Numerical Solutions of Differential Equations

Waleed A A Ali

Table of contents

# Introduction

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

# Material

All the material will be posted on the [Microsoft Teams Page](https://unioxfordnexus.sharepoint.com/:f:/r/sites/MathematicalInstitute-TCC/Shared%20Documents/Numerical%20Solutions%20to%20Differential%20Equations?csf=1&web=1&e=5yy5we) for the unit.

# Unit Outline

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

Meeting ID: 358 213 646 365 0

Passcode: Us2N6hB2

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

# Aims & Objectives

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

### Intended Learning Outcomes:

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

# Questions

For any questions, queries or issues that you see in the material, do not hesitate to contact me on [w.a.a.ali@bath.ac.uk](mailto:%20w.a.a.ali@bath.ac.uk).

# 1. Internal Workings of MATLAB

## 1.1 Floating-Point Arithmetic

Since computers have limited resources, only a finite strict subset 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 , since any real number is – in principle – truncated by the computer, giving rise to a new number denoted by , which does not necessarily coincide with the original number .

A computer represents a real number as a floating-point number in as

where:

* determines the sign of the number;
* is the base;
* is the exponent.
* is the mantissa (or significand). The mantissa has length which is the maximum number of digits that can be stored. Each term in the mantissa must satisfy for all and (to ensure that the same number cannot have different representations). The digits (with ) are often called the first significant digits of .

The set is therefore fully characterised by the basis , the number of significant digits and the range of values that can take which is with and .

A computer typically uses binary representation, meaning that the base is with the available digits (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 , which uses 64-bits as follows:

* 1 bit for (either 0 or 1) to determine the sign;
* 11 bits for (which can be );
* 52 bits for (since , it has to be equal to 1).

This gives a total of 64-bits with and . For 32-bit storage, the exponent is at most 7 and the mantissa has 23 digits. Note that 0 does not belong to since it cannot be represented in the form shown in [Equation 1.1](#eq-float) and it is therefore handled separately.

If a non-zero real number is replaced by its floating-point representation , then there will inevitably be a round-off error which is given by

where is the machine epsilon and provides the distance between 1 and its closest floating-point number greater than 1. In the binary case, (this can be found using eps(1)). In other words, the number is the maximum relative error that the computer can make while representing a real number by finite arithmetic.

Moreover, since and 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 are given by and , respectively. A positive number smaller than yields under-flow and a positive number greater than yields over-flow. The elements in are more dense near , and less dense while approaching . 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.

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

Since is a strict subset of , elementary algebraic operations on floating-point numbers do not inherit all the properties of analogous operations on . Precisely, commutativity still holds for addition and multiplication, i.e.  and . 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](https://www5.in.tum.de/~huckle/bugse.html) and the [Patriot Missile Fail](http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html) 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 , for instance, for a 1-D array, 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 ), 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 of the input and, as 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 behaves as as tends to infinity if

For example, the polynomial behaves like as tends to infinity since this term will be the fastest to grow. This can be written as as .

|  |
| --- |
| Couting flops |
| Let be given by  This function can be coded as fun in MATLAB as follows:  function [out]=fun(n)  out = 0;  for i=1:1:n   for j=1:1:n   out = out + i\*j;    end   end  end  For example, should perform the overall calculation  so fun(3) should output out=36.  This code requires the following operations:   * assignments:   + 1: out=0;   + : i=1:1:n;   + : for every i, j=1:1:n;   + : for every i, out=out+i\*j; * multiplications: i\*j; * additions: out=out+i\*j.   Therefore, for any , this code will need flops, meaning that the computational complexity is , i.e. the code runs in polynomial time. It is not uncommon to find algorithms that run in exponential time , like some recursive algorithms, or in logarithmic time . |

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

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

This will produce a simple time in seconds that MATLAB took from tic until toc, so if toc has not been types, then the timer will continue.

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

|  |
| --- |
| Code Profiling |
| Suppose that a code needs to be written that finds the 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 and displays the last term.   function [F]=Fib\_Iter(N)  S=ones(1,N);  for n=3:1:N   S(n)=S(n-1)+S(n-2);  end  F=S(end);  end   * **Recursively** by have a self-referential code that keeps referring back to itself to generate last term in the sequence from the previous terms.   function [F]=Fib\_Rec(N)  if N<3   F=1;  else   F=Fib\_Rec(N-1)+Fib\_Rec(N-2);  end  end  When running these codes for an input of , the times are very short, of the order of seconds but as gets larger, the recursive code starts to take much longer. Suppose the code efficiency is to be analysed for the input , this can be done using the profiler as follows:  >> profile on >> Fib\_Iter(40); >> profile off >> profile viewer  This will give a full breakdown of how many times every line was run and how much time it took. For Fib\_Iter(40), a total of 38 operations were performed, each taking such a short amount of time that it registers as “0 seconds”.    However, performing the profiler for Fib\_Rec(40) gives a *dramatically* different answer with the code taking nearly 247 seconds and having to call itself more than 102 million times.    This is why it is important to profile longer codes to see which parts take the longest time and which loops are the most time consuming. |

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