|_\infty \leq 1.$$

Calculating $\mathcal{I} + h_t A$:

$$\mathcal{I}+h_tA = \begin{pmatrix} 1-2\tilde{\alpha}-\tilde{v} & \tilde{\alpha} & 0 & \dots & 0 & 0 & 0 \\ \tilde{\alpha}+\tilde{v} & 1-2\tilde{\alpha}-\tilde{v} & \tilde{\alpha} & \dots & 0 & 0 & 0 \\ 0 & \tilde{\alpha}+\tilde{v} & 1-2\tilde{\alpha}-\tilde{v} & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1-2\tilde{\alpha}-\tilde{v} & \tilde{\alpha} & 0 \\ 0 & 0 & 0 & \dots & \tilde{\alpha}+\tilde{v} & 1-2\tilde{\alpha}-\tilde{v} & \tilde{\alpha} \\ 0 & 0 & 0 & \dots & 0 & \tilde{\alpha}+\tilde{v} & 1-2\tilde{\alpha}-\tilde{v} \end{pmatrix}$$

where $\tilde{\alpha} = \frac{\alpha h_t}{h_x^2}$ and $\tilde{v} = \frac{v h_t}{h_x}$. Taking the absolute value of all the terms and adding the rows

gives

$$\operatorname{abs}(\mathcal{I}+h_tA)=\begin{pmatrix}|1-2\tilde{\alpha}-\tilde{v}| & \tilde{\alpha} & 0 & \dots & 0 & 0 & 0\\ \tilde{\alpha}+\tilde{v} & |1-2\tilde{\alpha}-\tilde{v}| & \tilde{\alpha} & \dots & 0 & 0 & 0\\ 0 & \tilde{\alpha}+\tilde{v} & |1-2\tilde{\alpha}-\tilde{v}| & \dots & 0 & 0 & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots\\ 0 & 0 & 0 & \dots & 0 & \tilde{\alpha}+\tilde{v} & |1-2\tilde{\alpha}-\tilde{v}|\end{pmatrix}$$

$$\begin{aligned}\rightarrow\quad & \tilde{\alpha}+|1-2\tilde{\alpha}-\tilde{v}|\\ \rightarrow\ & 2\tilde{\alpha}+\tilde{v}+|1-2\tilde{\alpha}-\tilde{v}|\\ \rightarrow\ & 2\tilde{\alpha}+\tilde{v}+|1-2\tilde{\alpha}-\tilde{v}|.\\ & \vdots\\ \rightarrow\quad & \tilde{\alpha}+\tilde{v}+|1-2\tilde{\alpha}-\tilde{v}|\end{aligned}$$

The row sums of the absolute terms of this matrix are

$$a=\tilde{\alpha}+|1-2\tilde{\alpha}-\tilde{v}|,\quad b=2\tilde{\alpha}+\tilde{v}+|1-2\tilde{\alpha}-\tilde{v}|\quad\text{and}\quad c=\tilde{\alpha}+\tilde{v}+|1-2\tilde{\alpha}-\tilde{v}|.$$

Since $\tilde{\alpha}>0$ and $\tilde{v}>0$, then $b>c>a$, therefore, $\|\mathcal{I}+h_tA\|_\infty=b=2\tilde{\alpha}+\tilde{v}+|1-2\tilde{\alpha}-\tilde{v}|$. Consider the two cases $1-2\tilde{\alpha}-\tilde{v}>0$ and $1-2\tilde{\alpha}-\tilde{v}<0$.

1. If $1-2\tilde{\alpha}-\tilde{v}>0$, then $2\tilde{\alpha}+\tilde{v}<1$:

   $$\|\mathcal{I}+h_tA\|_\infty=|1-2\tilde{\alpha}-\tilde{v}|+2\tilde{\alpha}+\tilde{v}=1-2\tilde{\alpha}-\tilde{v}+2\tilde{\alpha}+\tilde{v}=1,$$

   therefore $\|\mathcal{I}+h_tA\|_\infty\leq 1$.

2. If $1-2\tilde{\alpha}-\tilde{v}<0$, then $2\tilde{\alpha}+\tilde{v}>1$:

   $$\|\mathcal{I}+h_tA\|_\infty=|1-2\tilde{\alpha}-\tilde{v}|+2\tilde{\alpha}+\tilde{v}=2\tilde{\alpha}+\tilde{v}-1+2\tilde{\alpha}+\tilde{v}=4\tilde{\alpha}+2\tilde{v}-1,$$

   therefore in this case, if $\|\mathcal{I}+h_tA\|_\infty$ needs to be less than or equal to 1, then

   $$\|\mathcal{I}+h_tA\|_\infty\leq 1\quad\Longrightarrow\quad 4\tilde{\alpha}+2\tilde{v}-1\leq 1\quad\Longrightarrow\quad 2\tilde{\alpha}+\tilde{v}\leq 1$$

   which contradicts with the assumption that $2\tilde{\alpha}+\tilde{v}>1$.

This means that the Euler method will produce a stable convergent solution if

$$2\tilde{\alpha}+\tilde{v}<1\quad\Longrightarrow\quad 2\alpha\frac{h_t}{h_x^2}+v\frac{h_t}{h_x}<1.$$

This means that a choice can be made with regards to the bounds of the different components, for instance, the values of h_x and h_t can be chosen such that

$$\tilde{\alpha}<\frac{1}{4}\quad\text{and}\quad\tilde{v}<\frac{1}{2}\quad\text{or}\quad\tilde{\alpha}<\frac{1}{3}\quad\text{and}\quad\tilde{v}<\frac{1}{3}$$

or any combination thereof provided that the choices satisfy the inequality $2\tilde{\alpha}+\tilde{v}<1$.

Bound for Convection-Diffusion

Consider the convection-diffusion equation

$$\frac{\partial u}{\partial t} = 0.1\frac{\partial^2 u}{\partial x^2} - 0.5\frac{\partial u}{\partial x} \quad \begin{matrix} t \in [0,10] \\ x \in [-2,2] \end{matrix}$$

$$u(x,0) = u_{init}(x) = 10, \quad u(-2,t) = u_l(t) = 1, \quad u(2,t) = u_r(t) = 0.$$

This can be discretised to give $\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = A\boldsymbol{U}$ where

$$\underbrace{\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} U_1(t) \\ U_2(t) \\ \vdots \\ U_{N-1}(t) \\ U_N(t) \end{pmatrix}}_{\boldsymbol{U}}$$

$$= \underbrace{\left[\frac{0.1}{h_x^2}\begin{pmatrix} -2 & 1 & \dots & 0 & 0 \\ 1 & -2 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & -2 & 1 \\ 0 & 0 & \dots & 1 & -2 \end{pmatrix} + \frac{0.5}{h_x}\begin{pmatrix} -1 & 0 & \dots & 0 & 0 \\ 1 & -1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & -1 & 0 \\ 0 & 0 & \dots & 1 & -1 \end{pmatrix}\right]}_{A} \underbrace{\begin{pmatrix} U_1(t) \\ U_2(t) \\ \vdots \\ U_{N-1}(t) \\ U_N(t) \end{pmatrix}}_{\boldsymbol{U}}$$

subject to

$$\boldsymbol{U}(0) = \begin{pmatrix} u_{init}(x_1) \\ u_{init}(x_2) \\ \vdots \\ u_{init}(x_{N-1}) \\ u_{init}(x_N) \end{pmatrix} \quad \text{where} \quad u_{init}(x) = 10.$$

As yet, the value of N has not been put forward since the stepsizes need to be established first. For a stable Euler method, the stepsizes h_t and h_x need to satisfy

$$2\alpha\frac{h_t}{h_x^2} + v\frac{h_t}{h_x} < 1 \quad \Longrightarrow \quad 2\frac{h_t}{h_x} + 5\frac{h_t}{h_x^2} < 10.$$

If $h_t = 2.5 \times 10^{-5}$ and $h_x = 0.02$ (which corresponds to $N_t = 40000$ and $N_x = 100$), then the Euler method will be stable.

A MATLAB Basics

This Appendix will cover some of the basic procedures in MATLAB.

A.1 Command Window

When MATLAB is opened, you will be faced with a window containing several parts.

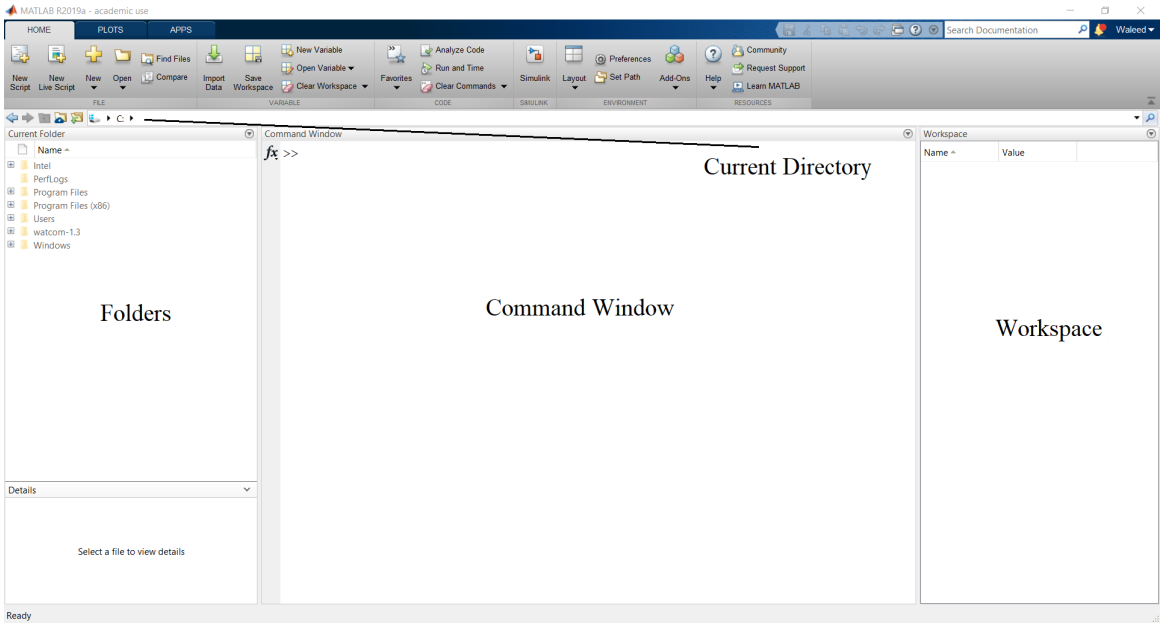


Figure A.1: Default MATLAB layout.

These different areas serve the following purpose:

- **Command Window**: This is the main window where the first line starts with >>. This is where commands are executed, note that once a command has been run (i.e. you pressed Enter), then what has been written cannot be edited or undone and therefore, this is a suitable space for running or executing codes only, not for writing extensive codes.
- **Directory**: This is the destination folder that MATLAB is going to refer to in either opening or saving codes. Note that all MATLAB files are saved as `.m` files.

- **Current Folder**: This displays the functions, figures, subfolders, scripts, codes, etc. that are in the current directory.
- **Workspace**: This displays all the the variables that have been used, along with their types (number, matrix, etc.) and their values.

A.2 Executing Commands in the Command Window

The command window will be where all the codes and functions are executed. It can also be used to perform quick calculations. Some examples of MATLAB syntax and built-in functions are shown below:

Mathematical Symbol	MATLAB Syntax
$+$	`+`
$-$	`-`
$\times$	`*`
$\div$	`/`
3^5	`3^5`
π	`pi`
e^2	`exp(2)`
$\sin(\pi)$	`sin(pi)`
$\sin^{-1}(\pi)$	`asin(pi)`
$\lfloor 3.6 \rfloor$	`floor(3.6)`
$\lceil 4.7 \rceil$	`ceil(4.7)`
$\lvert -4 \rvert$	`abs(-4)`
$1 + 2\mathrm{i}$	`1+2i`
i	`0+i`
$\Re(1 + 2\mathrm{i})$	`real(1+2i)`
$\Im(3 - 4\mathrm{i})$	`imag(3-4i)`
2×10^7	`2e7`
$147 \pmod 5$	`mod(147,5)`

All trigonometric functions follow the same syntax as `sin`, but bear in mind that by default, all the angles should be in *radians* and not in degrees. To use degrees, just put a `d` at the end of the trigonometric function, i.e. use `sind`, `cosd`, `asind`, etc.

The functions $\lceil \bullet \rceil$ and $\lfloor \bullet \rfloor$ are the ceiling and floor functions respectively. Their purpose is to round up to the nearest integer (ceiling) or round down to the nearest integer (floor). Standard rounding can be done using `round`.

Another important function is `mod` which find the remainder when dividing one number by another. For example, $147 \pmod 5$ is the remainder after dividing 147 by 5 which is 2.

```
>> 2+2
ans =
     4
>> sin(0)
ans =
     0
>> sin(pi/2)
ans =
     1
>> sin(30)
ans =
    -0.9880
>> sind(30)
ans =
     0.5000
>> pi
ans =
     3.1416
>> exp(1)
ans =
     2.7813
>> ceil(2.1)
ans =
     3
>> floor(6.9)
ans =
     6
>> round(2.3)
ans =
     2
>> mod(147,5)
ans =
     2
```

If the outcome of a calculation is an integer, then MATLAB will usually display it as an integer, if not, then by default, it will display the solution as a number to 4 decimal places. The number of decimal places can be increased by using `format long` and reversed by using `format short`.

> **Note**
>
> Note that any command executed in the Command Window will be applied globally, so if `format long` is used, it will apply to everything executed in the Command Window until it is reversed or MATLAB is restarted.

```
>> pi/2
ans =
     1.5708
>> format long
>>pi/2
ans =
     1.570796326794897
>> format short
>> pi/2
ans =
     1.5708
```

A.3 Defining Variables

MATLAB is a numerical programming language that relies on a "box" feature. This means that standard algebraic practices cannot be used, for instance, writing $2x = x + 1$ makes perfect sense mathematically and yields a solution of $x = 1$, however writing `2*x=x+1` makes no sense in MATLAB.

> **Note**
>
> A very important note to bear in mind here is that in MATLAB syntax, `2x` has no meaning. In order to multiply terms, the multiplication sign `*` needs to be used.

A "box" with a given name, which is always on the left hand side of the `=` sign, is assigned a value, which is on the right hand side, and the value can then be manipulated or changed, so there are no variables in MATLAB *per se*. In the following example, a "box" is given the name `x` and the number 3 is assigned to it, calculations can then be done by referring to the number that is in said box. The values within the boxes can be redefined by using the `=` sign again.

```
>> x=3
x =
     3
>> x+1
ans =
     4
>> x+x
ans =
     6
>> 3*x
ans =
     9
>> y=(2*x)^x
```

```
y =
     216
>> y+10
ans =
     226
```

On the other hand, $x = x + 2$ makes no sense mathematically but within MATLAB syntax (as is the case with most other programming languages), this simply means calculate `x+2` (which is on the right hand side of the `=` sign) using the value already in the box labelled `x` (which is 3), then redefine the value in that same box to take this new value, so the box labelled `x` is now assigned the value 5.

```
>> x=3
x =
     3
>> x=x+2
x =
     5
>> x=3*x
x =
     15
```

A.4 Naming Variables

There are certain rules with regards to what names can be used for the variables:

- Names can be of **any length** (within the bounds of reason of course to avoid confusion).
- Names are **case sensitive**, so `a` and `A` are two different variable names.
- Names must contain **no spaces**, underscores can be used instead. For example, `Bad Name` is not a viable variable name but `GoodName` and `Also_A_Good_Name` are both valid.
- Names must contain **no operators or symbols**, with the exception of the underscore, so **do not use** `! ? . , ; + - * / & # % $`.
- Names **can contain numbers** as long as they are not the first character. For example `1Forrest1` is not a viable variable name but `OneForrest1` or `Obi1Kenobi` are both viable.
- Names cannot be the same as already **existing functions**, for instance, a variable cannot be given the name `sin` since there is already a built-in function with that same name, however, one could use `Sin` since variable names are case sensitive (although this particular example is not recommended since it may cause confusion).

```
>> P_1=1
P_1 =
     1
>> P_2=P_1+2
P_2 =
     3
>> PP_3=P_1+p_2
Undefined function or variable 'p_2'.
>> PP_3=P_1+P_2
PP_3 =
     4
```

Typing `whos x` in the command window will give the properties of `x`, namely its size (in a matrix sense), storage allocation, class and attributes, but *not* its value. Typing `whos` on its own will give a list of all the variables that have been used along with their properties, alternatively, these can also be found in the Workspace.

A.5 Scripts & Functions

Within the command window, nothing can be edited once it has been executed which is inconvenient if the code is longer than a single line. In that case, it is best to use the *Editor*. By default, the Editor can be opened by clicking on *New Script*, this is a window in which any length of code can be written, saved and then executed with the *Run* button. If any changes need to be made then the editor window will allow that with ease, once changes are made, the code can be run again.

A `function` is very similar to a script but the difference between them is that a function can take in several inputs and produce several outputs and must always have the format:

```
function [output1,output2,...]=Function_Name(input1,input2,...)
    Body of the code
end
```

The `function` cannot always be executed with the *Run* button but will often need to be called in the Command Window to allow for the inputs to be placed.

The name of the function follows the same rules as the variable names mentioned before. One of the most important technicalities that has to be addressed is that the *functions and scripts that are used must be in the same as folder as is stated in the directory.*

When writing functions, or scripts of any kind, there are two important characteristics that need to be considered:

- **Commentary**: When writing codes, it is important to provide some comments on what is being done to give context and to allow for accessibility and reproducibility. This can be done by using `%` at the beginning of the line. This makes MATLAB ignore everything that comes after it, allowing for commentary of bits of code that need context. This is generally good practice in writing codes since the user can make comments about inputs, outputs, procedures, etc. without affecting the execution of the code.
- **Suppression**: On MATLAB, any line of code that is written will produce an output (many other coding languages do not unless prompted to do so). So in functions, performing an action will always produce an output whether it is needed or not. This is where semicolon `;` can be used. The semi-colon suppresses the output, this means that if there are several calculations to be made, sometimes the intermediate stages do not need to be seen, only the final answer, in this case the semicolon allows the calculation to be done but not printed out in the command window.

Example of `function`

Consider a cube with side length L (in m) and mass M (in kg), then the object will have density

$$\rho = \frac{M}{L^3}.$$

The following code calculates this density with the inputs being the mass M and length L with the output being the density rho:

```
function [rho]=Calculate_Density(M,L)

% M:  Mass of cube in kg
% L:  Side length of cube in m

rho = M/(L^3);

end
```

This function, which is called `Calculate_Density`, has two inputs, namely `M` and `L`, and one output, namely `rho`. Notice that the list of inputs must always be in round brackets `(...)` while the outputs should be in square brackets `[...]`.
To use this function, just type the name of the function in the command window with the inputs and outputs in *exactly the same order in which they appear* in the function and using the same set of brackets as well, i.e. `(...)` for inputs and `[...]` for outputs.

```
>> [rho] = Calculate_Density(100,20)
rho =
     0.0125
```

Expanding on this, suppose that a new function is desired where the user will input

the mass in pounds and the length in inches but the desired density should still be in kg m^{-2}. A few more lines can be added in that case.

```
1  function [rho]=Calculate_Density_Imperial(M,L)
2
3  % M:  Mass of cube in lbs
4  % L:  Side length of cube in inches
5
6  M = M/2.20462;     % Converts lbs to kg
7  L = L/39.3701;      % Converts inches to m
8
9  rho = M/(L^3);
10
11 end
```

Note that here, the same variable name has been used and then redefined. So initially, `M` will be input in pounds, say `M=50`, then at line 6, the same variable name is redefined, so the new mass will be $M = \frac{50}{2.20462} = 22.6796$, but the same name is used for both. Similarly for `L` when it is converted from inches to meters.
In this case, the function can be executed with a mass of 50lbs and a side length of 10in to give:

```
1  >> [rho] = Calculate_Density_Imperial(50,10)
2  rho =
3       1.3840e+03
```

One of the major differences in using scripts and functions is the assignment of variables and their declaration. In a script, if a variable C was given the value 3 (so `C=3` was in the script) then this value of C will be declared *globally*, meaning that it can be used in the command window and it will still take the same value. However in functions, the variables are declared *locally*, so if in a function the variable C was given the value 3, this will only hold within the function itself and no where outside it.

A.6 Exersises

! Excersise 1: Metric Cone

Write a MATLAB function that takes inputs h and r and outputs the volume of a cone (in cubic meters) with height h in meters and radius r in meters.
Test the code on a cone with height 5m, radius 3m (which should give a volume of $47.1238898 m^3$.

Solution 1

```
function [V]=Cone_Vol1(h,r)

% This function caculates the volume of a cone in m^3

% Inputs:
% h:  Height of the cone in m
% r:  Radius of the cone in m

% Output:
% V:  Volume of the cone in m^3

V=pi*(r^2)*h/3;

end
```

Code test with $h = 5$ and $r = 3$:

```
>> [V]=Cone_Vol1(5,3)
V =
     47.129
```

Excersise 2: Imperial Cone

Write a MATLAB function that takes inputs h and r and outputs the volume of a cone (in cubic meters) with height h in inches and diameter d in yards.
Test the code on a cone with height 10in, diameter 1yd (which should give a volume of 0.0556m^3.

Solution 2

```
function [V]=Cone_Vol2(h,d)

% This function caculates the volume of a cone in m^3

% Inputs:
% h:  Height of the cone in inches
% d:  Diamater of the cone in yards

% Convert h from inches to metres
h = h*0.0254;

% Convert d from yards to metres
d = d*0.9144;

% Radius of cone base is half the diameter
r = d/2;

% Output:
% V:  Volume of the cone in m^3

V=pi*(r^2)*h/3;

end
```

Code test with $h = 10$ and $d = 1$:

```
>> [V]=Cone_Vol2(10,1)
V =
     0.0556
```

B Arrays in MATLAB

MATLAB is one of the most versatile programming languages when it comes to working with vectors and matrices, hence the name MATLAB, particularly MATrix LABoratory. In MATLAB, vectors essentially represent lists and matrices represent tables.

B.1 Vectors

To form a vector, use square brackets and separate the terms using commas to form a row vector or semicolons to form a column vector.

```
>> v=[1,2,3,4]
v =
     1   2   3   4
>> u=[1;2;3;4]
u =
     1
     2
     3
     4
```

An algebraic sequence (a sequence where the consecutive terms differ by a fixed value) can be formed into a vector by using colons as `v=a:n:b`. This forms a vector `v` where the first term is `a`, then next term is `a+n`, then `a+2*n`, etc. until `b` is reached. If the sequence goes beyond `b`, then `b` is ignored and the last term before `b` will be the last term of the sequence. Note that `v=a:b` will produce a row vector from `a` to `b` in steps of 1.

```
>> u=[1:1:10]
u =
     1   2   3   4   5   6   7   8   9   10
>> v=[20:3:30]
v =
     20   23   26   29
>> w=[100:-20:-40]
w =
     100   80   60   40   20   0   -20   -40
```

Some useful operations that can be applied to vectors are: For a vector `v`:

- `abs(v)` takes the absolute value of all the terms of the vector `v`.
- `v'` takes the transpose of the vector $\boldsymbol{v}$, namely $\boldsymbol{v}^{\mathrm{T}}$, so it changes $\boldsymbol{v}$ from a row vector to a column vector and vice versa.
- `length(v)` finds how many terms there are in the vector `v`.
- `max(v)` finds the maximum value in the vector `v` while `min(v)` finds the minimum value.
- `[a,b]=max(v)` produces two outputs, `a` which is the maximum value in the vector `v` and `b` which is its location in `v`, similarly with `[a,b]=min(v)`. (Note that in MATLAB, array positions start from 1, unlike Python which starts from 0.)
- `sum(v)` takes the sum of all the terms in the vector `v`.
- `mean(v)` takes the mean of all the terms in the vector `v`.
- `median(v)` takes the median of all the terms in the vector `v`.
- `sort(v)` orders the terms of `v` in ascending order.
- `sort(v,'descend')` orders the terms of `v` in descending order.
- `norm(v)` finds the magnitude of the vector `v`. Recall that for a vector $\boldsymbol{v} = (v_1, v_2, \ldots, v_N)$, the magnitude of the vector $\boldsymbol{v}$ is given by:

$$|\boldsymbol{v}| = \sqrt{\sum_{n=1}^{N} |v_n|^2} = \sqrt{v_1^2 + v_2^2 + \cdots + v_N^2}.$$

- `norm(v,p)` finds the p-norm of the vector `v`. Recall that for a vector $v = (v_1, v_2, \ldots, v_N)$ and a positive integer p, the p-norm of $\boldsymbol{v}$, denoted $||\boldsymbol{v}||_p$ is given by

$$||\boldsymbol{v}||_p = \sqrt[p]{\sum_{n=1}^{N} |v_n|^p} = \sqrt[p]{v_1^p + v_2^p + \cdots + v_N^p}.$$

  Note that `norm(v)` is the default 2-norm whereas `norm(V,inf)` is the *sup-norm*[1] (also known as the *Chebyshev norm* or *infinity norm*).

[1] Recall that for a vector $\boldsymbol{v}$, the *sup-norm*, denoted $||\boldsymbol{v}||_\infty$ is the maximum absolute term in the vector, i.e. for a vector $v = (v_1, v_2, \ldots, v_N)$,

$$||\boldsymbol{v}||_\infty = \max_{n=1,2,\ldots,N} |v_n|.$$

```
>> v=[2,-8,6,-2,-9,4]
v =
     2    -8     6    -2    -9     4
>> abs(v)
ans =
     2     8     6     2     9     4
>> v'
ans =
     2
    -8
     6
    -2
    -9
     4
>> (v')'
ans =
     2    -8     6    -2    -9     4
>> length(v)
ans =
     6
>> max(v)
ans =
     6
>> [a,b]=max(v)
a =
     6
b =
     3
>> min(v)
ans =
    -9
>> [a,b]=min(v)
a =
    -9
b =
     5
>> sum(v)
ans =
    -7
>> mean(v)
ans =
    -1.1667
>> median(v)
ans =
     0
```

```
>> sort(v)
ans =
     -9   -8   -2    2    4    6
>> sort(v,'descend')
ans =
      6    4    2   -2   -8   -9
>> norm(v)
ans =
     14.3175
>> norm(v,1)
ans =
     31
>> norm(v,inf)
ans =
     9
```

B.2 Matrices

To form matrices, the same theme follows as with vectors where a comma indicates the next term on the same row and semicolons move to the next row. Be careful to ensure that *all the rows have the same number of terms*, similarly with the columns.

```
>> M=[1,2,3;4,5,6;7,8,9]
M =
     1    2    3
     4    5    6
     7    8    9
>> N=[1,2,3,4,5;6,7,8,9,10]
N =
     1    2    3    4    5
     6    7    8    9   10
>> P=[1,2,3;4,5,6;7,8]
Error using vertcat
Dimensions of arrays being concatenated are not consistent.
```

There are some operations that translate from vectors to matrices, for example, for a matrix `M`:

- `abs(M)` takes the absolute value of all the terms of the matrix `M`.
- `M'` takes the transpose of the matrix `M`.

Other functions as not as intuitive, for example, `length(M)` gives *only one* output which is *either* the number of rows *or* the number of columns, whichever is bigger. Whereas `size(M)`

gives two outputs with the first being the number of rows of `M` and the second is the number of columns of `M`.

Some matrix functions are done column-wise, for example, `max(M)` *does not* give the maximum value that appears in the matrix, instead it produces a row vector of maxima where the first term is maximum value of all the terms in the first column, the second is the maximum of the second column and so on. This same column-wise approach holds for other functions like `min(M)`, `sum(M)`, `mean(M)` and `sort(M)`; MATLAB works with the matrix as a collection of column vectors and applies these functions to each column separately. To find the maximum/minimum/sum of all th terms in the entire matrix, then the function will need to be used twice, so the maximum element in the whole matrix can be found by using `max(max(M))`.

Note that `[a,b]=max(M)` will give two outputs, the first output `a` is the vector `max(M)` as described above and the second output `b` is the vector of their locations. Similarly for `[a,b]=min(M)`.

Matrix norms are slightly more involved, in terms of their mathematical definition, than vector norms. For a matrix M of size $m \times n$ and a positive integer p, the matrix p-norm imposed by the vector p-norm is given by

$$||M||_p = \sup_{\boldsymbol{x} \in \mathbb{C}^n} \frac{||M\boldsymbol{x}||_p}{||\boldsymbol{x}||_p}$$

Calculating these explicitly can be very difficult since it requires using *all* possible vectors $\boldsymbol{x} \in \mathbb{C}^n$, however, the most useful norms have some closed forms:

- $||M||_1$ is the *maximum absolute column sum*;
- $||M||_\infty$ is the *maximum absolute row sum*;
- $||M||_2$ is the ***Spectral Radius*** of M (more specifically, it is the square root of the largest eigenvalue of the matrix $M^{\mathrm{H}}M$ where M^{H} is the Hermitian of M, or the complex conjugate transpose).

There are other norms that are not imposed by vector norms, like the ***Frobenius Norm*** which is the square root of the sum of the squares of the absolute valaue of all the terms, i.e.

$$||M||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|m_{ij}|^2}.$$

All these norms still use the same syntax as vector norms, i.e. using `norm(M,1)`, `norm(M,2)`, `norm(M,inf)` and `norm(M,'Fro')` (with `norm(M)` being the default 2-norm). This is why it is imperative to be mindful of the context since the same operation can have different meanings depending on whether the input was a vector or a matrix.

```
>> M=[-4,5;2,9;-6,10]
M =
    -4     5
     2     9
```

```
    -6   10
>> abs(M)
M =
     4   5
     2   9
     6   10
>> M'
ans =
    -4   2   -6
     5   9    10
>> size(M)
ans =
     3     2
>> length(M)
ans =
     3
>> max(M)
ans =
     2   10
>> max(max(M))
ans =
     10
>> [a,b]=max(M)
a =
     2   10
b =
     2   3
>> min(M)
ans =
    -6   5
>> min(min(M))
ans =
     -6
>> [a,b]=min(M)
a =
    -6   5
b =
     3   1
>> sum(M)
ans =
    -8   24
>> sum(sum(M))
ans =
     16
>> mean(M)
```

```
ans =
    -2.6667    8.0000
>> median(M)
ans =
    -4    9
>> sort(M)
ans =
    -6    5
    -4    9
     2   10
>> sort(M,'descend')
ans =
     2   10
    -4    9
    -6    5
>> norm(M)
ans =
    15.1099
>> norm(M,1)
ans =
    24
>> norm(M,inf)
ans =
    16
>> norm(M,'Fro')
ans =
    16.1864
```

B.3 Referencing Terms in Arrays

Elements of a vector (row or column) can be referred to by putting the index of the desired element in brackets after the vector's name. For example, `v(4)` is the 4^{th} element in the vector `v`.

> **MATLAB Indexing**
>
> Note that in MATLAB, indexing starts from 1, not from 0 like Python.

If the last element of a vector is desired where its size may not be known, then the index `end` can be used.

```
>> u=[9;7;0;1]
u =
     9
```

```
     7
     0
     1
>> u(1)
ans =
     9
>> u(4)
ans =
     1
>> u(end)
ans =
     1
>> u(6)
Index exceeds array bounds.
```

For matrices, there are two indices, the first denotes the row number and the second the column number:

$$\begin{pmatrix} (1,1) & (1,2) & (1,3) & \dots \\ (2,1) & (2,2) & (2,3) & \dots \\ (3,1) & (3,2) & (3,3) & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

So `M(2,3)` will output the element of `M` that is in row 2 and column 3. MATLAB also has the ability to refer to terms in matrices by using one index only. For instance, if a matrix M is of size 3×4, then $M(10)$ would refer to the "10^{th} element". Under usual circumstances, this is meaningless unless M is a vector, however, in this case, MATLAB can refer to the 10^{th} element where the elements start from 1 and work their way down columns as such:

$$\begin{pmatrix} (1) & (4) & (7) & (10) \\ (2) & (5) & (8) & (11) \\ (3) & (6) & (9) & (12) \end{pmatrix}$$

Therefore, the 10^{th} element of M would be the element in the 1^{st} row and 4^{th} column for the 3×4 matrix. Using this referencing system is certainly not recommended since it can cause issues with different sized matrices.

MATLAB can also refer to whole rows or whole columns, this is done by using `:`, for example `M(:,3)` will produce the 3^{rd} column whereas `M(1,:)` will produce the 1^{st} row.

```
>> M=[2,3,1,4;1,6,3,1;4,1,2,8]
M =
     2   3   1   4
     1   6   3   1
     4   1   2   8
>> M(2,3)
```

```
ans =
      3
>> M(3,1)
ans =
      4
>> M(end,3)
ans =
      2
>> M(end,end)
ans =
      8
>> M(:,2)
ans =
      3
      6
      1
>> M(3,:)
ans =
      4   1   2   8
>> M(:,end)
ans =
      4
      1
      8
>> M(2)
ans =
      1
>> M(4)
ans =
      3
>> M(12)
ans =
      8
```

B.4 Matrix Operations

Addition and subtraction of matrices (and vectors) follows the usual mathematical rules, namely, both matrices need to be of the same size and all the terms are added elementwise, i.e. the first term is added to the first term, the second to the second, etc.

```
>> A=[1,3,7;5,2,6;2,3,2]
A =
     1   3   7
     5   2   6
```

```
     2   3   2
>> B=[2,3,1;1,6,3;4,1,2]
B =
     2   3   1
     1   6   3
     4   1   2
>> A+B
ans =
     3   6   8
     6   8   9
     6   4   4
```

Matrices and vectors can be multiplied or divided by *a scalar value* using the `*` and `/` operations.

```
>> 2*A
ans =
     2   6  14
    10   4  12
     4   6   4
>> B/2
ans =
     1.00    1.50    0.50
     0.50    3.00    1.50
     2.00    0.50    1.00
```

Matrix multiplication is carried out using the `*` operator. Recall that for two matrices A, of size $m \times n$, and B, of size $p \times q$, the matrix product AB is only possible if $n = p$ (i.e. the number of columns of A is equal to the number of rows of B) and the resulting matrix AB will then be of size $m \times q$.

```
>> A*B
ans =
     33   28   24
     36   33   23
     15   26   15
```

Elementwise multiplication and division of matrices (also known as the **Hadamard Operations**) is also a possibility in MATLAB. So for matrices A and B of the same size, the elementwise product (denoted mathematically as $A \circ B$) produces a matrix that is of the same size as A and B where the first element is the product of the first element of A and the first element of B, the second element is the product of the second element of A and the second element of B and so on. This is done using a dot `.` before the operations, in other words, the elementwise product $A \circ B$ is written as `A.*B`, similarly for elementwise division using `./` and elementwise exponentiation using `.^`. Bear in mind this is *only possible if* the matrices/vectors are of the same size, just as in addition and multiplication.

```
>> A.*B
ans =
      2   9   7
      5  12  18
      8   3   4
>> A./B
ans =
      0.50  1.00   7.00
      5.00  0.33   2.00
      0.50  3.00   1.00
>> A.^2
ans =
      1  9   49
     25  4   36
      4  9    4
>> A^2
ans =
      30    30    39
      27    37    59
      21    18    36
```

There are some special matrices and matrix forms built into MATLAB such as:

- `[]`: empty vector/matrix which contains no terms, therefore has size 0×0 and is usually used as a placeholder.
- `zeros(a,b)`: forms a matrix of zeros with size `a` $\times$ `b`.
- `ones(a,b)`: forms a matrix of ones with size `a` $\times$ `b`.
- `eye(a,b)`: forms an identity matrix (ones on the main diagonal, zeros otherwise) of size `a` $\times$ `b`.
- `rand(a,b)`: forms a matrix of size `a` $\times$ `b` where all the elements are randomly chosen from a normal distribution whose entries lie between 0 and 1.
- `randi([M,N],a,b)`: forms a matrix of size `a` $\times$ `b` where all the elements are randomly chosen integers from a normal distribution whose entries lie between `M` and `N`.
- `diag(v)`: forms a square matrix whose diagonal entries are the elements of the vector `v`.

There are also some matrix operations that are very useful such as:

- `inv(A)`: find the inverse of the matrix `A`.
- `det(A)`: find the determinant of the matrix `A`.
- `trace(A)`: find the trace of the matrix `A` (which is the sum of the diagonal entries).

B.5 Substitution & Concatenation

Sometimes, vectors and matrices need to be augmented, either by adding, removing or changing some terms.

For both vectors and matrices, individual values can be substituted and redefined by referring to its index. For example, consider the vector $\boldsymbol{v}$ and suppose that its second element is to be changed, this can be done by using `v(2)=` to assign a new value that will overwrite the original value.

```
>> v=[1,3,7,5]
v =
     1   3   7   5
>> v(2)
ans =
     3
>> v(2)=8
v =
     1   8   7   5
>> v(4)=0
v =
     1   8   7   0
```

The same syntax can be used to redefine an element in terms of itself or in terms of others, like defining the second element as twice its original value or setting an element to be the sum of some other elements.

```
>> v(2)=10*v(2)
v =
     1   80   7   0
>> v(1)=v(3)
v =
     7   80   7   0
>> v(4)=v(1)+v(2)+v(3)
v =
     7   80   7   94
```

The same can be done with matrices as well where this replacement can either be done by elements, rows or columns.

```
>> M=[2,1;3,6]
M =
     2   1
     3   6
>> M(1,2)
```

```
ans =
     1
>> M(1,2)=4
M =
     2   4
     3   6
>> M(2,2)=0
M =
     2   4
     3   0
>> M(1,:)
ans =
     2   4
>> M(1,:)=[9,1]
M =
     9   1
     3   6
>> M(:,2)
ans =
     1
     6
>> M(:,2)=[4;0]
M =
     9   4
     3   0
```

Matrices and vectors can also be concatenated or cut, that simply means that terms can be added or removed, this is done by using the comma or semi-colon depending on the situation. Not only can terms be added, but whole rows and columns can be added as well but it is *critical* that the terms are added in a consistent fashion, meaning that if a new row is to be added, then it must be of the same size as all the other rows otherwise it will not make sense. To remove rows or columns, then simply assign an empty vector, namely `[]`, to the desired location.

```
>> A=[1,7]
A =
     1   7
>> A=[A,4]          % Add 4 to the end
A =
     1   7   4
>> A=[8,A]          % Add 8 to the start
A =
     8   1   7   4
>> A=[A;[0,5,7,9]]  % Add a new row
A =
```

```
     8   1   7   4
     0   5   7   9
>> A=[A,[0;1]]      % Add a new column
A =
     8   1   7   4   0
     0   5   7   9   1
>> A(:,3)=[]        % Remove third column
A =
     8   1   4   0
     0   5   9   1
>> A(1,:)=[]        % Remove first row
A =
     0   5   9   1
>> A(end)=[]        % Remove last term
A =
     0   5   9
```

B.6 Finding Terms

Sometimes, finding some terms is desired, say if the user needs to find all the values in a list that are greater than 5, or less than -1, or equal to 2. In this case, the comparative operators should be used which are:

Operation	MATLAB Syntax
Less than	<
Less than or equal to	<=
Equal to	==
Greater than	>
Greater than or equal to	>=
Not equal to	~=

These operators need to be used in conjunction with the `find` function. So for a given vector `v`, if the terms greater than 5 need to be found, then use `find(v>5)`, this will produce a vector of *indices* that denote the locations of the values that greater than 5. If there are no such values that satisfy the condition, then an empty vector will be produced, namely `[]`. This can be very useful if, say, all the values greater than 5 need to be multiplied by 10, or all the values that are less than -1 need to be changed to 0, or all the values that are equal to 2 need to be removed.

```
>> v=[1,2,-5,12,-3,2]
v =
     1   2   -5   12   -3   2
```

```
>> i=find(v>5)
ans =
     4
>> v(i)
ans =
     12
>> v(i)=10*v(i)
v =
     1    2    -5    120    -3    2
>> j=find(v<-1)
ans =
     3    5
>> v(j)
ans =
    -5    -3
>> v(j)=0
v =
     1    2    0    120    0    2
>> k=find(v==2)
ans =
     2    6
>> v(k)=[]
v =
     1    0    120    0
```

When finding terms in matrices, MATLAB tends to provide the location in the single index form rather than in the dual form. In other words, if a matrix is of size 3×3 and MATLAB needs to refer to the $(2, 3)$ element (second row, third column), it would display the index as the 7^{th} element. This is an important distinction that needs to be made.

```
>> M=[2,0,5;-1,2,9;-6,1,-8]
M =
     2    0    5
    -1    2    9
    -6    1   -8
>> m=find(M>5)
m =
     8
>> M(m)
and =
     9
>> M(m)=M(m)*10
M =
     2    0    5
    -1    2   90
```

```
    -6    1   -8
>> n=find(M<0)
n =
     2
     3
     9
>> M(n)
ans =
    -1
    -6
    -9
>> M(n)=0
M =
     2    0    5
     0    2   90
     0    1    0
```

An alternative way of finding terms would be to dispense with the `find` command altogether. This will produce a binary matrix showing the locations of the terms that satisfy the condition (with `1` being true and `0` being false).

```
>> A=[1,4,6,9,2;7,3,1,6,0]
A =
     1    4    6    9    2
     7    3    1    6    0
>> find(A>5)
ans =
     2
     5
     7
     8
>> A>5
ans =
     2×5 logical array
     0    0    1    1    0
     1    0    0    1    0
```

B.7 Exercises

! Exersise 1: Matrix Calculations

$$A = \begin{pmatrix} 1 & 2 \\ 5 & 8 \end{pmatrix} \quad ; \quad B = \begin{pmatrix} 4 & 0 & -4 \\ -1 & 0 & 1 \\ 2 & 1 & 3 \end{pmatrix} \quad ; \quad C = \begin{pmatrix} 1 & 0 & 4 \\ 2 & -2 & 6 \end{pmatrix}$$

$$\boldsymbol{u} = \begin{pmatrix} 1 \\ 8 \end{pmatrix} \quad ; \quad \boldsymbol{v} = \begin{pmatrix} 0 \\ 3 \\ 4 \end{pmatrix}$$

Using MATLAB, write a command/script to produce:

- The matrix AC.
- Element (2,3) of the matrix CB.
- Third element of the matrix $\boldsymbol{u}^{\mathrm{T}}C$.
- Element (1,2) of the matrix $\boldsymbol{u}\boldsymbol{v}^{\mathrm{T}}$.
- Trace of B^2.
- Maximum and minimum terms in $B\boldsymbol{v}$.
- 2-norm of $\boldsymbol{v}$.
- Frobenius norm of B.
- The determinant of B.
- The inverse of $134(C^{\mathrm{T}}C + \mathcal{I})$ where $\mathcal{I}$ is the identity matrix.
- The eigenvalues and eigenvectors of $\boldsymbol{v}\boldsymbol{u}^{\mathrm{T}}C$.

Solution 1

```
>> A=[1,2;5,8];
>> B=[4,0,-4;-1,0,1;2,1,3];
>> C=[1,0,4;2,-2,6];
>> u=[1;8];
>> v=[0;3;4];
>> A*C
ans =
     5    -4    16
     21   -16   68
>> D=C*B
D =
     12   4   8
     22   6   8
>> D(2,3)
ans =
     8
>> E=u'*C
E =
     17   -16   52
>> E(3)
ans =
     52
>> F=u*v'
F =
     0   3    4
     0   24   32
>> F(1,2)
ans =
     3
>> trace(B*B)
ans =
     11
>> G=B*v
G =
    -16
     4
     15
>> max(G)
ans =
     15
>> min(G)
ans =
    -16
>> norm(v,2)
ans =
     5
>> norm(B,'Fro')
ans =
     6.9282
>> det(B)
ans =
     0
>> H=134*(C'*C+eye(3))
H =
     804    -536    2144
    -536     670   -1608
```

C Loops

Loops are some of the most important features in any programming language and they fall under three types: `if`, `while` and `for` loops.

C.1 `if` Loops

An `if` command executes a loop if a certain condition is satisfied. This requires the use of comparative operators which are:

Operation	MATLAB Syntax
Less than	<
Less than or equal to	<=
Equal to	==
Greater than	>
Greater than or equal to	>=
Not equal to	~=

An `if` loops must have the following structure:

```
if compare <=> compare with

      do something

elseif compare <=> compare with

      do something else

else

     do something if none of the above conditions have been met

end
```

`if` Loop Example

Suppose a function is to be written which takes a number N as an input then in the command window, displays "The Good" if it is positive, "The Bad" if it is negative and "The Ugly" if it is zero[1].

```
function Good_Bad_Ugly(N)

if N>0                  % First check if the input N is positive

     disp('The Good')   % If N is positive, display 'The Good'

elseif N<0              % If N is not positive, check if it is negative

     disp('The Bad')    % If N is negative, display 'The Bad'

elseif N==0             % If N is neither positive nor negative, check
                        % if it zero

     disp('The Ugly')   % If N is zero, display 'The Ugly'

end

end
```

The `disp` command outputs the variables stated within the brackets, if the argument is single quotation marks, namely `'...'`, then it will be displayed verbatim. Note that here, the line will not start with `ans =` since it is was only asked to display and not specify variables. This function can be run within the command window as follows:

```
>> Good_Bad_Ugly(3)
The Good
>> Good_Bad_Ugly(-5)
The Bad
>> Good_Bad_Ugly(0)
The Ugly
```

In `if` loops, it is always a good idea to have a few `elseif` commands in order to have all the cases covered, this is because sometimes, MATLAB can misunderstand some inputs. For instance, suppose that the input is the complex number $1-2i$:

```
>> Good_Bad_Ugly(1-2i)
The Good
```

This does not make sense since the number $1-2i$ is neither positive nor negative, nor zero for that matter. In this case, MATLAB takes the real part only without being

prompted to do so, and prints the output and since the real part is 1, the output will be `The Good`. In order to accommodate for this, an extra condition can be added in the form of another `if` loop that considers this and displays "The Complex" if the number is complex.

```
function Good_Bad_Ugly(N)

if imag(N)~=0            % First, check if N has a non-zero imaginary
                         % part

     disp('The Complex') % If N does have a non-zero imaginary part,
                        % display 'The Complex'

else                     % Otherwise, run the code as before

     if N>0

          disp('The Good')

    elseif N<0

         disp('The Bad')

     elseif N==0

         disp('The Ugly')

    end

end

end
```

In this case, if the input as $1-2i$, then the output will be `The Imaginary`.

It is important to note that in `if` loops, the code will quit the loop after the first time the `if` condition is satisfied and will not check the other conditions.

`if` Loop Ordering

Suppose a function is to be written which takes an input N and displays "Multiple of 2" if it is a multiple of 2, "Multiple of 3" if it is a multiple of 3 and "Too high to count" otherwise. This function will require the use of the `mod` syntax; for numbers `N` and `b`,

[1] In reference to the 1966 film "The Good, the Bad and the Ugly."

`mod(N,b)` will produce `0` if `N` is a multiple of `b`.

```
function Mult(N)

if mod(N,2)==0      % Check if N is a multiple of 2

     disp('Multiple of 2')

elseif mod(N,3)==0  % Check if N is a multiple of 3

     disp('Multiple of 3')

else

     disp('Too high to count')

end

end
```

Run this code with the inputs $10, 15, 19$ and 24:

```
>> Mult(10)
Multiple of 2
>> Mult(15)
Multiple of 3
>> Mult(19)
Too high to count
>> Mult(24)
Multiple of 2
```

For the inputs $10, 15$ and 19, the results are as expected however with 24, only one output is produced, suggesting that 24 is a multiple of 2 only. The reason this is produced is because the `if` loop checked the first condition and since it was satisfied, it executed the code block underneath and quit the whole loop, not running through the others. That is why it is very important to be aware of the ordering of the `if` and `elseif` commands.

C.2 `while` Loops

The `while` loop is somewhat similar to the `if` loop in the sense that values of two terms are being compared but here, the loop will keep repeating until the condition is no longer satisfied.

`while` Loop Example

Suppose a function is to be written that takes two inputs, N and d and keeps subtracting d from N until it can no longer do so without becoming negative, the function should then output the last positive integer after this repeating operation. This code is the equivalent of finding the remainder of dividing a number N by d (or taking $N \pmod d$). For example, if $N = 9$ and $d = 4$, then $N - d = 5$, $N - 2d = 1$, $N - 3d = -3$, then the function would take the inputs $(N, d) = (9, 4)$ and outputs 1.

```
function [r]=Remainder(N,d)

M=N;            % Start with the number M being equal to N

while M-d>=0    % As long as M-d is non-negative, run the loop

     M=M-d;     % Since M-d is non-negative, find M-d
                % and let M be equal to this new value,
                % this keeps repeating until M-d<0

end

r=M;            % Set the remainder r to be this final value M

end
```

This can be used in the command window as follows (note that here, because there is only one output, then it does not need to be explicitly stated in square brackets):

```
>> [r]=Remainder(9,4)
r =
     1
>> [r]=Remainder(10,2)
r =
     0
>> Remainder(14515,135)
ans =
     70
>> Remainder(1e12,42578)
ans =
     20554
```

Suppose now that this code is to be modified so that it can also output the number of times d can be subtracted from N. For example, as before, if $(N, d) = (9, 4)$, the remainder is 1 and the number of times d must be subtracted from N to obtain this remainder is 2, this is the equivalent of finding the number of times the `while` loop

actually ran. This is a very common procedure and the way to tackle this is by use of a "counter". This is a variable that starts with the value 0 and every time the while loop is run, 1 is added to it. This modification can be done as follows.

```
function [r,counter]=Remainder(N,d)

M=N;                        % Start with the number M being equal to N

counter=0;                  % Start with the counter being 0

while M-d>=0                % As long as M-d is non-negative, run the loop

     M=M-d;                 % Since M-d is non-negative, find M-d
                            % and let M be equal to this new value

     counter=counter+1;     % Add 1 to the counter every time
                            % the while loop is run
end

r=M;                        % Set the remainder r to be this final value M

end
```

This can be used in the command window as follows (in this case, since there are two outputs, they both have to be stated, but they don't need to be of the same name, only the same order):

```
>> [r,counter]=Remainder(9,4)
r =
     1
counter =
     2
>> [r,c]=Remainder(10,2)
r =
     0
c =
     5
>> [R,C]=Remainder(14515,135)
R =
     70
C =
     107
>> [r,c]=Remainder(1e12,42578)
r =
     20554
c =
     23486307
```

Caution 1: Collatz Conjecture

In mathematics, there is a famous algorithm known as the *Collatz Conjecture*, the steps of the algorithm are as follows:

1. Pick any positive integer.
2. i. If the number is even, divide by 2.
   ii. If the number is odd, multiply by 3 and add 1.
3. Repeat Step 2.

For instance, if the input is the number 10, the sequence of numbers will be as follows:

$$10 \underset{\div 2}{\longrightarrow} 5 \underset{\times 3+1}{\longrightarrow} 16 \underset{\div 2}{\longrightarrow} 8 \underset{\div 2}{\longrightarrow} 4 \underset{\div 2}{\longrightarrow} 2 \underset{\div 2}{\longrightarrow} 1$$

Similarly, if the input is 21:

$$21 \underset{\times 3+1}{\longrightarrow} 64 \underset{\div 2}{\longrightarrow} 32 \underset{\div 2}{\longrightarrow} 16 \underset{\div 2}{\longrightarrow} 8 \underset{\div 2}{\longrightarrow} 4 \underset{\div 2}{\longrightarrow} 2 \underset{\div 2}{\longrightarrow} 1$$

Both number sequences end up at 1 from two different starting numbers of 10 and 21. (The algorithm is stopped at 1 since if the algorithm is carried on after reaching 1, then a loop will be formed going 4, 2, 1, 4, 2, 1, ...) The *Collatz Conjecture* states that regardless of the starting value, this sequence will *always* reach a 4-2-1 loop. This statement has been put forward in 1937 and has not yet been proven or disproven but has been computed for numbers larger than 10^{17}, all the numbers end at the 4-2-1 loop.

The `while` loop can be used in conjunction with the `if` loop in order to make a function that outputs the number of steps it takes to get to 1. This code can be checked by having an input of 10 and the output should be 6 since the algorithm required 6 steps before reaching 1, similarly, if the input is 21, then the output should be 7 and these can be used as test cases.
In writing codes, it is helpful to start with a pseudocode:

1. Read the input number.
2. As long as the number is greater than 1, do the following:
    i. If the number is even, divide by 2.
    ii. If the number is odd, multiply by 3 and add 1.
3. Repeat Step 2 until 1 is reached.

From this pseudocode, it is clear that Step 2 can be represented by an `if` loop. Steps 2 and 3 require the number to be greater than 1, since it is unknown when that will happen, the `while` loop can be used. Now, the pseudocode can be translated into MATLAB syntax with an input value of `a` and an output value `N` which is the number of staeps it takes to get to 1.

```
function [N]=Collatz(a)

N=0;                        % Start with N=0

while a>1                   % Perform the code block as long as the number
                            % is bigger than 1

     if mod(a,2)==0         % Check if the number is even

          a=a/2;            % If it is, redefine a as a/2

     else                   % Otherwise, if a is odd

          a=3*a+1;          % Redefine a as 3a+1

     end

     N=N+1;                 % Every time the code block is run, add 1 to N

end

end
```

This code can be checked using the test cases:

```
>> Collatz(10)
ans =
     6
>> Collatz(21)
ans =
     7
>> Collatz(1000)
ans =
     111
```

The function `Collatz` should only be able to take integer inputs. A custom error message can be made to ensure that; the following can be added in Line 2:

```
if mod(a,1)~=0
     error('a must be an integer')
end
```

C.3 Multiple Conditions for `if` & `while` Loops

Occasionally, multiple conditions may need to be satisfied when running `if` or `while` loops, this can be done with the `&&` for conjunctive conditions (equivalent to *and*) and `||` for disjunctive conditions (equivalent to *or*).

Collatz Isolation

For the function `Collatz` in Caution 1, the code should only be able to take any positive integer. An exclusion was introduced to produce an error message if the input was not an integer. Suppose that another condition is to be added that would produce the same error message if the input value is non-positive or not real. This can be done using the *or* syntax, which is `||`.

```
if imag(a)~=0 || mod(a,1)~=0 || a<=0 || imag(a)~=0
     error('a must be an integer')
end
```

C.4 `for` Loops

A `for` loop is different compared to the `while` and `if` loops since it does not require comparison, instead, it runs through a series of terms that have been predefined.

`for` Loop Example 1

Suppose a simple `for` loop is needed that takes an input value N and adds all the positive integers from 1 to N. So if $N = 10$, then the function would output the sum of the numbers from 1 to 10, namely 55. This can be written as follows:

```
function [Sum]=Summation(N)

Sum=0;

for i=1:1:N

     Sum=Sum+i;

end

end
```

This simple code starts with a `Sum=0`, then the variable `i` runs from 1 to N and adds itself onto `Sum`, the final result would be the sum of all the positive integers form 1 to N[2].

`for` Loop Example 2

Suppose a `for` loop is desired that takes a vector $\boldsymbol{v}$ as an input and outputs the vector $\boldsymbol{u}$ whose elements are the squares of $\boldsymbol{v}$[3].
The vector $\boldsymbol{v}$ will be a part of the input but the vector $\boldsymbol{u}$ needs to be *initialised*, meaning that $\boldsymbol{u}$ has to be predefined in some way. Since the size of $\boldsymbol{u}$ will be the same as $\boldsymbol{v}$, then the vector $\boldsymbol{u}$ can be initialised as a vector of zeros that is the same size as $\boldsymbol{v}$, this can be done using `u=zeros(size(v))`. The code can then be written by replacing the appropriate term in the list.

```
function [u]=Square(v)

u=zeros(size(v));

for i=1:1:N

     u(i)=v(i)^2;

end

end
```

[2]Bear in mind that this is a contrived example for the sake of demonstration. This exact procedure can be done in one single command `sum(1:1:10)`.

Alternatively, if the size of $\boldsymbol{u}$ is not known, then it can be initialised as an empty array `[]` and terms can be concatenated to it.

```
function [u]=Square2(v)

u=[];

for i=1:1:N

     u=[u,v(i)^2];

end

end
```

C.5 Exercises

! Excersise 1

Write a MATLAB function called `Fib` that takes an input N and produces a value F that is the N^{th} term of the Fibonacci sequence starting from 1,3 (recall that a Fibonacci sequence is a sequence where any term is the sum of the previous two terms). For example, if $N = 5$, then the first 5 terms of this Fibonacci sequence are $(1, 3, 4, 7, 11)$, meaning that the output should be $F = 11$. Use the following test cases to verify that the code produces the correct results:

- $N = 10$: $F = 123$;
- $N = 20$: $F = 15127$;
- $N = 50$: $F = 28143753123$.

[3] Just as before, this is intended to be a contrived example to show the working of a `for` loop. This procedure can be done in a single command as `u=v.^2` for elementwise exponentiation.

Solution 1

```
function [F]=Fib(N)

S=zeros(1,N);       % Initialise the sequence S as a list of N zeros

S(1)=1;             % Redefine the first term of S to be equal to 1

S(2)=3;             % Redefine the second term of S to be equal to 3

for n=3:1:N         % Starting from the third term onwards

     S(n)=S(n-1)+S(n-2); % Let the nth term of S be the sum of the
                         % previous two terms

end

F=S(end);       % Let F be the last term in the sequence S,
                % alteratively, F=S(N) can be used since it is known that
                % N is the last term

end
```

Exersise 2

Write a MATLAB function called `Fib2` that takes an input M and produces values c and G where G is the largest term of the Fibonacci sequence starting from 2,5 such that $G < M$ and the number of terms in the sequence up to that point is c. For example, if $M = 60$, start a Fibonacci sequence with the 2,5 until a number above M is reached and count the number terms. So if $M = 60$, then the sequence is $(2, 5, 7, 12, 19, 31, 50, 81)$, meaning that $G = 50$ (since it is the largest term in the sequence that is less than M) and $c = 6$ (since it takes 6 steps to get to 50). Use the following test cases to verify that the code produces the correct results:

- $M = 100$: $G = 81$, $c = 9$;
- $M = 1000$: $G = 898$, $c = 14$;
- $M = 10^9$: $G = 638162747$, $c = 42$.

Solution 2

```
function [c,G]=Fib2(M)

S=[2,5];        % Since, in principle, the number of terms is not known,
                % then define S as the seuqnece starting with 2 and 5

while S(end)<M % Run the while loop as long as the last term of the
                % sequence is less than M

     S=[S S(end)+S(end-1)];   % Redefine S in terms of itself; start
                              % with the sequnce S and append an extra
                              % term at the end that is the sum of the
                              % last term and the one before it

end

G=S(end-1);     % G will be the second to last term (since the last one
                % is bigger than M)

c=length(S);    % c is simply the length of S

end
```

D Plotting in MATLAB

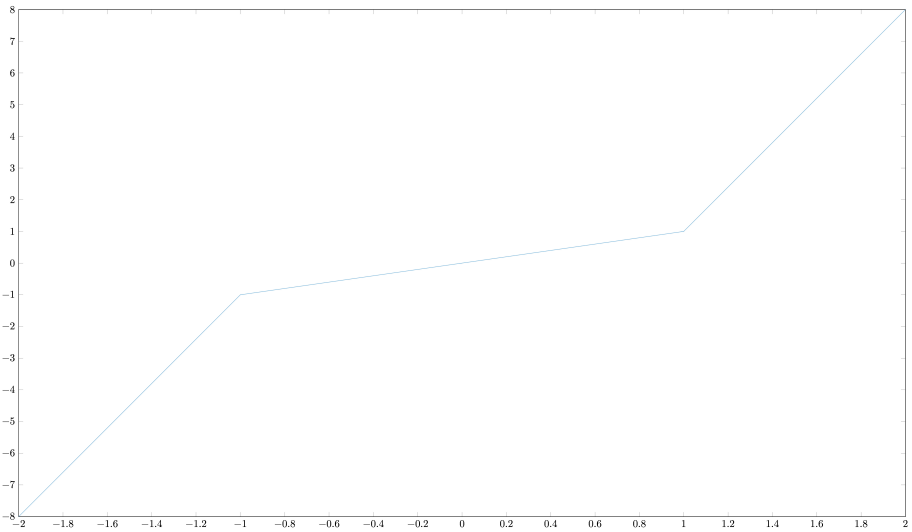
D.1 Forming Lists for Plotting

Suppose the function x^3 is to be plotted. First of all, a range of x values is needed, so if the function needs to be plotted in the interval $[-2, 2]$, then a vector needs to be formed that spans this particular domain, the more points there are, the smoother the function will be. This can be in done by using, say, `x=-2:1:2` which produces a vector `x` with 5 points, namely `x=[-2 -1 0 1 2]`.

Secondly, the values on the y-axis need to be formed. For every x value, the value on the y axis will be at x^3, this can be done using elementwise exponentiation as `y=x.^3`. In this case, the `x` and `y` vectors will be `x=[-2 -1 0 1 2]` and `y=[-8 -1 0 1 8]`.

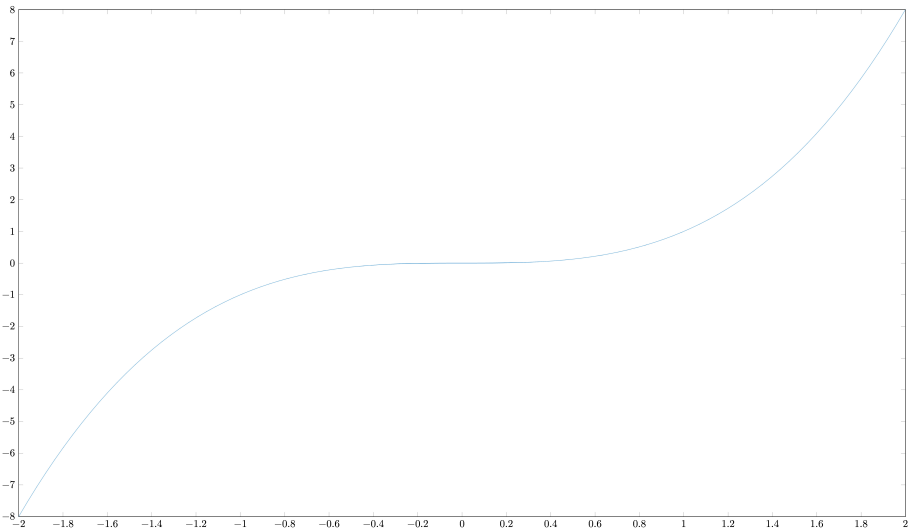
Now the plotting can commence. The `plot` function takes two arguments, the first is the set of coordinates on the horizontal axis and the second is the corresponding set of coordinates on the vertical axis. The `plot` function then plots the first against the second to form a set of points and connects them with lines. In other words, `plot(x,y)` draws points at the coordinates $(x(1), y(1)) = (-2, -8)$, $(x(2), y(2)) = (-1, -1)$, $(x(3), y(3)) = (0, 0)$, etc. and draws a line that connects all these points in the order they appear in.

```
>> x=-2:1:2;
>> y=x.^3;
>> plot(x,y)
```



Clearly, 5 points is not enough to plot a function accurately, so the domain vector x must be made finer by choosing smaller increments by saying something like `x=-2:0.1:2` (in this case, `x=[-5 -4.9 -4.8 -4.7 ... 4.7 4.8 4.9 5]`). A very convenient way of achieving this is by using the `linspace` function where `linspace(a,b)` forms a vector between `a` and `b` with 100 equally spaced points. If a different mesh is required, then add an extra argument `n` as `linspace(a,b,n)`, this forms a vector between `a` and `b` consisting of `n` equally spaced points. Therefore, the range of x values can be refined as `x=linspace(-2,2)`.

```
>> x=linspace(-2,2);
>> y=x.^3;
>> plot(x,y)
```



Notice that the semicolons are placed since the output does not need to be seen and it is therefore suppressed, otherwise MATLAB will output all 100 terms of `x` and `y` which not necessary.

D.2 Line Properties

The `plot` function has many additional options that can change the plotting colour, shape, style, line widths and many more (these can be referred to by simply typing `help plot` into the command window). Some of these options can be incorporated into a plot by adding them into the `plot` function itself as additional inputs as `plot(x,y,'Color','r','LineStyle','-','LineWidth',2)`.

Some of the available colours are:

Colour	`'Color'` Syntax
red	`'r'`
blue	`'b'`
green	`'g'`
cyan	`'c'`
magenta	`'m'`
yellow	`'y'`
black	`'k'`
white	`'w'`

Some of the available line styles are:

Line Style	`'LineStyle'` Syntax
Solid	`'-'`
Dashed	`'--'`
Dotted	`':'`
Chain	`'-.'`

The colours and line styles can be combined into one, so if a blue solid line is needed, then it can simply be done by using `'-b'` and the plotting command will be `plot(x,y,'-b')`.

D.3 Multiple Plots

It would stand to reason that if two different functions are to be plotted on the same figure space, say $y = x^2$ as a red solid line and $z = x^3$ as a blue dashed line for $x \in [-5, 5]$, then the following commands can be executed:

```
>> x=linspace(-5,5);
>> y=x.^2;
>> z=x.^3;
>> plot(x,y,'-r')
>> plot(x,z,'--b')
```

Unfortunately, MATLAB has a habit of overwriting plots every time the `plot` command is used, so in this case MATLAB would plot the graph of `y` then remove it and plot the graph of `z`. In order to avoid that, typing `hold on` before any `plot` command allows plotting more than one plot in the same figure space as well allowing some augmentation. This can be reverted by `hold off`.

```
>> hold on
>> x=linspace(-5,5);
>> y=x.^2;
>> z=x.^3;
>> plot(x,y,'-r')
>> plot(x,z,'--b')
>> hold off
```

D.3.1 Legends

When there is more than one line plotted in the same figure space, it is useful to have a legend to distinguish between the different plots. So if the functions y and z are plotted as above, then a legend can be added that labels them by simply using `legend('Function y','Function z')`. This labels the first plot with `Function y` and the second with `Function z`. Remember,

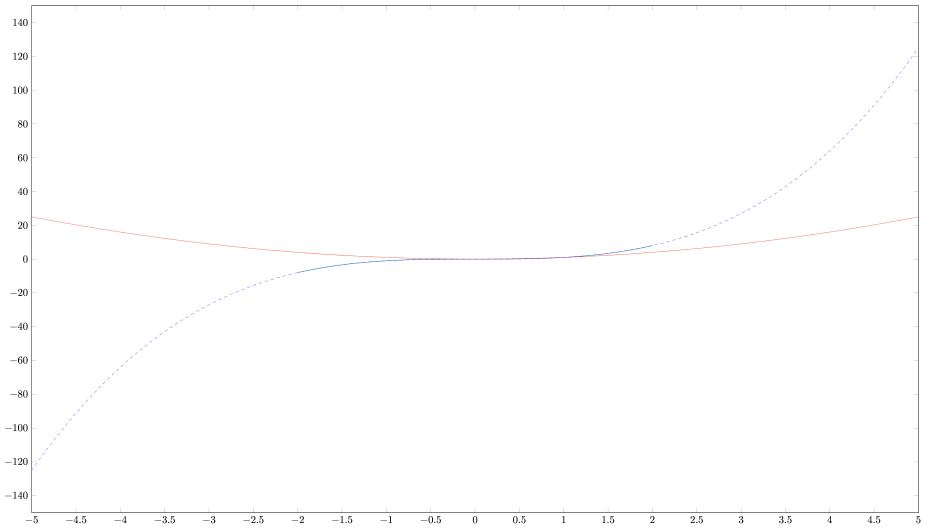
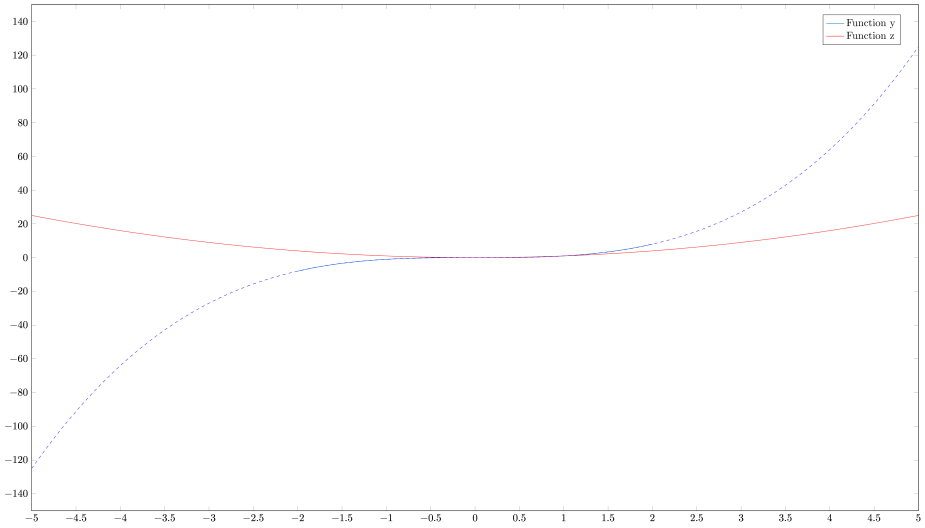


Figure D.1

quotation marks need to be inserted so they are displayed verbatim, otherwise MATLAB will produce an error since there are no variable with the names `Function y` or `Function z`.



D.4 Figure Properties

Some useful figure functions are:

- `clf`: Clears the figure space.
- `figure`: Opens a new figure window.
- `figure(n)`: Goes to figure window number n (and creates one if it is not open to begin with) and plots within that window.

The figures themselves can be augmented by introducing titles, grid lines and labelling the x- and y-axes, all these can be achieved as long as the `hold on` command is active:

- Title: `title('Put title here')`, the title must be in quotation marks.
- Grid: `grid on` and `grid off`.
- x-axis: `xlabel('Label for x axis')`.
- y-axis: `ylabel('Label for y axis')`.

MATLAB usually adjusts the axes so that the graphs fit but sometimes, the axes need to be readjusted according the user's preference, this can be done by using `axis([left right down up]` where `left` is the leftmost point, `right` is the rightmost point, etc.

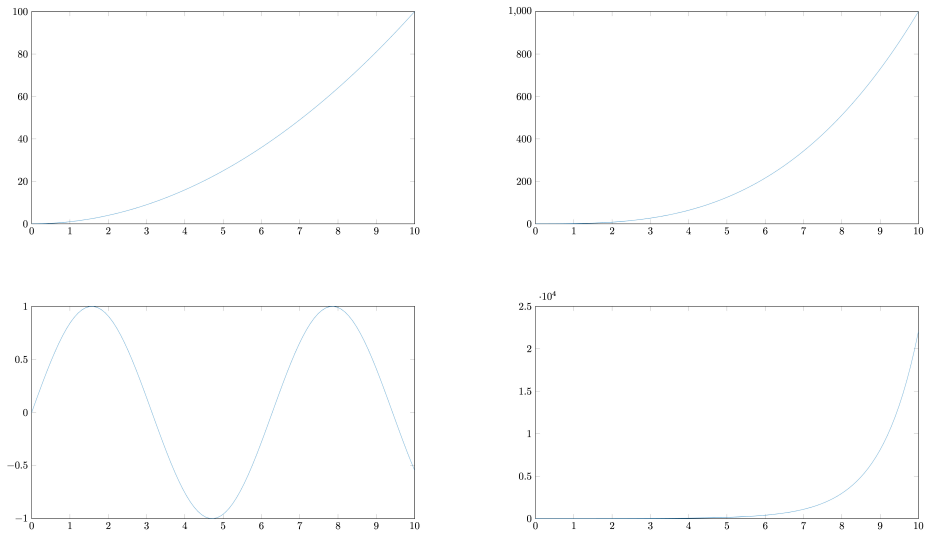
D.5 Subplots

Plotting multiple functions is very useful only if the axes can be maintained but if they are different, then the information can be quite distorted when interpreted graphically. In this case, subplots can be used to display more than one plot on the same figure space but on different sections. The command `subplot(a,b,n)` generates a grid of size $a \times b$ (a rows and b columns) and starts plotting in the n^{th} location where the top left is 1 and continues across the rows.

Suppose that for $x \in [0, 10]$, four functions are to be plotted: $y = x^2$ on the top left, $z = x^3$ on the top right, $w = \sin(x)$ on the bottom left and $u = \mathrm{e}^x$ on the bottom right. This means that a 2×2 grid is needed so the first two terms in `subplot` are 2. The function y has to be plotted after `subplot(2,2,1)` while z is to be plotted after `subplot(2,2,2)` and so on.

```
>> x=linspace(0,10);
>> y=x.^2;
>> z=x.^3;
>> w=sin(x);
>> u=exp(x);
>> subplot(2,2,1)
>> plot(x,y)
>> subplot(2,2,2)
>> plot(x,z)
>> subplot(2,2,3)
```

```
>> plot(x,w)
>> subplot(2,2,4)
>> plot(x,u)
```



One issue in this case is that all the subplots will behave independently, so turning on the grid in one subplot will not do the same for all the rest. Therefore, operations such as `grid on` and `hold on` need to be done for each of the subplots individually.

D.6 Aesthetics

Fonts in figures can usually be an issue since the default setting may not be to the user's liking. As seen in the figures above, the font on the axes is quite small which could make it difficult to read especially if the plots are to be in a report or dissertation. In that case, a special command needs to be run after `hold on` and before any plotting can commence. The command `set(gca,'FontSize',20,'FontName','Times')` sets the fontsize to 20 and the font to Times New Roman globally on all axes, legends and titles.

On MATLAB, the mathematical symbols will be displayed as regular text instead of mathematical symbols (like "x" instead of "x"). This can be adjusted by using LaTeX syntax by using dollar signs around the mathematical symbols. For example, the x- and y-axes can be labelled with "x" and "y" by using `xlabel('$x$','Interpreter','Latex')` and `ylabel('$y$','Interpreter','Latex')`. The same can be done in the title as `title('Plot of $x$ Against $y$','Interpreter','Latex')`.

The legend entries need slightly more work; if two functions y and z are plotted, then they can be labelled in maths typesetting by first defining legend in terms of a placeholder

variable as `Leg=legend('Function $y$','Function $z$')` then prescribing the interpreter as `set(Legend,'Interpreter')`. MATLAB usually places the legend on the top right corner by default but this can be modified by the `'Location'` argument and change it to `East`, `West`, `NorthEast`, `SouthWest` and so on, meaning that the new prescription for the legend would be `set(Legend,'Interpreter','Location','SouthWest')`.

Remember, this modification of font shapes, sizes and the different styles is only for aesthetic reasons and serves no purpose otherwise.

Lots of Plots

Suppose that the following need to be plotted:

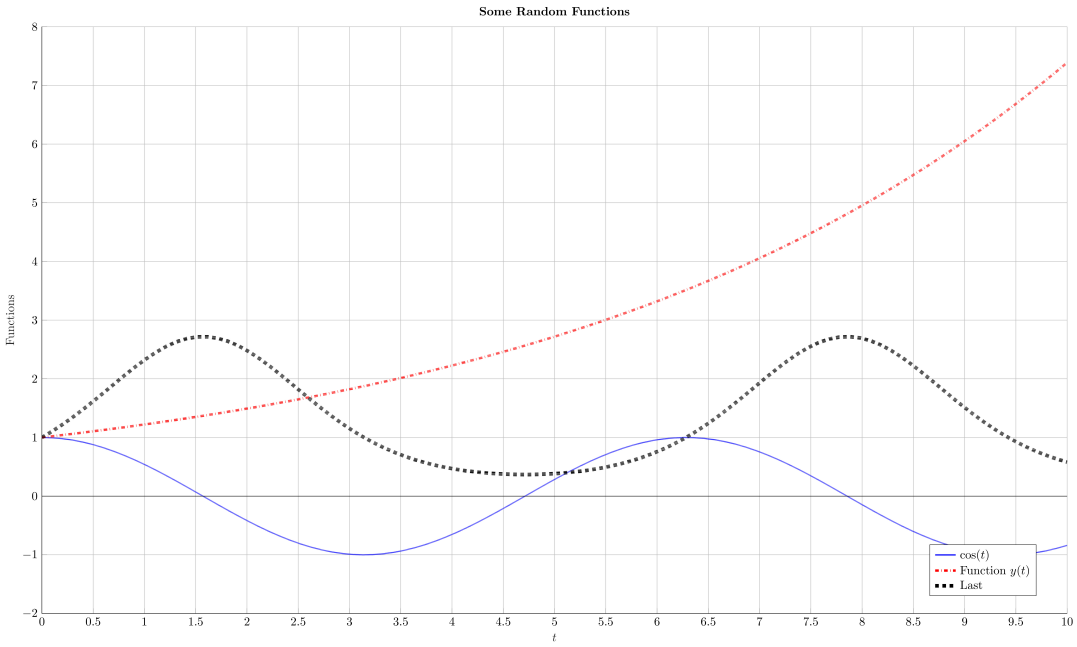
1. The function $x(t) = \cos(t)$ for $t \in [0, 10]$ as a blue solid line of thickness 1.
2. The function $y(t) = \mathrm{e}^{0.2t}$ for $t \in [0, 10]$ as a red chain of thickness 2.
3. The function $z(t) = \mathrm{e}^{\sin(t)}$ for $t \in [0, 10]$ as a black dashed line of thickness 3.
4. The legend appears in the bottom right corner and labels $x(t)$ as "$\cos(t)$", $y(t)$ as "Function $y(t)$" and $z(t)$ as "Last".
5. The title of the figure should be "Some Random Functions".
6. The horizontal axis labelled as "t".
7. The vertical axis labelled as "Functions".
8. The horizontal axis ranges from 0 to 10 and the vertical axis ranges from -2 to 8.
9. Axis lines are drawn to represent the horizontal and vertical axes.

Each of these can be executed separately by the following commands:

0. `t=linspace(0,10);`
1. `x=cos(t); plot(t,x,'-b','LineWidth',1)`
2. `y=exp(0.2*t); plot(t,y,'-.r','LineWidth',2)`
3. `z=exp(sin(t)); plot(t,z,'--k','LineWidth',3)`
4. `Leg=legend('$\cos(t)$','Function $y(t)$','Last');`
   `set(Leg,'Interpreter','Latex','Location','SouthEast')`
5. `title('Some Random Functions','Interpreter','Latex')`
6. `xlabel($t$,'Interpreter','Latex')`
7. `ylabel('Functions','Interpreter','Latex')`
8. `axis([0 10 -2 8])`
9. `plot([0 10],[0 0],'-k'); plot([0 0],[-2 8],'-k')`

A MATLAB script can be written to execute all these in order:

```
clf                 % Clears the figure before plotting

hold on             % Allows more than one plot in the same figure

grid on               % Produces a grid

set(gca,'FontSize',20,'FontName','Times')    % Sets the font golobally

t=linspace(0,10);   % Horizontal axis values

x=cos(t);           % Vector of values for the x function
y=exp(0.2*t);       % Vector of values for the y function
z=exp(sin(t));      % Vector of values for the z function

plot(t,x,'-b','LineWidth',1)    % Plots t against x
plot(t,y,'-.r','LineWidth',2)   % Plots t against y
plot(t,z,'--k','LineWidth',3)   % Plots t against z

title('Some Random Functions','Interpreter','Latex')   % Title

xlabel('$t$','Interpreter','Latex')          % Horizontal axis label
ylabel('Functions','Interpreter','Latex')   % Vertical axis label

axis([0 10 -2 8])           % Sets the axes
plot([0 10],[0 0],'-k')     % Plots the horizontal axis
plot([0 0],[-2 8],'-k')     % Plots the vertical axis

Leg=legend('$\cos(t)$','Function $y(t)$','Last');      % Sets the legend

set(Leg,'Interpreter','Latex','Location','SouthEast');  % Sets the font, interpreter and
```



All these commands can be executed in the command window rather than writing them in a script but if a mistake is made, then it cannot be undone and the entire stream of commands needs to be redone once again. Using a script on the other hand will allow for easy alteration.

D.7 Discrete Plots

The `plot` function does not just plot functions, all it needs are two vectors of the same length and it can plot them against one another. So if the graph is to be plotted as a series of points (discrete plot) rather than coordinates connected with a line, then the change in the `plot` function is quite straight forward, simply replace `'LineStyle'` with `'MarkerStyle'` and `'LineWidth'` with `'MarkerSize'`. This will use discrete points rather than connecting them with lines. The different marker styles are:

Marker Style	`'MarkerStyle'` Syntax
Dot ·	`'.'`
Cross ×	`'x'`
Asterisk ∗	`'*'`
Circle ∘	`'o'`
Crosshair +	`'+'`
Square □	`'s'`
Diamond ◇	`'d'`
Pentagram ⋆	`'p'`
Upward Triangle △	`'^'`

Marker Style	`'MarkerStyle'` Syntax
Downward Triangle ▽	`'v'`
Rightward Triangle ▷	`'>'`
Leftward Triangle ◁	`'<'`

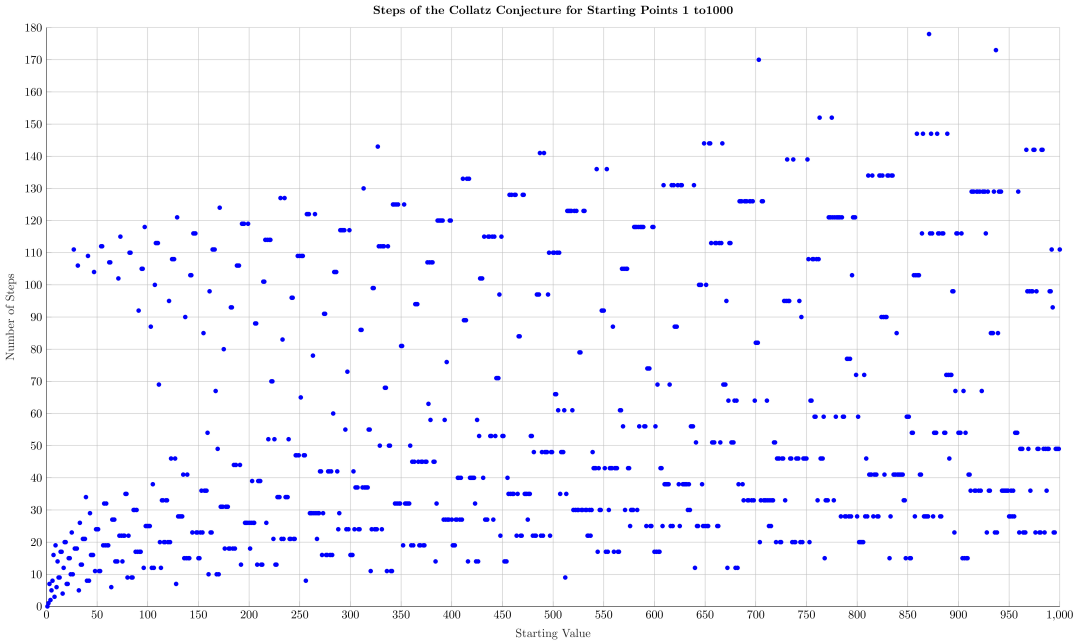
The colours work in the same way. These discrete plots can be combined with the line plot all in one command, for example, to plot a function with a red dashed line connecting circles, the plot command will be `plot(x,y,'--or')`.

Collatz Conjecture Plot

Consider to the Collatz conjecture from Section C.2, suppose that the number of steps it takes to reach 1 is to be plotted against the starting values, say from 1 to N where N will be the input. This will require the use of many of the tools developed so far.
First of all, a function that takes in a starting value and outputs the number of steps is needed, which that has already been done in the code `Collatz`. Since the inputs will be all the numbers from 1 to N, a `for` loop will be suitable for the job. Finally, the `plot` function with markers will be employed since connecting the points with lines will not make sense in this particular context.
In order for the `plot` function to work, it needs two vectors of the same length. For this particular example, the first vector is the list of numbers from 1 to N, which will be labelled `X` and will be on the x-axis, and the second is the vector of the number of steps for a starting value to decrease to 1 and this is labelled `Y`. The terms in the vector `Y` will have to be calculated individually by using the `Collatz` function. Of course, since the size of `Y` is the same size a `X`, it can be initialised by using `Y=zeros(size(X))`, the terms can then be substituted after they have been calculated. The code to execute this plotting procedure is as follows:

```
function Plot_Collatz(N)

X=1:1:N;            % List of starting values from 1 to N

Y=zeros(size(X));   % Initialise the vector Y

for i=X

     [y]=Collatz(i);     % Run the Collatz algorithm for the starting
                         % value i

    Y(i)=y;             % Record the the number of steps in the i-th
                        % element of the vector Y

end

clf
hold on
grid on
set(gca,'FontSize',20,'FontName','Times')

plot(X,Y,'.b','MarkerSize',10)

title(strcat('Steps of the Collatz Conjecture for Starting Points 1 to',' ',num2str(N)),

xlabel('Starting Value','Interpreter','Latex')
ylabel('Number of Steps','Interpreter','Latex')

end
```

The code can now be run in the command window using `Plot_Collatz(1000)` will give the following plot:



There are a few things that need to be observed in the above code:

- In Line 4, the `for` loop starts with `i=X`, this means that the values of `i` would run through all the values of the vector `X` in order. So the `for` loop does not need to take terms from a uniform set but it can be from any set of values and those will be taken in the order they appear.
- Line 6 runs the `Collatz` function for the input value `i` to produce a value `y` and this is then recorded in the vector `Y` in the i^{th} location in Line 8, hence `Y(i)=y`. Of course there will be no issues there since the size of `Y` is known and has already been initialised in Line 3 as a vector of zeros of the same size as `X`, the values are then replaced by the desired terms.
- Notice that here, the main function `Plot_Collatz` (also known as the *top level function*) refers to another function, namely `Collatz`. This code should be saved as a separate `.m` file and *has to be in the same directory* as `Plot_Collatz`, otherwise the code will not work. An alternative would be to put the `Collatz` function after the end of `Plot_collatz`.

```
function Plot_Collatz(N)

     Body of Plot_Collatz

end

function [n]=Collatz(a)

     Body of Collatz

end
```

- The `Collatz` function requires a single input, but in some cases, there could be many inputs and many outputs, in that case when calling the function, the sequence of inputs and outputs *must be in exactly the same order* as it appears in the function itself.

D.8 Plot Cheat Sheet

MATLAB Command	Purpose
`clf`	Clear figure space
`figure`	Opens a new figure space
`figure(n)`	Plots in figure space `n`
`hold on`	Allows more than one plot to be drawn on the same figure
`hold off`	Cancels `hold on`
`grid on`	Turns on the plot grid
`grid off`	Turns off the plot grid
`plot([a,b],[c,d])`	Plots a straight line from point (`a`,`c`) to (`b`,`d`)
`set(gca,'FontSize',20)`	Sets the global font size to 20
`set(gca,'FontName','Times')`	Sets the global font to Times
`axis([left right down up])`	Sets the axes where the x-axis goes from `left` to `right` and the y-axis from `down` to `up`
`title('Plot')`	Adds the title "Plot" to the figure
`xlabel('x')`	Labels the x-axis with "x"
`xlabel('$x$','Interpreter','Latex')`	Labels the x-axis with "x"
`Leg=legend('Plot 1','Plot 2',...)`	Gives the legend a handle "`Leg`" for further modification and labels the first plotted line as "Plot 1", the second as "Plot 2", etc.
`set(Leg,'Interpreter','Latex')`	Renders the legend in LaTeX, just like the labels

MATLAB Command	Purpose
`x=linspace(a,b)`	Generates a vector `x` with 100 points from `a` to `b`
`x=linspace(a,b,n)`	Generates a vector `x` with `n` points from `a` to `b`
`plot(x,y)`	Plots the vector `x` against the vector `y` as long as they are of the same size
`plot(x,y,'-b')`	Plots `x` against `y` with a blue line (continuous)
`plot(x,y,'-b','LineWidth',2)`	Plots `x` against `y` with a blue line of thickness 2
`plot(x,y,'xk')`	Plots `x` against `y` with black crosses (discrete)
`plot(x,y,'xk','MarkerSize',10)`	Plots `x` against `y` with black crosses of size 10

E Reading & Writing Data

Reading and writing data files can be important for importing data for analysis on MATLAB and exporting data for further processing elsewhere.

E.1 Writing Into Data Files

Data can be exported from MATLAB into a .dat or .txt file, both of which can be opened with *Notepad*.

Writing Data

Suppose that a list of values of x from 0 to 100 need to be exported along with a corresponding list of x^2, $\sin(x)$ and e^{-x} as seen here:

x	x^2	$\sin(x)$	e^x
1	1	0.84147	0.36788
2	4	0.90930	0.13534
⋮	⋮	⋮	⋮
99	9801	-0.9992	1.0112×10^{-43}
100	10000	-0.5063	3.7200×10^{-44}

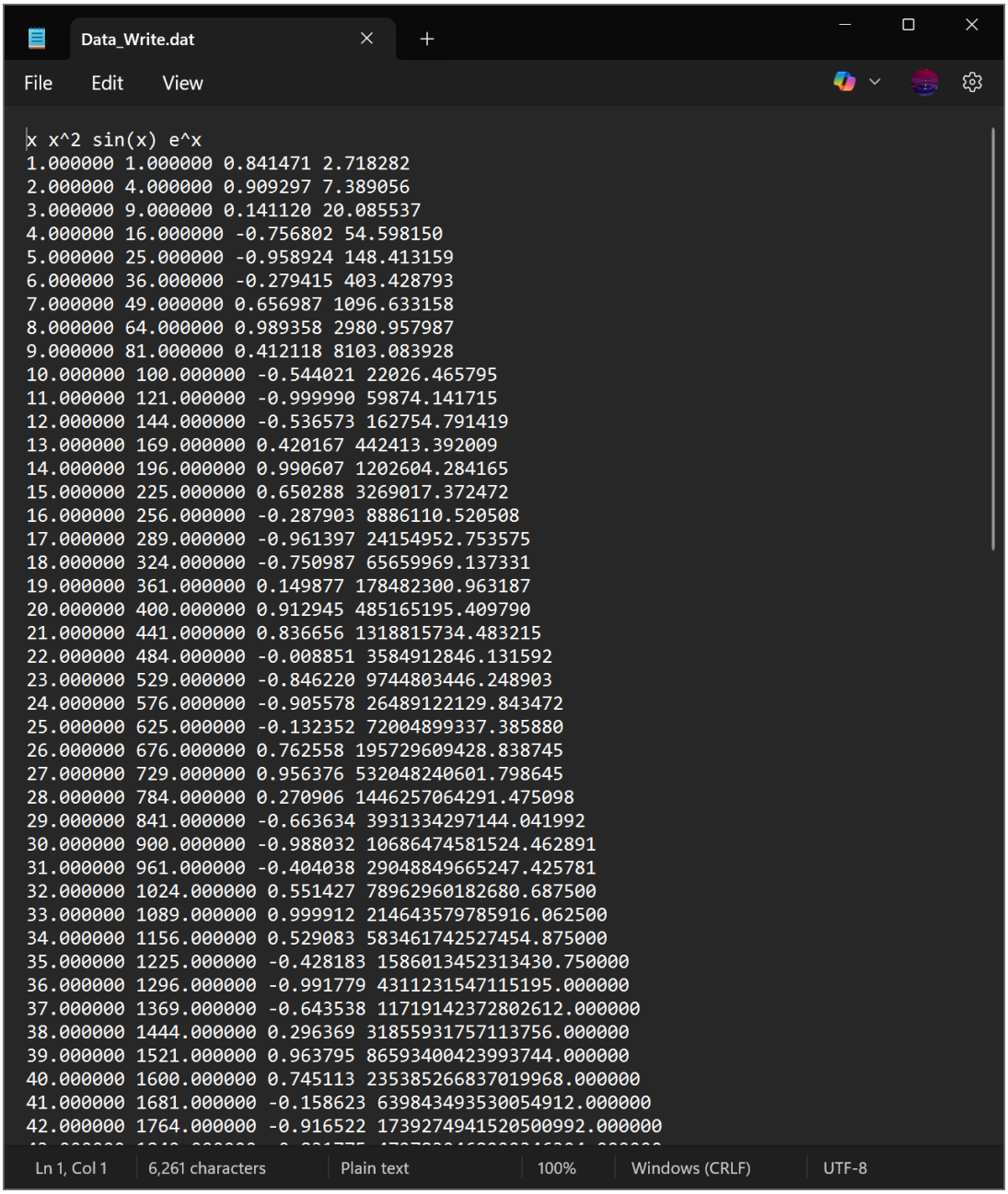
First, define each of these columns.

```
>> x=[1:1:100]';   % Column vector of values from 1 to 100

>> c1=x.^2;        % Column of x^2 terms

>> c2=sin(x);      % Column of sin(x) terms

>> c3=exp(x);      % Column of e^x terms

>> M=[x,c1,c2,c3]; % Form a matrix out of the columns
```

Now that the matrix is ready to be exported, a file needs to be opened with the desired name, say "Data_Write.dat" (.txt would also work). First, the file itself needs to be created in order to write the data into, this can be done by using `file_name=fopen('Data_Write.dat','w')`. The `'w'` indicates that MATLAB needs

to *write* the data into this file. The data can then be written into the the file using the `fprintf` command as `fprintf(file_name,'%f %f %f %f \r\n',M')`
The `%` sign determines the specification of the output and here, `%f` indicates that the output should be in the form of a floating point number. There are four columns so four specifiers need to be declared (hence `%f` appearing four times). The `\r\n` syntax indicates that MATLAB needs to move to the next line, otherwise, all the values will be printed on a single line (`\r\\n` needs to be used when opening using Microsoft Notepad, otherwise `\n` would suffice). The matrix is printed as `M'` instead of `M` since Notepad works on the reverse dimensions, so the rows on MATLAB are columns on Notepad and vice versa (for some obscure reason).
After writing all the data, the file needs to be closed so the data is not removed or overwritten using `fclose(file_name)`.
Without context, this data is meaningless so an additional row can be added before writing the data as a title for every column as `fprintf(file_name,'x  x^2  sin(x)  exp(x)  \r\n')`. All these can be combined into the following executable section:

```
x=[1:1:100]';       % Column vector of values from 1 to 100

c1=x.^2;            % Column of x^2 terms
c2=sin(x);          % Column of sin(x) terms
c3=exp(x);          % Column of e^x terms

M=[x,c1,c2,c3];     % Form a matrix out of the columns

my_file=fopen('Data_Write.dat','w');    % Open the file 'Data_Write.dat',
                                        % also works with 'Data_Write.txt'

fprintf(my_file,'x x^2 sin(x) e^x \r\n');
fprintf(my_file,'%f %f %f %f \r\n',M');

fclose(my_file);
```



E.1.1 Output Formats

When writing data, it is often times important to present the data in a certain form or with certain spacings. For example, $\sin(x)$ is better presented as a floating point and e^x is better presented in scientific notation. These can be done by changing the format after the `%` sign as follows:

Syntax	Display	Example
%f	Floating point	0.5 → 0.50000
%e	Scientific notation	pi → 3.1415e+00
%g	Floating Point with no trailing 0's	0.5000 → 0.5
%i	Integer	pi → 3

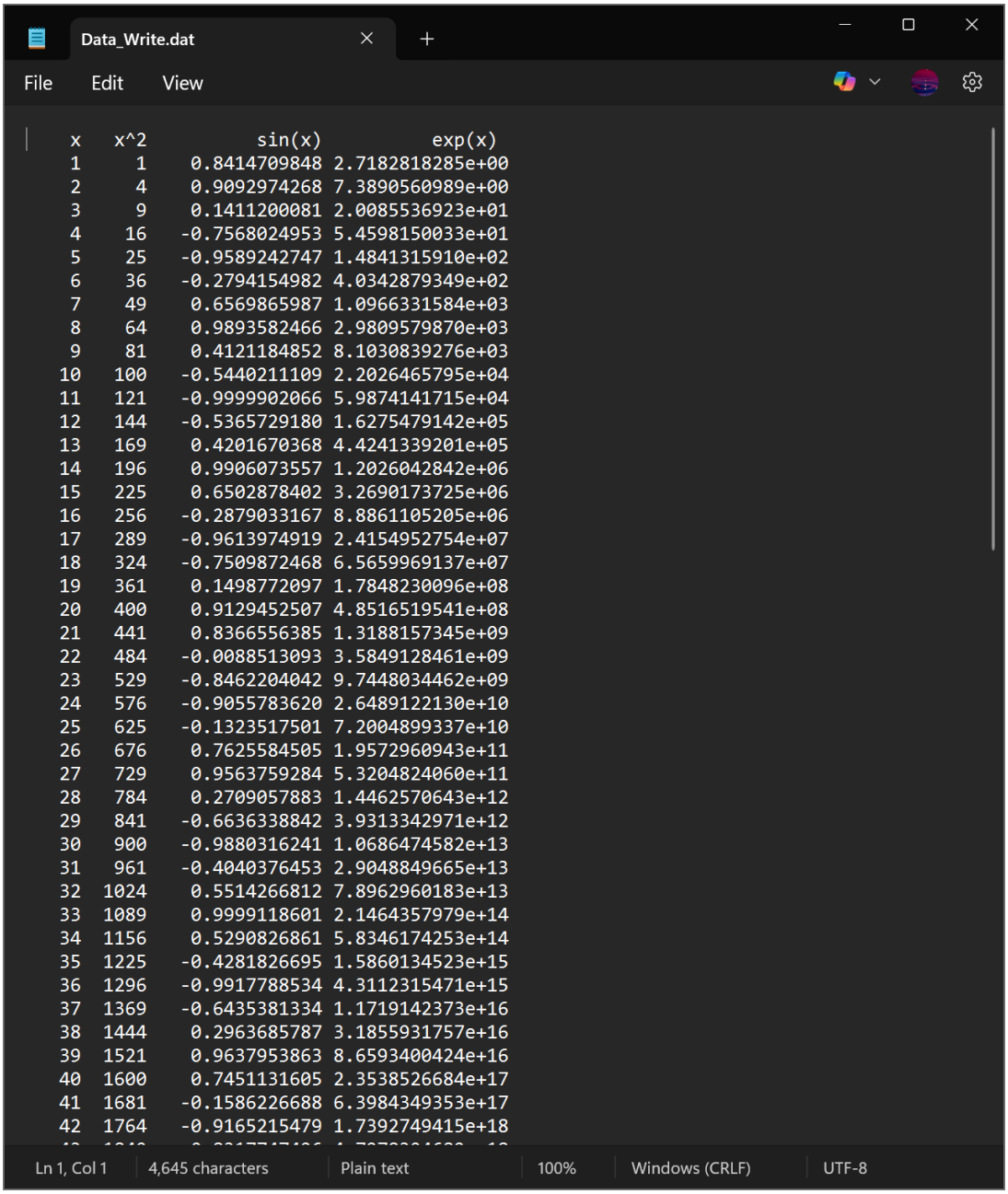
> **Note**
>
> There are many others that print numbers as strings (%s) or in hexadecimal notation (%x).

E.1.2 Alignment

The way in which the data is spaced out is important since it allows the data to be read more easily. By default, using %f will print the data as a floating point with six decimal places, one space will be added before the next item is printed. This can be changed to %15.10f which will print the data as a floating point but will dedicate 15 spaces to write the value to 10 decimal places.

> **Writing Better Data**
>
> The same code can be used as before with the alignment and decimal modifications.

```
>> x=[1:1:100]';

>> c1=x.^2;
>> c2=sin(x);
>> c3=exp(x);

>> M=[x,c1,c2,c3];

>> my_file=fopen('Data_Write.dat','w');

>> fprintf(my_file,'%5s %5s %15s %15s \r\n','x','x^2','sin(x)','exp(x)');

>> fprintf(my_file,'%5i %5i %15.10f %15.10e \r\n',M');

>> fclose(my_file);
```

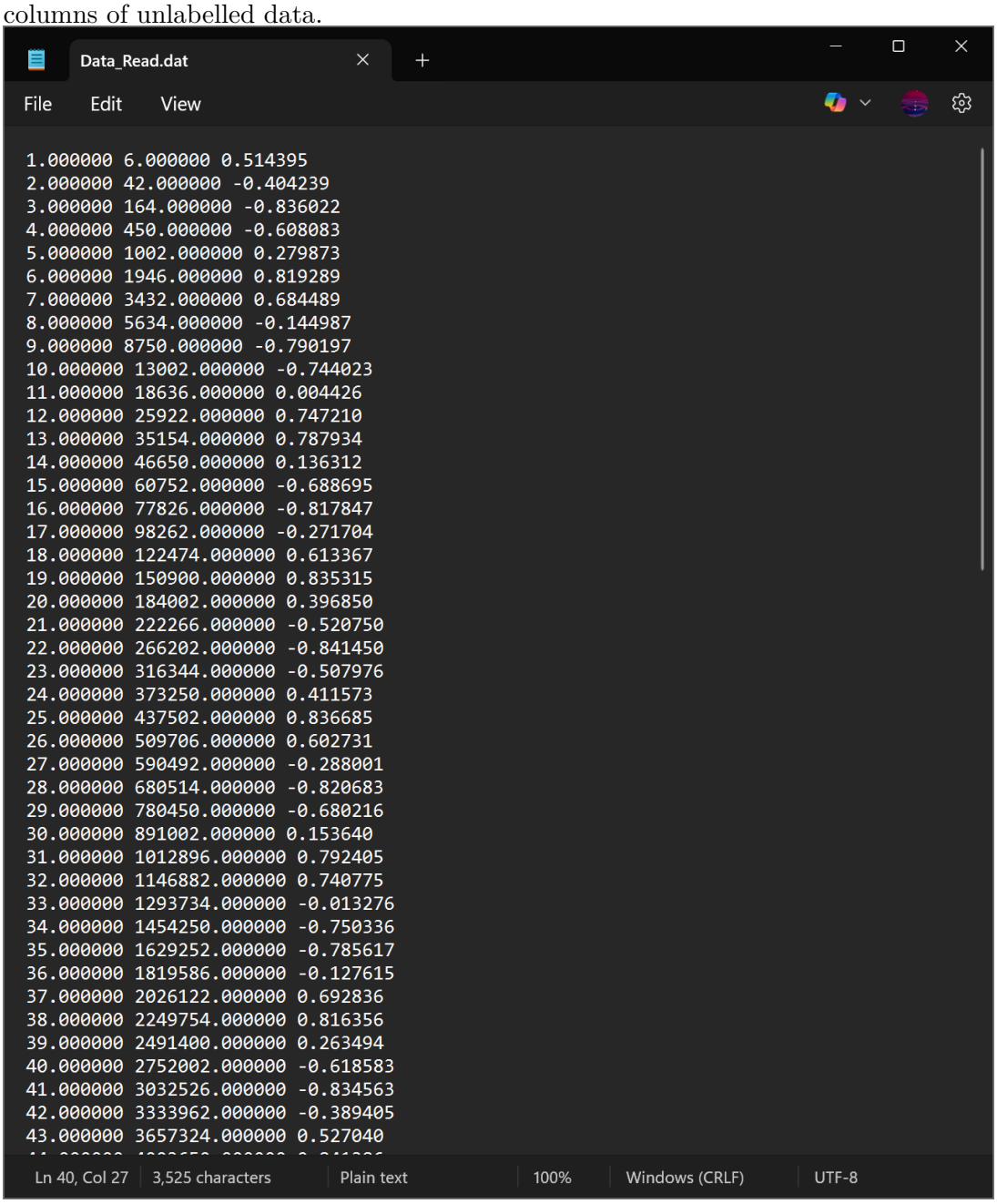


E.2 Reading From Data Files

Reading data from a .dat or .txt files is similar to writing.

Reading Data

Suppose that there is a data file called “Data_Read.dat” (or .txt) that has three columns of unlabelled data.



First, the file needs to be opened with `fopen` but in order to prepare it for reading, use the augmentation `'r'` (instead of `'w'` for writing). The format has to be specified, in this case, it would be `'%f %f %f'` since there are three terms that need to be read which are all placed into a row and separated by a space. The size of the data itself also needs to be specified as well, and since there are three columns, that could be

defined as `[3 Inf]` if the number of rows is unknown. The commands to read the data can be written as follows:

```
my_file=fopen('Data_Read.dat','r');

formatSpec = '%f %f %f';

Size_Data= [3 Inf];

M=fscanf(my_file,formatSpec,Size_Data);

M=M';

fclose(my_file);
```

This will produce an array `M` that contains all the data.

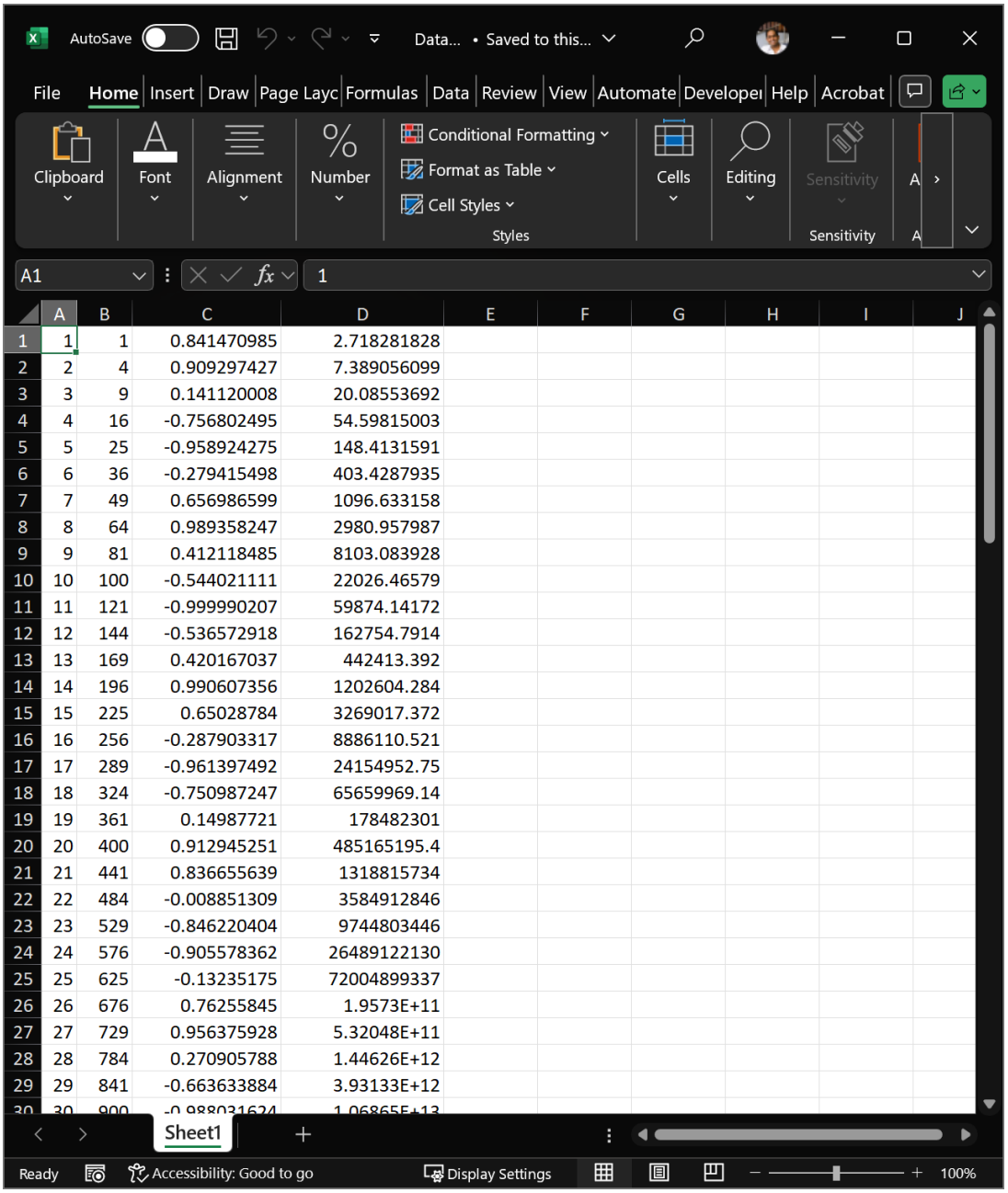
E.3 Reading & Writing Data with Excel

Writing data into Microsoft Excel is much simpler than .dat or .txt since spacing and formatting are built into excel. The difference is using **`writematrix`** and **`readmatrix`** instead of **`fprintf`** and **`fscanf`** and the file extension should be .xlsx and does not have to be opened and closed.

Reading & Writing with Excel

Suppose the data as before needs to be written into Excel, this can be done as follows:

```
x=[1:1:100]';

c1=x.^2;

c2=sin(x);

c3=exp(x);

M=[x,c1,c2,c3];

writematrix(M,'Data_Excel.xlsx');
```



	A	B	C	D
1	1	1	0.841470985	2.718281828
2	2	4	0.909297427	7.389056099
3	3	9	0.141120008	20.08553692
4	4	16	-0.756802495	54.59815003
5	5	25	-0.958924275	148.4131591
6	6	36	-0.279415498	403.4287935
7	7	49	0.656986599	1096.633158
8	8	64	0.989358247	2980.957987
9	9	81	0.412118485	8103.083928
10	10	100	-0.544021111	22026.46579
11	11	121	-0.999990207	59874.14172
12	12	144	-0.536572918	162754.7914
13	13	169	0.420167037	442413.392
14	14	196	0.990607356	1202604.284
15	15	225	0.65028784	3269017.372
16	16	256	-0.287903317	8886110.521
17	17	289	-0.961397492	24154952.75
18	18	324	-0.750987247	65659969.14
19	19	361	0.14987721	178482301
20	20	400	0.912945251	485165195.4
21	21	441	0.836655639	1318815734
22	22	484	-0.008851309	3584912846
23	23	529	-0.846220404	9744803446
24	24	576	-0.905578362	26489122130
25	25	625	-0.13235175	72004899337
26	26	676	0.76255845	1.9573E+11
27	27	729	0.956375928	5.32048E+11
28	28	784	0.270905788	1.44626E+12
29	29	841	-0.663633884	3.93133E+12

The same data can be read using `Data=readmatrix('Data_Excel.xlsx')`.

F Gaussian Elimination Method

The ***Gaussian Elimination Method*** is an algorithm that transforms the linear system $A\boldsymbol{x} = \boldsymbol{b}$ where $A \in \mathbb{C}^{N\times N}$ and $\boldsymbol{b} \in \mathbb{C}^N$ into an equivalent upper triangular system $U\boldsymbol{x} = \boldsymbol{g}$ after $N-1$ steps, where $U \in \mathbb{C}^{N\times N}$ is an upper triangular matrix and $\boldsymbol{g} \in \mathbb{C}^N$. This uses ***Elementary Row Operations*** (swapping rows, multiplying a row by a constant, adding two rows), after which point, the system $U\boldsymbol{x} = \boldsymbol{g}$ can solved by the backward substitution. Note that this method is possible when the elementary row operations are performed on both A and $\boldsymbol{b}$ simultaneously, so if rows i and j are swapped in A, the rows i and j must also be swapped in $\boldsymbol{b}$, simialry for the other operations.

The Gaussian elimination method can be performed as follows (the superscripts in brackets will be the step number):

> Parallel Example
>
> The algorithm will be explained and an example will be done in parallel to explain the steps with the matrix system $A\boldsymbol{x} = \boldsymbol{b}$ where
>
> $$A = \begin{pmatrix} 2 & -1 & 1 \\ -1 & 1 & 2 \\ 1 & 2 & -1 \end{pmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}.$$

1. **Establish the starting matrix**: If $a_{11} \neq 0$, then set $A^{(1)} = A$ and $\boldsymbol{b}^{(1)} = \boldsymbol{b}$ as

$$A^{(1)} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1j}^{(1)} & \dots & a_{1N}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & \dots & a_{2j}^{(1)} & \dots & a_{2N}^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{j1}^{(1)} & a_{j2}^{(1)} & \dots & a_{jj}^{(1)} & \dots & a_{jN}^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{N1}^{(1)} & a_{N2}^{(1)} & \dots & a_{Nj}^{(1)} & \dots & a_{NN}^{(1)} \end{pmatrix} \in \mathbb{R}^{N\times N} \quad \text{where} \quad a_{11}^{(1)} \neq 0$$

$$\text{and} \quad \boldsymbol{b}^{(1)} = \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} \\ \vdots \\ b_j^{(1)} \\ \vdots \\ b_N^{(1)} \end{pmatrix}.$$

If $a_{11} = 0$, then swap the first row with any other row whose first term is not zero and the result will be the starting matrix $A^{(1)}$.

$A^{(1)}$

$$A^{(1)} = A = \begin{pmatrix} 2 & -1 & 1 \\ -1 & 1 & 2 \\ 1 & 2 & -1 \end{pmatrix} \quad \text{and} \quad \boldsymbol{b}^{(1)} = \boldsymbol{b} = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}.$$

2. **Form the multiplier vector**: The desired outcome is to have the matrix A be upper triangular, i.e. all the terms below the diagonal should be 0. To achieve this, introduce a vector $\boldsymbol{m}_1$ of multipliers, whose i^{th} entry is given by

$$m_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}} \quad \text{for all} \quad i = 1, 2, \ldots, N,$$

hence the reason why the assumption $a_{11}^{(1)} \neq 0$ must be imposed. Essentially, the vector $\boldsymbol{m}_1$ is the first column of A divided the the first element of A.

$\boldsymbol{m}_1$

$$\boldsymbol{m}_1 = \frac{1}{a_{11}^{(1)}} \begin{pmatrix} a_{11}^{(1)} \\ a_{21}^{(1)} \\ a_{31}^{(1)} \end{pmatrix} = \begin{pmatrix} 1 \\ -\frac{1}{2} \\ \frac{1}{2} \end{pmatrix}$$

3. **Elimination terms in the first column**: For $j = 2, 3, \ldots, N$, multiply row 1 by $-m_{j1}$ and add it to row j to give the new row j:

$$\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \ldots & a_{1j}^{(1)} & \ldots & a_{1N}^{(1)} \\ a_{21}^{(1)} - m_{21}a_{11}^{(1)} & a_{22}^{(1)} - m_{21}a_{12}^{(1)} & \ldots & a_{2j}^{(1)} - m_{21}a_{1j}^{(1)} & \ldots & a_{2N}^{(1)} - m_{21}a_{1N}^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{j1}^{(1)} - m_{j1}a_{11}^{(1)} & a_{j2}^{(1)} - m_{j1}a_{12}^{(1)} & \ldots & a_{jj}^{(1)} - m_{j1}a_{1j}^{(1)} & \ldots & a_{jN}^{(1)} - m_{j1}a_{1N}^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{N1}^{(1)} - m_{N1}a_{11}^{(1)} & a_{N2}^{(1)} - m_{N1}a_{12}^{(1)} & \ldots & a_{Nj}^{(1)} - m_{N1}a_{1j}^{(1)} & \ldots & a_{NN}^{(1)} - m_{N1}a_{1N}^{(1)} \end{pmatrix}.$$

Row $\boldsymbol{j}$ Operations

$$\begin{pmatrix} 2 & -1 & 1 \\ -1 & 1 & 2 \\ 1 & 2 & -1 \end{pmatrix}$$

$$\xrightarrow[r_2\to-\left(-\frac{1}{2}\right)r_1+r_2]{} \begin{pmatrix} 2 & -1 & 1 \\ -\left(-\frac{1}{2}\right)(2)-1 & -\left(-\frac{1}{2}\right)(-1)+1 & -\left(-\frac{1}{2}\right)(1)+2 \\ 1 & 2 & -1 \end{pmatrix}$$

$$= \begin{pmatrix} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ 1 & 2 & -1 \end{pmatrix}$$

$$\xrightarrow[r_3\to-\left(\frac{1}{2}\right)r_1+r_3]{} \begin{pmatrix} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ -\left(\frac{1}{2}\right)(2)+1 & -\left(\frac{1}{2}\right)(-1)+2 & -\left(\frac{1}{2}\right)(1)-1 \end{pmatrix}$$

$$= \begin{pmatrix} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ 0 & \frac{5}{2} & -\frac{3}{2} \end{pmatrix}$$

Notice that by the definition of m_{j1}, the first element in every row must be equal to 0, therefore, this set of operation makes all the terms in the first column equal to 0 except the first. Define this new matrix as the second term in the iteration:

$$\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1j}^{(1)} & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(1)} - m_{21}a_{12}^{(1)} & \dots & a_{2j}^{(1)} - m_{21}a_{1j}^{(1)} & \dots & a_{2n}^{(1)} - m_{21}a_{1n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & a_{j2}^{(1)} - m_{j1}a_{12}^{(1)} & \dots & a_{jj}^{(1)} - m_{j1}a_{1j}^{(1)} & \dots & a_{jn}^{(1)} - m_{j1}a_{1n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(1)} - m_{n1}a_{12}^{(1)} & \dots & a_{nj}^{(1)} - m_{n1}a_{1j}^{(1)} & \dots & a_{nn}^{(1)} - m_{n1}a_{1n}^{(1)} \end{pmatrix}$$

$$\implies \begin{pmatrix} a_{11}^{(2)} & a_{12}^{(2)} & \dots & a_{1j}^{(2)} & \dots & a_{1n}^{(2)} \\ a_{21}^{(2)} & a_{22}^{(2)} & \dots & a_{2j}^{(2)} & \dots & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{j1}^{(2)} & a_{j2}^{(2)} & \dots & a_{jj}^{(2)} & \dots & a_{jn}^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{n1}^{(2)} & a_{n2}^{(2)} & \dots & a_{nj}^{(2)} & \dots & a_{nn}^{(2)} \end{pmatrix} = A^{(2)}$$

where for all $i, j = 2, 3, \dots, N$

$$a_{11}^{(2)} = a_{11}^{(1)} \quad ; \quad a_{1i}^{(2)} = a_{1i}^{(1)} \quad ; \quad a_{i1}^{(2)} = 0 \quad ; \quad a_{ij}^{(2)} = a_{ij}^{(1)} - m_{i1}a_{1j}^{(1)}$$

🔥 $A^{(2)}$

$$A^{(2)} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ 0 & \frac{5}{2} & -\frac{3}{2} \end{pmatrix}$$

4. **Modification of the right hand side**: The vector $\boldsymbol{b}$ has to also undergo the same operations as A, i.e. for $j = 2, \ldots, N$, let row j of $\boldsymbol{b}^{(1)}$ be row 1 multiplied by $-m_{j1}$ plus row j and the final vector is the vector $\boldsymbol{b}^{(2)}$.

🔥 $\boldsymbol{b}^{(1)}$

$$\boldsymbol{b}^{(1)} = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix} \xrightarrow[\substack{r_2 \to -\left(-\frac{1}{2}\right) r_1 + r_2 \\ r_3 \to -\left(\frac{1}{2}\right) r_1 + r_3}]{} \begin{pmatrix} 1 \\ -\left(-\frac{1}{2}\right)(1) + 1 \\ -\left(\frac{1}{2}\right)(1) + 2 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{3}{2} \\ \frac{3}{2} \end{pmatrix} = \boldsymbol{b}^{(2)}.$$

5. **Matrix representation of elimination**: This whole procedure can be written as $A^{(2)} = M^{(1)} A^{(1)}$ and $\boldsymbol{b}^{(2)} = M^{(1)} \boldsymbol{b}^{(1)}$ where

$$M^{(1)} = \begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ -m_{21} & 1 & 0 & \ldots & 0 \\ -m_{31} & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -m_{n1} & 0 & 0 & \ldots & 1 \end{pmatrix}.$$

🔥 $M^{(1)}$

$$M^{(1)} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix}$$

To check:

$$M^{(1)} A^{(1)} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & 1 \\ -1 & 1 & 2 \\ 1 & 2 & -1 \end{pmatrix} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ 0 & \frac{5}{2} & -\frac{3}{2} \end{pmatrix} = A^{(2)}$$

$$M^{(1)} \boldsymbol{b}^{(1)} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{3}{2} \\ \frac{3}{2} \end{pmatrix} = \boldsymbol{b}^{(2)}$$

6. **Repeat for other columns**: The process must now be repeated for the rest of the rows, specifically, those that have non-zero pivot points, i.e. the first point in a row

that is non-zero. This process can be done more simply by generating the M matrices in the same way as before without going through the starting steps. This process should be reapeated until the last row is reached.

Multiplier Matrices

The matrix $M^{(2)}$ can be generated in the same way as $M^{(1)}$, so

$$M^{(2)} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -5 & 1 \end{pmatrix}.$$

To check:

$$M^{(2)}A^{(2)} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -5 & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ 0 & \frac{5}{2} & -\frac{3}{2} \end{pmatrix} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ 0 & 0 & -14 \end{pmatrix} = A^{(3)}$$

$$M^{(2)}\boldsymbol{b}^{(2)} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -5 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ \frac{3}{2} \\ \frac{3}{2} \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{3}{2} \\ -6 \end{pmatrix} = \boldsymbol{b}^{(3)}$$

7. **Solve using backwards substitution**: After repeating for all other columns (a total of $N-1$ times), the final matrix $A^{(N)}$ will be an upper triangular matrix with non-zero terms on the diagonal and the system can then be solved by backwards substitution.

Backwards Substitution

$$A^{(1)}\boldsymbol{x} = \boldsymbol{b}^{(1)} \implies A^{(2)}\boldsymbol{x} = \boldsymbol{b}^{(2)} \implies A^{(3)}\boldsymbol{x} = \boldsymbol{b}^{(3)}$$

$$\implies \begin{pmatrix} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ 0 & 0 & -14 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{3}{2} \\ -6 \end{pmatrix} \implies \begin{matrix} 2x_1 - x_2 + x_3 = 1 \\ \frac{1}{2}x_2 + \frac{5}{2}x_3 = \frac{3}{2} \\ -14x_3 = -6 \end{matrix}$$

$$\implies \boldsymbol{x} = \frac{1}{7}\begin{pmatrix} 5 \\ 6 \\ 3 \end{pmatrix}.$$

The total number of operations in every step is given in the table below (the "steps" here refer to the matrix manipulation step and not exactly to the step numbers of the algorithm):

Step	Multiplications	Additions	Divisions
1	$(N-1)^2$	$(N-1)^2$	$N-1$
2	$(N-2)^2$	$(N-2)^2$	$N-2$
3	$(N-3)^2$	$(N-3)^2$	$N-3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Step	Multiplications	Additions	Divisions
$N-2$	4	4	2
$N-1$	1	1	1

This means that the total number of multiplications is

$$1+4+\cdots+(N-3)^2+(N-2)^2+(N-1)^2=\sum_{n=1}^{N-1}n^2=\frac{N(N-1)(2N-1)}{6},$$

similarly for the additions. Whereas the total number of divisions is

$$1+2+\cdots+(N-3)+(N-2)+(N-1)=\sum_{n=1}^{N-1}n=\frac{N(N-1)}{2}.$$

Therefore the total number of operations is

$$\frac{N(N-1)(2N-1)}{6}+\frac{N(N-1)(2N-1)}{6}+\frac{N(N-1)}{2}=\frac{2}{3}N^3-\frac{1}{2}N^2-\frac{1}{6}N.$$

This means that for large N, the Gaussian elimination algorithm requires $\mathcal{O}\left(\frac{2}{3}N^3\right)$ operations when A is a non-sparse matrix. This procedure is computationally expensive even for moderate sized matrices, this also assumes that the pivot points are non-zero, or more specifically, that the matrix has non-zero determinant. As an illustration of this computational complexity, if $N=10^6$ (which not atypical), then for a computer with the computing power of 1 Gigaflops per second, an $N\times N$ system will need 21 years to find a solution. A lot of more modern computational techniques are based on attempting to reduce this computational complexity, either by eliminating terms in some suitable way or chnaging the matrix in a more pallatable form.

Overall, every step of this process can be represented by a matrix transformation $M^{(n)}$. This means that in order to convert the matrix A into an upper triangular matrix U, the matrix transformations $M^{(1)}, M^{(2)}, \ldots, M^{(N-1)}$ have to be applied reverse order as

$$U=M^{(N-1)}M^{(N-2)}\ldots M^{(1)}A.$$

This can be written as

$$U=MA \quad \text{where} \quad M=M^{(N-1)}M^{(N-2)}\ldots M^{(1)}. \tag{F.1}$$

Notice that every matrix $M^{(n)}$ is lower triangular and this fact will be used later on in Section G.1.

G Matrix Decompositions

Consider the linear system

$$A\boldsymbol{x} = \boldsymbol{b} \quad \text{where} \quad A \in \mathbb{R}^{N\times N}, \quad \boldsymbol{x} \in \mathbb{R}^N, \quad \boldsymbol{b} \in \mathbb{R}^N.$$

In real-life situations, the matrix A does not always have a form that lends itself to being easily solvable (like a diagonal, triangular, sparse, etc.). However, there are ways in which a matrix can be broken down into several matrices, each of which can be dealt with separately and reducing computational time.

G.1 LU Factorisation

Returning to the Gaussian elimination procedure outlined in Appendix F, in order to convert the linear system $A\boldsymbol{x} = \boldsymbol{b}$ into the upper triangular matrix system $U\boldsymbol{x} = \boldsymbol{g}$, a series of transformations had to be done, each represented by a matrix $M^{(n)}$ giving the form

$$U = MA \quad \text{where} \quad M = M^{(N-1)}M^{(N-2)}\dots M^{(1)}.$$

For every $n = 1, 2, \dots, N-1$, the matrix $M^{(n)}$ is lower triangular which means that the product of all these matrices M must also be lower triangular. Note that since $M^{(n)}$ and M are both non-singular and lower triangular, then their inverses must also be lower triangular.

This means that the matrix A can be written as $A = LU$ where $L = M^{-1}$ is a lower triangular matrix and U is an upper triangular matrix. In fact U is the end result of the Gaussian elimination, while L has ones on the diagonal and the multipliers below the diagonal. This method is called the ***LU Decomposition*** of A.

In the cases when there might be pivoting issues (which is when the pivot points might be equal to 0 during the Gaussian Elimination), the LU decomposition will more precisely be the ***PLU Decomposition*** (or the ***LU Decomposition with Partial Pivoting***) where the method will produce an additional permutation matrix P where $PA = LU$. This matrix P will swap rows when needed in order to have non-zero pivot points and is in fact orthogonal (i.e. $P^{-1} = P^{\mathrm{T}}$).

The LU decomposition can be used to solve the linear system $A\boldsymbol{x} = \boldsymbol{b}$ by splitting the matrix A into two matrices with more manageable forms. Indeed, since $A = LU$, then the system becomes $LU\boldsymbol{x} = \boldsymbol{b}$, this can be solved as follows:

- Solve the lower triangular system $L\boldsymbol{y} = \boldsymbol{b}$ for $\boldsymbol{y}$ using forward substitution;

- Solve the upper triangular $U\boldsymbol{x} = \boldsymbol{y}$ for $\boldsymbol{x}$ using backwards substitution.

This is a much better way of solving the system since both equations involve a triangular matrix and this requires $\mathcal{O}(N^2)$ computations (forward and backward substitutions), which is much cheaper computationally compared to the full Gaussian elimination.

The advantage of using the LU decomposition is that if problems of the form $A\boldsymbol{x}_i = \boldsymbol{b}_i$ need to be solved with many different right hand sides $\boldsymbol{b}_i$ and a fixed A, then only one LU decomposition is needed, and the cost for solving the individual systems is only the repeated forward and back substitutions. Note that there are other strategies optimised for specific cases (i.e. symmetric positive definite matrices, banded matrices, tridiagonal matrices).

In MATLAB, the LU decomposition can be done by a simple `lu` command:

```
>> A=[5,0,1;1,2,1;2,1,1];
>> [L,U]=lu(A)
L =
     1.0000         0         0
     0.2000    1.0000         0
     0.4000    0.5000    1.0000
U =

     5.0000         0    1.0000
          0    2.0000    0.8000
          0         0    0.2000
>> L*U-A       % Verify if LU is equal to A
ans =
     0    0    0
     0    0    0
     0    0    0
```

Note that if the output for `L` is *not* lower triangular, that means there are some pivoting issues that had to be overcome and `L` had to change to accommodate for that to maintain the fact that $A = LU$. In this case, the PLU decomposition would be better suited to avoid that, this is done by adding one extra output to the `lu` command, in this case, A will actually be the product $A = P^{\mathrm{T}}LU$.

```
>>> A=[1,0,1;1,0,1;2,1,1];
>> [L,U]=lu(A)
L =
     0.5000    1.0000    1.0000
     0.5000    1.0000         0
     1.0000         0         0
U =

     2.0000    1.0000    1.0000
```

```
          0   -0.5000    0.5000
          0         0         0
>> L*U-A       % Verify if LU is equal to A even though
              % L is not lower triangular
ans =
     0   0   0
     0   0   0
     0   0   0
>> [L,U,P]=lu(A)
L =
     1.0000         0         0
     0.5000    1.0000         0
     0.5000    1.0000    1.0000
U =

     2.0000    1.0000    1.0000
          0   -0.5000    0.5000
          0         0         0
P =
     0   0   1
     0   1   0
     1   0   0
>> L*U-A       % Verify if P'LU is equal to A
ans =
     0   0   0
     0   0   0
     0   0   0
```

G.2 Orthogonality & QR Factorisation

Intuitively, the concept of orthogonality is crucial for defining the "amount of information" in a set of vectors; although this is also associated with the concept of linear independence, the "most informative" linearly independent vectors are those that are also orthogonal.

Recall that for a set of vectors $\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_M \in \mathbb{R}^N$ where $M \leq N$, the vectors are ***Orthogonal*** if $\langle \boldsymbol{q}_m, \boldsymbol{q}_n \rangle = 0$ for all $m \neq n$. The set of vectors is called ***Orthonormal*** if

$$\langle \boldsymbol{q}_m, \boldsymbol{q}_n \rangle = \delta_{mn} = \begin{cases} 0 & \text{if} \quad m \neq n \\ 1 & \text{if} \quad m = n. \end{cases}$$

If $N = M$, then the vectors form a linearly independent basis of $\mathbb{R}^N$.

A square matrix Q is called ***Orthogonal*** if all its columns are orthonormal to one another. Some of the properties of orthogonal matrices are:

- An orthogonal matrix Q satisfies $Q^{-1} = Q^{\mathrm{T}}$, therefore $Q^{\mathrm{T}}Q = QQ^{\mathrm{T}} = \mathcal{I}$;
- The determinant of an orthogonal matrix is 1 or -1;
- The product of two orthogonal matrices is orthogonal.
- Given a matrix $Q_1 \in \mathbb{R}^{M\times K}$ with $K < M$ and with orthonormal columns, there exists a matrix $Q_2 \in \mathbb{R}^{M\times(M-K)}$ such that $Q = [Q_1, Q_2]$ is orthogonal. In other words, for a "tall" rectangular matrix with orthonormal columns, there exist a set of vectors that can be concatenated with the matrix to form an orthogonal square matrix.
- Orthogonal matrices preserve the 2-norm of vectors and matrices. In other words, if $Q \in \mathbb{R}^{N\times N}$ is an orthogonal matrix, then for every $\boldsymbol{x} \in \mathbb{R}^N$ and $A \in \mathbb{R}^{N\times M}$:

$$\|Q\boldsymbol{x}\|_2 = \|\boldsymbol{x}\|_2 \quad ; \quad \|QA\|_2 = \|A\|_2.$$

There are two particularly relevant classes of orthogonal matrices:

- The ***Householder Reflection Matrix*** (named after Alston Scott Householder) is a reflection matrix on a plane that contains the origin. The reflection matrix is given by

$$P = \mathcal{I} - 2\boldsymbol{v}\boldsymbol{v}^{\mathrm{T}}$$

  where $\boldsymbol{v}$ is the unit vector that is normal to the hyperplane in which the reflection has been performed. The matrix P is in fact symmetric and orthogonal (i.e. $P^{-1} = P^{\mathrm{T}} = P$). Reflection transformations appear in many numerical linear algebra algorithms and their main use is to transform a vector $\boldsymbol{x} \in \mathbb{R}^N$ to another vector $\boldsymbol{y} \in \mathbb{R}^N$ with the same magnitude (meaning that for given vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^N$ with $\|\boldsymbol{x}\|_2 = \|\boldsymbol{y}\|_2$, there exists a reflection matrix P such that $P\boldsymbol{x} = \boldsymbol{y}$).

- The ***Givens Rotation Matrix*** (named after James Wallace Givens) represents a rotation in the plane that can be spanned by two vectors. The matrix of transformation is denoted $G(i, j; \theta)$ where the vector $G(i, j; \theta)\boldsymbol{x}$ is simply the vector $\boldsymbol{x}$ rotated θ radians anti-clockwise on a plane that is parallel to the (i, j)-plane. The matrix $G(i, j; \theta)$ is essentially an identity matrix with the (i, i) and (j, j) terms replaced by $\cos(\theta)$, the (i, j) term replaced by $\sin(\theta)$ and the (j, i) term replaced by $-\sin(\theta)$. For example, in $\mathbb{R}^5$, the matrix $G(2, 4; \theta)$ is

$$G(2,4;\theta) = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & \sin(\theta) & 0 & \cos(\theta) & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure G.1: Photo of (from the left): Jim Wilkinson, Wallace Givens, George Forsythe, Alston Householder, Peter Henrici, Fritz Bauer

Since both reflection and rotation matrices are orthogonal matrix transformations, a sequence of reflections and rotations can be represented by the matrix Q^{T} (which would also be orthogonal). To this end, any matrix $A \in \mathbb{R}^{M \times N}$ with $M \geq N$ can be transformed by $Q^{\mathrm{T}} \in \mathbb{R}^{M \times M}$ to give a block matrix with an upper triangular matrix occupying the first N rows with $M - N$ zero rows below it, i.e.

$$Q^{\mathrm{T}} A = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$$

where $R_1 \in \mathbb{R}^{N \times N}$ is an upper triangular square matrix. Equivalently, A can be written as $A = QR$ where Q is the orthogonal transformation matrix and R is a block rectangular matrix consisting of a square lower triangular matrix and a block zero matrix. This type of decomposition is called the ***QR Factorisation***. The full QR factorisation can be visually represented as follows:

There is a much more concise form of the QR factorisation where only the first several columns of Q are considered since the rest will be multiplied by 0 anyway, this gives an "economy version" of the QR factorisation written as $A = Q_1 R_1$ which be visually represented as follows:

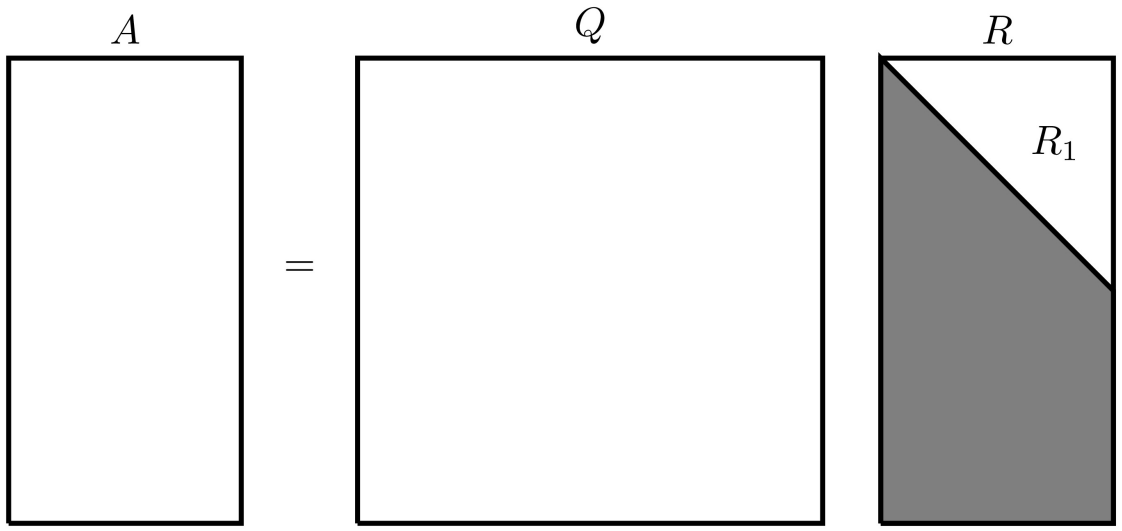


Figure G.2

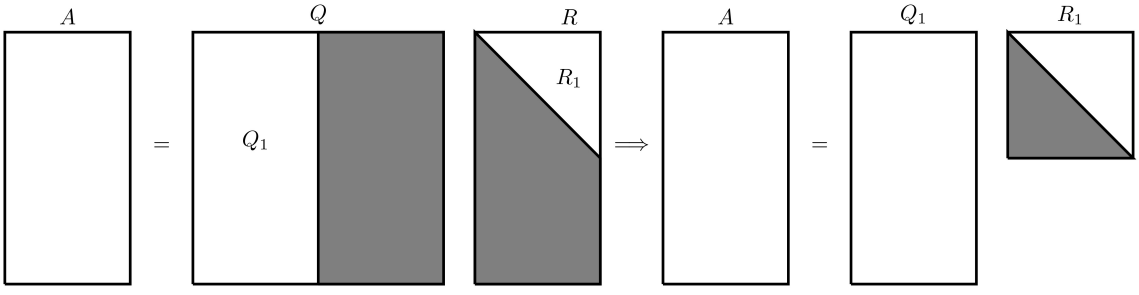


Figure G.3

The QR decomposition of a matrix can be performed on any matrix (square or rectangular). The following sections will show how this can be done using reflections and rotations.

G.2.1 QR Decomposition Using Reflections

The following will explain how the QR decomposition can be performed using reflection matrices on a square matrix $A \in \mathbb{R}^{N\times N}$. Denote the n^{th} column of the matrix A by $\boldsymbol{a}_n$, this means A can be written as

$$A = \begin{pmatrix} \vdots & \vdots & & \vdots \\ \boldsymbol{a}_1 & \boldsymbol{a}_2 & \dots & \boldsymbol{a}_N \\ \vdots & \vdots & & \vdots \end{pmatrix}.$$

The vector $\mathbf{e}_n$ will denote the n^{th} canonical basis vector, i.e. the vector with all its entries being equal to 0 except the element in location n which is equal to 1.

Paralell Example

This process will also be applied in parallel to the following matrix

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

In this case,

$$\boldsymbol{a}_1 = \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix}, \quad \boldsymbol{a}_2 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \quad \text{and} \quad \boldsymbol{a}_3 = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix}.$$

First, find a reflection matrix that transforms the first column of A into $(\alpha, 0, \dots, 0)^{\mathrm{T}}$ where $\alpha = \|\boldsymbol{a}_1\|_2$. Let $\boldsymbol{u} = \boldsymbol{a}_1 - \alpha\mathbf{e}_1$ and $\boldsymbol{v} = \frac{\boldsymbol{u}}{\|\boldsymbol{u}\|_2}$, then the first reflection matrix is

$$P_1 = \mathcal{I} - 2\boldsymbol{v}\boldsymbol{v}^{\mathrm{T}}.$$

This can be verified by checking that all the terms in the first column of the matrix $A_2 = P_1 A$ are zero except for the first term.

First Reflection Matrix

The 2-norm of the first column of A is $\alpha = \sqrt{3}$, then

$$\boldsymbol{u} = \boldsymbol{a}_1 - \alpha\mathbf{e}_1 = \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix} - \sqrt{3}\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} -1-\sqrt{3} \\ 1 \\ 1 \end{pmatrix}$$

$$\boldsymbol{v} = \frac{\boldsymbol{u}}{\|\boldsymbol{u}\|} = \frac{1}{\sqrt{6+2\sqrt{3}}}\begin{pmatrix} -1-\sqrt{3} \\ 1 \\ 1 \end{pmatrix}.$$

$$P_1 = \mathcal{I} - 2\boldsymbol{v}\boldsymbol{v}^{\mathrm{T}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \frac{2}{6+2\sqrt{3}} \begin{pmatrix} -1-\sqrt{3} \\ 1 \\ 1 \end{pmatrix} \begin{pmatrix} -1-\sqrt{3} & 1 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \frac{1}{3+\sqrt{3}} \begin{pmatrix} 4+2\sqrt{3} & -1-\sqrt{3} & -1-\sqrt{3} \\ -1-\sqrt{3} & 1 & 1 \\ -1-\sqrt{3} & 1 & 1 \end{pmatrix}$$

$$= \frac{1}{3+\sqrt{3}} \begin{pmatrix} -1-\sqrt{3} & 1+\sqrt{3} & 1+\sqrt{3} \\ 1+\sqrt{3} & 2+\sqrt{3} & -1 \\ 1+\sqrt{3} & -1 & 2+\sqrt{3} \end{pmatrix}$$

The matrix P_1 can be simplified to give

$$P_1 = \frac{1}{6} \begin{pmatrix} -2\sqrt{3} & 2\sqrt{3} & 2\sqrt{3} \\ 2\sqrt{3} & 3+\sqrt{3} & -3+\sqrt{3} \\ 2\sqrt{3} & -3+\sqrt{3} & 3+\sqrt{3} \end{pmatrix}$$

To verify that this matrix is valid, consider the product $A_2 = P_1 A$:

$$A_2 = P_1 A = \frac{1}{6} \begin{pmatrix} -2\sqrt{3} & 2\sqrt{3} & 2\sqrt{3} \\ 2\sqrt{3} & 3+\sqrt{3} & -3+\sqrt{3} \\ 2\sqrt{3} & -3+\sqrt{3} & 3+\sqrt{3} \end{pmatrix} = \begin{pmatrix} -1 & 1 & 1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix}$$

$$= \frac{1}{3} \begin{pmatrix} 3\sqrt{3} & \sqrt{3} & -\sqrt{3} \\ 0 & 2\sqrt{3} & -3+\sqrt{3} \\ 0 & 2\sqrt{3} & 3+\sqrt{3} \end{pmatrix},$$

indeed, all the terms in the first column are 0 except for the first.

Repeat the same process for the $(N-1) \times (N-1)$ bottom right submatrix of A_2 then once the new matrix P_2 is obtained (of size $(N-1) \times (N-1)$), place it at the bottom right of the $N \times N$ identity. When this process is repeated a total of $N-1$ times, the result will be an upper triangular matrix.

Second Reflection Matrix

Consider the matrix

$$A_2 = \frac{1}{3} \begin{pmatrix} 3\sqrt{3} & \sqrt{3} & -\sqrt{3} \\ 0 & 2\sqrt{3} & -3+\sqrt{3} \\ 0 & 2\sqrt{3} & 3+\sqrt{3} \end{pmatrix}.$$

Let B be the bottom right 2×2 submatrix of A_2,

$$A_2 = \tfrac{1}{3}\begin{pmatrix} 3\sqrt{3} & \sqrt{3} & -\sqrt{3} \\ 0 & \boxed{2\sqrt{3} & -3+\sqrt{3}} \\ 0 & 2\sqrt{3} & 3+\sqrt{3} \end{pmatrix} \quad \Longrightarrow \quad B = \begin{pmatrix} \frac{2\sqrt{3}}{3} & \frac{-3+\sqrt{3}}{3} \\ \frac{2\sqrt{3}}{3} & \frac{3+\sqrt{3}}{3} \end{pmatrix}.$$

Figure G.4

Repeat the same process as before with the matrix B: The 2-norm of the first column of B is $\beta = \frac{2\sqrt{6}}{3}$. Then

$$\boldsymbol{u} = \boldsymbol{b}_1 - \beta \mathbf{e}_1 = \begin{pmatrix} \frac{2\sqrt{3}}{3} \\ \frac{2\sqrt{3}}{3} \end{pmatrix} - \frac{2\sqrt{6}}{3}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{2\sqrt{3}}{3}\begin{pmatrix} 1-\sqrt{2} \\ 1 \end{pmatrix}$$

$$\boldsymbol{v} = \frac{\boldsymbol{u}}{\|\boldsymbol{u}\|} = \frac{1}{\sqrt{4-2\sqrt{2}}}\begin{pmatrix} 1-\sqrt{2} \\ 1 \end{pmatrix}$$

$$\tilde{P}_2 = \mathcal{I} - 2\boldsymbol{v}\boldsymbol{v}^{\mathrm{T}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \frac{2}{4-2\sqrt{2}}\begin{pmatrix} 1-\sqrt{2} \\ 1 \end{pmatrix}\begin{pmatrix} 1-\sqrt{2} & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \frac{1}{2-\sqrt{2}}\begin{pmatrix} 3-2\sqrt{2} & 1-\sqrt{2} \\ 1-\sqrt{2} & 1 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{pmatrix}.$$

Consider the product $\tilde{P}_2 B$:

$$\tilde{P}_2 B = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{pmatrix}\begin{pmatrix} \frac{2\sqrt{3}}{3} & \frac{-3+\sqrt{3}}{3} \\ \frac{2\sqrt{3}}{3} & \frac{3+\sqrt{3}}{3} \end{pmatrix} = \begin{pmatrix} \frac{2\sqrt{6}}{3} & \frac{\sqrt{6}}{3} \\ 0 & 1 \end{pmatrix}$$

which does change the matrix B into upper triangular form.
Let the matrix P_2 be the identity matrix with the bottom 2×2 submatrix replaced with $\tilde{P}_2$, i.e.

$$P_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ 0 & \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 2 & 0 & 0 \\ 0 & \sqrt{2} & \sqrt{2} \\ 0 & \sqrt{2} & -\sqrt{2} \end{pmatrix}.$$

The product $P_2 A_2$ should be lower triangular, indeed

$$P_2 A_2 = \frac{1}{6}\begin{pmatrix} 2 & 0 & 0 \\ 0 & \sqrt{2} & \sqrt{2} \\ 0 & \sqrt{2} & -\sqrt{2} \end{pmatrix}\begin{pmatrix} 3\sqrt{3} & \sqrt{3} & -\sqrt{3} \\ 0 & 2\sqrt{3} & -3+\sqrt{3} \\ 0 & 2\sqrt{3} & 3+\sqrt{3} \end{pmatrix} = \frac{1}{6}\begin{pmatrix} 6\sqrt{3} & 2\sqrt{3} & -2\sqrt{3} \\ 0 & 4\sqrt{6} & 2\sqrt{6} \\ 0 & 0 & -6\sqrt{2} \end{pmatrix}.$$

This sequence of steps will generate $N-1$ reflection matrices denoted $P_1, P_2, \ldots P_{N-1}$ which when applied to A in reverse order (i.e. the product is $P_{N-1} \ldots P_2 P_1 A$), must give an upper triangular matrix R. Since P_n are orthogonal for all $n = 1, 2, \ldots, N-1$, then their product will also be orthogonal.

Let $P = P_{N-1} \dots P_2 P_1$, then $R = PA$ meaning that $A = P^{-1}R$. Since P is orthogonal, then $P^{-1} = P^{\mathrm{T}}$ which will be equal to Q in the QR factorisation.

Final QR Decomposition

The matrices in question are

$$P_1 = \frac{1}{6}\begin{pmatrix} -2\sqrt{3} & 2\sqrt{3} & 2\sqrt{3} \\ 2\sqrt{3} & 3+\sqrt{3} & -3+\sqrt{3} \\ 2\sqrt{3} & -3+\sqrt{3} & 3+\sqrt{3} \end{pmatrix}, \quad P_2 = \frac{1}{2}\begin{pmatrix} 2 & 0 & 0 \\ 0 & \sqrt{2} & \sqrt{2} \\ 0 & \sqrt{2} & -\sqrt{2} \end{pmatrix}$$

The matrix product $P_2 P_1 A$ should give the matrix R which is upper triangular, indeed

$$R = P_2 P_1 A = \frac{1}{6}\begin{pmatrix} 6\sqrt{3} & 2\sqrt{3} & -2\sqrt{3} \\ 0 & 4\sqrt{6} & 2\sqrt{6} \\ 0 & 0 & -6\sqrt{2} \end{pmatrix}.$$

Let

$$P = P_2 P_1 = \frac{1}{12}\begin{pmatrix} 2 & 0 & 0 \\ 0 & \sqrt{2} & \sqrt{2} \\ 0 & \sqrt{2} & -\sqrt{2} \end{pmatrix}\begin{pmatrix} -2\sqrt{3} & 2\sqrt{3} & 2\sqrt{3} \\ 2\sqrt{3} & 3+\sqrt{3} & -3+\sqrt{3} \\ 2\sqrt{3} & -3+\sqrt{3} & 3+\sqrt{3} \end{pmatrix}$$
$$= \begin{pmatrix} -\frac{\sqrt{3}}{3} & \frac{\sqrt{3}}{3} & \frac{\sqrt{3}}{3} \\ \frac{\sqrt{6}}{3} & \frac{\sqrt{6}}{6} & \frac{\sqrt{6}}{6} \\ 0 & \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{pmatrix}.$$

Therefore

$$Q = P^{-1} = P^{\mathrm{T}} = \begin{pmatrix} -\frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{3} & 0 \\ \frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{6} & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{6} & -\frac{\sqrt{2}}{2} \end{pmatrix},$$

hence giving the QR decomposition of A as

$$\underbrace{\begin{pmatrix} -1 & 1 & 1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} -\frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{3} & 0 \\ \frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{6} & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{6} & -\frac{\sqrt{2}}{2} \end{pmatrix}}_{Q} \underbrace{\begin{pmatrix} \sqrt{3} & \frac{\sqrt{3}}{3} & -\frac{\sqrt{3}}{3} \\ 0 & \frac{2\sqrt{6}}{3} & \frac{\sqrt{6}}{3} \\ 0 & 0 & -\sqrt{2} \end{pmatrix}}_{R}$$

G.2.2 QR Decomposition Using Rotations

The following will explain how the QR decomposition can be performed using rotation matrices on a square matrix $A \in \mathbb{R}^{N \times N}$.

Parallel Example

This process will also be applied in parallel to the following matrix

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

The rotation matrices should make all the terms in the lower triangular part of the matrix equal to zero. Starting with the lower left most element a_{N1}, this element can be eliminated by using the rotation matrix $G(1, N; \theta)$ where $\theta = \arctan\left(-\frac{a_{N1}}{a_{11}}\right)$. When applied to A, this should eliminate the term a_{N1}.

First Rotation Matrix

For the matrix

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

The angle θ will be $\theta = \arctan\left(-\frac{a_{31}}{a_{11}}\right) = \arctan(1) = \frac{\pi}{4}$. Therefore the rotation matrix will be

$$G_1 = G\left(1, 3; \frac{\pi}{4}\right) = \begin{pmatrix} \cos\left(\frac{\pi}{4}\right) & 0 & -\sin\left(\frac{\pi}{4}\right) \\ 0 & 1 & 0 \\ \sin\left(\frac{\pi}{4}\right) & 0 & \cos\left(\frac{\pi}{4}\right) \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} \\ 0 & 1 & 0 \\ \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{pmatrix}.$$

This can be verified by considering the product $A_2 = G_1 A$:

$$A_2 = G_1 A = \begin{pmatrix} \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \\ 0 & 1 & 0 \\ -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{pmatrix} \begin{pmatrix} -1 & 1 & 1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} -\sqrt{2} & 0 & 0 \\ 1 & 1 & -1 \\ 0 & \sqrt{2} & \sqrt{2} \end{pmatrix}$$

which does eliminate a_{31}.

This process can be repeated for all other terms in the lower triangular section to reduce A into an upper triangular matrix. In these cases, to eliminate the element in position (m, n), the angle $\theta = \arctan\left(-\frac{a_{mn}}{a_{nn}}\right)$ and the rotation matrix is $G(n, m; \theta)$.

Second & Third Rotation Matrices

Repeat the same process as above to the matrix A_2 to eliminate the term in position

(2,1): $\theta_2 = \arctan\left(-\frac{a_{21}}{a_{11}}\right) = \arctan\left(\frac{1}{\sqrt{2}}\right)$ and $G_2 = G(1,2;\theta_2)$ is

$$G_2 = G(1,2;\theta_2) = \begin{pmatrix} \cos(\theta_2) & -\sin(\theta_2) & 0 \\ \sin(\theta_2) & \cos(\theta_2) & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{6}}{3} & -\frac{\sqrt{3}}{3} & 0 \\ \frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{3} & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Applying G_2 to A_2 should eliminate the (2,1) element, indeed

$$A_3 = G_2 A_2 = \begin{pmatrix} -\sqrt{3} & -\frac{\sqrt{3}}{3} & \frac{\sqrt{3}}{3} \\ 0 & \frac{\sqrt{6}}{3} & -\frac{\sqrt{6}}{3} \\ 0 & \sqrt{2} & \sqrt{2} \end{pmatrix}.$$

Finally, the term in position (2,3) needs to be eliminated: $\theta_3 = \arctan\left(-\frac{a_{32}}{a_{22}}\right) = \arctan\left(\sqrt{3}\right)$ and $G_3 = G(2,3;\theta_3)$ is

$$G_3 = G(2,3;\theta_3) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_3) & -\sin(\theta_3) \\ 0 & \sin(\theta_3) & \cos(\theta_3) \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & \sqrt{3} \\ 0 & -\sqrt{3} & 1 \end{pmatrix}.$$

Applying G_3 to A_3 should eliminate the (3,2) element, indeed

$$G_3 A_3 = \frac{1}{2}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & \sqrt{3} \\ 0 & -\sqrt{3} & 1 \end{pmatrix}\begin{pmatrix} -\sqrt{3} & -\frac{\sqrt{3}}{3} & \frac{\sqrt{3}}{3} \\ 0 & \frac{\sqrt{6}}{3} & -\frac{\sqrt{6}}{3} \\ 0 & \sqrt{2} & \sqrt{2} \end{pmatrix} = \frac{1}{3}\begin{pmatrix} -3\sqrt{3} & -\sqrt{3} & \sqrt{3} \\ 0 & 2\sqrt{6} & \sqrt{6} \\ 0 & 0 & 3\sqrt{2} \end{pmatrix}.$$

This process will generate a sequence of *at most* $\frac{1}{2}N(N-1)$ rotation matrices (since this is the number of terms that need to be eliminated). Suppose that M rotation matrices are needed where $M \in \left\{1, 2, \ldots, \frac{1}{2}N(N-1)\right\}$, then when these are applied to A in reverse order (the product $G_M G_{M-1} \ldots G_2 G_1 A$), then the result should be the upper triangular matrix R. Let $G = G_M G_{M-1} \ldots G_2 G_1$, then $R = GA$. Since all the rotation matrices are orthogonal, then their product must also be orthogonal, therefore if $Q = G^{-1} = G^{\mathrm{T}}$, then $A = QR$, hence giving the QR decomposition of A.

Final QR Decomposition

The matrices in question are

$$G_1 = \begin{pmatrix} \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \\ 0 & 1 & 0 \\ -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{pmatrix}, \quad G_2 = \begin{pmatrix} \frac{\sqrt{6}}{3} & -\frac{\sqrt{3}}{3} & 0 \\ \frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{3} & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad G_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{\sqrt{3}}{2} \\ 0 & -\frac{\sqrt{3}}{2} & \frac{1}{2} \end{pmatrix}.$$

The product of the rotation matrices is

$$G = G_3G_2G_1 = \begin{pmatrix} \frac{\sqrt{3}}{3} & -\frac{\sqrt{3}}{3} & -\frac{\sqrt{3}}{3} \\ \frac{\sqrt{6}}{3} & \frac{\sqrt{6}}{6} & \frac{\sqrt{6}}{6} \\ 0 & \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{pmatrix}.$$

Therefore

$$Q = G^{-1} = G^{\mathrm{T}} = \begin{pmatrix} \frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{3} & 0 \\ -\frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{6} & \frac{\sqrt{2}}{2} \\ \frac{-\sqrt{3}}{3} & \frac{\sqrt{6}}{6} & -\frac{\sqrt{2}}{2} \end{pmatrix}$$

hence giving the QR decomposition of A as

$$\underbrace{\begin{pmatrix} -1 & 1 & 1 \\ 1 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} \frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{3} & 0 \\ -\frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{6} & \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{3}}{3} & \frac{\sqrt{6}}{6} & -\frac{\sqrt{2}}{2} \end{pmatrix}}_{Q} \underbrace{\begin{pmatrix} -\sqrt{3} & -\frac{\sqrt{3}}{3} & \frac{\sqrt{3}}{3} \\ 0 & \frac{2\sqrt{6}}{3} & \frac{\sqrt{6}}{3} \\ 0 & 0 & \sqrt{2} \end{pmatrix}}_{R}.$$

Generally, the QR decomposition of a matrix is unique up to sign differences (as seen from the examples above where some of the rows and columns have different signs but in the end, the result will be the same).

G.2.3 QR Decomposition in MATLAB

In MATLAB, the QR decomposition can be done with the `qr` function.

```
>> A=[4,6,1;0,1,-1;0,1,2]
A =
     4     6     1
     0     1    -1
     0     1     2
>> [Q,R]=qr(A)
Q =
     1.0000         0         0
          0   -0.7071   -0.7071
          0   -0.7071    0.7071
R =
     4.0000    6.0000    1.0000
          0   -1.4142   -0.7071
          0         0    2.1213
```

If the matrix is rectangular, then the economy version of the QR decomposition can be found using `qr(A,"econ")`.

G.3 Eigenvalue Decomposition

For a matrix $A \in \mathbb{C}^{N\times N}$, the value $\lambda \in \mathbb{C}$ and non-zero vector $\boldsymbol{v} \in \mathbb{C}^N$ are known as the ***Eigenvalue*** and ***Eigenvector***, respectively, if they satisfy the relationship $A\boldsymbol{v} = \lambda\boldsymbol{v}$. These can be written in eigenpair notation as $\{\lambda; \boldsymbol{v}\}$.

In MATLAB, to find the eigenvalues and eigenvectors of a matrix `A`, use `[V,E]=eig(A)`. This will produce a matrix `V` whose columns are the eigenvectors of `A` and a diagonal matrix `E` whose entries are the corresponding eigenvalues where the (n, n) element of `E` is the eigenvalue that corresponds to the eigenvector in column n of `V`. However, if only `eig(A)` is run without specifying the outputs, MATLAB will produce a column vector of eigenvalues only.

```
>> A=[-2,-4,2;-2,1,2;4,2,5]
A =
    -2  -4   2
    -2   1   2
     4   2   5
>> eig(A)
ans =
    -5
     3
     6
>> [V,E]=eig(A)
V =
     0.8165   0.5345   0.0584
     0.4082  -0.8018   0.3505
    -0.4082  -0.2672   0.9347

E =
    -5   0   0
     0   3   0
     0   0   6
```

Therefore, the matrix A has the following eigenpairs

$$\left\{-5\,;\begin{pmatrix}0.8165\\0.4082\\-0.4082\end{pmatrix}\right\}\quad,\quad\left\{3\,;\begin{pmatrix}0.5345\\-0.8018\\-0.2672\end{pmatrix}\right\}\quad,\quad\left\{6\,;\begin{pmatrix}0.0584\\0.3505\\0.9347\end{pmatrix}\right\}.$$

Notice that the eigenvectors are not represented in the most pleasant form, the reason is that MATLAB normalises eigenvectors by default, meaning that the magnitude of every eigenvector is 1. In order to convert this to a more palatable form, the columns should be individually multiplied or divided by any scalar value[1]. The easiest way to do this is to, first

[1] Remember that any scalar multiple of an eigenvector is still an eigenvector.

of all, divide every individual column by its minimum value, then any other manipulations can be carried out afterwards.

```
>> v1=V(:,1)/min(V(:,1))
ans =
    -2
    -1
     1
>> v2=V(:,2)/min(V(:,2))
ans =
    -0.6667
     1.0000
     0.3333
>> v2=3*v2
ans =
    -2
     3
     1
>> v3=V(:,3)/min(V(:,3))
ans =
     1
     6
     16
```

This produces a far more appealing set of eigenpairs:

$$\left\{-5\,;\begin{pmatrix}-2\\-1\\1\end{pmatrix}\right\}\quad,\quad\left\{3\,;\begin{pmatrix}-2\\3\\1\end{pmatrix}\right\}\quad,\quad\left\{6\,;\begin{pmatrix}1\\6\\16\end{pmatrix}\right\}.$$

G.3.1 Eigendecomposition

Suppose that the matrix $A \in \mathbb{C}^{N\times N}$ has N linearly independent eigenvectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_N$ with their associated eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_N$. Let V be the matrix whose columns are the eigenvectors of A and let Λ be the diagonal matrix whose entries are the corresponding eigenvalues (in the same way that MATLAB produces the matrices `E` and `V`). In other words, if the matrix A has the eigenpairs

$$\{\lambda_1; \boldsymbol{v}_1\}\,,\quad \{\lambda_2; \boldsymbol{v}_2\}\,,\quad \ldots\{\lambda_N; \boldsymbol{v}_N\}\,,$$

then the matrices V and Λ are

$$V = \begin{pmatrix}\vdots & \vdots & & \vdots\\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & \ldots & \boldsymbol{v}_N\\ \vdots & \vdots & & \vdots\end{pmatrix}\quad\text{and}\quad \Lambda = \begin{pmatrix}\lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_N\end{pmatrix}.$$

The matrix A can then be written as $A = V\Lambda V^{-1}$ and this is called the ***Eigendecomposition*** of A. If V is an orthogonal matrix (as MATLAB produces it), then the eigendecomposition of A is $A = V\Lambda V^{\mathrm{T}}$.

This particular decomposition of matrices is useful when the matrix A acts as a repeated transformation in a vector space. For example, suppose that the vector $\boldsymbol{y}$ can be found by applying the matrix transformation A on the vector $\boldsymbol{x}$ 100 times, this means that $\boldsymbol{y} = A^{100}\boldsymbol{x}$. Under usual circumstances, calculating A^{100} is incredibly cumbersome but if the eigendecomposition of A is used, then the problem can be reduced into taking the power of a diagonal matrix instead. Indeed,

$$\boldsymbol{y} = A^{100}\boldsymbol{x}$$

$$\boldsymbol{y} = \underbrace{AA\ldots A}_{100 \text{ times}}\boldsymbol{x}$$

$$\boldsymbol{y} = \underbrace{\left(V\Lambda V^{-1}\right)\left(V\Lambda V^{-1}\right)\ldots\left(V\Lambda V^{-1}\right)}_{100 \text{ times}}\boldsymbol{x}$$

$$\boldsymbol{y} = V\Lambda V^{-1}V\Lambda V^{-1}V\Lambda V^{-1}\boldsymbol{x}$$

$$\boldsymbol{y} = V\Lambda^{100}V^{-1}\boldsymbol{x}.$$

Therefore, instead of calculating A^{100}, the matrix Λ^{100} can be calculated instead which will be much easier since Λ is a diagonal matrix (remember that the power of a diagonal matrix is just the power of its individual terms). If V is orthogonal, then the calculation will be simpler since the matrix V does not need to be inverted, only its transpose taken.

Luckily, MATLAB can perform this decomposition as seen with the `eig` command.

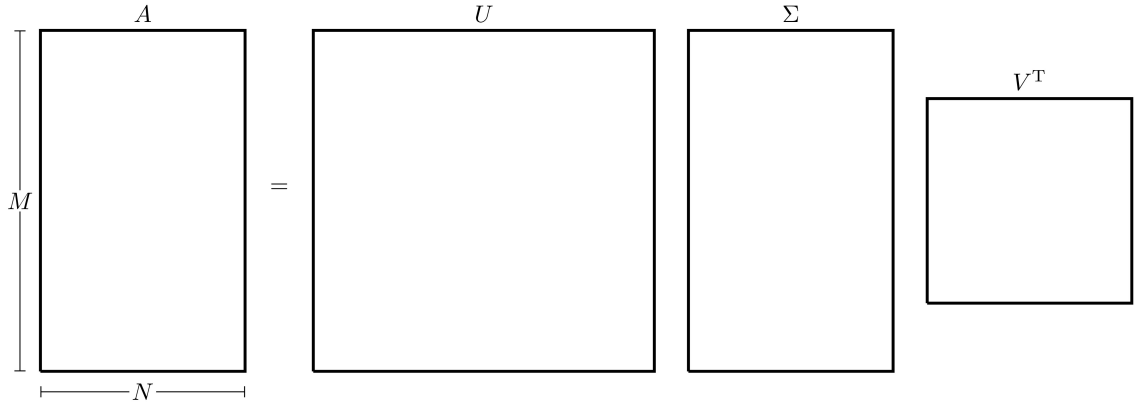
G.4 Singular Value Decomposition (SVD)

What happens if a square matrix A does not have a full system of eigenvectors? What happens if A is a rectangular matrix? In cases like this, some of the previous decompositions can fail, however there is one more way in which these issues can be resolved and it is by using the ***Singular Value Decomposition***.

For $A \in \mathbb{R}^{M\times N}$, orthogonal matrices $U \in \mathbb{R}^{M\times M}$ and $V \in \mathbb{R}^{N\times N}$ can always be found such that $AV = U\Sigma$ where $\Sigma \in \mathbb{R}^{M\times N}$ is a diagonal matrix that can be written as $\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_p)$ where $p = \min\{M, N\}$ whose entries are positive and arranged in descending order, i.e.

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0.$$

Since V is an orthogonal matrix, then A can be written as $A = U\Sigma V^{\mathrm{T}}$, this form is called the ***Singular Value Decomposition*** (SVD) of A. If $M > N$, this can be illustrated as follows:



The scalar values σ_i are called the ***Singular Values of A***, the columns of U are called ***Left Singular Vectors*** and the columns of V are called ***Right Singular Vectors***. In a vector sense, the SVD of A given by $A = U\Sigma V^{\mathrm{T}}$ can be written as $A\boldsymbol{v}_i = \sigma_i \boldsymbol{u}_i$ for all $i = 1, 2, \ldots, p$ (where $\boldsymbol{u}_i$ and $\boldsymbol{v}_i$ are the columns of U and V respectively).

Properties of the SVD

- The SVD of a matrix $A \in \mathbb{C}^{M\times N}$ requires $\mathcal{O}(MNp)$ computations (where $p = \min\{M, N\}$).
- The singular values are also useful when calculating the 2-norm of a matrix. Recall that for a matrix $A \in \mathbb{C}^{M\times N}$, the 2-norm of A can be written in terms of the spectral radius of $A^{\mathrm{H}}A$ as
$$\|A\|_2 = \sqrt{\rho(A^{\mathrm{H}}A)}$$
where the spectral radius is the largest eigenvalue in absolute value. This can also be written in terms of the singular values as
$$\|A\|_2 = \sqrt{\sigma_{max}(A)}$$
where $\sigma_{max}(A)$ represents the largest singular value of matrix A, which (as per the the way in which the singular values have been arranged) is going to be σ_1.
- If $A \in \mathbb{R}^{N\times N}$, then the eigenvalues of AA^{T} and $A^{\mathrm{T}}A$ are equal to the squares of the singular values of A, indeed, if $A = U\Sigma V^{\mathrm{T}}$, then
$$AA^{\mathrm{T}} = \left(U\Sigma V^{\mathrm{T}}\right)\left(U\Sigma V^{\mathrm{T}}\right)^{\mathrm{T}} = U\Sigma V^{\mathrm{T}}V\Sigma^{\mathrm{T}}U^{\mathrm{T}} = U\Sigma^2 U^{\mathrm{T}}$$
$$A^{\mathrm{T}}A = \left(U\Sigma V^{\mathrm{T}}\right)^{\mathrm{T}}\left(U\Sigma V^{\mathrm{T}}\right) = V\Sigma^{\mathrm{T}}U^{\mathrm{T}}U\Sigma V^{\mathrm{T}} = V\Sigma^2 V^{\mathrm{T}}$$
since Σ is a diagonal square matrix.
- Let $r, s \in \mathbb{N}$ and $\tau \in \mathbb{R}$, suppose that the singular values $\sigma_1, \sigma_2, \ldots, \sigma_p$ of A satisfy
$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_s > \tau \geq \sigma_{s+1} \geq \cdots \geq \sigma_r > \sigma_{r+1} = \sigma_{r+2} = \cdots = \sigma_p = 0.$$
Then r is the ***Rank*** of A and s is the ***τ-rank*** of A. In fact, if $\tau = \varepsilon_M$ (the machine precision), then s is called the ***Numerical Rank*** of A.

- Specific singular vectors span specific subspaces defined in connection to A. For instance, if the rank of A is r, then $A\boldsymbol{v}_i = \mathbf{0}$ for all $i = r+1, \ldots, N$. As a consequence, the vectors $\boldsymbol{v}_{r+1}, \boldsymbol{v}_{r+2}, \ldots, \boldsymbol{v}_N$ span the null-space of A, denoted by

$$\text{null}(A) = \left\{ \boldsymbol{x} \in \mathbb{R}^N : A\boldsymbol{x} = \mathbf{0} \right\}.$$

- If $A = U\Sigma V^{\mathrm{T}}$, then A can be rewritten as

$$A = \sum_{i=1}^{r} E_i$$

  where $E_i = \sigma_i \boldsymbol{u}_i {\boldsymbol{v}_i}^{\mathrm{T}}$ is a rank-1 matrix. It can be seen that

$$\|E_i\| = \|\sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T\|_2 = \sigma_i.$$

  Since the norm of a matrix is a measure of the "magnitude" of a matrix, it can be said that A is made up of very specific elementary rank-1 matrices, in such a way that E_1 is the most "influential" one.

The singular value decomposition of the matrix $A \in \mathbb{R}^{M \times N}$ can be done by following these steps:

Parallel Example

These steps will be applied in parallel to the matrix

$$A = \begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix}.$$

1. Calculate the eigenpairs of AA^{T} and $A^{\mathrm{T}}A$.

Eigenpairs

The eigenpairs of AA^{T} are

$$\left\{ 25; \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\} \quad \text{and} \quad \left\{ 9; \begin{pmatrix} -1 \\ 1 \end{pmatrix} \right\}.$$

Similarly, the eigenpairs of $A^{\mathrm{T}}A$ are

$$\left\{ 25; \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \right\}, \quad \left\{ 9; \begin{pmatrix} 1 \\ -1 \\ 4 \end{pmatrix} \right\} \quad \text{and} \quad \left\{ 0; \begin{pmatrix} -2 \\ 2 \\ 1 \end{pmatrix} \right\}.$$

2. Normalise the eigenvectors by dividing by their 2-norm (this will in fact be the default output from MATLAB's `eig` function).

Normalise Eigenvectors

The *normalised* eigenpairs of AA^{T} are

$$\left\{25; \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}\right\} \quad \text{and} \quad \left\{9; \frac{1}{\sqrt{2}}\begin{pmatrix}-1\\1\end{pmatrix}\right\}.$$

Similarly, the *normalised* eigenpairs of $A^{\mathrm{T}}A$ are

$$\left\{25; \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\\0\end{pmatrix}\right\}, \quad \left\{9; \frac{1}{\sqrt{18}}\begin{pmatrix}1\\-1\\4\end{pmatrix}\right\} \quad \text{and} \quad \left\{0; \frac{1}{3}\begin{pmatrix}-2\\2\\1\end{pmatrix}\right\}.$$

3. The matrix of singular values Σ must be of the same size as A, i.e. $\Sigma \in \mathbb{R}^{M\times N}$, where the diagonal terms are the square roots of the eigenvalues of AA^{T} and $A^{\mathrm{T}}A$ (only the ones that are shared by the two matrix products) arranged in descending order. There will only be p diagonal terms where $p = \min\{M, N\}$.

Terms of Σ

The matrix Σ must be of size 2×3. The eigenvalues of AA^{T} and $A^{\mathrm{T}}A$ are 25 and 9. Therefore the matrix Σ and is given by

$$\Sigma = \begin{pmatrix}\sqrt{25} & 0 & 0\\ 0 & \sqrt{9} & 0\end{pmatrix} = \begin{pmatrix}5 & 0 & 0\\ 0 & 3 & 0\end{pmatrix}.$$

4. The matrix $U \in \mathbb{R}^{M\times M}$ will be the matrix whose columns are the normalised eigenvectors of $A^{\mathrm{T}}A$ arranged in the same order as the values appear in Σ. Note that if $\boldsymbol{v}$ is a normalised eigenvector, then $-\boldsymbol{v}$ will also be a normalised eigenvector, therefore this will give rise to 2^M possible cases for U (which will be narrowed down later).

Matrix U

The normalised eigenpairs of $A^{\mathrm{T}}A$ are

$$\left\{25; \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}\right\} \quad \text{and} \quad \left\{9; \frac{1}{\sqrt{2}}\begin{pmatrix}-1\\1\end{pmatrix}\right\}.$$

If $\boldsymbol{u}_1$ is the first normalised eigenvector and $\boldsymbol{u}_2$ is the second normalised eigenvector (i.e. $\boldsymbol{u}_1 = (1\ ,\ 1)^{\mathrm{T}}$ and $\boldsymbol{u}_2 = (-1\ ,\ 1)^{\mathrm{T}}$), then the matrix $U \in \mathbb{R}^{2\times 2}$ can take one of four

possible forms

$$U_1 = \begin{pmatrix} \boldsymbol{u}_1 & \boldsymbol{u}_2 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}, \qquad U_2 = \begin{pmatrix} \boldsymbol{u}_1 & -\boldsymbol{u}_2 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$U_3 = \begin{pmatrix} -\boldsymbol{u}_1 & \boldsymbol{u}_2 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} -1 & -1 \\ -1 & 1 \end{pmatrix}, \quad U_4 = \begin{pmatrix} -\boldsymbol{u}_1 & -\boldsymbol{u}_2 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} -1 & 1 \\ -1 & -1 \end{pmatrix}.$$

5. The matrix $V \in \mathbb{R}^{N \times N}$ will be the matrix whose columns are the normalised eigenvectors of AA^{T} arranged in the same order as the values appear in Σ. Just as before, there will technically be 2^N choices of V. In this case, one choice of U or V should be fixed.

Matrix V

The normalised eigenpairs of $A^{\mathrm{T}}A$ are

$$\left\{25; \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}\right\}, \quad \left\{9; \frac{1}{\sqrt{18}} \begin{pmatrix} 1 \\ -1 \\ 4 \end{pmatrix}\right\} \quad \text{and} \quad \left\{0; \frac{1}{3} \begin{pmatrix} -2 \\ 2 \\ 1 \end{pmatrix}\right\}.$$

Since V has a larger size than U, fix V as the matrix whose columns are the normalised eigenvectors of AA^{T} with no sign changes. This can be accommodated for later on by picking an appropriate choice for U. Then

$$V = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{18}} & -\frac{2}{3} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{18}} & \frac{2}{3} \\ 0 & \frac{4}{\sqrt{18}} & \frac{1}{3} \end{pmatrix}.$$

6. The correct choice for the matrix U can be found in one of two ways:

- **Trial & Error**: Perform the multiplication $U\Sigma V^{\mathrm{T}}$ for the different choices of U until the correct one is found that gives A. Alternatively, U can be fixed and the different choices for V can be investigated.

Trial & Error

Consider the product $U\Sigma V^{\mathrm{T}}$ for the different choices of U and see which one gives the

matrix A:

$$U_1\Sigma V^{\mathrm{T}} = \begin{pmatrix} 2 & 3 & -2 \\ 3 & 2 & 2 \end{pmatrix} \neq A$$

$$U_2\Sigma V^{\mathrm{T}} = \begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix} = A$$

$$U_3\Sigma V^{\mathrm{T}} = \begin{pmatrix} -3 & -2 & -2 \\ -2 & -3 & 2 \end{pmatrix} \neq A$$

$$U_4\Sigma V^{\mathrm{T}} = \begin{pmatrix} -2 & -3 & 2 \\ -3 & -2 & -2 \end{pmatrix} \neq A$$

Therefore the correct choice for U is U_2.

- **Pseudo-Inversion**: First, consider the expression $A = U\Sigma V^{\mathrm{T}}$, multiplying both sides by V on the right gives $AV = U\Sigma$ (since V is orthogonal meaning that $V^{\mathrm{T}}V = \mathcal{I}$). Since Σ is rectangular in general, it does not have an inverse but it does have a ***Pseudo-Inverse***[2]. Since Σ is a diagonal matrix, then the pseudo-inverse will also be a diagonal matrix with the diagonal entries being the reciprocals of the singular values. For example, if

$$\Sigma = \begin{pmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 \\ 0 & 0 & \sigma_3 & 0 \end{pmatrix},$$

then the pseudo-inverse of Σ is

$$\Sigma^{+} = \begin{pmatrix} \frac{1}{\sigma_1} & 0 & 0 \\ 0 & \frac{1}{\sigma_2} & 0 \\ 0 & 0 & \frac{1}{\sigma_3} \\ 0 & 0 & 0 \end{pmatrix}.$$

Similarly if

$$\Sigma = \begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \\ 0 & 0 & 0 \end{pmatrix},$$

then the pseudo-inverse of Σ is

$$\Sigma^{-} = \begin{pmatrix} \frac{1}{\sigma_1} & 0 & 0 & 0 \\ 0 & \frac{1}{\sigma_2} & 0 & 0 \\ 0 & 0 & \frac{1}{\sigma_3} & 0 \end{pmatrix}.$$

Therefore multiplying both sides of $AV = U\Sigma$ by Σ^{+} on the right will give the desired expression for U which is $U = AV\Sigma^{+}$.

[2] For a matrix $B \in \mathbb{C}^{M\times N}$ with $M < N$, then the pseudo-inverse is the matrix $B^{+} \in \mathbb{C}^{N\times M}$ such that $BB^{+} = \mathcal{I} \in \mathbb{R}^{M\times M}$. Similarly, if $B \in \mathbb{C}^{M\times N}$ with $M > N$, the pseudo-inverse is the matrix $B^{-} \in \mathbb{C}^{N\times M}$ such that $B^{-}B = \mathcal{I} \in \mathbb{R}^{N\times N}$. Note that if a matrix is square and invertible, then the pseudo-inverse is the inverse.

Pseudo-Inverse

The pseudo-inverse of $\Sigma \in \mathbb{R}^{2\times 3}$ is $\Sigma^+ \in \mathbb{R}^{3\times 2}$ where its diagonal terms are the reciprocals of those in Σ, i.e.

$$\Sigma = \begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix} \quad \Longrightarrow \quad \Sigma^+ = \begin{pmatrix} \frac{1}{5} & 0 \\ 0 & \frac{1}{3} \\ 0 & 0 \end{pmatrix}.$$

This can be verified by showing that $\Sigma\Sigma^+ = \mathcal{I}$. To find U, calculate

$$U = AV\Sigma^+ = \begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{18}} & -\frac{2}{3} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{18}} & \frac{2}{3} \\ 0 & \frac{4}{\sqrt{18}} & \frac{1}{3} \end{pmatrix} \begin{pmatrix} \frac{1}{5} & 0 \\ 0 & \frac{1}{3} \\ 0 & 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

7. This finally gives all the matrices required for the SVD of A.

SVD of A

$$\underbrace{\begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}}_{U} \underbrace{\begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix}}_{\Sigma} \underbrace{\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{18}} & -\frac{1}{\sqrt{18}} & \frac{4}{\sqrt{18}} \\ -\frac{2}{3} & \frac{2}{3} & \frac{1}{3} \end{pmatrix}}_{V^{\mathrm{T}}}.$$

Note that if the SVD of a matrix A is known, it can also be useful in finding pseudo inverse of A:

$$\begin{aligned} & A = U\Sigma V^{\mathrm{T}} \\ \underset{\times V}{\Rightarrow} \quad & AV = U\Sigma V^{\mathrm{T}}V \\ \underset{V^{-1}=V^{\mathrm{T}}}{\Rightarrow} \quad & AV = U\Sigma \\ \underset{\times\Sigma^+}{\Rightarrow} \quad & AV\Sigma^+ = U\Sigma\Sigma^+ \\ \underset{\Sigma\Sigma^+=\mathcal{I}}{\Rightarrow} \quad & AV\Sigma^+ = U \\ \underset{\times U^{\mathrm{T}}}{\Rightarrow} \quad & AV\Sigma^+U^{\mathrm{T}} = U^{\mathrm{T}} \\ \underset{UU^{\mathrm{T}}=\mathcal{I}}{\Rightarrow} \quad & AV\Sigma^+U^{\mathrm{T}} = \mathcal{I}. \end{aligned}$$

Therefore, the matrix $A^+ = V\Sigma^+U^{\mathrm{T}}$ is the pseudo-inverse of A.

Pseudo-Inverse of A

Find the pseudo-inverse of A where

$$A = \begin{pmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{pmatrix}.$$

The SVD of A is

$$A = \underbrace{\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}}_{U} \underbrace{\begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \end{pmatrix}}_{\Sigma} \underbrace{\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{18}} & -\frac{1}{\sqrt{18}} & \frac{4}{\sqrt{18}} \\ -\frac{2}{3} & \frac{2}{3} & \frac{1}{3} \end{pmatrix}}_{V^{\mathrm{T}}}.$$

The pseudo-inverse of A is

$$A^{+} = V\Sigma^{+}U^{\mathrm{T}} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{18}} & -\frac{2}{3} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{18}} & \frac{2}{3} \\ 0 & \frac{4}{\sqrt{18}} & \frac{1}{3} \end{pmatrix} \begin{pmatrix} \frac{1}{5} & 0 \\ 0 & \frac{1}{3} \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} = \frac{1}{45} \begin{pmatrix} 7 & 2 \\ 2 & 7 \\ 10 & -10 \end{pmatrix}.$$

G.4.0.1 SVD in MATLAB

In MATLAB, the SVD of a matrix can be found with the `SVD` command.

```
>> A=[3, 2, 2; 2, 3, -2]
A =
     3   2   2
     2   3  -2
>> [U,S,V]=svd(A)
U =
    -0.7071    0.7071
    -0.7071   -0.7071
S =
     5.0000         0   0
          0    3.0000   0
V =
    -0.7071    0.2357   -0.6667
    -0.7071   -0.2357    0.6667
    -0.0000    0.9428    0.3333
>> U*S*V'-A    % Check is A=USV'
ans =
     1.0e-14 *
          0         0   -0.0222
    -0.0222   -0.1332    0.0666
```

Notice that sometimes, due to round-off error, `U*S*V'-A` may not exactly be equal to the zero matrix, but it is still close enough to it.