

Numerical Solutions of Differential Equations

Waleed A A Ali

Table of contents

Introduction	4
1 Internal Workings of MATLAB	5
1.1 Floating-Point Arithmetic	5
1.2 Computational Complexity	6
I Linear Algebra	12
2 Linear Algebra	13
2.1 Solving Linear Systems of Equations	14
2.1.1 Computational Stability of Linear Systems	15
2.2 Direct Methods	17
2.2.1 Direct Substitution	18
2.2.2 Forward/Backward Substitution	19
2.2.3 TDMA Algorithm	20
2.2.4 Cramer's Rule	21
2.2.5 Gaussian Elimination Method	22
2.3 Iterative Methods	28
2.3.1 Constructing an Iterative Method	29
2.3.2 Computational Cost & Stopping Criteria	31
2.4 In-Built MATLAB Procedures	32
II Solving Initial Value Problems	36
3 The Euler Method	37
3.1 Steps of the Euler Method	38
3.2 Accuracy	41
3.3 Set of IVPs	44
3.4 Higher Order IVPs	46
3.4.1 Sets of Higher Order IVPs	49
3.4.2 Stability of a Set of ODEs	50
3.5 Limitations of the Euler Method	51
3.5.1 Bounds on the Step size	53
3.5.2 Exact Bound	54
3.5.3 Estimated Bound	55
3.6 MATLAB Code	60

4 The Modified Euler Method	64
4.1 Steps of the Modified Euler Method	64
4.2 Accuracy of the Modified Euler Method	66
4.3 MATLAB Code	68
5 The Runge-Kutta Method	72
5.1 MATLAB Code	74
6 MATLAB's In-Built Procedures	78
7 Implicit IVP Solvers	82
7.1 Backwards Euler Method	82
7.2 Stability of the Backwards Euler Method	83
7.3 Order of Accuracy	84
7.4 Stiff Differential Equations	84
III Solving Boundary Value Problems	87
8 Boundary Value Problems	88
8.1 Example of Boundary Value Problems	88
8.2 Finite Difference Method for Boundary Value Problems	90
8.3 Existence & Uniqueness of Solutions to BVPs	90
8.3.1 Finite Difference Approximations to the Derivatives	92
8.3.2 Discretisation of the Differential Equation	94
8.3.3 Steps of The Finite Difference Method	96
8.4 MATLAB Code	99
8.5 Comparison Between Forward, Backward & Centred Difference Approximations	102
8.6 MATLAB's In-Built Procedures	103
9 Mixed Value Problems	107
9.1 Finite Difference Method for MVPs	107
10 Symmetric Boundary Conditions	112
10.1 Finite Difference Method for Symmetric Boundary Value Problems	112
IV Solving Partial Differential Equations	116
11 Heat Equation	117
11.1 The Method of Lines for the Heat Equation	117
11.2 Linear Advection Equation	122
11.3 Convection-Diffusion Equation	125
11.4 Asymptotic Stability	126
11.4.1 Stability of the Euler Method for the Advection Equation	128
11.4.2 Stability of the Euler Method for the Heat Equation	130
11.5 Stability of the Convection-Diffusion Equation	132

Introduction

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

All lectures will be held online on **Fridays, 10.00 - 12.00**.

Lecture	Date	Topic
1	17/10	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• Solving linear systems using direct methods• Solving linear systems using iterative methods
3	31/10	<ul style="list-style-type: none">• Euler method for IVPs
4	07/11	<ul style="list-style-type: none">• Modified Euler method• Runge-Kutta method• Backwards Euler method• solving stiff IVPs
5	14/11	<ul style="list-style-type: none">• Solving BVPs using the finite difference method
6	21/11	<ul style="list-style-type: none">• Solving MVPs and symmetric BVPs
7	28/11	<ul style="list-style-type: none">• Method of lines• Apply MoL for diffusion and/or convection
8	05/12	Stability of the method of lines

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

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

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 which is $E \in (L, U)$ with $L < 0$ and $U > 0$.

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).

This gives a total of 64-bits with $L = -1022$ and $U = 1023$. 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.

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 which is given by

$$\frac{|x - fl(x)|}{|x|} \leq \frac{1}{2}\varepsilon_M$$

where $\varepsilon_M = \beta^{1-t}$ is the machine epsilon and provides the distance between 1 and its closest floating-point number greater than 1. In the binary case, $\varepsilon_M \approx 2.2204 \cdot 10^{-16}$ (this can be found using `eps(1)`). In other words, the number $u = \frac{1}{2}\varepsilon_M$ is the maximum relative error that the computer can make while representing a real number by finite arithmetic.

Moreover, since L and U are finite, one cannot represent numbers whose absolute value is either arbitrarily large or arbitrarily small. Precisely, the smallest and the largest positive real numbers of \mathcal{F} are given by $x_{\min} = \beta^{L-1}$ and $x_{\max} = \beta^U(1 - \beta^{-t})$, respectively. A positive number smaller than x_{\min} yields under-flow and a positive number greater than x_{\max} yields over-flow. 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. The largest and smallest values that can be represented as floating-point numbers in MATLAB can be found using the `realmin` and `realmax` commands.

```

1 >> realmin
2 ans =
3     2.2251e-308
4 >> realmax
5 ans =
6     1.7977e308

```

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, see [Software Bugs](#) and the [Patriot Missile Fail](#) among others.

1.2 Computational Complexity

The ***Computational Complexity*** of a function can be defined as the relationship between the size of the input and the difficulty of running the function 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 basic, or **Floating-Point Operations**, such as additions, subtractions, multiplications, divisions and assignments. Floating-point operations, also called *flops*, usually measure 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 $f(x) \sim \mathcal{O}(p(x))$ as x tends to infinity if

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

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 $f(x) \sim \mathcal{O}(x^4)$ as $x \rightarrow \infty$.

Counting flops

Let $f : \mathbb{N} \rightarrow \mathbb{N}$ 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:

```

1 function [out]=fun(n)
2
3 out = 0;
4
5 for i=1:1:n
6
7     for j=1:1:n
8
9         out = out + i*j;
10
11    end
12
13 end
14
15 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 `tic` `toc`:

```

1 >> tic;
2 >> Run code or code block
3 >> 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.

🔥 Code Profiling

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.

```
1 function [F]=Fib_Iter(N)
2
3 S=ones(1,N);
4
5 for n=3:1:N
6
7     S(n)=S(n-1)+S(n-2);
8
9 end
10
11 F=S(end);
12
13 end
```

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

```
1 function [F]=Fib_Rec(N)
2
3 if N<3
4
5     F=1;
6
7 else
8
9     F=Fib_Rec(N-1)+Fib_Rec(N-2);
10
11 end
12
13 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

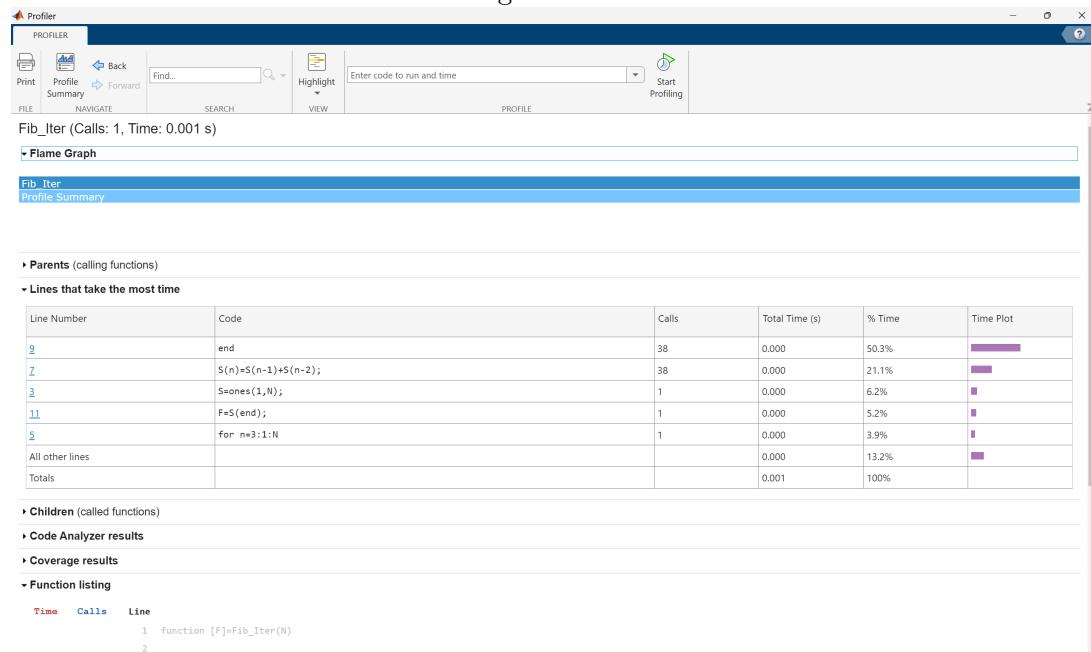
longer. Suppose the code efficiency is to be analysed for the input $N = 40$, this can be done using the profiler as follows:

```

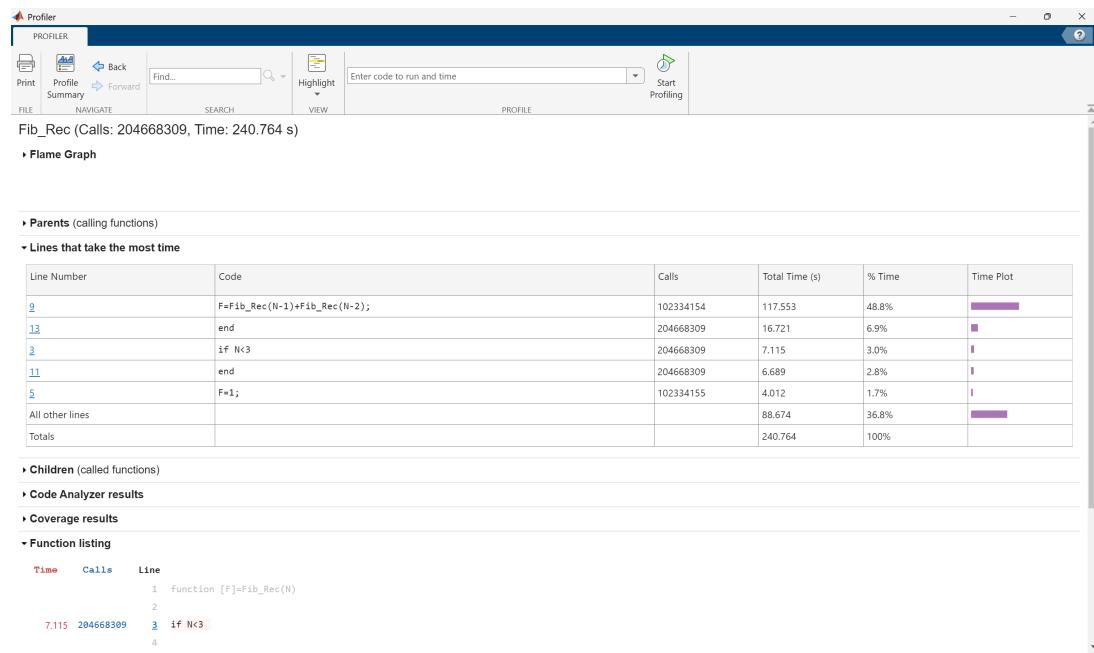
1 >> profile on
2 >> Fib_Iter(40);
3 >> profile off
4 >> profile viewer

```

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.

i 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

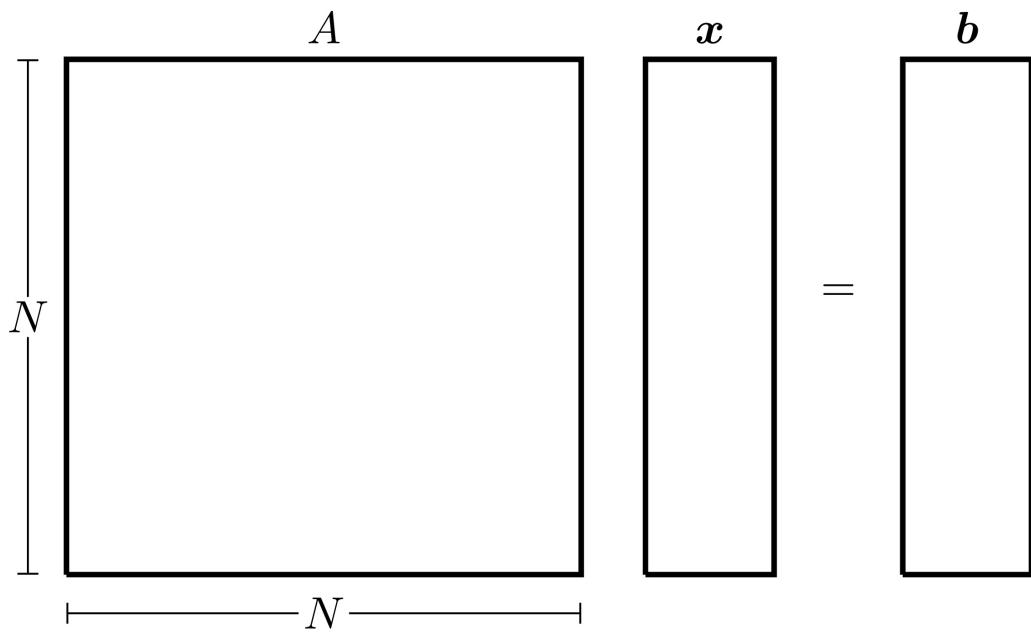
Linear Algebra

2 Linear Algebra

In this unit, the following Numerical Linear Algebra processes will be studies:

- Solving linear systems of the form

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

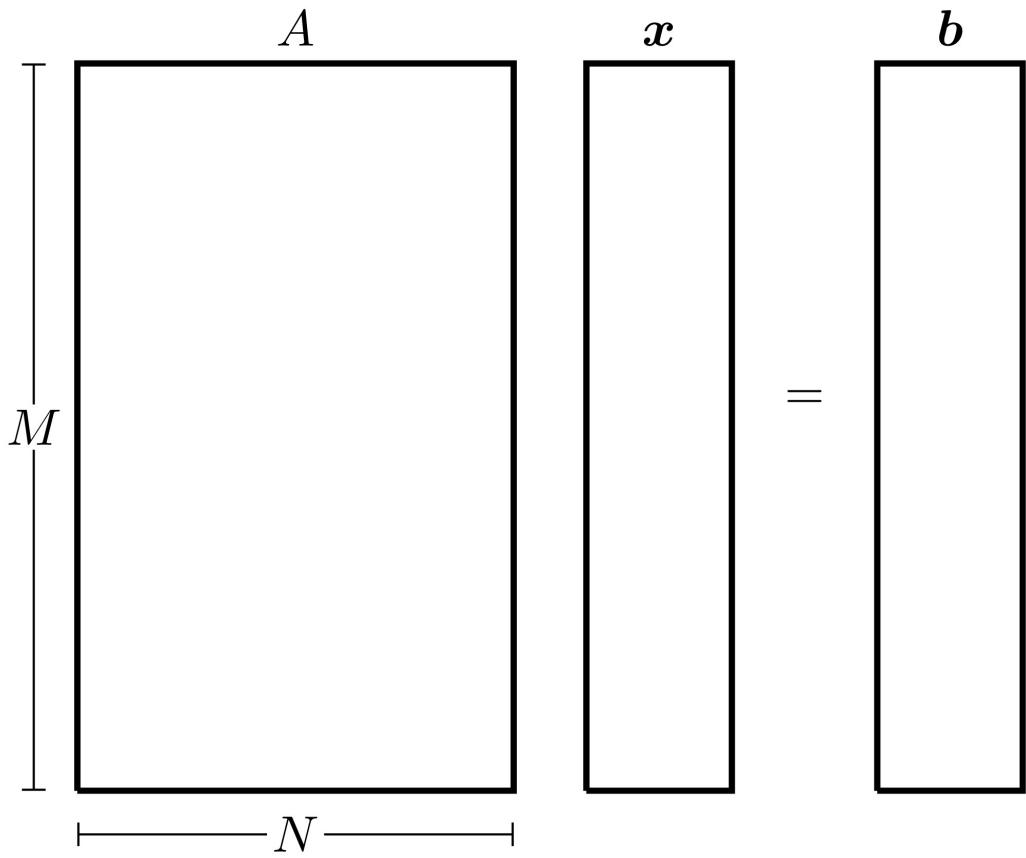


This can be done by using a *Direct Method* if the solution of the system can be obtained in a finite number of steps or an *Iterative Method* if the solution, in principle, requires an infinite number of steps.

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.

- Solving, or at least understanding, the nature of *overdetermined* problems of the form

$$Ax = b \quad \text{where} \quad A \in \mathbb{R}^{M \times N}, \quad \mathbf{x} \in \mathbb{R}^N, \quad \mathbf{b} \in \mathbb{R}^M \quad \text{with} \quad M > N.$$



This problem often arises when fitting models to a large data sets, where \boldsymbol{x} represents parameters in the model and \boldsymbol{b} represents the results (or measurements) of some experiments.

- Solving eigenvalue problems of the form

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

and the generalisations thereof.

2.1 Solving Linear Systems of Equations

The first part of linear algebra is solving the square linear system

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

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 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 be written in matrix form as:

$$\underbrace{\begin{pmatrix} 1 & 1 & 1 \\ 1 & -2 & 0 \\ 0 & 1 & -1 \end{pmatrix}}_A \underbrace{\begin{pmatrix} a \\ b \\ c \end{pmatrix}}_x = \underbrace{\begin{pmatrix} 20 \\ 0 \\ 10 \end{pmatrix}}_b.$$

There are many 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:
 - Backslash

2.1.1 Computational Stability of Linear Systems

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

Consider the linear system

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

In real-life applications, the matrix A is usually fully known and often invertible while the vector b may not be known exactly and its measurement may often include rounding errors.

Suppose that the vector \mathbf{b} has a small error $\delta\mathbf{b}$, then the solution \mathbf{x} will also have a small error $\delta\mathbf{x}$, meaning that the system will in fact be

$$A(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}. \quad (2.1)$$

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

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

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

i Note 1: Submultiplicative Property of Matrix Norms

For a matrix A and a vector \mathbf{x} ,

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

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

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

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 .

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

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

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

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 \mathbf{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 $A\mathbf{x} = \mathbf{b}$ may be solved using an *iterative method* which generate a sequence of approximations \mathbf{x}_n to \mathbf{x} while ensuring that each iteration is easy to perform and that \mathbf{x}_n rapidly tends to \mathbf{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 \mathbf{x} accumulate for large $\kappa_p(A)$, but the number of computation required to find \mathbf{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, `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. $A^{-1} = A^T$;
- $\kappa_p(A^T) = \kappa_p(A^{-1}) = \kappa_p(A)$;
- $\kappa_p(\lambda A) = \kappa_p(A)$;
- $\kappa_p(AB) \leq \kappa_p(A)\kappa_p(B)$.

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 $A\mathbf{x} = \mathbf{b}$ where

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

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

$$\begin{aligned} A\mathbf{x} = \mathbf{b} \quad \Rightarrow \quad & \begin{pmatrix} a_1 & & & \\ & a_2 & & \\ & & \ddots & \\ & & & a_{N-1} \\ & & & & a_N \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{N-1} \\ b_N \end{pmatrix} \\ & a_1x_1 = b_1 \quad x_1 = \frac{b_1}{a_1} \\ & a_2x_2 = b_2 \quad x_2 = \frac{b_2}{a_2} \\ \Rightarrow & \vdots \quad \Rightarrow \quad \vdots \\ & a_{N-1}x_{N-1} = b_{N-1} \quad x_{N-1} = \frac{b_{N-1}}{a_{N-1}} \\ & a_Nx_N = b_N \quad x_N = \frac{b_N}{a_N}. \end{aligned}$$

The solution can be written explicitly as $x_n = \frac{b_n}{a_n}$ for all $n = 1, 2, \dots, N$. 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 Substitution

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

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

Solving the system using direct substitution:

$$\begin{aligned} A\mathbf{x} = \mathbf{b} \quad \Rightarrow \quad & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \\ 4 \end{pmatrix} \\ \Rightarrow \quad & \begin{pmatrix} x_1 \\ 2x_2 \\ -x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \\ 4 \end{pmatrix} \quad \Rightarrow \quad \begin{array}{l} x_1 = 4 \\ x_2 = 1 \\ x_3 = -4. \end{array} \end{aligned}$$

2.2.2 Forward/Backward Substitution

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

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

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

and $a_1, a_2, \dots, a_N \neq 0$ (so that the determinant is non-zero). The matrix A is upper triangular in this case and will require backwards substitution:

$$\begin{aligned} A\mathbf{x} = \mathbf{b} \implies & \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1,N-1} & a_{1N} \\ a_{22} & \dots & a_{2,N-1} & \dots & a_{2N} \\ \ddots & & \vdots & & \\ & & a_{N-1,N-1} & a_{N-1,N} & \\ & & & a_{NN} & \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{N-1} \\ b_N \end{pmatrix} \\ & \begin{array}{lllllll} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1,N-1}x_{N-1} & + & a_{1N}x_N & = b_1 \\ a_{22}x_2 & + & \dots & + & a_{2,N-1}x_{N-1} & + & a_{2N}x_N & = b_2 \\ \vdots & & & & & & & & \\ & & & a_{N-1,N-1}x_{N-1} & + & a_{N-1,N}x_N & = b_{N-1} \\ & & & & & a_{NN}x_N & = b_N \end{array} \end{aligned}$$

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:

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

A similar version can be obtained for the forward substitution for lower triangular matrices. 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, $\frac{1}{2}(N^2 - N)$ multiplications and $\frac{1}{2}(N^2 - N)$ 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 $A\mathbf{x} = \mathbf{b}$ where

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

This problem can be solved by suing backward substitution:

$$\begin{aligned} A\mathbf{x} = \mathbf{b} \implies \begin{pmatrix} 1 & 2 & 1 \\ 0 & -1 & 4 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \implies & \begin{aligned} x_1 + 2x_2 + x_3 &= 1 \\ -x_2 + 4x_3 &= 0 \\ -x_3 &= 1 \end{aligned} \\ -x_2 + 4x_3 = 0 \underset{x_3=-1}{\implies} & -x_2 - 4 = 0 \implies x_2 = -4 \\ x_1 + 2x_2 + x_3 = 1 \underset{x_2=-4, x_3=-1}{\implies} & x_1 - 8 - 1 = 1 \implies x_1 = 10. \end{aligned}$$

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 $A\mathbf{x} = \mathbf{b}$ where

$$A = \begin{pmatrix} m_1 & r_1 & & & & \\ l_2 & m_2 & r_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & l_{N-1} & m_{N-1} & r_{N-1} & \\ & & & l_N & m_N & \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{N-1} \\ b_N \end{pmatrix}.$$

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 has explicit formulas for the solution using a forward and backwards sweep that convert the tridiagonal system into an upper triangular system as follows:

$$\begin{aligned} R_n &= \begin{cases} \frac{r_1}{m_1} & n = 1 \\ \frac{r_n}{m_n - l_n R_{n-1}} & n = 2, 3, \dots, N-1 \end{cases} \\ B_n &= \begin{cases} \frac{b_1}{m_1} & n = 1 \\ \frac{b_n - l_n B_{n-1}}{m_n - l_n R_{n-1}} & n = 2, 3, \dots, N \end{cases} \end{aligned}$$

$$x_n = \begin{cases} B_n - R_n x_{n+1} & n = 1, 2, 3, \dots, N-1 \\ B_N & n = N. \end{cases}$$

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 other similar algorithms to the TDMA algorithm, like those solve pentadiagonal systems and those where the tridiagonal matrix starts or ends with a full row.

2.2.4 Cramer's Rule

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

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

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

where A_n is the matrix obtained from A by replacing the n^{th} column by \mathbf{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 $A\mathbf{x} = \mathbf{b}$ where

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

The determinant of A is equal to 7. Using Cramer's rule, the solution $\mathbf{x} = (x_1, x_2, x_3)^T$ 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 Gigalops per second (10^9 flops), then a matrix system of size 20×20 will require 1620 years to compute.

2.2.5 Gaussian Elimination Method

The **Gaussian Elimination Method** is an algorithm that transforms the linear system $A\mathbf{x} = \mathbf{b}$ where $A \in \mathbb{C}^{N \times N}$ and $\mathbf{b} \in \mathbb{C}^N$ into an equivalent upper triangular system $U\mathbf{x} = \mathbf{g}$ after $N-1$ steps, where $U \in \mathbb{C}^{N \times N}$ is an upper triangular matrix and $\mathbf{g} \in \mathbb{C}^N$. This uses **Elementary Row Operations** (swapping rows, multiplying a row by a constant, adding two rows), after which point, the system $U\mathbf{x} = \mathbf{g}$ can solved by the backward substitution. Note that this method is possible when the elementary row operations are performed on both A and \mathbf{b} simultaneously, so if rows i and j are swapped in A , the rows i and j must also be swapped in \mathbf{b} , similarly 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 $A\mathbf{x} = \mathbf{b}$ where

$$A = \begin{pmatrix} 2 & -1 & 1 \\ -1 & 1 & 2 \\ 1 & 2 & -1 \end{pmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}.$$

1. **Establish the starting matrix:** If $a_{11} \neq 0$, then set $A^{(1)} = A$ and $\mathbf{b}^{(1)} = \mathbf{b}$ as

$$A^{(1)} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1j}^{(1)} & \dots & a_{1N}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & \dots & a_{2j}^{(1)} & \dots & a_{2N}^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{j1}^{(1)} & a_{j2}^{(1)} & \dots & a_{jj}^{(1)} & \dots & a_{jN}^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{N1}^{(1)} & a_{N2}^{(1)} & \dots & a_{Nj}^{(1)} & \dots & a_{NN}^{(1)} \end{pmatrix} \in \mathbb{R}^{N \times N} \quad \text{where} \quad a_{11}^{(1)} \neq 0$$

$$\text{and} \quad \mathbf{b}^{(1)} = \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} \\ \vdots \\ b_j^{(1)} \\ \vdots \\ b_N^{(1)} \end{pmatrix}.$$

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 $A^{(1)}$.

🔥 $A^{(1)}$

$$A^{(1)} = A = \begin{pmatrix} 2 & -1 & 1 \\ -1 & 1 & 2 \\ 1 & 2 & -1 \end{pmatrix} \quad \text{and} \quad \mathbf{b}^{(1)} = \mathbf{b} = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}.$$

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 \mathbf{m}_1 of multipliers, whose i^{th} entry is given by

$$m_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}} \quad \text{for all } i = 1, 2, \dots, N,$$

hence the reason why the assumption $a_{11}^{(1)} \neq 0$ must be imposed. Essentially, the vector \mathbf{m}_1 is the first column of A divided by the first element of A .

 m_1

$$\mathbf{m}_1 = \frac{1}{a_{11}^{(1)}} \begin{pmatrix} a_{11}^{(1)} \\ a_{21}^{(1)} \\ a_{31}^{(1)} \end{pmatrix} = \begin{pmatrix} 1 \\ -\frac{1}{2} \\ \frac{1}{2} \end{pmatrix}$$

3. **Elimination terms in the first column:** For $j = 2, 3, \dots, N$, multiply row 1 by $-m_{j1}$ and add it to row j to give the new row j :

$$\left(\begin{array}{ccccccc} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1j}^{(1)} & \dots & a_{1N}^{(1)} \\ a_{21}^{(1)} - m_{21}a_{11}^{(1)} & a_{22}^{(1)} - m_{21}a_{12}^{(1)} & \dots & a_{2j}^{(1)} - m_{21}a_{1j}^{(1)} & \dots & a_{2N}^{(1)} - m_{21}a_{1N}^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{j1}^{(1)} - m_{j1}a_{11}^{(1)} & a_{j2}^{(1)} - m_{j1}a_{12}^{(1)} & \dots & a_{jj}^{(1)} - m_{j1}a_{1j}^{(1)} & \dots & a_{jN}^{(1)} - m_{j1}a_{1N}^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{N1}^{(1)} - m_{N1}a_{11}^{(1)} & a_{N2}^{(1)} - m_{N1}a_{12}^{(1)} & \dots & a_{Nj}^{(1)} - m_{N1}a_{1j}^{(1)} & \dots & a_{NN}^{(1)} - m_{N1}a_{1N}^{(1)} \end{array} \right).$$

 Row j Operations

$$\begin{aligned} & \left(\begin{array}{ccc} 2 & -1 & 1 \\ -1 & 1 & 2 \\ 1 & 2 & -1 \end{array} \right) \\ \xrightarrow[r_2 \rightarrow -(-\frac{1}{2})r_1 + r_2]{} & \left(\begin{array}{ccc} 2 & -1 & 1 \\ -\left(-\frac{1}{2}\right)(2) - 1 & -\left(-\frac{1}{2}\right)(-1) + 1 & -\left(-\frac{1}{2}\right)(1) + 2 \\ 1 & 2 & -1 \end{array} \right) \\ &= \left(\begin{array}{ccc} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ 1 & 2 & -1 \end{array} \right) \\ \xrightarrow[r_3 \rightarrow -(\frac{1}{2})r_1 + r_3]{} & \left(\begin{array}{ccc} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ -\left(\frac{1}{2}\right)(2) + 1 & -\left(\frac{1}{2}\right)(-1) + 2 & -\left(\frac{1}{2}\right)(1) - 1 \end{array} \right) \\ &= \left(\begin{array}{ccc} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ 0 & \frac{5}{2} & -\frac{3}{2} \end{array} \right) \end{aligned}$$

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:

$$\begin{aligned} & \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1j}^{(1)} & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(1)} - m_{21}a_{12}^{(1)} & \dots & a_{2j}^{(1)} - m_{21}a_{1j}^{(1)} & \dots & a_{2n}^{(1)} - m_{21}a_{1n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & a_{j2}^{(1)} - m_{j1}a_{12}^{(1)} & \dots & a_{jj}^{(1)} - m_{j1}a_{1j}^{(1)} & \dots & a_{jn}^{(1)} - m_{j1}a_{1n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(1)} - m_{n1}a_{12}^{(1)} & \dots & a_{nj}^{(1)} - m_{n1}a_{1j}^{(1)} & \dots & a_{nn}^{(1)} - m_{n1}a_{1n}^{(1)} \end{pmatrix} \\ \implies & \begin{pmatrix} a_{11}^{(2)} & a_{12}^{(2)} & \dots & a_{1j}^{(2)} & \dots & a_{1n}^{(2)} \\ a_{21}^{(2)} & a_{22}^{(2)} & \dots & a_{2j}^{(2)} & \dots & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{j1}^{(2)} & a_{j2}^{(2)} & \dots & a_{jj}^{(2)} & \dots & a_{jn}^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{n1}^{(2)} & a_{n2}^{(2)} & \dots & a_{nj}^{(2)} & \dots & a_{nn}^{(2)} \end{pmatrix} = A^{(2)} \end{aligned}$$

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 \mathbf{b} has to also undergo the same operations as A , i.e. for $j = 2, \dots, N$, let row j of $\mathbf{b}^{(1)}$ be row 1 multiplied by $-m_{j1}$ plus row j and the final vector is the vector $\mathbf{b}^{(2)}$.

$\mathbf{b}^{(1)}$

$$\mathbf{b}^{(1)} = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix} \xrightarrow{\begin{array}{l} r_2 \rightarrow -\left(-\frac{1}{2}\right)r_1 + r_2 \\ r_3 \rightarrow -\left(\frac{1}{2}\right)r_1 + r_3 \end{array}} \begin{pmatrix} 1 \\ -\left(-\frac{1}{2}\right)(1) + 1 \\ -\left(\frac{1}{2}\right)(1) + 2 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{3}{2} \\ \frac{3}{2} \end{pmatrix} = \mathbf{b}^{(2)}.$$

5. **Matrix representation of elimination:** This whole procedure can be written as

$A^{(2)} = M^{(1)}A^{(1)}$ and $\mathbf{b}^{(2)} = M^{(1)}\mathbf{b}^{(1)}$ where

$$M^{(1)} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ -m_{21} & 1 & 0 & \dots & 0 \\ -m_{31} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -m_{n1} & 0 & 0 & \dots & 1 \end{pmatrix}.$$

🔥 $M^{(1)}$

$$M^{(1)} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix}$$

To check:

$$M^{(1)}A^{(1)} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & 1 \\ -1 & 1 & 2 \\ 1 & 2 & -1 \end{pmatrix} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ 0 & \frac{5}{2} & -\frac{3}{2} \end{pmatrix} = A^{(2)}$$

$$M^{(1)}\mathbf{b}^{(1)} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{3}{2} \\ \frac{3}{2} \end{pmatrix} = \mathbf{b}^{(2)}$$

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:

$$M^{(2)}A^{(2)} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -5 & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ 0 & \frac{5}{2} & -\frac{3}{2} \end{pmatrix} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ 0 & 0 & -14 \end{pmatrix} = A^{(3)}$$

$$M^{(2)}\mathbf{b}^{(2)} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -5 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ \frac{3}{2} \\ \frac{3}{2} \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{3}{2} \\ -6 \end{pmatrix} = \mathbf{b}^{(3)}$$

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

$$\begin{aligned} A^{(1)}\mathbf{x} = \mathbf{b}^{(1)} &\implies A^{(2)}\mathbf{x} = \mathbf{b}^{(2)} \implies A^{(3)}\mathbf{x} = \mathbf{b}^{(3)} \\ \implies \begin{pmatrix} 2 & -1 & 1 \\ 0 & \frac{1}{2} & \frac{5}{2} \\ 0 & 0 & -14 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} &= \begin{pmatrix} 1 \\ \frac{3}{2} \\ -6 \end{pmatrix} \implies \begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ \frac{1}{2}x_2 + \frac{5}{2}x_3 &= \frac{3}{2} \\ -14x_3 &= -6 \end{aligned} \\ \implies \mathbf{x} &= \frac{1}{7} \begin{pmatrix} 5 \\ 6 \\ 3 \end{pmatrix}. \end{aligned}$$

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
$N - 2$	4	4	2
$N - 1$	1	1	1

This means that the total number of multiplications is

$$1 + 4 + \dots + (N - 3)^2 + (N - 2)^2 + (N - 1)^2 = \sum_{n=1}^{N-1} n^2 = \frac{N(N - 1)(2N - 1)}{6},$$

similarly for the additions. Whereas the total number of divisions is

$$1 + 2 + \dots + (N - 3) + (N - 2) + (N - 1) = \sum_{n=1}^{N-1} n = \frac{N(N - 1)}{2}.$$

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 is 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 changing the matrix in a more palatable 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 $M^{(1)}, M^{(2)}, \dots, M^{(N-1)}$ have to be applied reverse order as

$$U = M^{(N-1)}M^{(N-2)} \dots M^{(1)}A.$$

This can be written as

$$U = MA \quad \text{where} \quad M = M^{(N-1)}M^{(N-2)} \dots M^{(1)}. \quad (2.2)$$

Notice that every matrix $M^{(n)}$ is lower triangular and this fact will be used later on in [?@sec-LU](#).

2.3 Iterative Methods

For a large matrix A , solving the system $A\mathbf{x} = \mathbf{b}$ 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 \mathbf{x}_k to the exact solution \mathbf{x} . The hope is that the iterative method converges to the exact solution, i.e.

$$\lim_{k \rightarrow \infty} \mathbf{x}^{(k)} = \mathbf{x}.$$

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

$$\mathbf{x}^{(k+1)} = B\mathbf{x}^{(k)} + \mathbf{g} \quad \text{for } k \geq 0,$$

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

$$\mathbf{x}^{(k+1)} = B\mathbf{x}^{(k)} + \mathbf{g} \quad (2.3)$$

$$\Rightarrow \lim_{k \rightarrow \infty} \mathbf{x}^{(k+1)} = B\mathbf{x} + \mathbf{g}.$$

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

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

Subtracting these gives

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

For the sake of argument, consider the case when the matrix B is symmetric. In order to find a bound for the error, take the 2-norm of the error equation

$$\|\mathbf{e}^{(k+1)}\|_2 = \|B\mathbf{e}^{(k)}\|_2 \quad \Rightarrow \quad \|\mathbf{e}^{(k+1)}\|_2 = \|B\mathbf{e}^{(k)}\|_2.$$

By the submultiplicative property of matrix norms given in Note 1 and the fact that the 2-norm of a symmetric matrix is equal to the spectral radius (largest eigenvalue in absolute terms), the error $\|\mathbf{e}^{(k+1)}\|$ can be bounded above

$$\|\mathbf{e}^{(k+1)}\|_2 = \|B\mathbf{e}^{(k)}\|_2 \leq \|B\|\|\mathbf{e}^{(k)}\|_2 = \rho(B)\|\mathbf{e}^{(k)}\|_2.$$

This can be iterated backwards, so for $k \geq 0$,

$$\|\mathbf{e}^{(k+1)}\|_2 \leq \rho(B)\|\mathbf{e}^{(k)}\|_2 \leq \rho(B)^2\|\mathbf{e}^{(k-1)}\|_2 \leq \dots \leq \rho(B)^{k+1}\|\mathbf{e}^{(0)}\|_2.$$

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

2.3.1 Constructing an Iterative Method

A general technique to devise an iterative method to solve $A\mathbf{x} = \mathbf{b}$ 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

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

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

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

which is of the form given in [?@eq-Bxg](#) where $B = P^{-1}(P - A) = I - P^{-1}A$ and $\mathbf{g} = P^{-1}\mathbf{b}$. It would then stand to reason that if the iterative procedure was of the form

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

(as in Equation 2.3), 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

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

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 of A (i.e. not including the diagonal) and U is the strictly upper triangular part of $A = D + L + U$.

i Note

For example

$$\underbrace{\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}}_A = \underbrace{\begin{pmatrix} a & & \\ & e & \\ & & i \end{pmatrix}}_D + \underbrace{\begin{pmatrix} 0 & b & c \\ 0 & 0 & f \\ 0 & 0 & 0 \end{pmatrix}}_L + \underbrace{\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}}_U.$$

The matrix P can be chosen as a combination of D , L and U in some way to generate the matrix B . There are many ways in which this can be done, each with their own advantages:

- **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 $\mathbf{x}^{(k+1)} = B\mathbf{x}^{(k)} + D^{-1}\mathbf{b}$ for $k \geq 0$, which can be written component-wise as

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

If A is strictly diagonally dominant by rows², then the Jacobi method converges (i.e. $\rho(B) < 1$, where $B = I - D^{-1}A$). Note that each component $x_i^{(k+1)}$ of the new vector $\mathbf{x}^{(k+1)}$ is computed independently of the others, meaning that the update is simultaneous which makes this method suitable for parallel programming.

²A 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

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

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

- **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 $\mathbf{x}^{(k+1)} = B\mathbf{x}^{(k)} + (D + L)^{-1}\mathbf{b}$ which can be written component-wise as

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

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 $\omega \neq 0$
- 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 $A\mathbf{x} = \mathbf{b}$: 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 $\mathcal{O}(N^2)$. 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 $\mathcal{O}(kN^2)$; 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 $\mathbf{x}^{(k)}$ 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. $\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$. More precisely, an iterative method can be stopped at the first iteration step $k = k_{\min}$ for which

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

When the above estimate is satisfied, it is guaranteed that

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

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. $\delta^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$. More precisely, iterative method would stop after the first iteration step $k = k_{\min}$ for which

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

If B is symmetric and positive definite, then

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

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

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

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

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

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

$$\underbrace{\begin{pmatrix} 1 & 1 & 1 \\ 1 & -2 & 0 \\ 0 & 1 & -1 \end{pmatrix}}_A \underbrace{\begin{pmatrix} a \\ b \\ c \end{pmatrix}}_x = \underbrace{\begin{pmatrix} 20 \\ 0 \\ 10 \end{pmatrix}}_b.$$

This can be solved as follows:

```
1 >> A=[1,1,1;1,-2,0;0,1,-1];
2 >> b=[20;0;10];
3 >> A\b
4 ans =
5     15.0000
6     7.5000
7    -2.5000
```

Note

The [MATLAB website](#) shows the following flowcharts for how $A \setminus b$ classifies the problem before solving it.

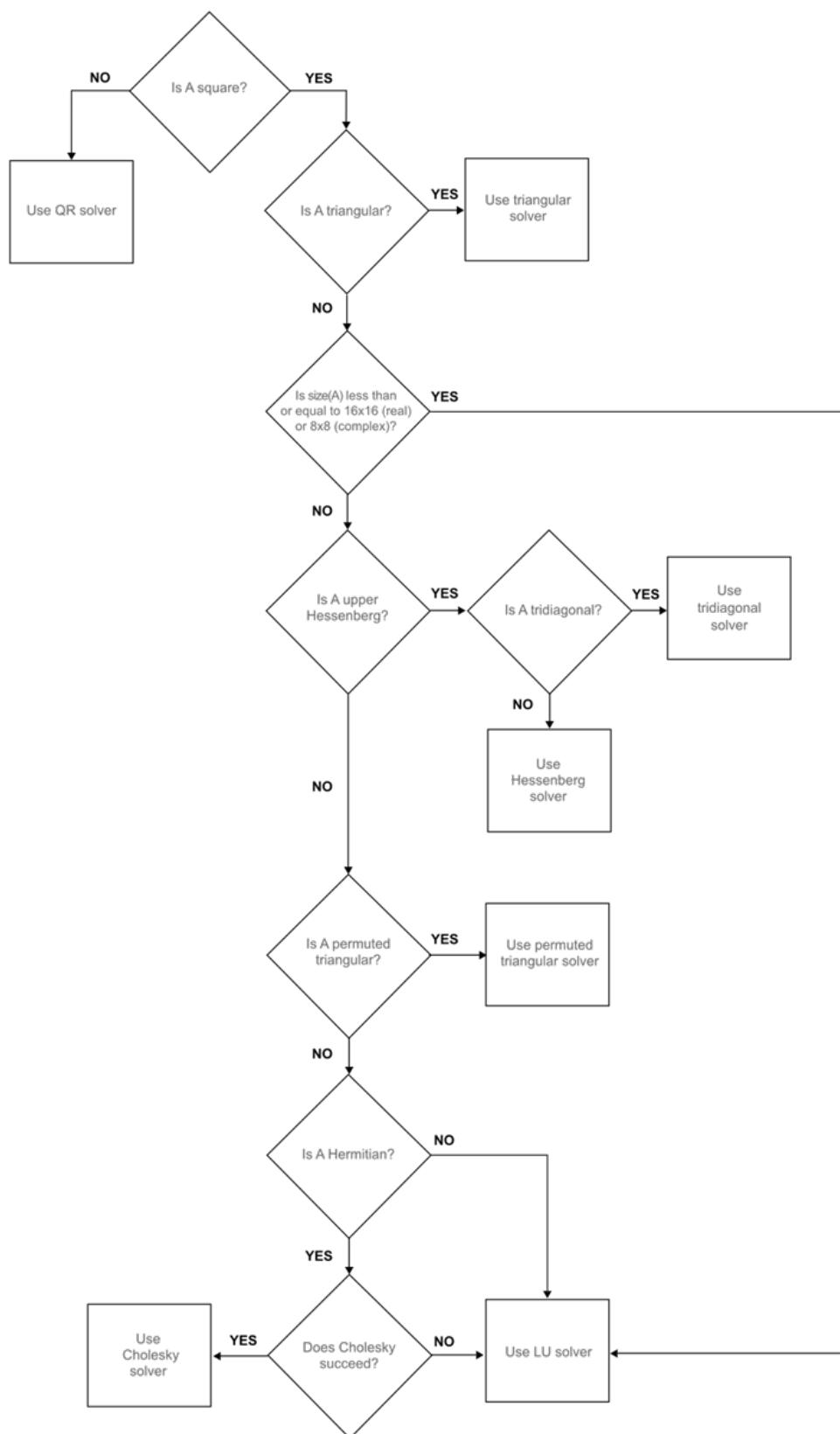


Figure 2.1: If the matrix A is full.

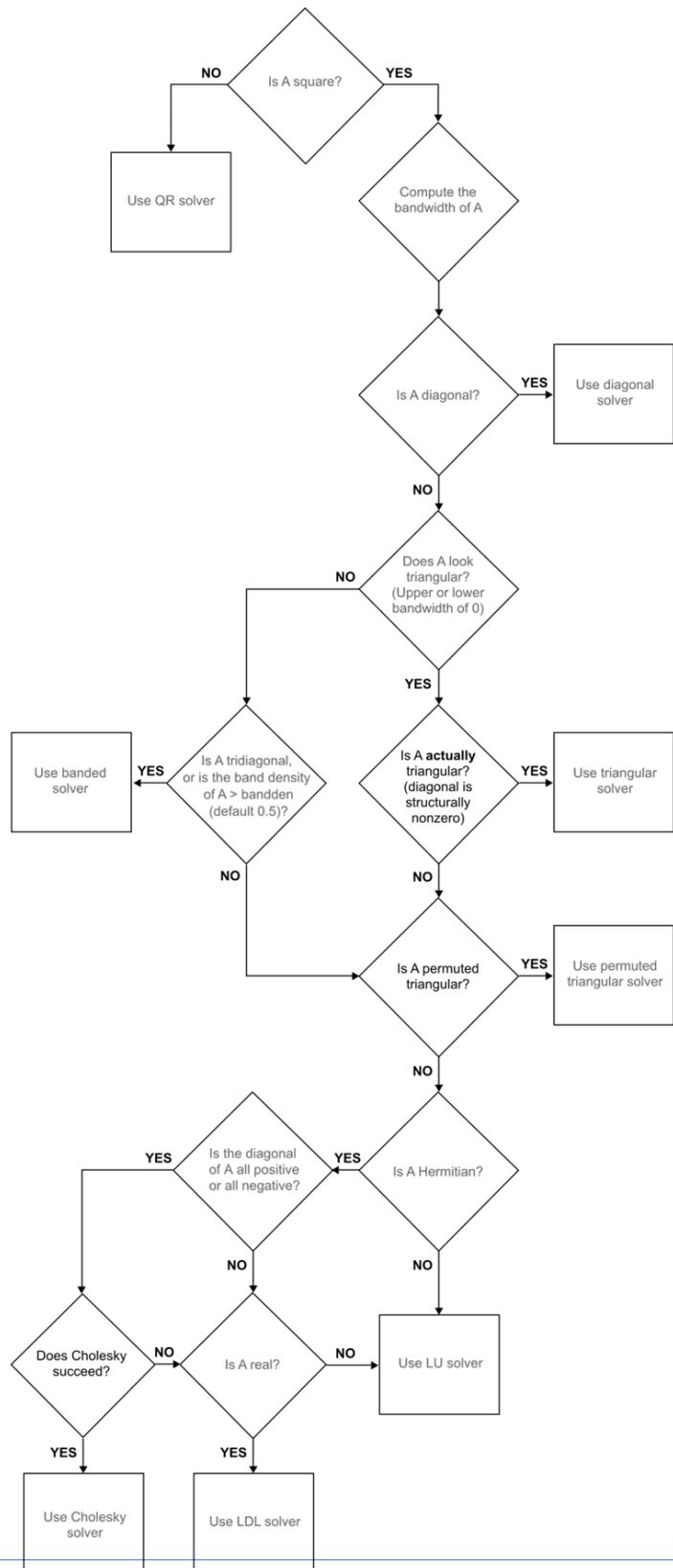


Figure 2.2: If the matrix A is sparse.

Part II

Solving Initial Value Problems

3 The Euler Method

Consider the first order ordinary differential equation (ODE)

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

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***)

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

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***.

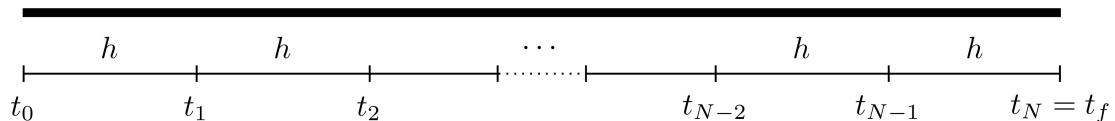
Consider the following first order IVP

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

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 constant¹ width h (sometimes also denoted δt), which is known as the ***Stepsize***. The discretised interval will be the set of points

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



¹In 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 $\frac{dy}{dt}$, 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

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

Parallel Example

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

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

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

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

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

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

Inerval 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,

$$y'(t_0) = f(t_0, y_0) \implies y'(0) = f(0, 0) = 6 - 2(0) = 6.$$

So the starting gradient is 6.

3. The next step is to find the value of y at the subsequent time $t_1 = t_0 + h$. For this purpose, consider the Taylor series expansion of y at $t = t_1$,

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

Note

The term $\mathcal{O}(h^3)$ 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,

$$Y_1 = y_0 + hy'(t_0). \quad (3.1)$$

This determines the value of Y_1 which is an approximation to $y(t_1)$.

Calculating 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 $n = 1, 2, \dots, N - 1$

$$Y_{n+1} = Y_n + hy'(t_n) \quad \text{where} \quad y'(t_n) = f(t_n, Y_n).$$

🔥 Calculating \mathbf{Y}_n

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

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

$$\Rightarrow \quad Y_2 = Y_1 + hy'(t_1) = 2.4 + (0.4)(1.2) = 2.88$$

$$Y_3 : \quad y'(t_2) = f(t_2, Y_2) \quad \Rightarrow \quad y'(0.8) = f(0.8, 2.88) = 6 - 2(2.88) = 0.24$$

$$\Rightarrow \quad Y_3 = Y_2 + hy'(t_2) = 2.88 + (0.4)(0.24) = 2.976$$

$$Y_4 : \quad y'(t_3) = f(t_3, Y_3) \quad \Rightarrow \quad y'(1.2) = f(1.2, 2.976) = 6 - 2(2.976) = 0.048$$

$$\Rightarrow \quad Y_4 = Y_3 + hy'(t_3) = 2.976 + (0.4)(0.048) = 2.9952$$

$$Y_5 : \quad y'(t_4) = f(t_4, Y_4) \quad \Rightarrow \quad y'(1.6) = f(1.6, 2.9952) = 6 - 2(2.9952) = 0.0096$$

$$\Rightarrow \quad Y_5 = Y_4 + hy'(t_4) = 2.9952 + (0.4)(0.0096) = 2.99904$$

5. The solution to the IVP can now be approximated by the function that passes through the points

$$(t_0, Y_0), \quad (t_1, Y_1), \quad \dots \quad (t_N, Y_N).$$

🔥 Solution to the IVP

The approximate solution to the IVP

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

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 $n \in \{1, 2, \dots, N\}$ 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 $t_1 = t_0 + h$,

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

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

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

This can be done for all the locations to give a list of local truncation errors $e_1, e_2, e_3, \dots, e_N$. 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 $e = \mathcal{O}(h^2)$.

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

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

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

$$\begin{aligned}
E = |y(t_f) - Y_N| &\leq \underbrace{\sum_{n=1}^N e_n}_{\substack{\text{sum of all} \\ \text{local truncation} \\ \text{errors}}} = \sum_{n=1}^N \frac{h^2}{2!} |y''(\tau_n)| = h^2 \sum_{n=1}^N \frac{1}{2} |y''(\tau_n)|. \\
\implies E &\leq h^2 \sum_{n=1}^N \frac{1}{2} |y''(\tau_n)| \tag{3.2}
\end{aligned}$$

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.

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

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

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

This can also be expressed differently as

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

Therefore, since

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

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.

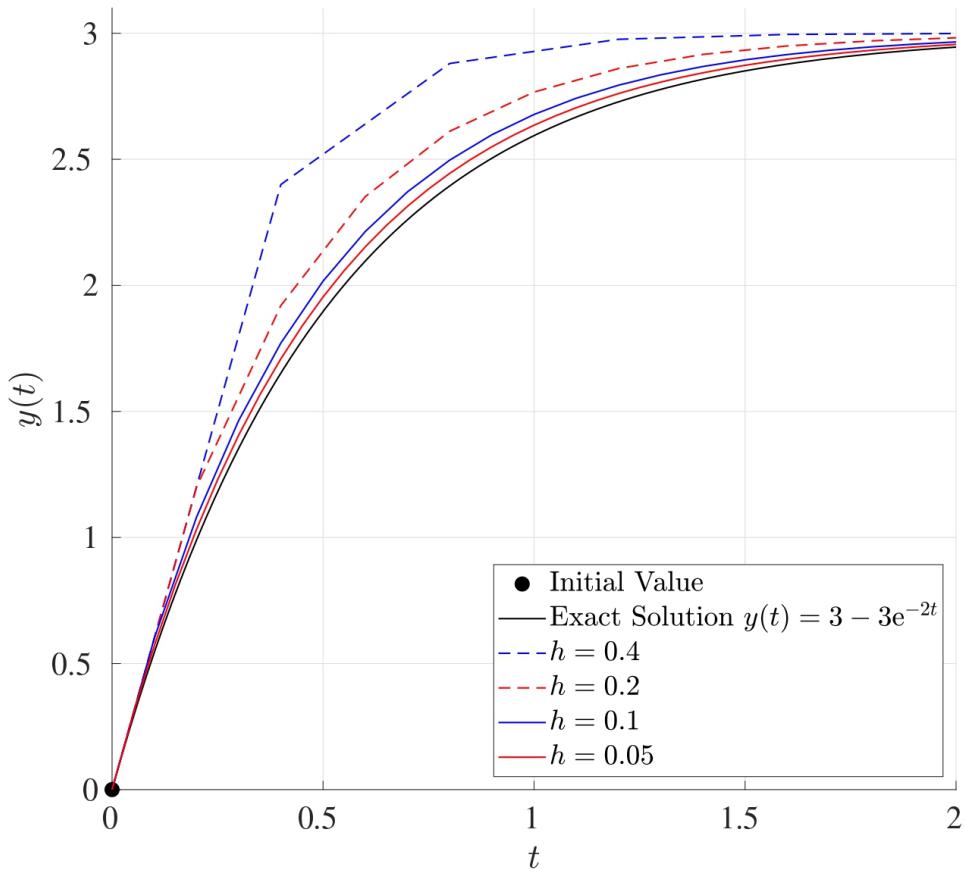
In conclusion, the local truncation error of the Euler method is $e = \mathcal{O}(h^2)$ while the global integration error $E = \mathcal{O}(h)$ when h is small.

🔥 Different Stepsizes

Returning to the IVP

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

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 $[t_0, t_f]$:

$$\begin{aligned} \frac{dy_1}{dt} &= a_{11}y_1 + a_{12}y_2 + \cdots + a_{1K}y_K + b_1, & y_1(t_0) &= \tilde{y}_1 \\ \frac{dy_2}{dt} &= a_{21}y_1 + a_{22}y_2 + \cdots + a_{2K}y_K + b_2, & y_2(t_0) &= \tilde{y}_2 \\ &\vdots && \\ \frac{dy_K}{dt} &= a_{K1}y_1 + a_{K2}y_2 + \cdots + a_{KK}y_K + b_K, & y_K(t_0) &= \tilde{y}_K \end{aligned}$$

where, for $i, j = 1, 2, \dots, K$, 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

$$\begin{aligned} \frac{d\mathbf{y}}{dt} &= A\mathbf{y} + \mathbf{b} \quad \text{with} \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad t \in [t_0, t_f] \\ \text{where } \mathbf{y}(t) &= \begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_K(t) \end{pmatrix}, \quad A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1K} \\ a_{21} & a_{22} & \cdots & a_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ a_{K1} & a_{K2} & \cdots & a_{KK} \end{pmatrix}, \\ \mathbf{b} &= \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_K \end{pmatrix}, \quad \mathbf{y}_0 = \begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \\ \vdots \\ \tilde{y}_K \end{pmatrix}. \end{aligned}$$

In this case, $\mathbf{y}(t)$ is the unknown solution vector, A is a matrix of constants, \mathbf{y}_0 is the vector of initial values and \mathbf{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, \dots, t_N) where $t_n = t_0 + nh$ for $n = 0, 1, \dots, N$. Let \mathbf{Y}_n be the approximation to the function vector \mathbf{y} at the time $t = t_n$, then

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

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

Sets of IVPs

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

$$\begin{aligned}\frac{dy}{dt} &= y + 2z, \quad y(0) = 1 \\ \frac{dz}{dt} &= \frac{3}{2}y - z, \quad z(0) = 0\end{aligned}$$

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

$$\frac{d\mathbf{y}}{dt} = A\mathbf{y} + \mathbf{b} \quad \text{with} \quad \mathbf{y}(0) = \mathbf{y}_0.$$

In this case,

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

Let $N = 5$, so

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

The Euler iteration will be

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

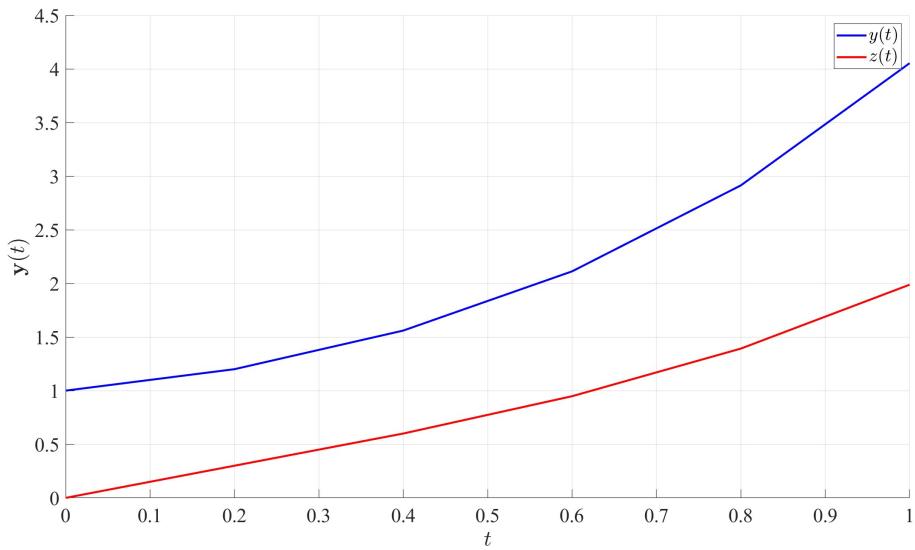
This can be written as

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

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

$$\begin{aligned}\mathbf{Y}_1 &= \mathbf{Y}_0 + 0.2 [A\mathbf{Y}_0 + \mathbf{b}_0] = \begin{pmatrix} 1 \\ 0 \end{pmatrix} + 0.2 \left[\begin{pmatrix} 1 & 2 \\ \frac{3}{2} & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right] = \begin{pmatrix} 1.2 \\ 0.3 \end{pmatrix} \\ \mathbf{Y}_2 &= \mathbf{Y}_1 + 0.2 [A\mathbf{Y}_1 + \mathbf{b}_1] = \begin{pmatrix} 1.2 \\ 0.3 \end{pmatrix} + 0.2 \left[\begin{pmatrix} 1 & 2 \\ \frac{3}{2} & -1 \end{pmatrix} \begin{pmatrix} 1.2 \\ 0.3 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right] = \begin{pmatrix} 1.56 \\ 0.6 \end{pmatrix} \\ \mathbf{Y}_3 &= \mathbf{Y}_2 + 0.2 [A\mathbf{Y}_2 + \mathbf{b}_2] = \begin{pmatrix} 1.56 \\ 0.6 \end{pmatrix} + 0.2 \left[\begin{pmatrix} 1 & 2 \\ \frac{3}{2} & -1 \end{pmatrix} \begin{pmatrix} 1.56 \\ 0.6 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right] = \begin{pmatrix} 2.112 \\ 0.948 \end{pmatrix} \\ \mathbf{Y}_4 &= \mathbf{Y}_3 + 0.2 [A\mathbf{Y}_3 + \mathbf{b}_3] = \begin{pmatrix} 2.112 \\ 0.948 \end{pmatrix} + 0.2 \left[\begin{pmatrix} 1 & 2 \\ \frac{3}{2} & -1 \end{pmatrix} \begin{pmatrix} 2.112 \\ 0.948 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right] = \begin{pmatrix} 2.9136 \\ 1.3920 \end{pmatrix} \\ \mathbf{Y}_5 &= \mathbf{Y}_4 + 0.2 [A\mathbf{Y}_4 + \mathbf{b}_4] = \begin{pmatrix} 2.9136 \\ 1.3920 \end{pmatrix} + 0.2 \left[\begin{pmatrix} 1 & 2 \\ \frac{3}{2} & -1 \end{pmatrix} \begin{pmatrix} 2.9136 \\ 1.3920 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right] = \begin{pmatrix} 4.0531 \\ 1.9877 \end{pmatrix}\end{aligned}$$

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



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 $[t_0, t_f]$

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

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

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

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

$$\begin{aligned} y_1(t) &= \frac{dy}{dt} \\ y_2(t) &= y'_1(t) = \frac{d^2y}{dt^2} \\ y_3(t) &= y'_2(t) = \frac{d^3y}{dt^3} \\ &\vdots \\ y_{K-3}(t) &= y'_{K-4}(t) = \frac{d^{K-3}y}{dt^{K-3}} \\ y_{K-2}(t) &= y'_{K-3}(t) = \frac{d^{K-2}y}{dt^{K-2}} \\ y_{K-1}(t) &= y'_{K-2}(t) = \frac{d^{K-1}y}{dt^{K-1}} \end{aligned}$$

Notice that

$$\begin{aligned} \frac{dy_{K-1}}{dt} &= \frac{d^Ky}{dt^K} = -a_{K-1}\frac{d^{K-1}y}{dt^{K-1}} - \dots - a_2\frac{d^2y}{dt^2} - a_1\frac{dy}{dt} - a_0y + f(t) \\ &= -a_{K-1}y_{K-1} - \dots - a_2y_2 - a_1y_1 - a_0y + f(t) \end{aligned}$$

Let \mathbf{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 $\mathbf{y}' = A\mathbf{y} + \mathbf{b}$ as follows:

$$\begin{aligned} \frac{d\mathbf{y}}{dt} &= \frac{d}{dt} \begin{pmatrix} y \\ y_1 \\ y_2 \\ \vdots \\ y_{K-3} \\ y_{K-2} \\ y_{K-1} \end{pmatrix} = \begin{pmatrix} y' \\ y'_1 \\ y'_2 \\ \vdots \\ y'_{K-3} \\ y'_{K-2} \\ y'_{K-1} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{K-2} \\ y_{K-1} \\ \frac{d^Ky}{dt^K} \end{pmatrix} \\ &= \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{K-2} \\ y_{K-1} \\ -a_{K-1}y_{K-1} - \dots - a_2y_2 - a_1y_1 - a_0y + f(t) \end{pmatrix} \end{aligned}$$

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

The initial condition vector will be

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

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

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

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

Let $u = \frac{dy}{dt}$, $v = u' = \frac{d^2y}{dt^2}$ and $w = v' = \frac{d^3y}{dt^3}$. The derivatives of u, v and w are:

$$u' = v$$

$$v' = w$$

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

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

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

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

The initial condition vector will be

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

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

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

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

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

$$\begin{aligned} \frac{d\mathbf{y}}{dt} &= \frac{d}{dt} \begin{pmatrix} y \\ y' \\ z \\ z' \\ z'' \\ z''' \end{pmatrix} = \begin{pmatrix} y' \\ y'' \\ z' \\ z'' \\ z''' \\ z'''' \end{pmatrix} = \begin{pmatrix} y' \\ -y - 6y' + \sin(t) \\ z' \\ z'' \\ 5y - 2y' + 8z'' + e^{2t} \\ z'''' \end{pmatrix} \\ &= \underbrace{\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ -1 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 5 & -2 & 0 & 0 & 8 \end{pmatrix}}_A \underbrace{\begin{pmatrix} y \\ y' \\ z \\ z' \\ z'' \\ z''' \end{pmatrix}}_y + \underbrace{\begin{pmatrix} 0 \\ \sin(t) \\ 0 \\ 0 \\ 0 \\ e^{2t} \end{pmatrix}}_b. \end{aligned}$$

The vector of initial values would be

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

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

$$\frac{d\mathbf{y}}{dt} = A\mathbf{y}.$$

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

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

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

Definition 3.1. The initial value problem

$$\frac{d\mathbf{y}}{dt} = A\mathbf{y} + \mathbf{b} \quad \text{with} \quad \mathbf{y}(0) = \mathbf{y}_0$$

is said to be **Asymptotically Stable** if $\mathbf{y} \rightarrow \mathbf{0}$ as $t \rightarrow \infty$, in other words, all functions in \mathbf{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*

$$\frac{d\mathbf{y}}{dt} = A\mathbf{y} + \mathbf{b}$$

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 asymptotically stable.

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

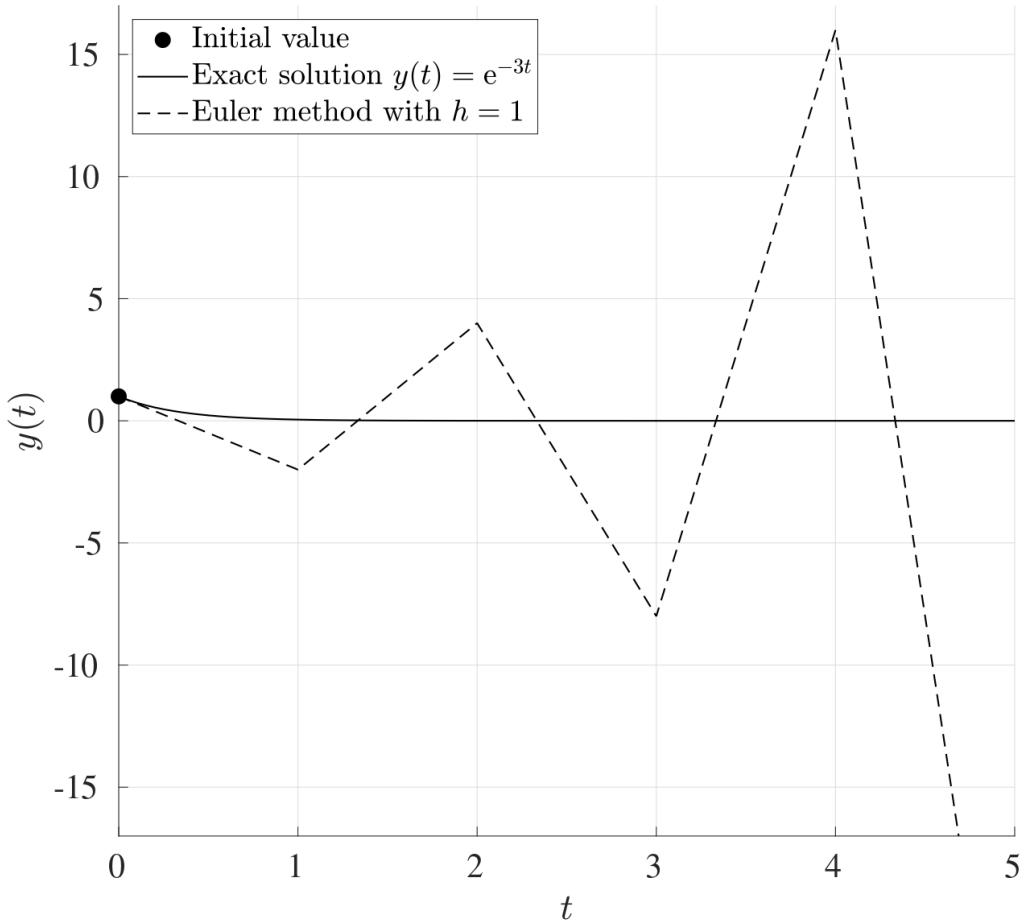
3.5 Limitations of the Euler Method

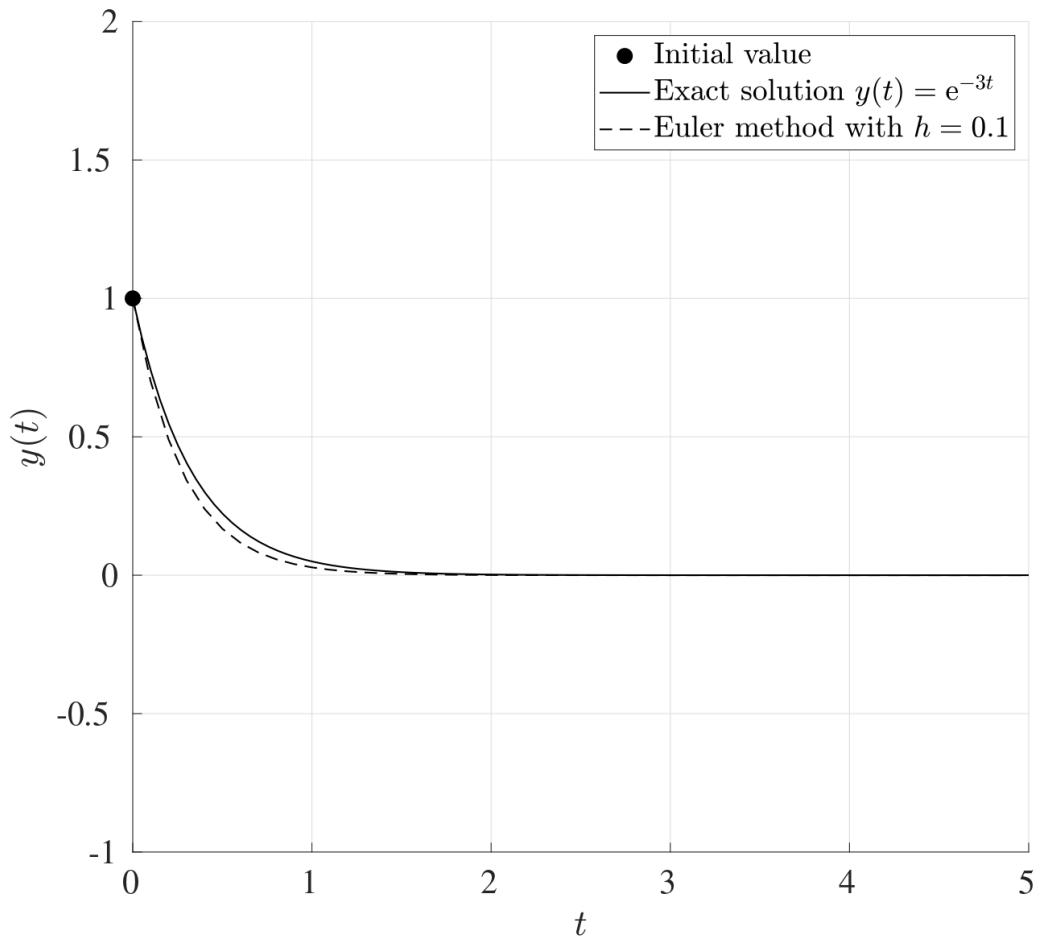
In some cases, if the stepsize h is taken to be too large, then the Euler method can give misleading results.

For example, consider the initial value problem:

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

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:

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

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 .

³In 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

$$\frac{dy}{dt} = Ay + b \quad \text{with} \quad y(0) = y_0.$$

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.)

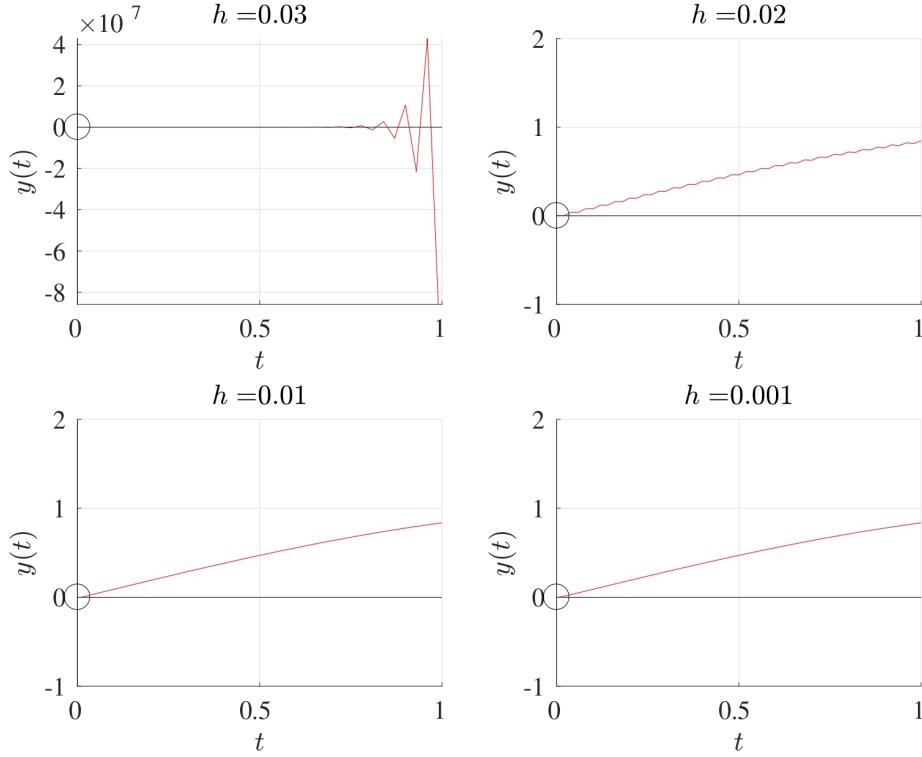
If the stepsize is too large, then the method may not converge but on the other hand if it is too low, then the iteration will take a considerable amount of time to perform. Therefore an “optimal” stepsize is needed to obtain sufficiently accurate solutions.

Different Stepsizes

Consider the following initial value problem

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

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

$$\frac{dy}{dt} = Ay + b \quad \text{with} \quad y(0) = y_0.$$

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

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

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

In other words, if the initial value problem is asymptotically stable, then the Euler method is stable if and 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

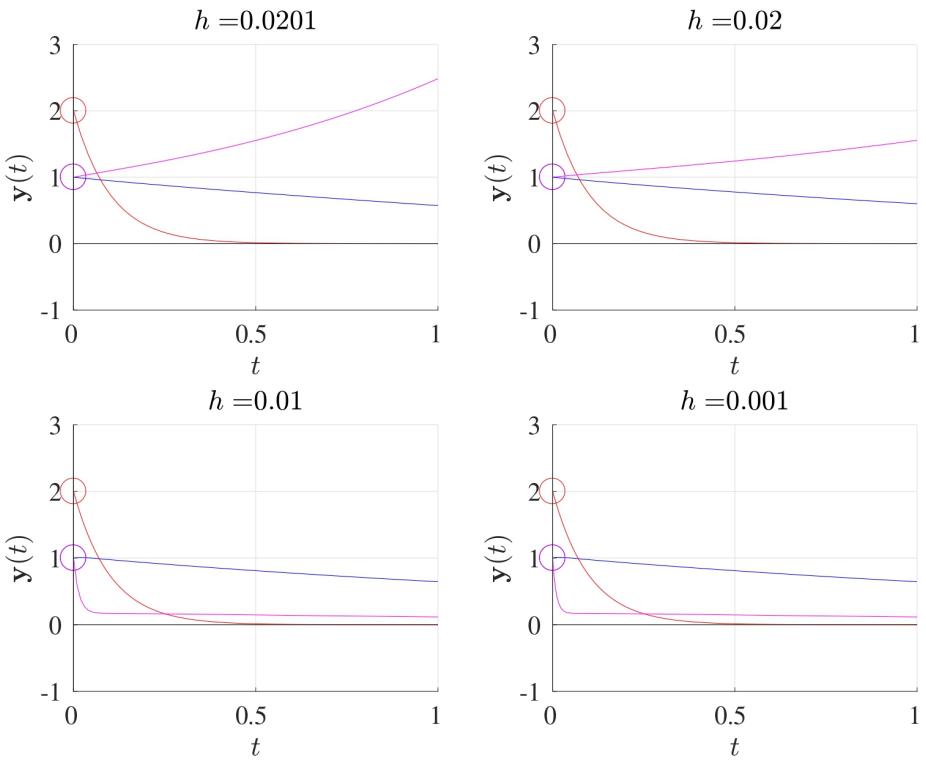
Consider the system of differential equations $y' = Ay$ with $y(0) = y_0$ where

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

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

$$\begin{aligned} h_0 &= 2 \min \left\{ \frac{1}{|\lambda_k|} \right\} = 2 \min \left\{ \frac{1}{|-0.4575|}, \frac{1}{|-100.5425|}, \frac{1}{|-10|} \right\} \\ &= 2 \min \{2.0858, 0.0099, 0.1\} = 2 \times 0.0099 = 0.0199. \end{aligned}$$

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 $\mathbf{x} = (x_1, x_2, \dots, x_n)$, the **sup-norm** of \mathbf{x} is the maximum absolute value in the vector, i.e.

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

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 \mathbf{x} and matrix M given by

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

The sup-norm of \mathbf{x} is simply the largest absolute element which is 9, therefore $\|\mathbf{x}\|_\infty = 9$. 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:

$$\begin{pmatrix} 5 & 2 & 4 & 1 \\ -9 & 5 & 3 & -7 \\ 6 & 0 & -1 & 4 \\ 9 & 5 & -2 & 4 \end{pmatrix} \xrightarrow{|\bullet|} \begin{pmatrix} 5 & 2 & 4 & 1 \\ 9 & 5 & 3 & 7 \\ 6 & 0 & 1 & 4 \\ 9 & 5 & 2 & 4 \end{pmatrix} \begin{cases} \rightarrow 12 \\ \rightarrow 24 \\ \rightarrow 11 \\ \rightarrow 20 \end{cases} \text{ maximum is } 24.$$

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

$$\frac{d\mathbf{y}}{dt} = A\mathbf{y} + \mathbf{b} \quad \text{with} \quad \mathbf{y}(0) = \mathbf{y}_0$$

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 $\mathbf{y}' = A\mathbf{y}$ 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:

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

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 $\mathbf{y}' = A\mathbf{y}$ 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:

$$\xrightarrow{|\bullet|} \begin{pmatrix} |1-h| & 0 & 0 & 0 & 0 \\ h & |1-h| & 0 & 0 & 0 \\ 0 & h & |1-h| & 0 & 0 \\ 0 & 0 & h & |1-h| & 0 \\ 0 & 0 & 0 & h & |1-h| \end{pmatrix} \rightarrow \begin{array}{l} |1-h| \\ |1-h|+h \\ |1-h|+h \\ |1-h|+h \\ |1-h|+h \end{array}$$

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 $\mathbf{y}' = A\mathbf{y}$ where

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

Find the sup-norm:

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

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

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

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

- ii. Suppose that $a < b$, then

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

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 $-2 \pm 2.65i$ 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 $[0, 1]$:

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

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.

```

1 function IVP_Euler
2
3 %% Solve a set of first order IVPs using Euler
4
5 % This code solves a set of IVP when written explicitly
6 % on the interval [t0,tf] subject to the initial conditions
7 % y(0)=y0. The output will be the graph of the solution(s)
8 % and the vector value at the final point tf. Note that the
9 % IVPs do not need to be linear or homogeneous.
10
11 %% Lines to change:
12
13 % Line 28 : t0 - Start time
14 % Line 31 : tf - End time
15 % Line 34 : N - Number of subdivisions
16 % Line 37 : y0 - Vector of initial values
17 % Line 105+ : Which functions to plot, remembering to assign
18 %               a colour, texture and legend label

```

```

19 % Line 125+ : Set of differential equations written
20 % explicitly. These can also be non-linear and
21 % include forcing terms. These equations can
22 % also be written in matrix form if the
23 % equations are linear.
24
25 %% Set up input values
26
27 % Start time
28 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 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     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

```

```

109 % Legend labels
110 legend('$y_1(t)$','$y_2(t)$','$y_3(t)$')
111 set(legend,'Interpreter','Latex')
112
113 % Display the values of the vector y at tf
114 disp(strcat('The vector y at t=',num2str(tf),' is:'))
115 disp(Y(:,end))
116
117 end
118
119 function [dydt]=DYDT(t,y,K)
120
121 % When the equation are written in explicit form
122
123 dydt=zeros(K,1);
124
125 dydt(1)=2*y(1)+y(2)+y(3)+cos(t);
126
127 dydt(2)=sin(y(1))+exp(-y(2)+y(3));
128
129 dydt(3)=y(1)*y(2)-y(3);
130
131 % If the set of equations is linear, then these can be
132 % written in matrix form as dydt=A*y+b(t). For example, if
133 % the set of equations is:
134 % dudt = 7u - 2v + w + exp(t)
135 % dvdt = 2u + 3v - 9w + cos(t)
136 % dwdt = 2v + 5w + 2
137 % Then:
138 % A=[7,-2,1;2,3,-9;0,2,5];
139 % b=@(t) [exp(t);cos(t);2];
140 % dydt=A*y+b(t)
141
142 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 be 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),

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

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;

$$\begin{aligned} y(t_1) - y(t_0) &= \int_{t_0}^{t_1} y'(t) dt \\ &= \int_{t_0}^{t_1} y'(t_0) dt \\ &= hy'(t_0) \end{aligned}$$

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

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 $y'(t_1)$, namely

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

Thus

$$\begin{aligned}
 y(t_1) - y(t_0) &= \int_{t_0}^{t_1} y'(t) \, dt \\
 &= \int_{t_0}^{t_1} \frac{1}{2} (y'(t_0) + y'(t_1)) \, dt \\
 &= \frac{h}{2} (y'(t_0) + y'(t_1)) \\
 \implies y(t_1) &= y(t_0) + \frac{h}{2} (y'(t_0) + y'(t_1)).
 \end{aligned}$$

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

🔥 Modified Euler Method

Consider the differential equation

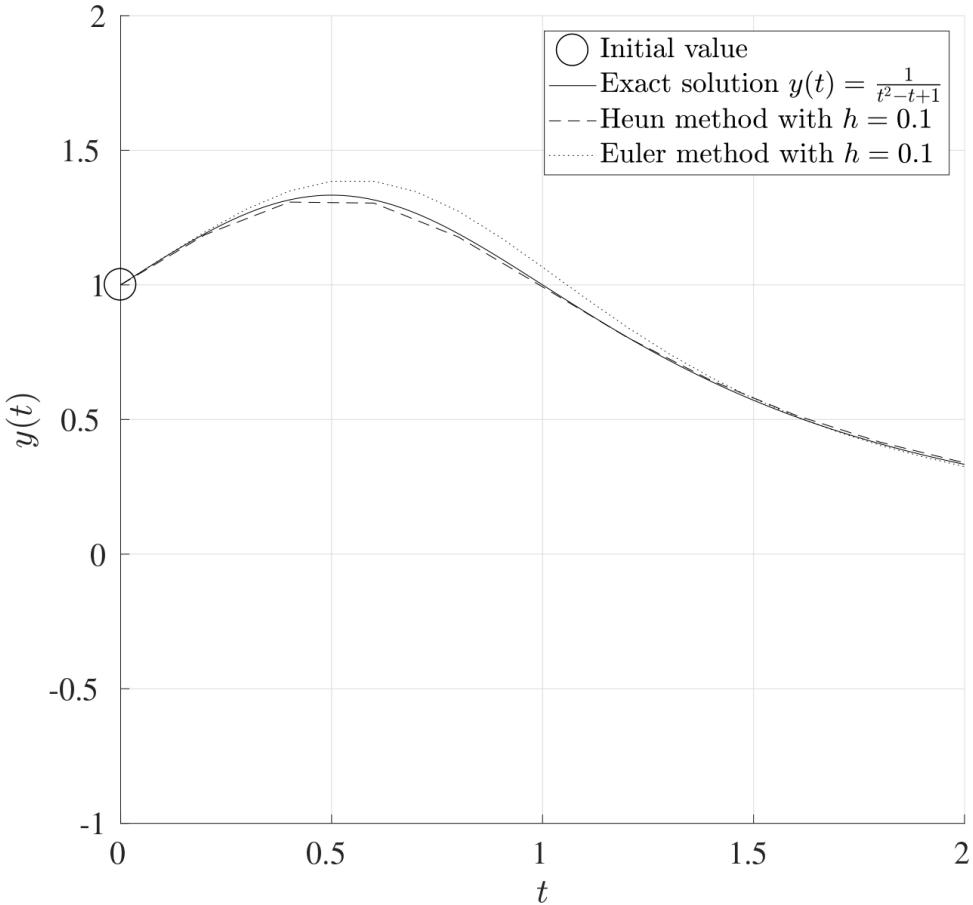
$$\frac{dy}{dt} = (1 - 2t)y^2 \quad \text{with } y(0) = 1, \quad t \in [0, 2].$$

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 $t_{0.5} = t_0 + \frac{1}{2}h$:

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

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

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

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

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.

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

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

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

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

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

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

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

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

Hence

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

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,

$$\begin{aligned} \underbrace{y(t_1)}_{\approx Y_1} &= \underbrace{y(t_0)}_{=Y_0} + \frac{h}{2} [\underbrace{y'(t_1)}_{=f(t_1, \tilde{Y}_1)} + \underbrace{y'(t_0)}_{=f(t_0, Y_0)}] + \mathcal{O}(h^3) \\ &\implies Y_1 = Y_0 + \frac{h}{2} [f(t_1, \tilde{Y}_1) + f(t_0, Y_0)]. \end{aligned} \quad (4.3)$$

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

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

Therefore the *local truncation error* $e = \mathcal{O}(h^3)$ 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 $E = \mathcal{O}(h^2)$ 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 $[0, 1]$:

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

i 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.

```
1 function IVP_Mod_Euler
2
3 %% Solve a set of first order IVPs using Modified Euler
4
5 % This code solves a set of IVP when written explicitly
6 % on the interval [t0,tf] subject to the initial conditions
7 % y(0)=y0. The output will be the graph of the solution(s)
8 % and the vector value at the final point tf. Note that the
9 % IVPs do not need to be linear or homogeneous.
10
11 %% Lines to change:
12
13 % Line 28 : t0 - Start time
14 % Line 31 : tf - End time
15 % Line 34 : N - Number of subdivisions
16 % Line 37 : y0 - Vector of initial values
17 % Line 115+ : Which functions to plot, remembering to assign
18 %               a colour, texture and legend label
19 % Line 135+ : Set of differential equations written
20 %               explicitly. These can also be non-linear and
21 %               include forcing terms. These equations can
22 %               also be written in matrix form if the
23 %               equations are linear.
24
25 %% Set up input values
26
27 % Start time
28 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

```

```

119 % Legend labels
120 legend('$y_1(t)$','$y_2(t)$','$y_3(t)$')
121 set(legend,'Interpreter','Latex')
122
123 % Display the values of the vector y at tf
124 disp(strcat('The vector y at t=',num2str(tf),' is:'))
125 disp(Y(:,end))
126
127 end
128
129 function [dydt]=DYDT(t,y,K)
130
131 % When the equation are written in explicit form
132
133 dydt=zeros(K,1);
134
135 dydt(1)=2*y(1)+y(2)+y(3)+cos(t);
136
137 dydt(2)=sin(y(1))+exp(-y(2)+y(3));
138
139 dydt(3)=y(1)*y(2)-y(3);
140
141 % If the set of equations is linear, then these can be
142 % written in matrix form as dydt=A*y+b(t). For example, if
143 % the set of equations is:
144 % dudt = 7u - 2v + w + exp(t)
145 % dvdt = 2u + 3v - 9w + cos(t)
146 % dwdt = 2v + 5w + 2
147 % Then:
148 % A=[7,-2,1;2,3,-9;0,2,5];
149 % b=@(t) [exp(t);cos(t);2];
150 % dydt=A*y+b(t)
151
152 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 $\mathcal{O}(h^2)$. 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 $\mathcal{O}(h^p)$ 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 $\mathcal{O}(h^4)$, this is formulated as follows:

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

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.

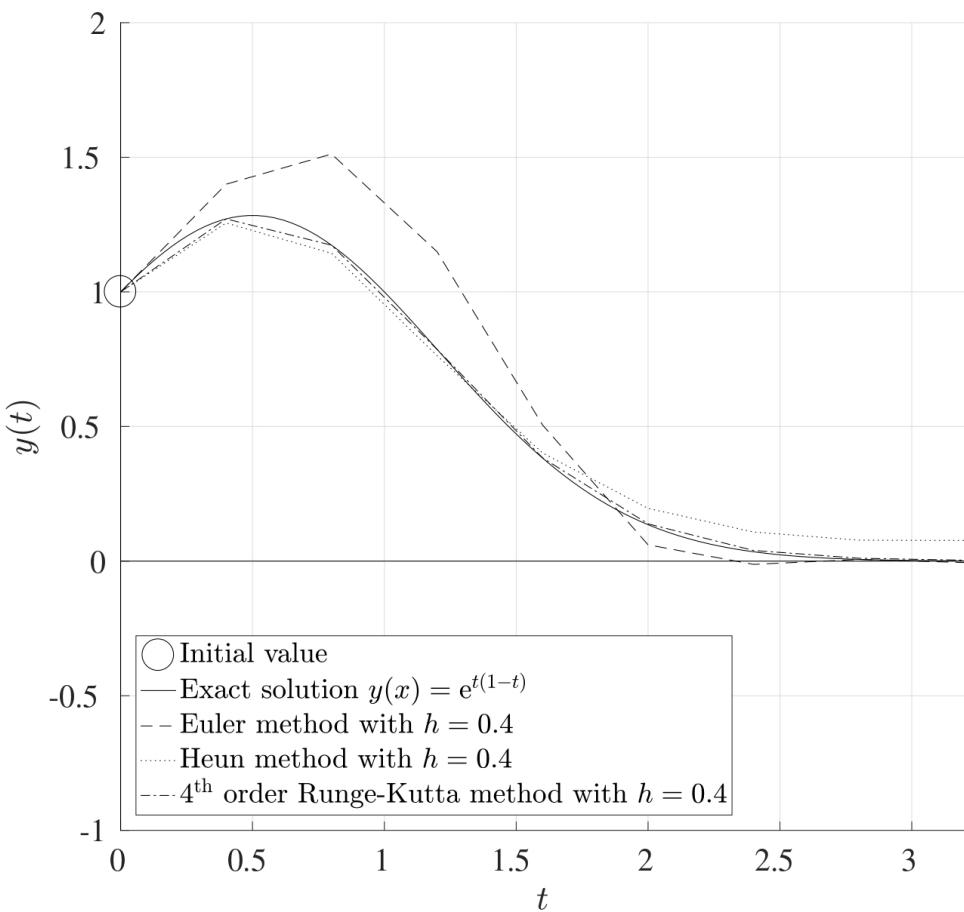
🔥 Runge-Kutta Method

Consider the differential equation

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

The exact solution to this differential equation is known to be

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



5.1 MATLAB Code

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

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

i 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.

```
1 function IVP_RK4
2
3 %% Solve a set of first order IVPs using RK4
4
5 % This code solves a set of IVP when written explicitly
6 % on the interval [t0,tf] subject to the initial conditions
7 % y(0)=y0. The output will be the graph of the solution(s)
8 % and the vector value at the final point tf. Note that the
9 % IVPs do not need to be linear or homogeneous.
10
11 %% Lines to change:
12
13 % Line 28 : t0 - Start time
14 % Line 31 : tf - End time
15 % Line 34 : N - Number of subdivisions
16 % Line 37 : y0 - Vector of initial values
17 % Line 109+ : Which functions to plot, remembering to assign
18 %               a colour, texture and legend label
19 % Line 129+ : Set of differential equations written
20 %               explicitly. These can also be non-linear and
21 %               include forcing terms. These equations can
22 %               also be written in matrix form if the
23 %               equations are linear.
24
25 %% Set up input values
26
27 % Start time
28 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 desired 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:'))

```

```

119 disp(Y(:,end))
120
121 end
122
123 function [dydt]=DYDT(t,y,K)
124
125 % When the equation are written in explicit form
126
127 dydt=zeros(K,1);
128
129 dydt(1)=2*y(1)+y(2)+y(3)+cos(t);
130
131 dydt(2)=sin(y(1))+exp(-y(2)+y(3));
132
133 dydt(3)=y(1)*y(2)-y(3);
134
135 % If the set of equations is linear, then these can be
136 % written in matrix form as dydt=A*y+b(t). For example, if
137 % the set of equations is:
138 % dudt = 7u - 2v + w + exp(t)
139 % dvdt = 2u + 3v - 9w + cos(t)
140 % dwdt = 2v + 5w + 2
141 % Then:
142 % A=[7,-2,1;2,3,-9;0,2,5];
143 % b=@(t) [exp(t);cos(t);2];
144 % dydt=A*y+b(t)
145
146 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`:

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

```
1 function IVP_InBuilt
2
3 %% Solve a set of first order IVPs using In-Built codes
4
5 % This code solves a set of IVP when written explicitly
6 % on the interval [t0,tf] subject to the initial conditions
7 % y(0)=y0. The output will be the graph of the solution(s)
8 % and the vector value at the final point tf. Note that the
9 % IVPs do not need to be linear or homogeneous.
10
11 %% Lines to change:
12
13 % Line 28    : t0 - Start time
14 % Line 31    : tf - End time
15 % Line 43    : T_Span - Time span for evaluation
16 % Line 46    : y0 - Vector of initial values
17 % Line 85+   : Which functions to plot, remembering to assign
18 %               a colour, texture and legend label
19 % Line 105+  : Set of differential equations written
```

```

20 % explicitly. These can also be non-linear and
21 % include forcing terms. These equations can
22 % also be written in matrix form if the
23 % equations are linear.
24
25 %% Set up input values
26
27 % Start time
28 t0=0;
29
30 % End time
31 tf=1;
32
33 % Time span
34 % In-built methods tend to use adaptive meshing; decreasing
35 % the stepsize near locations with drastic derivative
36 % changes and increasing near small derivative changes.
37 % Sometimes this is not desired but a uniform meshing is
38 % required from the start time t0 to the end time tf being
39 % split into N equal sub intervals. This can be changed
40 % here:
41 % Adaptive meshing: T_Span=[t0 tf]
42 % Specific meshing: T_Span=linspace(t0,tf,N)
43 T_Span=[t0 tf];
44
45 % Column vector initial values y0=y(t0)
46 y0=[0;1;0];
47
48 %% Set up IVP solver parameters
49
50 % Number of differential equations
51 K=length(y0);
52
53 %% Use solver
54
55 % Set the solver tolerance
56 tol=odeset('RelTol',1e-6);
57
58 % Solve the IVP using ode45 or ode23
59 [T,Y]=ode45(@(t,y) DYDT(t,y,K),T_Span,y0,tol);
60
61 % Convert T and Y to columns for consistency
62 T=T';
63 Y=Y';
64

```

```

65 %% Setting plot parameters
66
67 % Clear figure
68 clf
69
70 % Hold so more than one line can be drawn
71 hold on
72
73 % Turn on grid
74 grid on
75
76 % Setting font size and style
77 set(gca,'FontSize',20,'FontName','Times')
78
79 % Label the axes
80 xlabel('$t$','Interpreter','Latex')
81 ylabel('$\mathbf{y}(t)$','Interpreter','Latex')
82
83 % Plot the desired solutions. If all the solutions are
84 % needed, then consider using a for loop in that case
85 plot(T,Y(1,:),'-b','LineWidth',2)
86 plot(T,Y(2,:),'-r','LineWidth',2)
87 plot(T,Y(3,:),'-k','LineWidth',2)
88
89 % Legend labels
90 legend('$y_1(t)$')
91 set(legend,'Interpreter','Latex')
92
93 % Display the values of the vector y at tf
94 disp(strcat('The vector y at t=',num2str(tf), ' is:'))
95 disp(Y(:,end))
96
97 end
98
99 function [dydt]=DYDT(t,y,K)
100
101 % When the equation are written in explicit form
102
103 dydt=zeros(K,1);
104
105 dydt(1)=2*y(1)+y(2)+y(3)+cos(t);
106
107 dydt(2)=sin(y(1))+exp(-y(2)+y(3));
108
109 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-linearity 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

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

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

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

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

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

This can be rearranged to give

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

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

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

whereas the standard Euler method in matrix form is

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

The Euler method requires *explicit calculations* using matrix multiplications but the back-wards Euler method requires matrix inversion instead.

7.2 Stability of the Backwards Euler Method

Consider the initial value problem in its scalar form

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

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

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

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

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

The backwards Euler then yields

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

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

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

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:

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

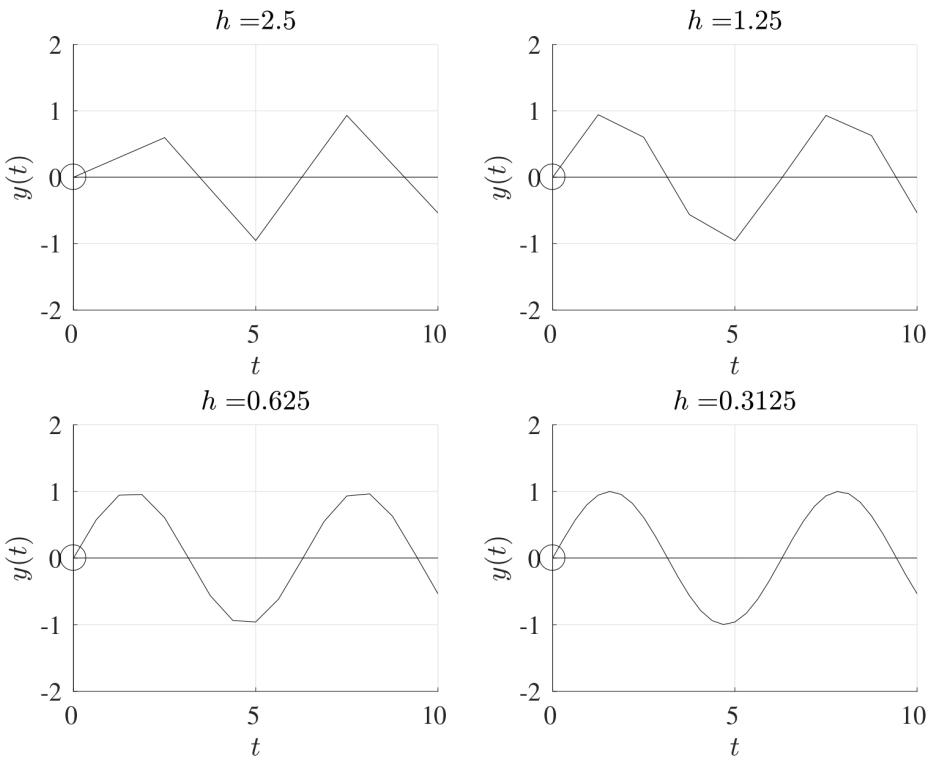
This means that the method is stable for stepsizes h that satisfy $|1 - h\lambda| > 1$ and since $\lambda < 0$ for an asymptotically stable system, then this inequality is *always* satisfied. This means that the backwards Euler method is stable *for all* stepsizes $h > 0$, no matter how large.

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

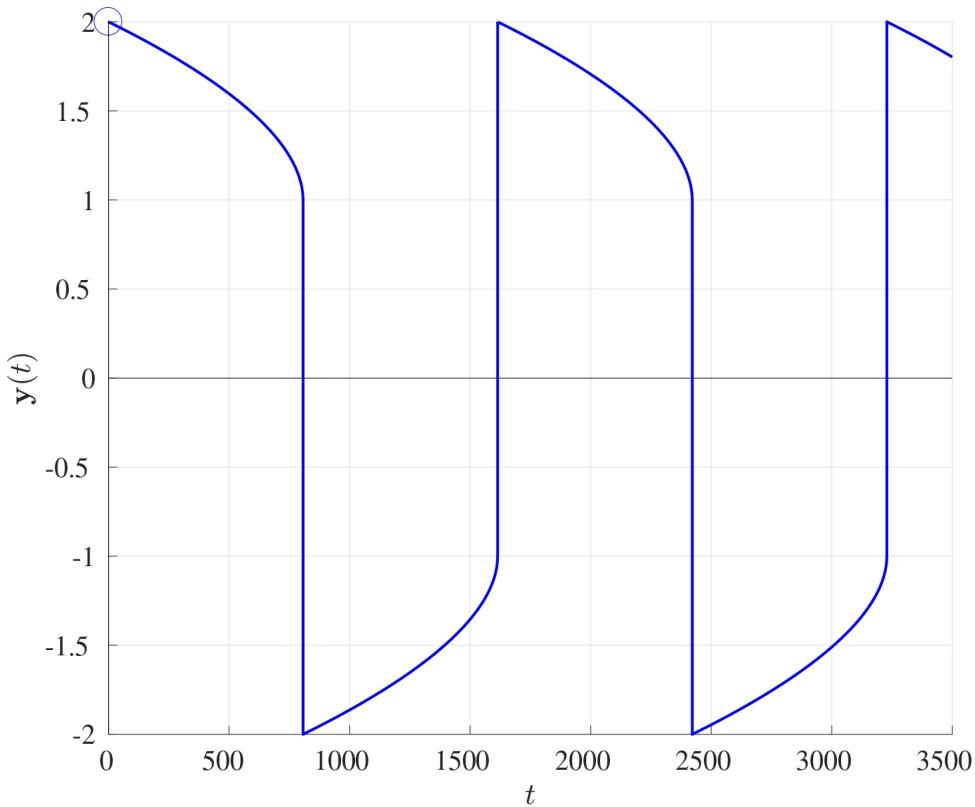
the command `ode15s` and this can be implemented exactly as `ode45`. This solves sets of differential equations implicitly using numerical differentiation of orders 1 to 5.

🔥 Stiff IVPs

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

$$\begin{aligned}\frac{dy_1}{dt} &= y_2 & y_1(0) &= 2 \\ \frac{dy_2}{dt} &= 1000(1 - y_1^2)y_2 - y_1 & y_2(0) &= 0.\end{aligned}$$

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 your 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

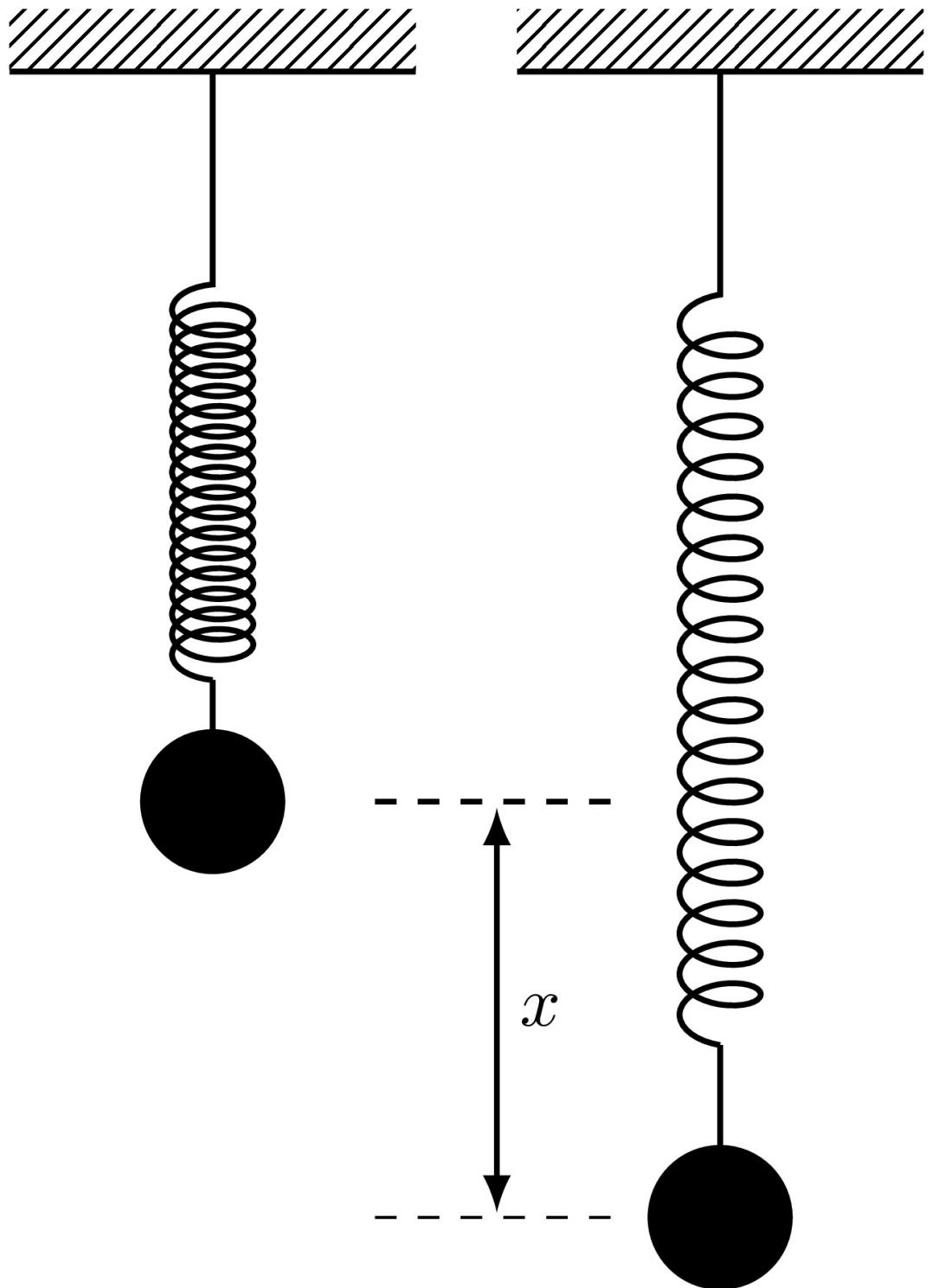
8 Boundary Value Problems

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

$$a = -\frac{d^2x}{dt^2} \implies F = -m\frac{d^2x}{dt^2}$$

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

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

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

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

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

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

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

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 from 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:

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

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

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:

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

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

These two conditions provide an expression 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:

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

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

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*.

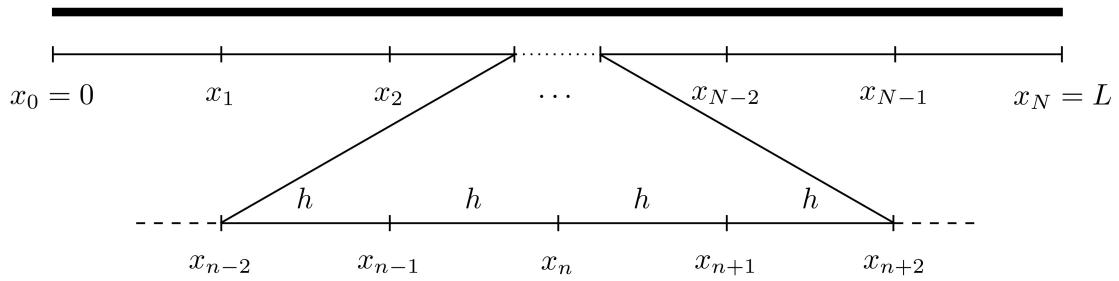
8.3.1 Finite Difference Approximations to the Derivatives

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

$$\frac{df}{dx}(X) \approx \frac{f(X+h) - f(X-h)}{2h}.$$

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, \dots, x_N where $x_n = nh$.



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

- **Forward Difference:**

$$\frac{du}{dx}(x_n) \approx \frac{u(x_{n+1}) - u(x_n)}{h}$$

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

- **Backward Difference:**

$$\frac{du}{dx}(x_n) \approx \frac{u(x_n) - u(x_{n-1})}{h}$$

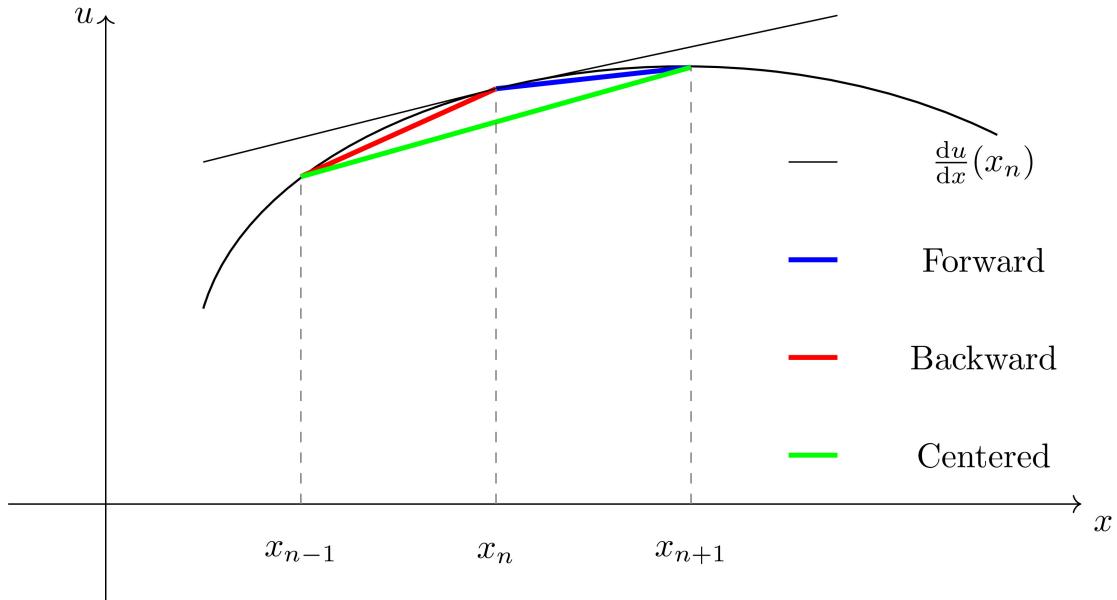
$$\frac{d^2u}{dx^2}(x_n) \approx \frac{u(x_n) - 2u(x_{n-1}) + u(x_{n-2})}{h^2}$$

- **Centred Difference:**

$$\frac{du}{dx}(x_n) \approx \frac{u(x_{n+1}) - u(x_{n-1})}{2h}$$

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

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



i Second Derivative Expression

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

$$\frac{du}{dx}(x_n) \approx \frac{u(x_{n+1}) - u(x_{n-1})}{2h}.$$

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

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

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:**

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

- **Backward Difference:**

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

- **Centred Difference:**

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

These approximations will form the basis for solving the BVP.

8.3.2 Discretisation of the Differential Equation

Returning to the differential equation

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

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

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

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:

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

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

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

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 $U_N \approx u(x_N) = u_r$. 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 $n = 2, 3, \dots, N - 2$, 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 all unknown. In summary, all of these $N - 1$ equations are:

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

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

$$\underbrace{\begin{pmatrix} \beta_1 & \gamma_1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ \alpha_2 & \beta_2 & \gamma_2 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & \alpha_3 & \beta_3 & \gamma_3 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & \alpha_4 & \beta_4 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \beta_{N-4} & \gamma_{N-4} & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & \alpha_{N-3} & \beta_{N-3} & \gamma_{N-3} & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & \alpha_{N-2} & \beta_{N-2} & \gamma_{N-2} \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & \alpha_{N-1} & \beta_{N-1} \end{pmatrix}}_A \underbrace{\begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ \vdots \\ U_{N-4} \\ U_{N-3} \\ U_{N-2} \\ U_{N-1} \end{pmatrix}}_{\mathbf{U}} = \underbrace{\begin{pmatrix} f(x_1) - \alpha_1 u_l \\ f(x_2) \\ f(x_3) \\ f(x_4) \\ \vdots \\ f(x_{N-4}) \\ f(x_{N-3}) \\ f(x_{N-2}) \\ f(x_{N-1}) - \gamma_{N-1} u_r \end{pmatrix}}_{\mathbf{g}}.$$

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

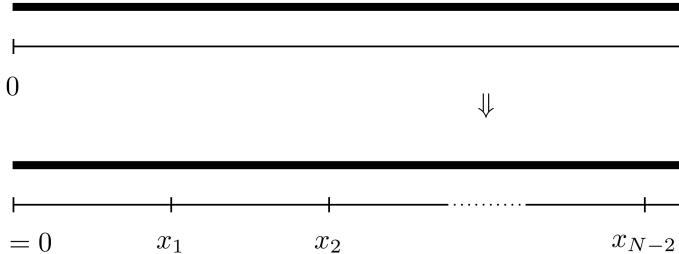
Carrying out matrix inversions by hand can become increasingly cumbersome if A is larger than 2×2 and therefore this process should be done computationally. This can be solved in MATLAB by using either $\mathbf{U}=\text{inv}(\mathbf{A})*\mathbf{g}$ or $\mathbf{U}=\mathbf{A}\backslash\mathbf{g}$. The *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 $x_0, x_1, x_2, \dots, x_N$ 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 $n = 0, 1, 2, \dots, N$.
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, $U_0, U_1, U_2, \dots, U_N$.
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 $A\mathbf{U} = \mathbf{g}$ and solve using MATLAB's backlash operator.

BVP Example

Consider the boundary value problem

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

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

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

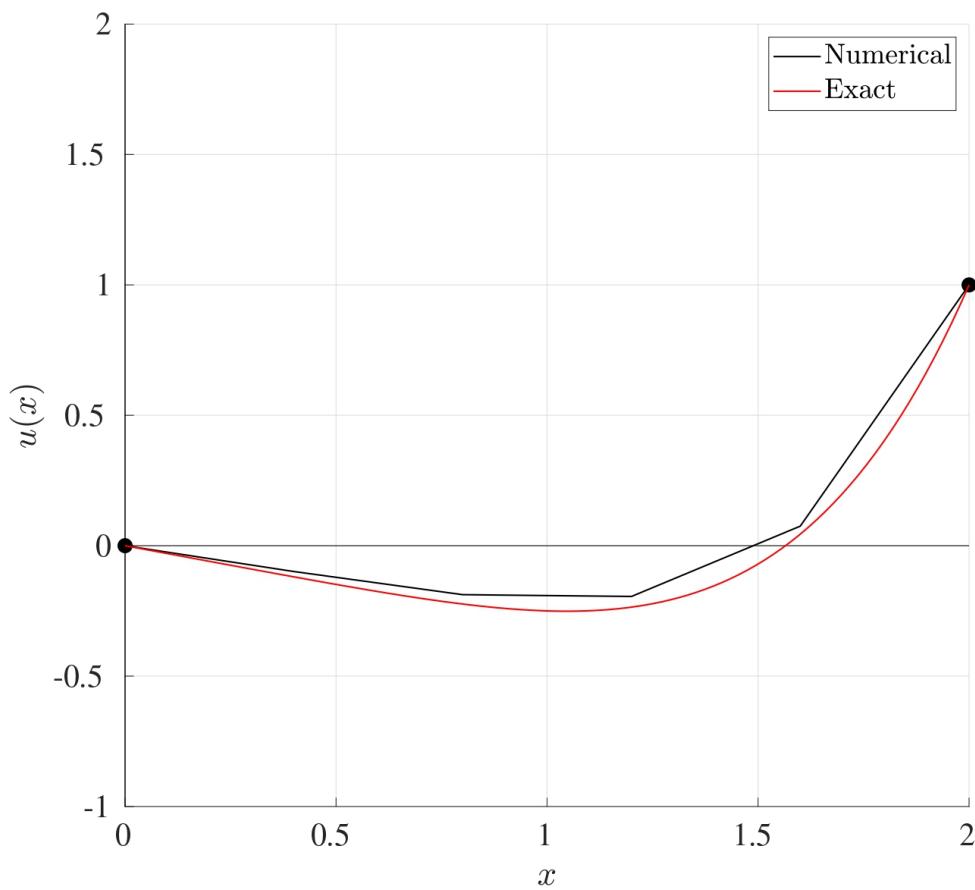
$$\beta_n = -\frac{2a(x_n)}{h^2} + c(x_n) = -12.5$$

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

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

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

This system can be solved using $\mathbf{U}=\text{inv}(A)*\mathbf{g}$ or $\mathbf{U}=A\backslash\mathbf{g}$. 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

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

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

```
1 function BVP_CD
2
3 %% Solve BVPs using centered differences
4
5 % The bvp is written in the form
6 % a(x) u'' + b(x) u' + c(x) u = f(x) on x in [x0,L]
7 % with the boundary conditions u(x0)=ul and u(L)=ur.
8 % After the centered difference approximation is
9 % used, the system will be written in the form AU=g.
10
11 %% Lines to change:
12
13 % Line 26 : x0 - Start point
14 % Line 29 : L - End point
15 % Line 32 : N - Number of subdivisions
16 % Line 35 : xl - Left boundary value
17 % Line 38 : xr - Right boundary value
18 % Line 119 : Expression for the function a(x)
19 % Line 127 : Expression for the function b(x)
20 % Line 135 : Expression for the function c(x)
21 % Line 143 : Expression for the function f(x)
22
23 %% Set up input values
24
25 % Start point
26 x0=0;
27
28 % End point
29 L=10;
30
31 % Number of subdivisions
32 N=50;
33
34 % Boundary value at x=x0
35 ul=1;
36
37 % Boundary value at x=L
```

```

38 ur=-1;
39
40 %% Set up BVP solver parameters
41
42 % Interval width
43 h=(L-x0)/N;
44
45 % X = Vector of locations
46 % (x1, x2, x3, ..., xN) (notice the start is x1 NOT x0)
47 X=x0+h:h:L;
48
49 % Evaluate the functions a(x), b(x), c(x) and f(x) at X
50 aX=a(X);
51 bX=b(X);
52 cX=c(X);
53 fX=f(X);
54
55 % Find the expressions for alpha, beta and gamma at X
56 alpha=aX/(h^2)-bX/(2*h);
57 beta=-2*aX/(h^2)+cX;
58 gamma=aX/(h^2)+bX/(2*h);
59
60 % Set up the vector g on the right hand side
61 g=zeros(N-1,1);
62 g(1)=fX(1)-alpha(1)*ul;
63 g(N-1)=fX(N-1)-gamma(N-1)*ur;
64 for j=2:1:N-2
65     g(j)=fX(j);
66 end
67
68 % Set up the matrix A on the left hand side (LHS_A is
69 % to avoid confusion with the function a(x))
70 A=zeros(N-1,N-1);
71 A(1,1)=beta(1);
72 A(1,2)=gamma(1);
73 A(N-1,N-1)=beta(N-1);
74 A(N-1,N-2)=alpha(N-1);
75 for j=2:1:N-2
76     A(j,j-1)=alpha(j);
77     A(j,j)=beta(j);
78     A(j,j+1)=gamma(j);
79 end
80
81 % Solve for the unknown vector U (it is then readjusted
82 % from a column vector to a row vector for plotting)

```

```

83 U=A\g;
84 U=U';
85
86 % Add the missing term x0 to the start of the vector x
87 X=[x0,X];
88
89 % Add the left and right boundary values to the vector U
90 U=[ul,U,ur];
91
92 %% Setting plot parameters
93
94 % Clear figure
95 clf
96
97 % Hold so more than one line can be drawn
98 hold on
99
100 % Turn on grid
101 grid on
102
103 % Setting font size and style
104 set(gca,'FontSize',20,'FontName','Times')
105
106 % Label the axes
107 xlabel('$t$', 'Interpreter', 'Latex')
108 ylabel('$u(t)$', 'Interpreter', 'Latex')
109
110 % Plot solution
111 plot(X,U, '-k', 'LineWidth', 2)
112
113 end
114
115 function [A]=a(X)
116 A=zeros(size(X));
117 for i=1:1:length(X)
118     x=X(i);
119     A(i)=1;
120 end
121 end
122
123 function [B]=b(X)
124 B=zeros(size(X));
125 for i=1:1:length(X)
126     x=X(i);
127     B(i)=2;

```

```

128 end
129 end
130
131 function [C]=c(X)
132 C=zeros(size(X));
133 for i=1:1:length(X)
134     x=X(i);
135     C(i)=exp(-x);
136 end
137 end
138
139 function [F]=f(X)
140 F=zeros(size(X));
141 for i=1:1:length(X)
142     x=X(i);
143     F(i)=sin(x);
144 end
145 end

```

8.5 Comparison Between Forward, Backward & Centred Difference Approximations

The main difference between the different differencing schemes is 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

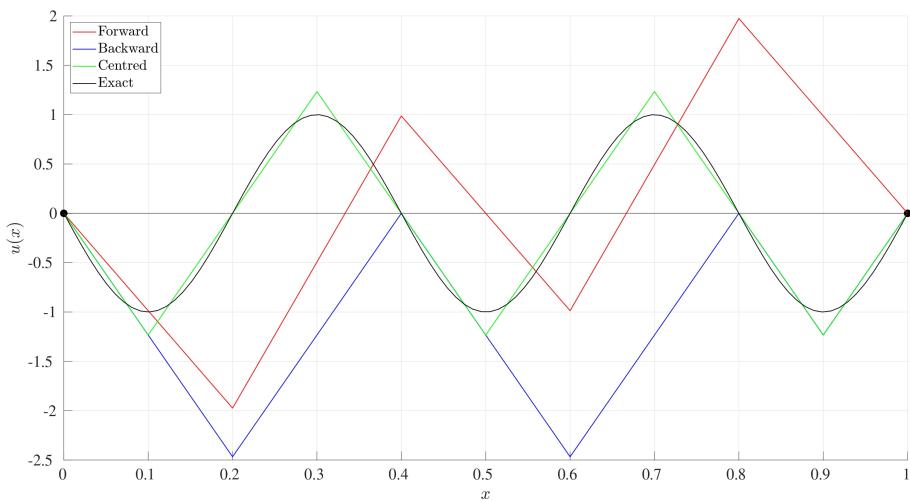
Consider the BVP

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

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

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

using `bvp4c`.

```

1 function BVP_InBuilt
2
3 %% Solve BVPs using bvp4c
4
5 % The bvp is written in the form
6 % a(x) u'' + b(x) u' + c(x) u = f(x) on x in [x0,L]
7 % with the boundary conditions u(x0)=ul and u(L)=ur.
8
9 %% Lines to change:
10
11 % Line 24 : x0 - Start point
12 % Line 27 : L - End point
13 % Line 30 : N - Number of spatial points
14 % Line 33 : xl - Left boundary value
15 % Line 36 : xr - Right boundary value
16 % Line 44 : Expression for the function a(x)
17 % Line 45 : Expression for the function b(x)
18 % Line 46 : Expression for the function c(x)
19 % Line 47 : Expression for the function f(x)
20
21 %% Set up input values
22
23 % Start point
24 x0=0;
25
26 % End point
27 L=10;
28
29 % Number of spatial points
30 N=50;
31
32 % Boundary value at x=x0
33 ul=1;
34
35 % Boundary value at x=L
36 ur=-1;
37
38 %% Set up BVP solver parameters
39
40 % Set up solving space
41 X=linspace(x0,L,N);
42
43 % Define the functions in the BVP
44 a= @(x) 1;
45 b= @(x) 2;

```

```

46 c= @(x) exp(-x);
47 f= @(x) sin(x);
48
49 %% Set up BVP solving parameters
50
51 % First, write the second order ODE as a set of first order
52 % ODEs:
53 % U'=V
54 % V'=(-b(x)*V-c(x)*U+f(x))/a(x)
55
56 % Second order BVPs can have more than one solution
57 % and vector v is the initialising vector of solutions.
58 % It can be kept as a vector of zeros
59 v=[0 0];
60
61 % Initialise vectors for space and v
62 init=bvpinit(X,v);
63
64 % Solve the bvp subject to the boundary values and
65 % initial guesses
66 sol=bvp4c(@(x,u) DUDT(x,u,a,b,c,f),@(x0,L) BCs(x0,L,ul,ur),init);
67
68 % Evaluate the solution at the grid points
69 U=deval(sol,X);
70
71 % Convert U to columns for consistency
72 U=U';
73
74 %% Setting plot parameters
75
76 % Clear figure
77 clf
78
79 % Hold so more than one line can be drawn
80 hold on
81
82 % Turn on grid
83 grid on
84
85 % Setting font size and style
86 set(gca,'FontSize',20,'FontName','Times')
87
88 % Label the axes
89 xlabel('$t$','Interpreter','Latex')
90 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 of length 5 with density ρ , convective velocity v , specific heat capacity C_p , thermal conductivity k_f and heat source f :

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

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 $\frac{dT}{dx}(5) = 0$ 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 and 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

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

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 $n = 0, 1, 2, \dots, N$. This differential equation can be discretised using the centred difference approximation just as before to give

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

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

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

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

$$\frac{du}{dx}(0) = \tilde{u}_l \quad \text{and} \quad u(L) = u_r.$$

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:

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

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

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

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

$$\frac{du}{dx}(0) = \tilde{u}_l.$$

Using the centred finite difference approximation on the derivative gives

$$\tilde{u}_l = \frac{du}{dx}(0) = \frac{du}{dx}(x_0) \approx \frac{u(x_1) - u(x_{-1})}{2h} \approx \frac{U_1 - U_{-1}}{2h} \implies \frac{U_1 - U_{-1}}{2h} \approx \tilde{u}_l$$

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:

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

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

$$\underbrace{\begin{pmatrix} \beta_0 & \gamma_0 + \alpha_0 & 0 & \dots & 0 & 0 & 0 \\ \alpha_1 & \beta_1 & \gamma_1 & \dots & 0 & 0 & 0 \\ 0 & \alpha_2 & \beta_2 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \beta_{N-3} & \gamma_{N-3} & 0 \\ 0 & 0 & 0 & \dots & \alpha_{N-2} & \beta_{N-2} & \gamma_{N-2} \\ 0 & 0 & 0 & \dots & 0 & \alpha_{N-1} & \beta_{N-1} \end{pmatrix}}_A \underbrace{\begin{pmatrix} U_0 \\ U_1 \\ U_2 \\ \vdots \\ U_{N-3} \\ U_{N-2} \\ U_{N-1} \end{pmatrix}}_U = \underbrace{\begin{pmatrix} f(x_0) + 2h\alpha_0 \tilde{u}_l \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{N-3}) \\ f(x_{N-2}) \\ f(x_{N-1}) - \gamma_{N-1} u_r \end{pmatrix}}_g.$$

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

$$u(0) = u_l \quad \text{and} \quad \frac{du}{dx}(L) = \tilde{u}_r,$$

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

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

$$\underbrace{\begin{pmatrix} \beta_1 & \gamma_1 & 0 & \dots & 0 & 0 & 0 \\ \alpha_2 & \beta_2 & \gamma_2 & \dots & 0 & 0 & 0 \\ 0 & \alpha_3 & \beta_3 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \beta_{N-2} & \gamma_{N-2} & 0 \\ 0 & 0 & 0 & \dots & \alpha_{N-1} & \beta_{N-1} & \gamma_{N-1} \\ 0 & 0 & 0 & \dots & 0 & \alpha_N + \gamma_N & \beta_N \end{pmatrix}}_A \underbrace{\begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ \vdots \\ U_{N-2} \\ U_{N-1} \\ U_N \end{pmatrix}}_U = \underbrace{\begin{pmatrix} f(x_1) - \alpha_1 u_l \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_{N-2}) \\ f(x_{N-1}) \\ f(x_N) - 2h\gamma_N \tilde{u}_r \end{pmatrix}}_{\mathbf{g}}.$$

Mixed Value Problem

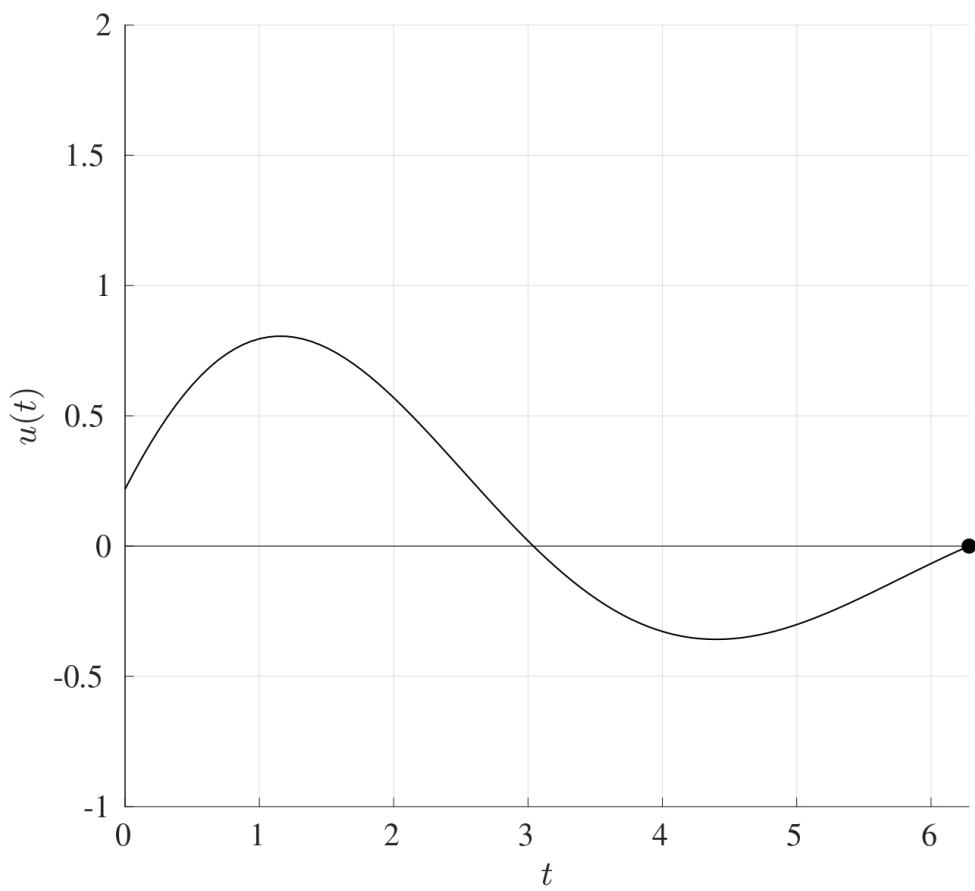
Consider the differential equation for a damped harmonic oscillator

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

with the mixed conditions

$$\frac{du}{dt}(0) = 1 \quad \text{and} \quad u(2\pi) = 0.$$

This MVP is to determine the trajectory of the mass if the launching speed at the start is 1, which is $\frac{du}{dt}(0) = 1$, 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

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

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

$$T(-L) = T(L) \quad \text{and} \quad k_f \frac{dT}{dx}(L) = \alpha(T(L) - T_{air})$$

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 $k_f \frac{dT}{dx}(L) = \alpha(T(L) - T_{air})$). 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

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

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 $n = 0, 1, 2, \dots, N$. This differential equation can be discretised using the centred difference approximation (just as in Section 8.2) to give

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

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

This gives a set of $N-1$ equations in $N+1$ unknowns, namely $U_0, U_1, U_2, \dots, U_N$. 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

$$u(-L) = u(L) \quad \text{and} \quad \frac{du}{dx}(L) = pu(L) + q$$

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

$$\alpha_N U_{N-1} + \beta_N U_N + \gamma_N U_{N+1} = f(x_N). \quad (10.1)$$

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

$$\frac{du}{dx}(x_N) = \frac{du}{dx}(L)pu(L) + q \approx pU_N + q.$$

The LHS can be rewritten in terms of its centred differencing approximation as

$$\frac{du}{dx}(x_N) \approx \frac{u(x_{N+1}) - u(x_{N-1})}{2h} \approx \frac{U_{N+1} - U_{N-1}}{2h}.$$

Combining these two can give an expression for U_{N+1} as:

$$\frac{U_{N+1} - U_{N-1}}{2h} \approx pU_N + q \implies U_{N+1} = U_{N-1} + 2hpU_N + 2hq.$$

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 $AU = g$ where

$$\underbrace{\begin{pmatrix} \beta_1 & \gamma_1 & 0 & \dots & 0 & 0 & \alpha_1 \\ \alpha_2 & \beta_2 & \gamma_2 & \dots & 0 & 0 & 0 \\ 0 & \alpha_3 & \beta_3 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \beta_{N-2} & \gamma_{N-2} & 0 \\ 0 & 0 & 0 & \dots & \alpha_{N-1} & \beta_{N-1} & \gamma_{N-1} \\ 0 & 0 & 0 & \dots & 0 & \alpha_N + \gamma_N & \beta_N + 2hp\gamma_N \end{pmatrix}}_A \underbrace{\begin{pmatrix} U_1 \\ U_2 \\ U_3 \\ \vdots \\ U_{N-2} \\ U_{N-1} \\ U_N \end{pmatrix}}_U = \underbrace{\begin{pmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_{N-2}) \\ f(x_{N-1}) \\ f(x_N) - 2hq\gamma_N \end{pmatrix}}_g.$$

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 .

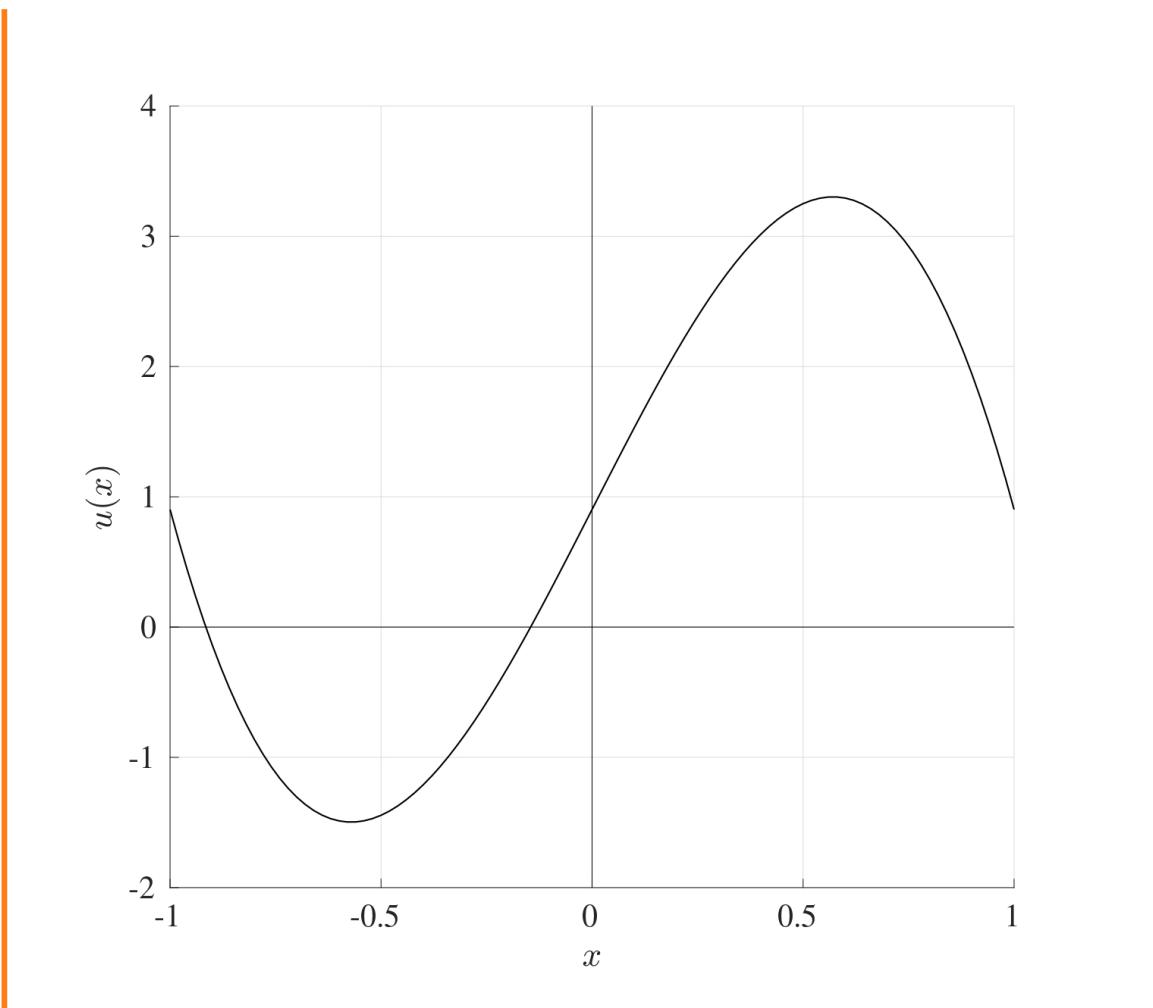
Symmetric Boundary Value Problem

Consider the conduction problem

$$-\frac{d^2T}{dx^2} = 40 \sin(x) \quad \text{in} \quad -1 < x < 1$$

with the conditions

$$T(-1) = T(1) \quad \text{and} \quad \frac{dT}{dx}(1) = \frac{1}{2}(T(1) - 25).$$



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

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \quad \text{with } 0 < x < L \quad \text{and } t > 0$$

where $u = u(x, t)$ is the temperature at location x at time t and α is the thermal diffusivity¹. 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 $x_0, x_1, x_2, \dots, x_N$ where $x_n = nh$ and the spatial interval width is $h_x = \frac{L}{N}$.



¹The thermal diffusivity will always be regarded as a constant and usually takes the form $\alpha = \frac{k}{\rho C_p}$ where k is the thermal conductivity, ρ is the density of the material and C_p 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

$$\frac{\partial u}{\partial t}(x_n, t) \approx \frac{dU_n}{dt}(t)$$

and this holds for $n = 1, 2, \dots, N - 1$ 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

$$\alpha \frac{\partial^2 u}{\partial x^2}(x_n, t) \approx \frac{\alpha}{h_x^2} [U_{n-1}(t) - 2U_n(t) + U_{n+1}(t)].$$

This holds for $n = 1, 2, \dots, N - 1$ 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

$$\frac{dU_n}{dt} = \frac{\alpha}{h_x^2} [U_{n-1} - 2U_n + U_{n+1}]$$

for all $n = 1, 2, \dots, N - 1$ 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 $\frac{dU}{dt} = AU + b$ where

$$\underbrace{\frac{d}{dt} \begin{pmatrix} U_1(t) \\ U_2(t) \\ U_3(t) \\ \vdots \\ U_{N-3}(t) \\ U_{N-2}(t) \\ U_{N-1}(t) \end{pmatrix}}_U = \underbrace{\frac{\alpha}{h_x^2} \begin{pmatrix} -2 & 1 & 0 & \dots & 0 & 0 & 0 \\ 1 & -2 & 1 & \dots & 0 & 0 & 0 \\ 0 & 1 & -2 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & -2 & 1 & 0 \\ 0 & 0 & 0 & \dots & 1 & -2 & 1 \\ 0 & 0 & 0 & \dots & 0 & 1 & -2 \end{pmatrix}}_A \underbrace{\begin{pmatrix} U_1(t) \\ U_2(t) \\ U_3(t) \\ \vdots \\ U_{N-3}(t) \\ U_{N-2}(t) \\ U_{N-1}(t) \end{pmatrix}}_U + \underbrace{\frac{\alpha}{h_x^2} \begin{pmatrix} u_l(t) \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ u_r(t) \end{pmatrix}}_b$$

subject to the *initial condition*

$$\mathbf{U}_0 = \begin{pmatrix} U_1(0) \\ U_2(0) \\ U_3(0) \\ \vdots \\ U_{N-3}(0) \\ U_{N-2}(0) \\ U_{N-1}(0) \end{pmatrix} \approx \begin{pmatrix} u(x_1, 0) \\ u(x_2, 0) \\ u(x_3, 0) \\ \vdots \\ u(x_{N-3}, 0) \\ u(x_{N-2}, 0) \\ u(x_{N-1}, 0) \end{pmatrix} = \begin{pmatrix} u_{init}(x_1) \\ u_{init}(x_2) \\ u_{init}(x_3) \\ \vdots \\ u_{init}(x_{N-3}) \\ u_{init}(x_{N-2}) \\ u_{init}(x_{N-1}) \end{pmatrix}.$$

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

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial t^2}, \quad x \in [0, 1], \quad t > 0$$

$$u(x, 0) = u_{init}(x) = \begin{cases} 0 & 0 \leq x < 0.4 \\ 1 & 0.4 \leq x < 0.6 \\ 0 & 0.6 \leq x \leq 1 \end{cases}, \quad u(0, t) = u_l(t) = 2, \quad u(1, t) = u_r(t) = 2.$$

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 $\frac{d\mathbf{U}}{dt} = A\mathbf{U} + \mathbf{b}$ where

$$\frac{d}{dt} \underbrace{\begin{pmatrix} U_1(t) \\ U_2(t) \\ U_3(t) \\ U_4(t) \\ U_5(t) \end{pmatrix}}_{\mathbf{U}} = \frac{\alpha}{h^2} \underbrace{\begin{pmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{pmatrix}}_A \underbrace{\begin{pmatrix} U_1(t) \\ U_2(t) \\ U_3(t) \\ U_4(t) \\ U_5(t) \end{pmatrix}}_{\mathbf{U}} + \frac{\alpha}{h^2} \underbrace{\begin{pmatrix} u_l(t) \\ 0 \\ 0 \\ 0 \\ u_r(t) \end{pmatrix}}_{\mathbf{b}}$$

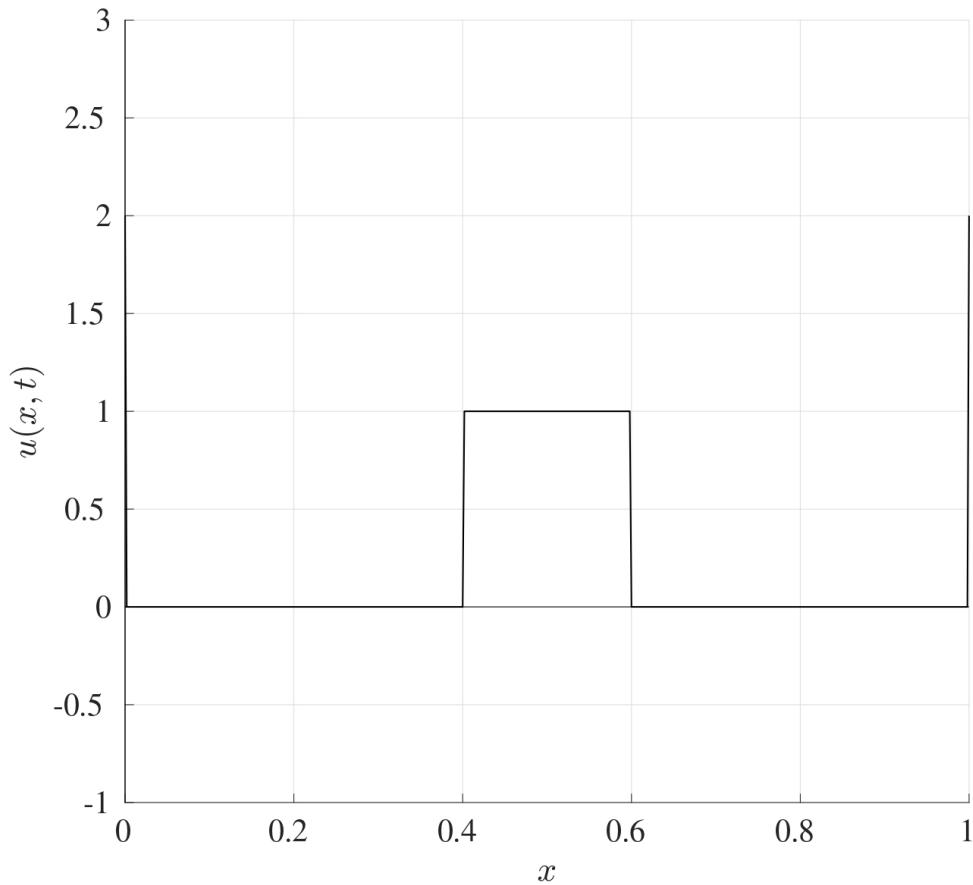
The differential equation

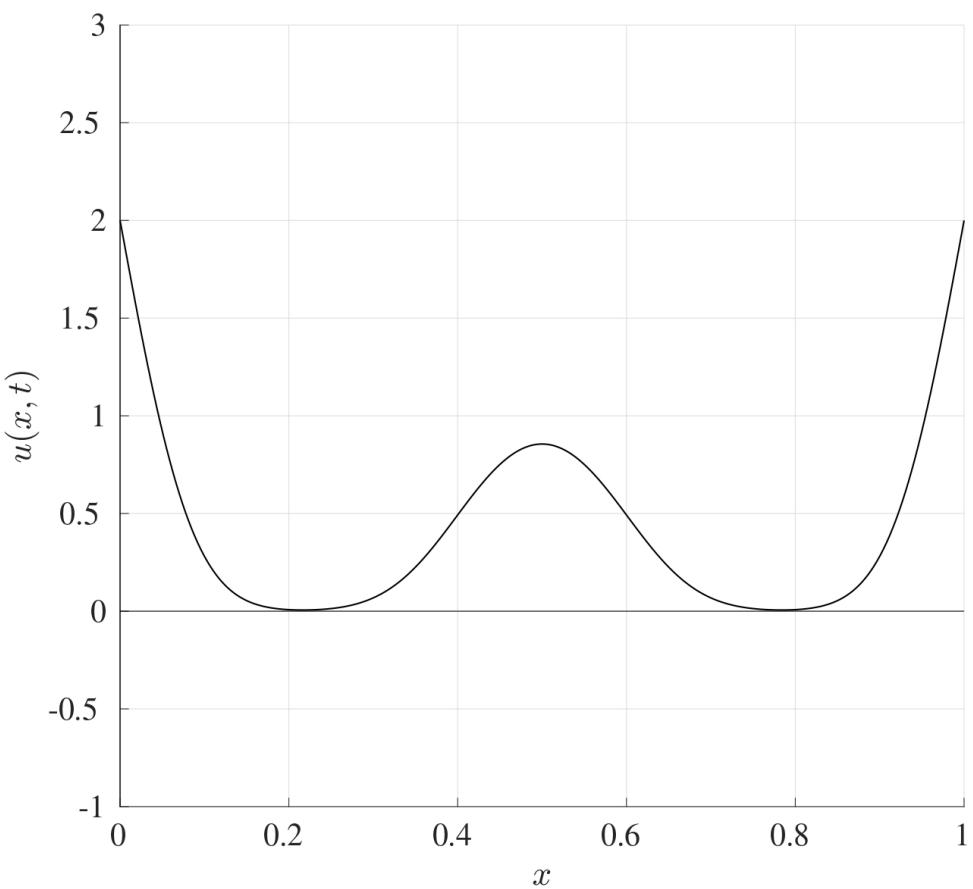
$$\frac{d\mathbf{U}}{dt} = A\mathbf{U} + \mathbf{b}$$

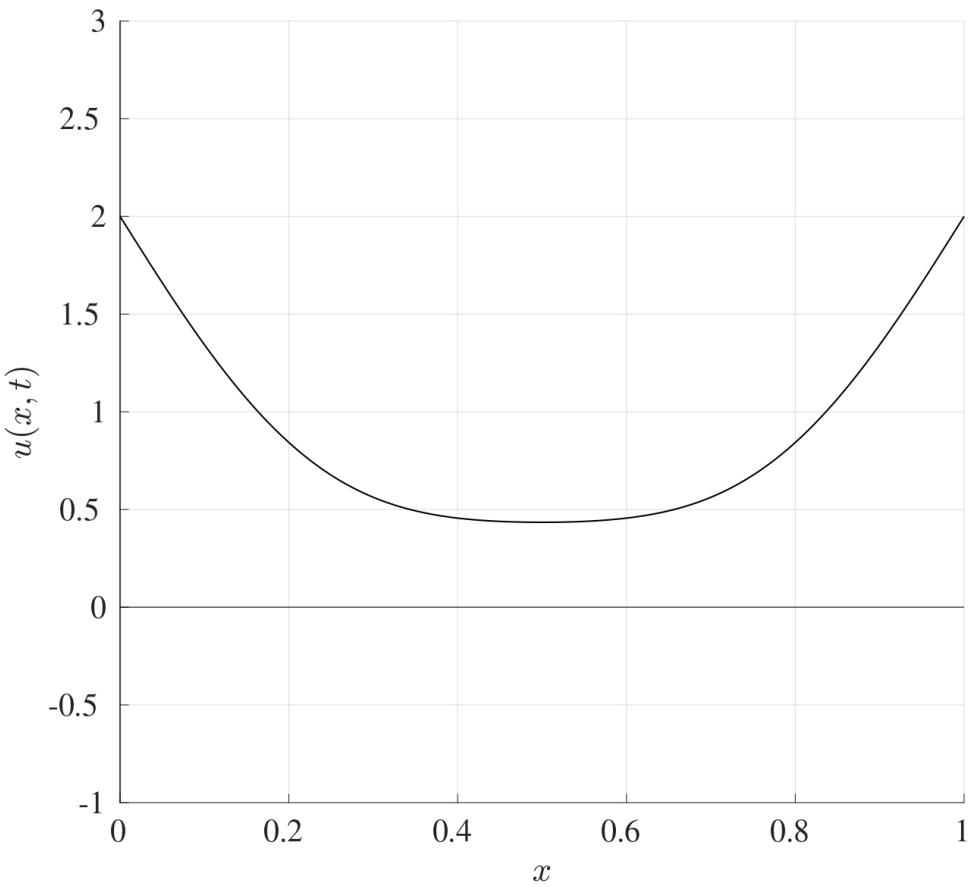
can be solved using the Euler method with the initial condition

$$\mathbf{U}(0) = \begin{pmatrix} u_{init}(x_1) \\ u_{init}(x_2) \\ u_{init}(x_3) \\ u_{init}(x_4) \\ u_{init}(x_5) \end{pmatrix}$$

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 $h_t = 0.02$ ($N_t = 50000$).







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

$$\frac{\partial u}{\partial t} = -v \frac{\partial u}{\partial x} \quad \text{with } 0 < x < L \quad \text{and } t > 0$$

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

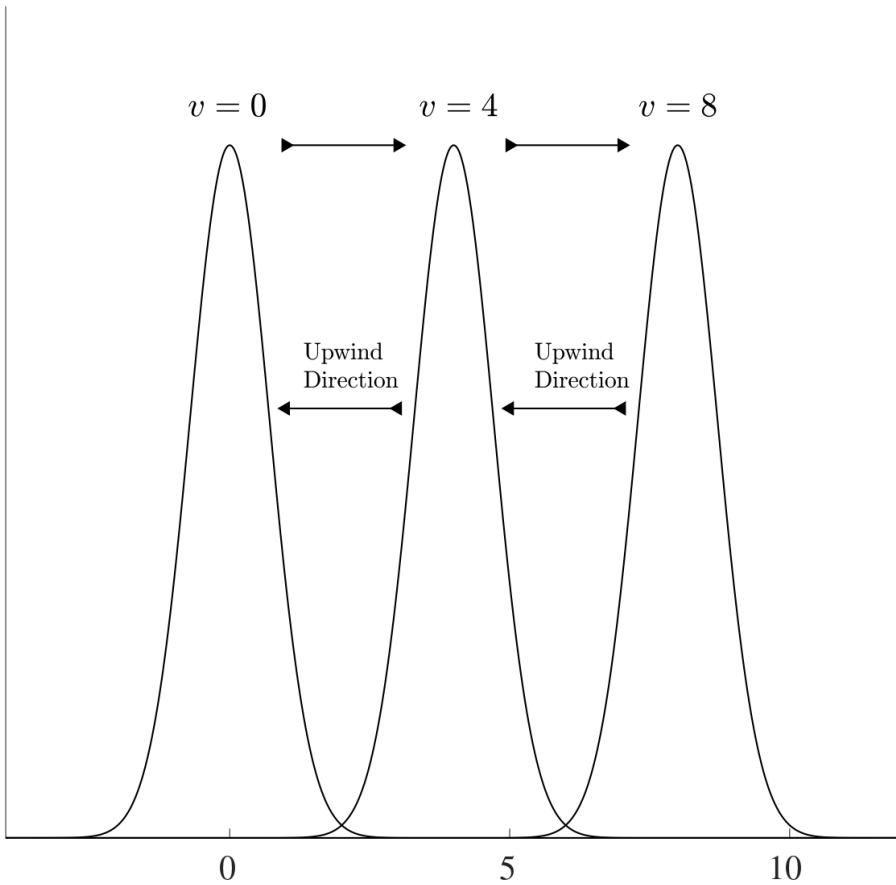
$$u(x, t) = u_{init}(x - vt),$$

this can be verified from the partial differential equation as follows:

$$\frac{\partial u}{\partial t} = -v \frac{\partial u}{\partial x} \quad \text{at} \quad u(x, t) = u_{init}(x - vt)$$

$$\begin{aligned} \text{LHS} &= \frac{\partial}{\partial t} u(x, t) = \frac{\partial}{\partial t} (u_{init}(x - vt)) = -vu'_{init}(x - vt) \\ \text{RHS} &= \frac{\partial}{\partial x} u(x, t) = \frac{\partial}{\partial x} (u_{init}(x - vt)) = -vu'_{init}(x - vt). \end{aligned}$$

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 $n = 0, 1, 2, \dots, N$, 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

$$\frac{dU_n}{dt} = \frac{v}{h_x} (U_{n-1} - U_n) \quad \text{for } n = 1, 2, \dots, N$$

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 $\frac{d\mathbf{U}}{dt} = A\mathbf{U} + \mathbf{b}$ where

$$\frac{d}{dt} \underbrace{\begin{pmatrix} U_1(t) \\ U_2(t) \\ U_3(t) \\ \vdots \\ U_{N-2}(t) \\ U_{N-1}(t) \\ U_N(t) \end{pmatrix}}_{\mathbf{U}} = \frac{v}{h_x} \underbrace{\begin{pmatrix} -1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 1 & -1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & -1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & -1 & 0 & 0 \\ 0 & 0 & 0 & \dots & 1 & -1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 & -1 \end{pmatrix}}_A \underbrace{\begin{pmatrix} U_1(t) \\ U_2(t) \\ U_3(t) \\ \vdots \\ U_{N-2}(t) \\ U_{N-1}(t) \\ U_N(t) \end{pmatrix}}_{\mathbf{U}}$$

$$+ \frac{v}{h_x} \underbrace{\begin{pmatrix} u_l(t) \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \end{pmatrix}}_b$$

and the initial condition is

$$\mathbf{U}_0 = \underbrace{\begin{pmatrix} U_1(0) \\ U_2(0) \\ U_3(0) \\ \vdots \\ U_{N-3}(0) \\ U_{N-2}(0) \\ U_{N-1}(0) \end{pmatrix}}_{\mathbf{U}_0} \approx \underbrace{\begin{pmatrix} u(x_1, 0) \\ u(x_2, 0) \\ u(x_3, 0) \\ \vdots \\ u(x_{N-3}, 0) \\ u(x_{N-2}, 0) \\ u(x_{N-1}, 0) \end{pmatrix}}_{\mathbf{U}_0} = \underbrace{\begin{pmatrix} u_{init}(x_1) \\ u_{init}(x_2) \\ u_{init}(x_3) \\ \vdots \\ u_{init}(x_{N-3}) \\ u_{init}(x_{N-2}) \\ u_{init}(x_{N-1}) \end{pmatrix}}_{\mathbf{U}_0}$$

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

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} - v \frac{\partial u}{\partial x} \quad \text{with } 0 < x < L, \quad t > 0.$$

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

$$\frac{dU_n}{dt}(t) = \frac{\alpha}{h_x^2} [U_{n-1}(t) - 2U_n(t) + U_{n+1}(t)] - \frac{v}{h_x} [U_n(t) + U_{n-1}(t)] \quad \text{for } n = 1, 2, \dots, N-1.$$

This system can be written in the form $\frac{dU}{dt} = A\mathbf{U} + \mathbf{b}$ where

$$A = \frac{\alpha}{h_x^2} \begin{pmatrix} -2 & 1 & 0 & \dots & 0 & 0 & 0 \\ 1 & -2 & 1 & \dots & 0 & 0 & 0 \\ 0 & 1 & -2 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & -2 & 1 & 0 \\ 0 & 0 & 0 & \dots & 1 & -2 & 1 \\ 0 & 0 & 0 & \dots & 0 & 1 & -2 \end{pmatrix} + \frac{v}{h_x} \begin{pmatrix} -1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 1 & -1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & -1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & -1 & 0 & 0 \\ 0 & 0 & 0 & \dots & 1 & -1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 & -1 \end{pmatrix}$$

$$\mathbf{U} \begin{pmatrix} U_1(t) \\ U_2(t) \\ \vdots \\ U_{N-2}(t) \\ U_{N-1}(t) \end{pmatrix}, \quad \mathbf{b} = \frac{\alpha}{h_x^2} \begin{pmatrix} u_l(t) \\ 0 \\ \vdots \\ 0 \\ u_r(t) \end{pmatrix} + \frac{v}{h_x} \begin{pmatrix} u_l(t) \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}.$$

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 Example

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:

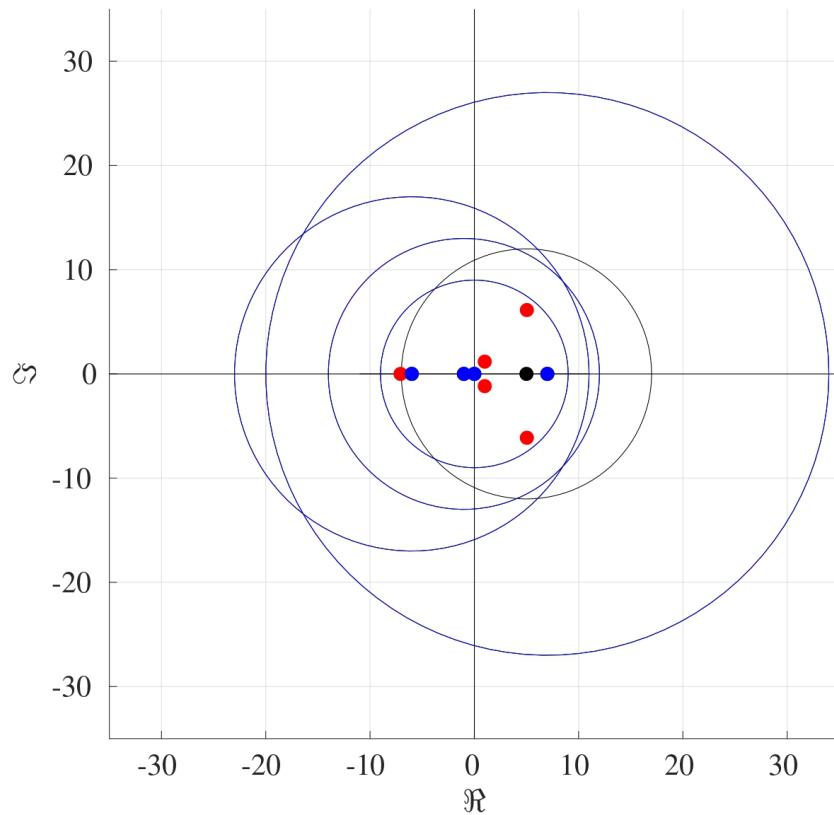
1. Indicate the locations of the diagonal terms (namely $-1, 5, 0, -6, 7$) on the complex

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,

$$\text{abs}(A) = \begin{pmatrix} 1 & 3 & 4 & 2 & 4 \\ 0 & 5 & 4 & 7 & 1 \\ 4 & 2 & 0 & 3 & 0 \\ 6 & 6 & 4 & 6 & 1 \\ 7 & 4 & 7 & 9 & 7 \end{pmatrix} \rightarrow \begin{array}{l} 3 + 4 + 2 + 4 = 13 \rightarrow R_1 \\ 0 + 4 + 7 + 1 = 12 \rightarrow R_2 \\ 4 + 2 + 3 + 0 = 9 \rightarrow R_3 \\ 6 + 6 + 4 + 1 = 17 \rightarrow R_4 \\ 7 + 4 + 7 + 9 = 27 \rightarrow R_5 \end{array}$$

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.

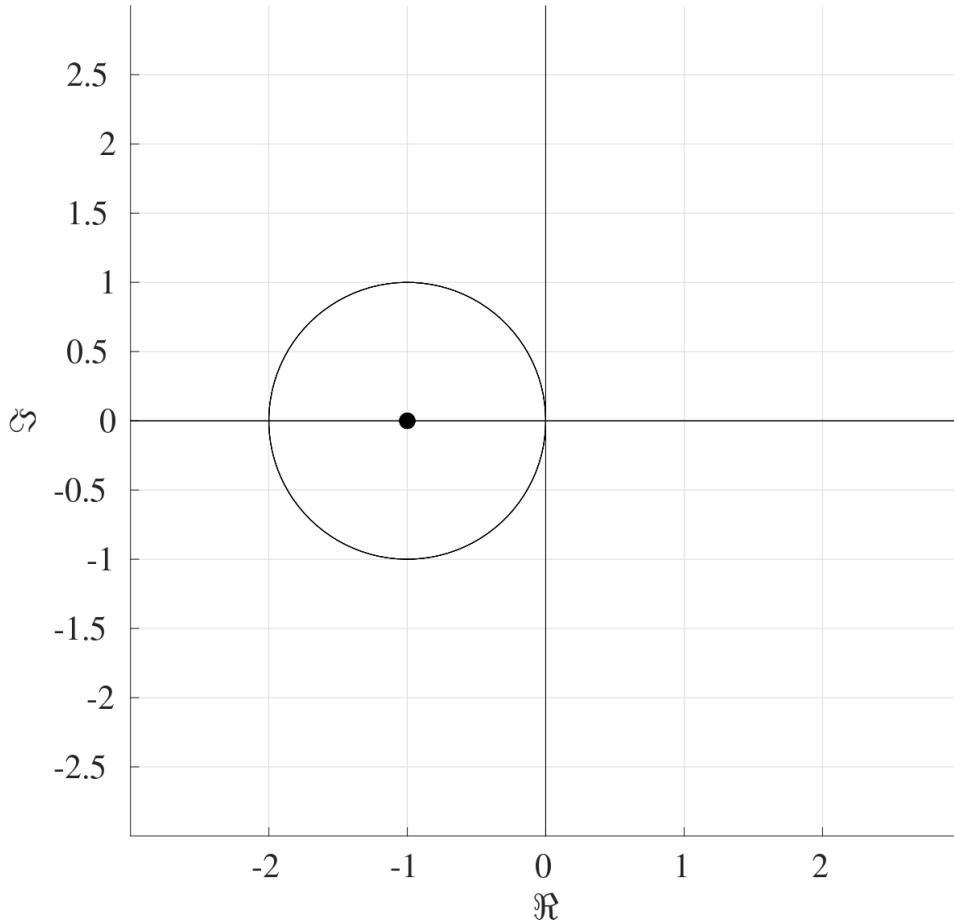


11.4.1 Stability of the Euler Method for the Advection Equation

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 $\frac{d\mathbf{U}}{dt} = A\mathbf{U} + \mathbf{b}$

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 $\mathcal{I} + h_t A$:

$$\mathcal{I} + h_t A = \mathcal{I} + \frac{vh_t}{h_x} A_2 = \begin{pmatrix} 1 - \tilde{v} & 0 & 0 & \dots & 0 & 0 & 0 \\ \tilde{v} & 1 - \tilde{v} & 0 & \dots & 0 & 0 & 0 \\ 0 & \tilde{v} & 1 - \tilde{v} & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & \tilde{v} & 1 - \tilde{v} \end{pmatrix}.$$

where $\tilde{v} = \frac{vh_t}{h_x}$. Now taking the absolute value of all the terms and taking the row sums gives:

$$\text{abs}\left(\mathcal{I} + \frac{vh_t}{h_x} A_2\right) = \begin{pmatrix} |1 - \tilde{v}| & 0 & 0 & \dots & 0 & 0 & 0 \\ \tilde{v} & |1 - \tilde{v}| & 0 & \dots & 0 & 0 & 0 \\ 0 & \tilde{v} & |1 - \tilde{v}| & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & \tilde{v} & |1 - \tilde{v}| \end{pmatrix} \rightarrow \begin{array}{l} |1 - \tilde{v}| \\ \rightarrow \tilde{v} + |1 - \tilde{v}| \\ \rightarrow \tilde{v} + |1 - \tilde{v}| \\ \vdots \\ \vdots \\ \rightarrow \tilde{v} + |1 - \tilde{v}| \end{array}$$

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 $1 - \tilde{v} < 0$.

$$\|\mathcal{I} + h_t A\|_\infty = |1 - \tilde{v}| + \tilde{v} = 1 - \tilde{v} + \tilde{v} = 1.$$

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

$$\|\mathcal{I} + h_t A\|_\infty \leq 1 \implies 2\tilde{v} - 1 \leq 1 \implies \tilde{v} \leq 1$$

which contradicts with the assumption that $\tilde{v} > 1$.

Therefore, the Euler method will produce a convergent solution if

$$\tilde{v} < 1 \implies v \frac{h_t}{h_x} < 1.$$

In terms of number of spatial and temporal points N_x and N_t respectively, this restriction would be

$$v \frac{t_f - t_0}{L - x_0} \frac{N_x}{N_t} < 1$$

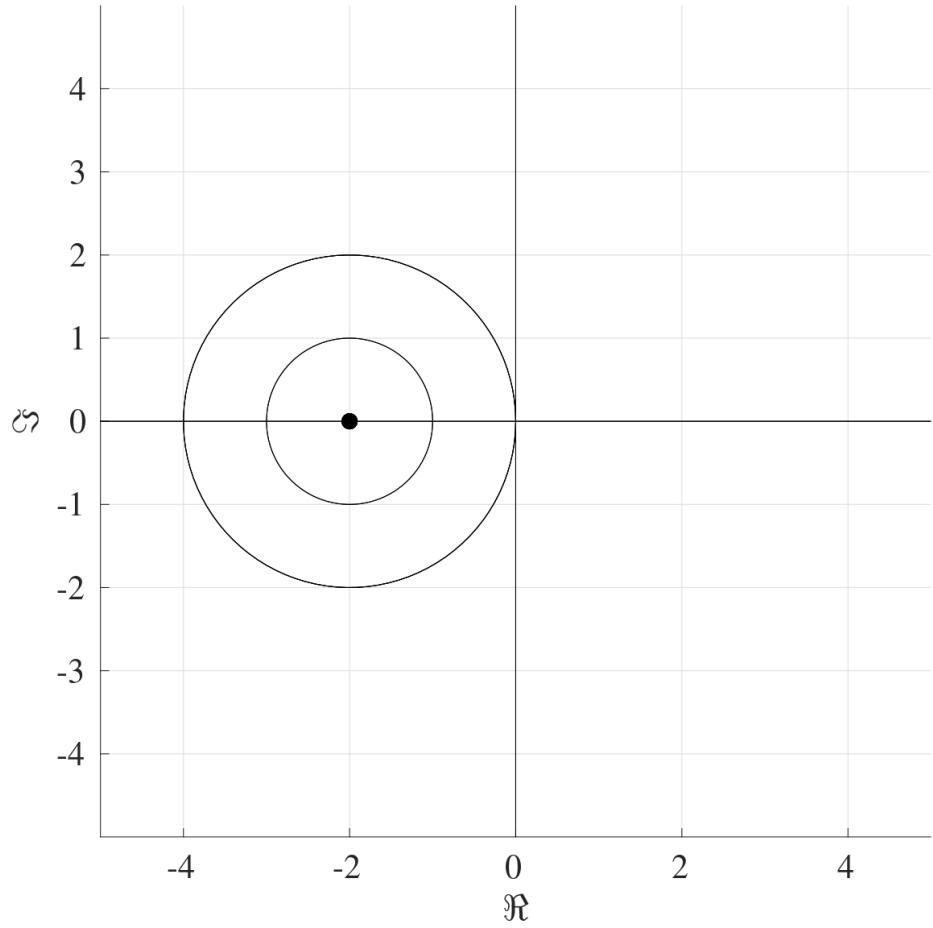
So for a fixed velocity v , if the time step h_t is to be halved, then the spatial step would also need to be halved as well.

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 $\frac{dU}{dt} = A\mathbf{U} + \mathbf{b}$ 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:

$$\text{abs}\left(\mathcal{I} + \frac{\alpha h_t}{h_x^2} A_1\right) = \begin{pmatrix} |1 - 2\tilde{\alpha}| & \tilde{\alpha} & 0 & \dots & 0 & 0 & 0 \\ \tilde{\alpha} & |1 - 2\tilde{\alpha}| & \tilde{\alpha} & \dots & 0 & 0 & 0 \\ 0 & \tilde{\alpha} & |1 - 2\tilde{\alpha}| & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & \tilde{\alpha} & |1 - 2\tilde{\alpha}| \end{pmatrix} \rightarrow \begin{array}{l} \tilde{\alpha} + |1 - 2\tilde{\alpha}| \\ 2\tilde{\alpha} + |1 - 2\tilde{\alpha}| \\ 2\tilde{\alpha} + |1 - 2\tilde{\alpha}| \\ \vdots \\ \tilde{\alpha} + |1 - 2\tilde{\alpha}| \end{array}$$

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

$$\|\mathcal{I} + h_t A\|_\infty \leq 1 \implies 4\tilde{\alpha} - 1 \leq 1 \implies \tilde{\alpha} \leq \frac{1}{2}$$

which contradicts with the assumption that $\tilde{\alpha} > \frac{1}{2}$.

This means that the Euler method produces a stable convergent solution if

$$\tilde{\alpha} < \frac{1}{2} \implies \alpha \frac{h_t}{h_x^2} < \frac{1}{2}.$$

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.

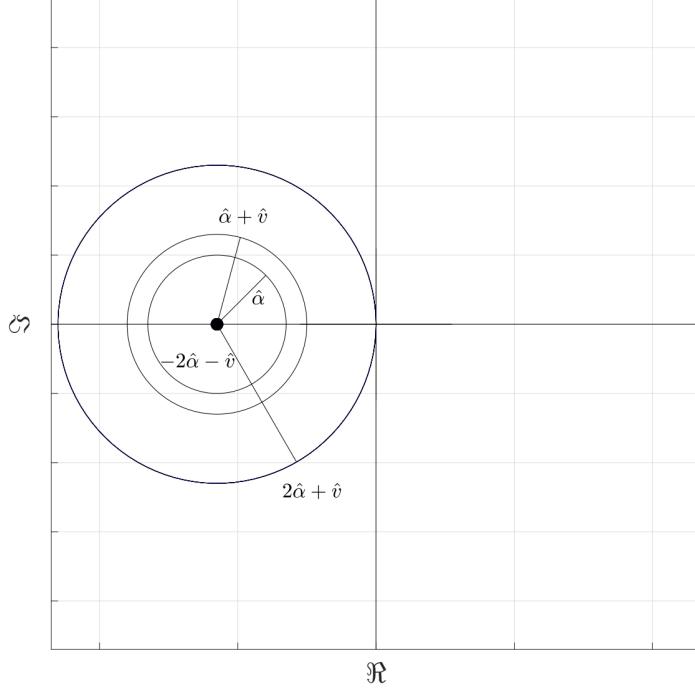
When discretised, the convection-diffusion equation can be written as $\frac{d\mathbf{U}}{dt} = A\mathbf{U} + \mathbf{b}$ where the matrix A is given by

$$A = \frac{\alpha}{h_x^2} \begin{pmatrix} -2 & 1 & \dots & 0 & 0 \\ 1 & -2 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & -2 & 1 \\ 0 & 0 & \dots & 1 & -2 \end{pmatrix} + \frac{v}{h_x} \begin{pmatrix} -1 & 0 & \dots & 0 & 0 \\ 1 & -1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & -1 & 0 \\ 0 & 0 & \dots & 1 & -1 \end{pmatrix}.$$

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 $\frac{d\mathbf{U}}{dt} = A\mathbf{U} + \mathbf{b}$ 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 $\mathcal{I} + h_t A$:

$$\mathcal{I} + h_t A = \begin{pmatrix} 1 - 2\tilde{\alpha} - \tilde{v} & \tilde{\alpha} & 0 & \dots & 0 & 0 & 0 \\ \tilde{\alpha} + \tilde{v} & 1 - 2\tilde{\alpha} - \tilde{v} & \tilde{\alpha} & \dots & 0 & 0 & 0 \\ 0 & \tilde{\alpha} + \tilde{v} & 1 - 2\tilde{\alpha} - \tilde{v} & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 - 2\tilde{\alpha} - \tilde{v} & \tilde{\alpha} & 0 \\ 0 & 0 & 0 & \dots & \tilde{\alpha} + \tilde{v} & 1 - 2\tilde{\alpha} - \tilde{v} & \tilde{\alpha} \\ 0 & 0 & 0 & \dots & 0 & \tilde{\alpha} + \tilde{v} & 1 - 2\tilde{\alpha} - \tilde{v} \end{pmatrix}$$

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

$$\text{abs}(\mathcal{I} + h_t A) = \begin{pmatrix} |1 - 2\tilde{\alpha} - \tilde{v}| & \tilde{\alpha} & 0 & \dots & 0 & 0 & 0 \\ \tilde{\alpha} + \tilde{v} & |1 - 2\tilde{\alpha} - \tilde{v}| & \tilde{\alpha} & \dots & 0 & 0 & 0 \\ 0 & \tilde{\alpha} + \tilde{v} & |1 - 2\tilde{\alpha} - \tilde{v}| & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & \tilde{\alpha} + \tilde{v} & |1 - 2\tilde{\alpha} - \tilde{v}| \end{pmatrix}$$

$$\begin{aligned} &\rightarrow \tilde{\alpha} + |1 - 2\tilde{\alpha} - \tilde{v}| \\ &\rightarrow 2\tilde{\alpha} + \tilde{v} + |1 - 2\tilde{\alpha} - \tilde{v}| \\ &\rightarrow 2\tilde{\alpha} + \tilde{v} + |1 - 2\tilde{\alpha} - \tilde{v}|. \\ &\quad \vdots \\ &\rightarrow \tilde{\alpha} + \tilde{v} + |1 - 2\tilde{\alpha} - \tilde{v}| \end{aligned}$$

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, $\|\mathcal{I} + h_t A\|_\infty = b = 2\tilde{\alpha} + \tilde{v} + |1 - 2\tilde{\alpha} - \tilde{v}|$. 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 $2\tilde{\alpha} + \tilde{v} < 1$:

$$\|\mathcal{I} + h_t A\|_\infty = |1 - 2\tilde{\alpha} - \tilde{v}| + 2\tilde{\alpha} + \tilde{v} = 1 - 2\tilde{\alpha} - \tilde{v} + 2\tilde{\alpha} + \tilde{v} = 1,$$

therefore $\|\mathcal{I} + h_t A\|_\infty \leq 1$.

2. If $1 - 2\tilde{\alpha} - \tilde{v} < 0$, then $2\tilde{\alpha} + \tilde{v} > 1$:

$$\|\mathcal{I} + h_t A\|_\infty = |1 - 2\tilde{\alpha} - \tilde{v}| + 2\tilde{\alpha} + \tilde{v} = 2\tilde{\alpha} + \tilde{v} - 1 + 2\tilde{\alpha} + \tilde{v} = 4\tilde{\alpha} + 2\tilde{v} - 1,$$

therefore in this case, if $\|\mathcal{I} + h_t A\|_\infty$ needs to be less than or equal to 1, then

$$\|\mathcal{I} + h_t A\|_\infty \leq 1 \implies 4\tilde{\alpha} + 2\tilde{v} - 1 \leq 1 \implies 2\tilde{\alpha} + \tilde{v} \leq 1$$

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

$$2\tilde{\alpha} + \tilde{v} < 1 \implies 2\alpha \frac{h_t}{h_x^2} + v \frac{h_t}{h_x} < 1.$$

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

$$\frac{\partial u}{\partial t} = 0.1 \frac{\partial^2 u}{\partial x^2} - 0.5 \frac{\partial u}{\partial x} \quad t \in [0, 10] \\ x \in [-2, 2]$$

$$u(x, 0) = u_{init}(x) = 10, \quad u(-2, t) = u_l(t) = 1, \quad u(2, t) = u_r(t) = 0.$$

This can be discretised to give $\frac{d\mathbf{U}}{dt} = A\mathbf{U}$ where

$$\frac{d}{dt} \underbrace{\begin{pmatrix} U_1(t) \\ U_2(t) \\ \vdots \\ U_{N-1}(t) \\ U_N(t) \end{pmatrix}}_{\mathbf{U}} = \underbrace{\left[0.1 \frac{1}{h_x^2} \begin{pmatrix} -2 & 1 & \dots & 0 & 0 \\ 1 & -2 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & -2 & 1 \\ 0 & 0 & \dots & 1 & -2 \end{pmatrix} + 0.5 \frac{1}{h_x} \begin{pmatrix} -1 & 0 & \dots & 0 & 0 \\ 1 & -1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & -1 & 0 \\ 0 & 0 & \dots & 1 & -1 \end{pmatrix} \right]}_A \underbrace{\begin{pmatrix} U_1(t) \\ U_2(t) \\ \vdots \\ U_{N-1}(t) \\ U_N(t) \end{pmatrix}}_{\mathbf{U}}$$

subject to

$$\mathbf{U}(0) = \begin{pmatrix} u_{init}(x_1) \\ u_{init}(x_2) \\ \vdots \\ u_{init}(x_{N-1}) \\ u_{init}(x_N) \end{pmatrix} \quad \text{where } u_{init}(x) = 10.$$

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

$$2\alpha \frac{h_t}{h_x^2} + v \frac{h_t}{h_x} < 1 \quad \implies \quad 2 \frac{h_t}{h_x} + 5 \frac{h_t}{h_x^2} < 10.$$

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.