Department of Chemical Engineering
The University of Queensland

MATLAB PROGRAMMING

A Primer for the Process Engineer

Lars Keld Nielsen, Robert B. Newell, Ian T. Cameron, Tony Howes, 2001

Published by

The Department of Chemical Engineering
The University of Queensland
4072 Australia

CHAPTER 1

Introduction

Computational mathematics

The term computational mathematics is unlikely to inspire much enthusiasm with you as a second year process engineering students. You might think that both elements - computing and mathematics - are relatively irrelevant, abstract, and difficult for the process engineer. Yet, within the next year or so you will realise that both are essential tools for you to understand advanced engineering subjects and later as a professional engineer you will be using as an integral part of problem solving.

The reason why they may seem so irrelevant, abstract and difficult is that they are typically taught out of context as separate entities. You learn about them as separate entities removed from the engineering problems they are supposed to help you solve. The computer is for the engineer what the toolbox is for the tradesperson - a collection of tools that enables him or her to do his or her job. Imagine spending two years as an apprentice learning about the hammer without ever been shown the nail and the pieces of wood it is going to be used to join.

The reason for this odd way of teaching math and computing is partly historical and partly due to lack of good tools. An integrated approach to solving process engineering problems using computational mathematics involves

1. Formulating the physical problem, for example by using a mass balance.
2. Formulating the problem as a mathematical problem.
3. Choosing an appropriate numerical method to solve the problem.
4. Implementing the numerical method on the computer.
5. Presenting the results in a suitable form.

Steps 2, 3 and 4 each represents a level of abstraction away from the physical problem. Until recently, the final level of abstraction was too large to be fully understood for most first and second year students.

The development of powerful computational and symbolic math packages such as MATLAB, Maple and Mathematica has partly rectified this situation. These packages almost totally remove the abstraction between the numerical method formulation and the computational solution. They also go some way to reduce the abstraction between the analytical mathematical formulation and the numerical formulation.

This text will teach you how to use MATLAB as a tool to solve process engineering problems. The text is divided into three parts. The first part of the book will teach you about the various building blocks used to write MATLAB programs (Chapter 2 to 6) and how efficiently to write programs (Chapter 7). The second part is divided into chapters based on the type of mathematical problem considered. Each chapter describes process engineering examples resulting in the type of mathematical problem considered, followed by a description of the numerical methods used to solve the problem and how these methods are

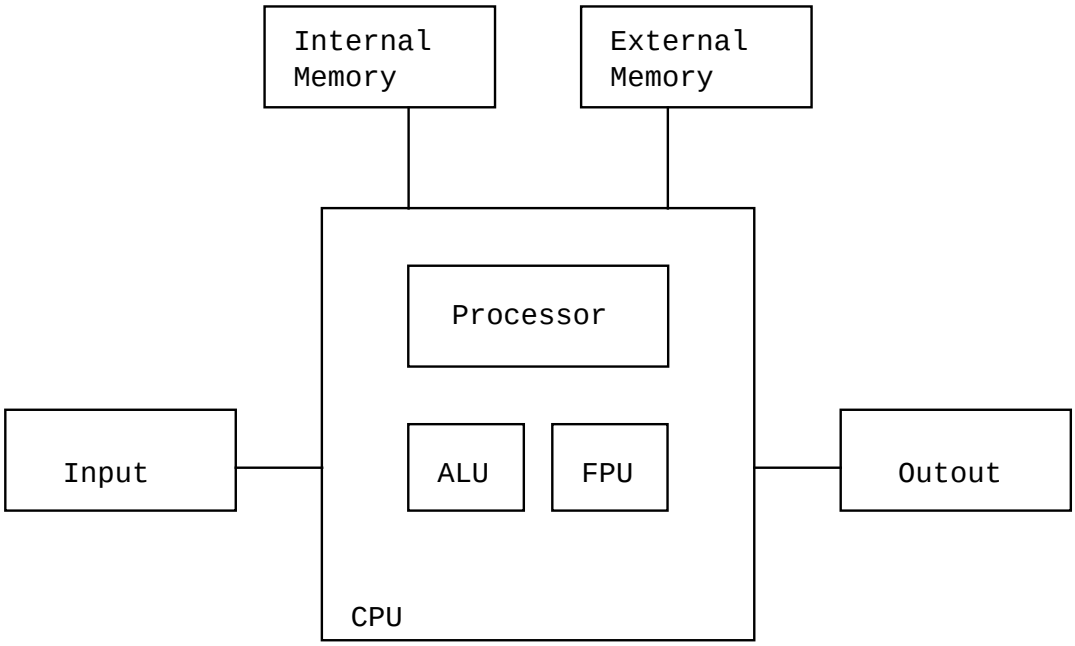
implemented in MATLAB. Finally, in Part 3 we will look at the use of MATLAB during the various stages of process design (synthesis, analysis, and optimisation).

Computing systems

Before discussing MATLAB, a brief discussion of computers is in its place. A computer is a machine designed to perform operations on a series of physical devices (hardware) according to a set of instructions called a program (software).

Computer Hardware

All computers - from microcomputers (e.g. Personal Computers, PCs) to supercomputers - have a common internal organisation.



The processor is the part of the computer that controls all other parts. It accepts input values (from a device such as a keyboard) and stores them in memory. It also interprets the instructions in a computer program. If we want to add two values, the processor will retrieve the values from memory and send them to the arithmetic logic unit, or ALU. The ALU performs the addition, and the processor then stores it in memory. The processing unit and the ALU together are called the central processing unit, or CPU. In the PC, the CPU is implemented on a single integrated circuit chip, also called a microprocessor. In newer chips (such as the Intel 80486 chip), a floating point unit ("math co-processor") is integrated on the same chip allowing for faster floating point operations, or flops. Flops are used in many programs including traditional calculation programs (spreadsheets) and in generating the graphics display on the monitor.

The newer microprocessor chip may also include some small amount of memory, or cache, used for temporary storage by the CPU. Another larger temporary memory area - typically random access memory, or RAM - is located on the main printed circuit board ("motherboard"). Finally, a more permanent storage is available on a hard disk, floppy disk, CD ROMS, tape drives, etc.

Apart from the keyboard, input can come from a pointing device (e.g. mouse), a microphone, a video camera, a modem, electrical signal sampling card, etc. Typical output devices other than the monitor include printers and modems.

Computer software

Computer software contains the instructions or commands that we want the computer to perform. Software can be divided into three broad categories: operating systems, software tools, and programming languages.

Operating systems. The operating system provides an interface between the user and the hardware by providing a (hopefully) convenient and efficient environment in which you can select and execute the software on your system. The operating system also provides a range of utilities to perform functions such as printing files, copying files from one disk to another, and listing files that you have saved on a diskette. On the PC, the Windows operating system is the most common operating system (but not the only one available).

Software tools. Software tools are programs written by other programmers to perform common operations in specific areas. For example, text editors help you enter and format text for reports, etc. They range in sophistication from simple editors with little added functionality; over word processors that allow you to include tables, charts, graphics, equations and include spelling and grammar checkers; to desktop publishing packages that provide the tools for generating professional-looking documents.

Other common types of packages are spreadsheet programs used to do relatively simple calculations, database management programs used to manage large amounts of data, and graphics packages used to present data.

There are also many specific engineering software tools, such as computer aided design packages, statistical packages, and flowsheet packages. Other engineering software includes finite-element and finite-difference packages used to solve fluid flow and stress analysis problems.

Modern software tools are typically very versatile and allow you a large range of opportunities. Still, there are problems that are so specialised or that combines so many different elements, that none of the existing packages will do the job properly. In this case, you must turn to a programming language, in order to create your own software tool.

Programming languages. Machine language is the language that is understood by the computer hardware. Since computer designs are based on two state technology (on-off), machine language is binary (0-1).

Machine language is not very suited for writing large programs. Hence, high-level languages have developed in which the instructions are more english-like. High level languages such as Pascal, C, Fortran, and Basic allow you the flexibility of doing anything within the realm of the computer. However, it generally requires many steps to perform the overall tasks of interest, such as "read some data from a floppy disk, manipulate them, and plot them on the screen". Thus, the flexibility of a programming language comes at the price of having to develop long pieces of programming code.

Why using MATLAB?

MATLAB is a mathematical computational software tool with a substantial potential as a mathematical programming language as well. MATLAB stands for "matrix laboratory" and

MATLAB's basic element is the matrix. You might initially think: "Matrices? Don't they belong to linear algebra over in the Math department? Chemical engineers do not need matrices!". In terms of the types of computing required in designing, analysing, optimising, and controlling chemical process plants, however, matrix calculations are by far the most dominant. In fact, the existence of large libraries of programming code for solving matrix problems is one of the main reasons why Fortran until very recently remained the most common programming language used by chemical engineers.

It should be stressed that many computing problems in chemical engineering can be solved more readily using one of the existing software packages. In your course, you'll use programs such as ASPEN (flowsheet simulations) to solve special problems. The need for a more flexible environment arises when solving highly specialised problems. Many companies have developed highly sophisticated models of their plant or parts of their plant. Generally, these models can only be solved using custom designed software; hence a programming environment is needed. Another need for a programming language arises, when you wish to integrate the solution of various types of problems, where no single software package offers a full solution.

As mentioned, Fortran used to be one of the most popular programming languages for mathematical computation. There are several advantages of MATLAB compared to Fortran. First of all, MATLAB's basic element is the matrix and most common matrix manipulations are build into the language itself. For example, consider performing the matrix multiplication

$$\underline{\underline{A}} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \underline{\underline{B}} = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 0 \end{pmatrix} \quad \underline{\underline{C}} = \underline{\underline{A}}\,\underline{\underline{B}}$$

and displaying the result on the screen. In MATLAB a program doing this would look something like

```
A=[1,2;3,4];
B=[5,6,7;8,9,0];
C=A*B
```

whereas the corresponding Fortran programming would look like

```
DOUBLE A(2,2), B(2,3), C(2,3)
A(1,1)=1
A(1,2)=2
A(2,1)=3
A(2,2)=4
B(1,1)=5
B(1,2)=6
B(1,3)=7
B(2,1)=8
B(2,2)=9
B(2,3)=0
DO 100 I=1,2
DO 200 J=1,3
C(I,J) = 0
DO 300 K=1,2
C(I,J) = C(I,J) + A(I,K)*B(K,J)
300     CONTINUE
200     CONTINUE
```

```
      100   CONTINUE
      WRITE(6,'(4F10.4)') C
```

Secondly, MATLAB is an interactive programming environment. In MATLAB, the above matrix multiplication would be performed directly from the MATLAB prompt. The Fortran code on the other hand would be written using an editor, then compiled, and then finally run.

Finally, MATLAB has an impressive set of build-in graphical functions. Thus, you can immediately inspect your result graphically.

All up MATLAB is a much more user-friendly environment for developing programs and allows engineers to develop programs much, much faster than when using Fortran or C. For certain types of calculations, however, there can be a substantial loss in program speed. Hence, MATLAB allows for the integration of Fortran and C code functions into MATLAB programs. Thus, when developing large programs a typical program development cycle may involve writing the program in MATLAB, identifying serious bottlenecks, and rewriting these bottlenecks in C or Fortran.

The possibility of integrating C functions into MATLAB also adds another dimension to the environment. MATLAB is extremely good at doing matrix manipulations and creating graphics, but offers little or no other functionality, such as integrating with hardware. If for example you wish to directly monitor and control a piece of process equipment, there is no direct way of doing this from MATLAB. C on the other hand is a very versatile programming language, in which code for accessing hardware is readily written. By integrating C functions in MATLAB you can at the same time make use of the strong matrix manipulation capabilities of MATLAB and the versatility of C.

Versions of MATLAB

Here we will be using MATLAB Version 4. There is a student version of MATLAB, which is identical to the professional except for four features:

- each matrix is limited to 1024 elements
- the features for preparing hardcopy of graphics is not available
- a math coprocessor is not required but will be used if available
- a Signals and Systems Toolbox is included for student use

Developing an Algorithm

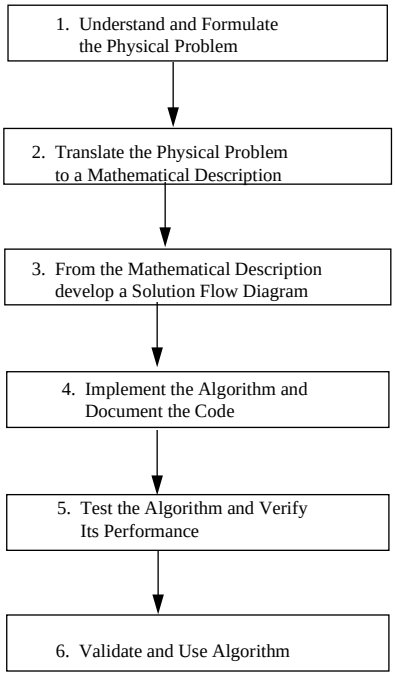
One of the key tasks in solving engineering problems is the development and implementation of an algorithm to solve the problems. What is an algorithm?

We can define an algorithm as:

"a method or procedure of computation which usually involves a series of steps".

This is nothing more than the general procedure we use in any problem solving exercise. However the important thing in computing is to have a structured and organized approach.

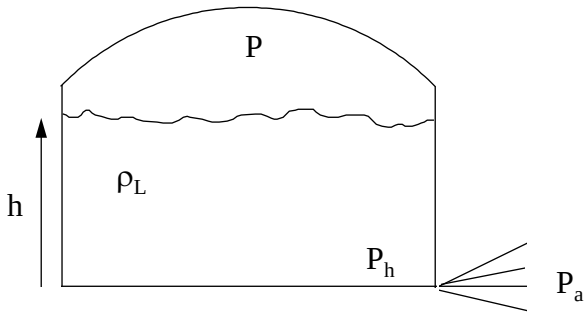
In solving any engineering problem which requires computation there is a series of overall steps. These are shown in the following figure.



Let's look at each of these steps by way of a particular example. The example is the calculation of liquid loss from a hole in the base of a storage tank.

Phase 1: Understanding the Physical Problem

The situation is represented in the following diagram.



Flow given by Bernoulli's equation. Key parameters are tank pressure, P; liquid density D_L; height of liquid h and the atmospheric pressure. Flowrate is related to the pressure difference $(P_h - P_a)$.

Phase 2: Physical Problem to Mathematical Description

Bernoulli's equation applied across the hole gives:

$$P_h - P_a = \frac{1}{2} u^2 \rho_L$$

$$u = \left[\frac{2}{\rho_L} (P_h - P_a) \right]^{\frac{1}{2}}$$

where D_L = liquid density (kg/m^3); P_h = Pascals; P_a = 101300 Pascals.

Now P_h = P + liquid head = P + D_L gh.

where h = metres, g = 9.81 m/s^2.

So,

$$u = \left[\frac{2}{\rho_L} \left(P + \rho_L g h - P_a \right) \right]^{\frac{1}{2}}$$

Mass flowrate is: $G = u.A_h.D_L$ (kg/s)

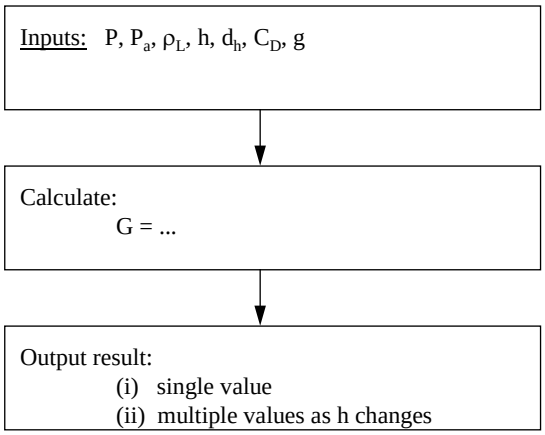
where A_h = flow area (m^2) or hole diameter = d_h

$$G = C_D A_h \rho_L \left[\frac{2(P - P_a)}{\rho_L} + 2gh \right]^{\frac{1}{2}}$$

C_D = loss (or concentration) coefficient; 0.61 (sharp hole), 0.8 (smooth hole)

This is the equation which calculates the flowrate of lost liquid.

Phase 3: Solution Flow Diagram



Phase 4: Implement Algorithm and Document Code

Below is the implementation in MATLAB. Note the following:

(i) Code documentation:

- What it does
- Who wrote it

- What date
- What are the required inputs
- What is given as output(s)
- Are there other routines used in this code
- Any revisions made, date, by whom

(ii) Comments: Put comments in the code so it is readable

(iii) Checks: For stupid input or other potential errors

(iv) Indentation can help clearly identify subsections of your algorithm.

PLEASE MAKE SURE YOU DO THIS IN YOUR CODE!

The following MATLAB code shows the implementation of the algorithm.

```
%*****************************************************************
% This routine computes the outflow of a liquid from a pressurized
% tank using a form of the Bernoulli equation.
%
% Inputs are:
%            P     = vapour space pressure (kPa)
%            h     = liquid height (m)
%            rho_l = liquid density (kg/m^3)
%            d_h   = hole diameter (mm)
%
% Outputs are:
%            G     = liquid flowrate (kg/s)
%
% Author:
%            I.T. Cameron, CAPE Centre, Univ. of Qld
% Date:
%            July 1996
%
% Revisions:   none
%
%*****************************************************************
```

```
% Parameters
  g = 9.81;  P_a = 101300;  C_d = 0.61;
% Get input from user
  P     = input('Give the vapour space pressure (kPa)  ');
  h     = input('Give the liquid height (m)  ');
  d_h   = input('Give the hole diameter (mm)  ');
  rho_l = input('Give the liquid density (kg/m^3)  ');
% compute flow area (m^2)
  A_h   = d_h^2/4*1e-6 ;
% convert input from kilopascals to pascals
  P = P*1000;
% compute the liquid flow
  G = C_d*A_h*rho_l*sqrt(2*(P-P_a)/rho_l +2*g*h);
% display output
  fprintf('Liquid flowrate is %6.2f (kg/s) \n', G)
```

Phase 5: Test and Verify the Algorithm

As a simple test use the following input.

P = 200 kPa

h = 4 metres

d_h = 25 mm

D_L = 1200 kg/m^3

Output result gives: liquid flowrate is 1.78 kg/s

This example gives you a good idea of what is expected as a standard for this subject. Keep in mind the basic principles, be structured in your approach, document and comment your code and test it to see if it is correct. Check it!.

CHAPTER 2

The MATLAB environment

Starting and finishing MATLAB

Starting MATLAB brings up the command screen. After some general information you will see the MATLAB prompt (¯), which indicates that MATLAB is waiting for you to enter a command.

As you enter commands, MATLAB will keep them in memory for a while. Thus, if you wish to send the same command a bit later, you can use the up arrow to scroll back through previous commands.

When finished working you leave MATLAB by entering **quit** or **exit**.

Initialising matrices

MATLAB provides many ways of initialising matrices. Two of them are using explicit lists and using the colon operator.

Explicit lists

The simplest way to define a matrix is to use a list of numbers in square brackets. The A and B matrices from before

$$\underline{\underline{A}} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \underline{\underline{B}} = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 0 \end{pmatrix}$$

can be written as (try it out)

¯ A = [1,2;3,4] or ¯ A = [1 2 ; 3 4]

¯ B = [5,6,7;8,9,0] or ¯ B = [5 6 7 ; 8 9 0]

That is, a matrix is "filled" row-wise with a semicolon separating each row. The column values can be separated by spaces or by commas. Remember, scalars and vectors are just matrices where either one or both dimensions are 1. The name of the matrix must start with a letter and can contain up to 19 characters that are letters, digits, or the underscore.

When defining a matrix as above, MATLAB will display the variable name and the value of the matrix on the next line. MATLAB will echo the answer of any statement, unless the statement is followed by a semicolon as in (try it out)

¯ A = [1,2;3,4];

A matrix can also be defined by listing each row on a separate line as in

```
B = [ 5,6,7
      8,9,0]
```

If there are too many numbers in a row of the matrix to fit a line, you can end one line with an ellipsis containing three or more periods followed by a carriage return, and then continue the statement below. For example,

```
B =   [      5,6, ...
     7;8,9,0]
```

An ellipsis can be used in general when one need a statement extending more than one line.

Previously defined matrices can also be used to define a new matrix. The statement

```
C = [A;5,6]
```

will create the matrix

$$\underline{\underline{C}} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

In MATLAB it is valid to have an empty matrix. An empty matrix can be defined as

```
D=[]
```

Notice that an empty matrix is different from a matrix containing only zeros.

The colon operator

The colon operator has many powerful uses in MATLAB. In terms of defining matrices, the colon operator is used as

```
start:interval:end
```

to define equispaced row vectors. For example, the statement

```
E = 3.5:-0.5:1.0
```

defines the vector

```
E = [ 3.5 3.0 2.5 2.0 1.5 1.0]
```

If no interval is given, an interval of 1 is assumed, as in

```
F = 1:5        (=[ 1 2 3 4 5])
```

Displaying matrices

The content of a given matrix can be displayed simply by entering the name of the matrix. The name of the matrix will be displayed followed by its content. It is possible to control the format of the output using the format command. For example,

- format long

will change the default 5 digit format to a 15 digit format. Try to display the E matrix in long format.

help command

In order to investigate the other display formats, we'll use the on-line help to get a list of the available formats. To see how help works just write

- help help

As you can see, there are a number of help options. For now we will just use help on a specific command, but keep the help command in mind for later. Used proper the help command can keep you from having to skim through these notes to find some command (you'll properly learn more details from the on-line help anyway). Now, entering

- help format

gives a short description of how to use format and the various options available. Try to display matrix F using the different formats. In MATLAB the **format** command can also be accessed via the Options menu.

In addition to entering the matrix name, you can also use the **disp** command. The disp command can be used to display text enclosed in single quotation marks. It can also display the content of a matrix without displaying the name. Try

- temp=37;
- disp('The temperature is');disp(temp);disp('degrees');

Information on matrices

Having entered a number of matrices, we now may want to see what matrices are available. The command who will give you a list of your variables, while the command **whos** will give you a more detailed description of your variables. If you need to use the dimension of a given matrix somewhere in your program, you can use the **size** and **length** functions,

```
[rows,columns]=size(B)

len = length(B)
```

Notice that the **size** function returns two separate 1x1 matrices: one containing the number of rows and one containing the number of columns. In order to keep both values you must specify two matrices on the left hand side in a square bracket separated by commas. The **length** function returns the largest of the two dimensions.

Extracting matrices

Individual elements in a matrix can be extracted using a subscript reference in parenthesis after the matrix name. Thus,

```
¯ a12=A(1,2)
```

will extract the element in row 1, column 2 of matrix A and place it in the (scalar) matrix a12.

The values in the parenthesis do not have to be scalars, but can be vectors. Thus,

```
¯ B([1 3],2)
```

will extract row 1 and 3, column 2.

As discussed above, you can use the colon operator to generate a vector. Thus,

```
¯ B(1:2,1)
```

will extract row 1 and 2, column 1.

You can also use intervals other than 1. Thus,

```
¯ B(2:-1:1,1)
```

will again extract row 1 and 2, column 1, but now the rows in the resulting matrix will have been swapped around.

A special case is the colon operator on its own denoting the full row or column. Thus,

```
¯ B(:,2)
```

will extract all rows, column 2.

The colon operator can also be used to recast matrices. Thus,

```
¯ G=B(:)
```

is one long column vector with the first column first, followed by the second column, and so on. The similar construct used on the left hand side,

```
¯ A(:) = 1:4
```

will result in A keeping its dimensions but being filled with the values of the vector on the right hand side.

Another function is the **diag** function that extracts diagonals (and create diagonal matrices). Please refer to the on-line help for use of this function.

Exercise 2.1

Give the size and contents of the following matrices. Check your answer using MATLAB. A very useful tool for storing the work that you do on these exercises is the **diary** function, which allows you to save what you write as a file. Use **help diary** to see how this is implemented

```
1.    A = [1,0,0,0,0,1];
2.    B = [ 2; 4; 6; 10];
3.    C = [5 3 5; 6 2 -3];
4.    D = [          3 4
            5 7
            9 10];
5.    E = [ 3 5 10 0; 0 0 ...
                     0 3; 3 9 9 8];
6.    T = [ 4 24 9];
      Q = [ T 0 T];
7.    X = [3 6];
      Y = [ D; X];
8.    R = [C; X, 5];
9.    V = [C(2,1); B];
10.   A(2,1) = -3;
```

Exercise 2.2

Give the size and contents of the following matrices. Use the following matrix G where referenced:

$$\mathbf{G} = \begin{pmatrix} 0.6 & 1.5 & 2.3 & -0.5 \\ 8.2 & 0.5 & -0.1 & -2.0 \\ 5.7 & 8.2 & 9.0 & 1.5 \\ 0.5 & 0.5 & 2.4 & 0.5 \\ 1.2 & -2.3 & -4.5 & 0.5 \end{pmatrix}$$

Check your answer using MATLAB.

```
1.    A = G(:,2);
2.    B = G(4,:);
3.    C = [10:15];
4.    D = [4:9;1:6];
5.    E = [-5,5];
6.    F = [0.0:0.1:1.0];
7.    T1 = G(4:5,1:3);
8.    T2 = G(1:2:5,:);
```

Special values and matrices

Many matrices are used so frequently that special functions are used to create them.

Computer information

- **computer.** Contains the computer type

- **eps.** This variable contains the floating-point precision for the computer being used. A computer can only represent a discrete number of values on the real axis. The floating point precision is per definition the difference between 1.0 and the next larger decimal value that the computer can represent.
- **realmax, realmin.** Largest and smallest floating point number represented by the machine.

Special constants

- **pi.** The value of π is automatically stored in this variable.
- **i,j.** These variables are initially set to the value $\sqrt{-1}$. See later for a discussion of complex numbers.
- **Inf.** This variable is MATLAB's representation of infinity, typically caused by division with zero.
- **NaN.** This value stands for Not-a-Number the result of an undefined calculation, e.g. zero divided by zero.
- **ans.** This variable stores values computed by an expression but not stored in a variable name.

Time and dates

- **clock.** This function returns the current time in a six-element row vector containing year, month, day, hour, minute, and seconds.
- **etime.** elapsed time function.

  ¯ etime(t2,t1)

  returns the time in seconds between the two clock time vectors, t1 and t2.
- **date.** Returns the current date in a character string, such as 30-Jun-94.
- **cputime.** Elapsed cputime in seconds since MATLAB was started.
- **tic, toc.** Stopwatch timer functions. **tic** will start the stopwatch timer. **toc** will give the time since tic was activated.

Elementary matrices

- **zeros, ones, and eye.** The function zeros creates a matrix full of zeros. If n,m are scalars zeros(n,m) creates a matrix of zeros with n rows and m columns. If B is a matrix zeros(B) creates a matrix of zeros of same dimensions as matrix B. The function ones work the same way, except now all entries in the matrix is ones. The function eye creates an identity matrix. For example, eye(n) creates a n x n square matrix with ones in the diagonal and zero elsewhere.
- **linspace and logspace.** linspace(a,b,N) and logspace(a,b,N) creates row vectors of length N, where the elements that are equally spaced - in a linear or log-linear sense - in the interval [a,b].

Plotting matrices

The facilities for plotting data in MATLAB are extensive. Here we shall only look at simple x-y plots. Firstly, create the vectors

- x = 1:10
- y= [54.2,58.5,63.8,64.2,67.3,71.5,88.3,90.1,90.6,89.5,90.4]

then plot the data

- plot(x,y)

A plot function will create a new figure window. In the figure window you have several options in the menu. The fastest way to jump between the figure window and the command window is using alt+tab (hold down alt and press tab once at a time to step through the other windows available. When you see the window of interest, release the alt button). If you want to clear the graphics, you simple close the figure window using close from the file menu. If you want to keep a figure and use a new window for a later plot you select new figure from the file menu (this will open a new empty window).

Returning to the above plot we may wish to add some text and a grid. The following commands

- plot(x,y)
- title('Laboratory Experiment 1')
- xlabel('Time, s')
- ylabel('Temperature, K')
- grid

would add these extra features.

Dealing with files

So far, all work has been done in memory. To be able to save your work and write programs, you'll need to have a rough understanding of how files are accessed in MATLAB. MATLAB looks for files two different places: in the current working directory and in the MATLABPATH. When looking for a file (say a command file), MATLAB will first check for it in the current working directory and then it will search the MATLABPATH. When saving a file, MATLAB will save it in the working directory unless told otherwise.

Current working directory. The current working directory will initially be the directory from which MATLAB was started. You can see what directory that is by entering

- cd

You can list the content of the working directory by entering

- dir

In order to change the working directory you can enter

- cd new_directory (e.g. cd A:\)

On a stand alone machine, you can use any directory as your working directory. In the networked environment at Uni, however, the machines do not have a harddisk. On these machines, Windows and MATLAB are loaded from a Novell network server. On the

networked machines you should make a habit of changing the current working directory to the floppy disk drive A:, or the U: directory, where each student has a small amount of hard-disk space. All your work will then be saved to this directory.

The **what** command is useful for finding out what is in the current working directory. It lists the MAT-files, M-files, and MEX-files in the directory. MAT, M and MEX-files are files with the extensions MAT, M, and MEX, respectively (e.g myfile1.mat, myfile2.m, and myfile3.mex). The mat-files stores matrices in a compact binary format (see below), the m-files stores script and function files (see next chapter), and mex files are used when linking C or Fortran programs to MATLAB.

See also **exist**, **lookfor**, and **which**.

MATLABPATH. The MATLABPATH is a string variable initialised at start-up containing a list of all the directories, where MATLAB commands can be found. If you issue the command

- path

you'll see a listing of the MATLABPATH.

On a stand alone machine you may change the content of the MATLABPATH. On a networked machine you wont be able to access the MATLAB startup file, thus you'll probably not want to change the MATLABPATH. Use help for further details on the path command.

Saving and loading. Having worked for a while you may wish to save the matrices you have created and continue another day. The command

- save

will save all your matrices in the default file matlab.mat in the current working directory. If you give a file name as in

- save mydata

all your matrices will be saved in the file mydata.mat in the current working directory. You can also choose to save only some of your matrices. Later you can load all the matrices again by entering

- load or
- load mydata

The save and load commands normally works on MAT files which have the data stored in a compact binary format. Sometimes you may wish to save data in text (i.e. ascii) format to use in another program or load a text file created somewhere else into MATLAB. In these cases, you'll use the switch -ascii at the end of the command. Please refer to the on-line help for further details.

Packing and clearing

There is a limit to how much information can be stored in memory. Throughout a long session small amounts of memory can get lost by fragmentation. The packing command, **pack**, will free up this memory without losing any of the matrices stored. The clearing command, **clear**, can be used to remove matrices no longer needed. clear can be used to clear specific variables

- clear A (clears Matrix A only)

or all variables

- clear (clear all matrices)

Writing M-files

M-files are fundamental for programming in MATLAB. There are two types of M-files: script files and function files. Function files will be discussed at a later point.

Format

Script files are simply a collection of commands as illustrated in the following listing of the file TEST.M.

```
%TEST       This is a test program.
%
%           TEST defines two vectors and create a plot of them
%

% Define x and y
x = 1:10  % Time in seconds
y= [54.2,58.5,63.8,64.2,67.3,71.5,88.3,90.1,90.6,89.5,90.4] % Temperature in degrees C
% Plot y versus x
plot(x,y)
```

In a standard layout of an M-script file, the first part contains a description of what the file does. Description lines starts with a % sign, which tells MATLAB to ignore this line. The description section is a useful reminder when you later want to use the same script file again. It also forms the on-line help information for the file, i.e. if you issue the command

- help test

from the MATLAB prompt, the first lines of the TEST.M file (upto the first line which doesn't start with a %) will be shown on the screen. The first line of the description should contain the name and a short description (the command **lookfor** searches this line first line only).

The second part contains the various commands you wish to issue. You should also add comment lines here, to explain what individual variables are and what different subsection of

the script does. This helps you to remember what exactly you did and helps other people who might use your M-file. Remember that although everything seems very clear when you write it, it might not be very clear at all when you need to use it again half a year later.

Execution

M-files are executed simply by writing their name at the prompt, i.e.

```
– test
```

tells MATLAB to find the file TEST.M and execute the commands in this file one-by-one. When finished MATLAB will return a prompt.

Notice that MATLAB is case sensitive, i.e. **test** is not the same as **TEST** or **Test**. All functions and M-files must be entered in small characters.

Advantage of using M-files

The advantage of using M-files is that you can keep a long series of commands for use later. Also you can test out a series of commands to see if they do the job and if not you can go back and edit them.

Editing

You use a standard text editor to create and edit M-files. The menu item new M-file in the file menu will open the standard Windows NOTEPAD editor with an untitled text file. You enter the description and the commands in this editor and save using save in the file menu. NOTEPAD will by default try to add the extension txt to your file, so you must write the full file name in the dialog box, e.g. TEST.M. Hereafter you can choose either to exit NOTEPAD or to jump (alt+tab) back to MATLAB to test out your new M-file.

The advantage of jumping is that the editor will remain open, so you can easily jump back and change the M-file if there are any mistakes in it. Remember though that the changes you make are only implemented when the file is saved again. If you wish to open an existing M-file from MATLAB, you can use the Open M-file item on the file menu.

Exercise 2.3

In a study of the effect of various factors on the growth performance of activated sludge, the oxygen uptake rate was measured at various temperatures.

Temperature °C	Oxygen uptake rate grams oxygen per gram dry weight
5	0.01
10	0.04
15	0.10
20	0.20
25	0.25
30	0.28
35	0.30
40	0.25
45	0.02

Write a script M-file that generates a plot of these data, including title, labels, and grid. Print the graph that this generates

Demo

Now is a good time to see some examples of the MATLAB language at work. Enter

```
demo
```

and browse through the various demos offered. The demos show both the command and the result.

Summary

In this chapter the MATLAB environment has been introduced. You have learned how to start and end a MATLAB session, how to initialise a matrix, how to display a matrix, how to find information about matrices, how to use special matrices, how to create simple plot, how to deal with files, how to pack the environment and clear matrices, and how to write M-files.

A number of special characters, commands, and functions have been introduced. They were

Special Characters

[ ]	forms matrices
( )	forms subscripts
,	separates subscripts and matrix elements
;	separates matrix elements and suppresses printing
>>	Prompts user for next command
...	continues command to new line
!	Gives access to DOS commands
%	Indicates a comment
:	generates matrices

Commands and functions

ans	Last computed value
clc	clears command screen
clear	clears workspace or specific matrices
clg	clears graphics
clock	current time
computer	Contains the computer type
cputime	Elapsed cputime in seconds since MATLAB was started
date	current date in a character string, such as 30-Jun-94
disp	displays matrix or text
eps	floating-point precision
etime	elapsed time function
exit	terminates MATLAB
eye	identity matrix
format	formats output
grid	draws grid on plot
help	invokes on-line help
i, j	The value $\sqrt{-1}$
Inf	Infinity

length	prints the greater of row and column dimension
linspace, logspace	linear or logspaced vector
load	loads matrices from a file
matlab	initiates MATLAB
NaN	Not-a-Number
ones	matrix full of ones
pi	The value of π
plot	generates a linear x-y plot
quit	terminates MATLAB
realmax, realmin	Largest and smallest floating point number
save	saves variables in a file
size	prints row and column dimensions
tic, toc	Stopwatch timer functions
title	adds title to a plot
what	lists M-files on disk
who	lists variables in memory
whos	lists variables in memory in details
xlabel	adds x-axis label to a plot
ylabel	adds y-axis label to a plot
zeros	matrix full of zeros

CHAPTER 3

Matrix and array computations

MATLAB uses two different ways of calculating with matrices. Matrix computations are matrix operations performed according to the appropriate rules for linear algebraic matrix operations. Array computations are computations performed element by element. To illustrate the difference consider the following multiplications of the two matrices

$$\underline{\underline{A}} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \underline{\underline{B}} = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

A matrix multiplication would be written

```
C = A * B
```

and result in

$$\underline{\underline{C}} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

An array multiplication would written

```
C = A .* B
```

and result in

$$\underline{\underline{C}} = \begin{pmatrix} 5 & 12 \\ 21 & 32 \end{pmatrix}$$

Notice how a period or dot (.) is used to indicate an array operation.

Matrix computations are used whenever we employ techniques from the linear algebra to solve problems. Array computations are used when we wish to evaluate the similar expressions for a range of different values, i.e. the expression could be written as a series of scalar expressions, but we use an array computation as a condensed way of writing all the expressions.

Matrix computation

The basic element of MATLAB is the matrix and MATLAB provides a number of standard matrix operations as well as several advanced operations. Here we will only discuss the simple ones.

- ' The transpose operator calculates conjugated transpose, i.e. rows and columns are swapped around (transposed) and the conjugate of the matrix taken (only of importance for complex numbers)
- + - Matrices of identical dimensions are added or subtracted element for element. If one of the matrices is a scalar, MATLAB will promote it to a matrix of the appropriate dimensions. For example

¯ A = 3 + [1,2;3,4]

results in

A = [4,5;6,7]

Otherwise if dimensions are not identical an error occurs.

- ***** Matrix multiplication. Recall that matrix multiplications can only be performed between matrices of identical inner dimensions, i.e.

  ¯ C = A*B

  is only valid if the number of columns in A equals the number of rows in B. If A has dimension l-by-m and B has dimension m-by-n, then C has dimension l-by-n.

- ^ Matrix power. X^p is X to the power p, if p is a scalar. If p is an integer, the power is computed by repeated multiplication. If p is not an integer, the calculation involves eigenvalues and eigenvectors. If x is a scalar and P is a matrix, x^P is x raised to the matrix power P using eigenvalues and eigenvectors. If both X and P are matrices an error occurs.

- **det** d=det(X) calculates the determinant of matrix X.

- **trace** trace(X) calculates the sum of diagonal elements.

- **inv** inv(X) calculates the inverse of a matrix

There is no direct function for the dot product of two vectors. However, X * Y' calculates X.Y

Exercise 3.1

Use MATLAB to define the following matrices

$$\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 0 & -1 \\ 3 & 0 \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} 1 & 3 \\ -1 & 5 \end{pmatrix} \quad \mathbf{C} = \begin{pmatrix} 3 & 2 \\ -1 & -2 \\ 0 & 2 \end{pmatrix} \qquad \mathbf{D} = \begin{pmatrix} 1 & 2 \end{pmatrix} \qquad \mathbf{I} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

then compute the matrices

1. AB	2. DB
3. BC'	4. (CB)D'
5. B^{-1}	6. BB^{-1}
7. $B^{-1}B$	8. AC'
9. $(AC')^{-1}$	10. $(AC')^{-1}(AC')$
11. IB	12. BI

Exercise 3.2

Amino acids in proteins contain molecules of oxygen (O), carbon (C), nitrogen (N), sulphur (S), and hydrogen (H) as indicated in the table

Amino Acid	O	C	N	S	H
Alanine	2	3	1	0	7
Arginine	2	6	4	0	15
Asparagine	3	4	2	0	8
Aspartate	4	4	1	0	6
Cysteine	2	3	1	1	7
Glutamate	4	5	1	0	8
Glutamine	3	5	2	0	10
Glycine	2	2	1	0	5
Histidine	2	6	3	0	10
Isoleucine	2	6	1	0	13
Leucine	2	6	1	0	13
Lysine	2	6	2	0	15
Methionine	2	5	1	1	11
Phenylalanine	2	9	1	0	11
Proline	2	5	1	0	10
Serine	3	3	1	0	7
Threonine	3	4	1	0	9
Tryptophan	2	11	2	0	11
Tyrosine	3	9	1	0	11
Valine	2	5	1	0	11

Given the molecular weights for oxygen (15.9994), carbon (12.011), nitrogen (14.00674), sulfur (32.066), and hydrogen (1.00794), calculate the molecular weight of the amino acids using MATLAB.

Array computations

Array computations are not true matrix operations. Array computations are performed individually element by element. Array computations can be performed on a single matrix, between a matrix and a scalar, or between two matrices of identical dimensions. A dot notation is used to distinguish between matrix and array computation where necessary.

- + - Matrices of identical dimensions are always added or subtracted element by element. Thus there is no difference between a matrix addition and an array addition. If one of the matrices is a scalar, MATLAB promotes it to a matrix of the appropriate dimensions. For example

  ```
  >> A = 3 + [1,2;3,4]
  ```

  results in

A = [4,5;6,7]

Otherwise, if dimensions are not identical an error occurs.

- **.*** Array multiplication.
- **./** Array division.
- **.^** Array power. X.^p will take each element of X to the power p, if p is a scalar. If p is a matrix of same dimension of X, each element in X is lifted to the power dictated by corresponding element in P.

Common math functions

Most common math functions are implemented in MATLAB. All work on an element by element basis.

- **abs(x)** calculates the absolute value of x
- **rem(x,y)** calculates the remainder of x/y,
- **sign(x)** returns the value -1 if x less than zero, 0 if x is zero, and 1 if x is greater than zero.
- **sqrt(x)** computes the squareroot of x
- **exp(x)** computes the natural base exponential of x
- **log(x)** computes the natural base logarithm of x
- **log10(x)** computes the common base logarithm of x
- **round(x)** rounds x to the nearest integer
- **fix(x)** rounds x to the nearest integer towards zero
- **floor(x)** rounds x to the nearest integer towards $-\infty$
- **ceil(x)** rounds x to the nearest integer towards ∞.

Trigonometric functions

Sine, cosine, and tangent are computed using **sin(x)**, **cos(x)**, and **tan(x)**. The inverse functions are calculated as **asin(x)**, **acos(x)**, and **atan(x)**. asin and atan return values in the interval $[-\pi/2,\pi/2]$, while acos returns values in the interval $[0,\pi]$.

Hyperbolic functions

Hyperbolic sine, cosine and tangent are computed using **sinh(x)**, **cosh(x)**, and **tanh(x)**. The inverse functions are calculated as **asinh(x)**, **acosh(x)**, and **atanh(x)**.

For more functions see the quick reference table.

Exercise 3.3

Plot the function

$$f(x) = x^{3.5}$$

for x in the interval 0 to 5.

Exercise 3.4

Evaluate the following expressions, and then check your answer by entering the expressions in MATLAB.

1. round(-2.6)
2. fix (-2.6)
3. floor(-2.6)
4. ceil(-2.6)
5. sign(-2.6)
6. abs(round(-2.6))
7. sqrt(floor(10.7))
8. floor(ceil(10.8))
9. log10(100)+log10(0.001)
10. abs(-5:5)
11. round([0:0.3:2,1:0.75:4])

Exercise 3.5

Plot the function

$$f(x) = x^{3.5}$$

for x in the interval 0 to 2π using 30 equidistant points.

Precedence of arithmetic operations

Since several operations can be combined in a single arithmetic expression, it is important to know in which order operations are performed. MATLAB follows the standard algebraic precedence, i.e.

Precedence	Operation
1	parentheses
2	exponentiation, left to right
3	multiplication and division, left to right
4	addition and subtraction, left to right

The left to right precedence is used whenever to operations of same precedence is performed, e.g.

```
- 3^3^3
```

is calculated as

```
-(3^3)^3
```

Exercise 3.6

Give the MATLAB commands to compute the following values. Assume that the variables in the equation are scalars and have been assigned values. Try to use as few parentheses as possible

1. $\text{friction} = \frac{\upsilon^2}{30s}$

2. $\text{factor} = 1 + \frac{b}{v} + \frac{c}{v^2}$

3. $\text{slope} = \frac{y_2 - y_1}{x_2 - x_1}$

4. $\text{resistance} = \frac{1}{\frac{1}{r_1} + \frac{1}{r_2} + \frac{1}{r_3}}$

5. $\text{loss} = \text{fp}\frac{1}{d}\frac{v^2}{2}$

Exercise 3.7

Give the values in vector C after executing the following statements, when A and B contain the values shown. Check your answers using MATLAB.

A = [2 -1 5 0] B = [3 2 -1 4]

1. C = A-B
2. C = B + A -3
3. C = 2*A+A.^B
4. C = B./A
5. C = B.\A
6. C = A.^B
7. C = (2).^B+A
8. C = 2*B/2.0.*A

Data analysis

We will use some data analysis functions to further illustrate the difference between matrix computations and element by element computations. Data analysis is an essential part of evaluating the data collected from engineering experiments.

MATLAB provides a range of standard data analysis functions as well as a range of more advanced analysis functions, such as convolution, digital filters, and Fourier transforms. Furthermore, many of the so-called Toolboxes (add-on functions bought separately) are concerned with the analysis of data.

Here we will only look as the simple statistics functions. Common for these functions is that they work on vectors. A matrix will be treated as a set of separate column vectors.

Min, max, sort, and median

min, max. Minimum and maximum component. min and max will also return the indices of the minimum or maximum element. For example,

```
» X=magic(4)  (Note: What is this? Use help if you are unsure)
X =
   16    2    3   13
    5   11   10    8
    9    7    6   12
    4   14   15    1
» [minimum,index]=min(X)
minimum =
    4    2    3    1
index =
    4    1    1    4
```

sort. Sort elements in ascending order. Sort returns both the sorted matrix and an index matrix indicating the permutations perfromed. For example,

```
>> X=magic(4)
X                                                    =
        16       2       3       13
        5        11      10      8
        9        7       6       12
        4        14      15      1
» [Y,I]=sort(X)
Y =
    4    2    3    1
    5    7    6    8
    9   11   10   12
   16   14   15   13
I =
    4    1    1    4
    2    3    3    2
    3    2    2    3
    1    4    4    1
```

median. The median is the value in the middle of a group, assuming that the group is sorted. If the number of values is even the median value is the average value of the values in position N/2-1 and N/2. For example,

```
» X=magic(4)
X =
   16    2    3   13
    5   11   10    8
    9    7    6   12
    4   14   15    1
» med=median(X)
med =
```

7 9 8 10

Sums and products

sum. Sum of elements. For example,

sum(1:5) = 15

prod. Products of elements. For example,

prod(1:5) = 120

cumsum, cumprod. The cumulative sum and product. For example

cumsum(1:5)=[1 3 6 10 15]

cumprod(1:5)=[1 2 6 24 120]

Exercise 3.8

Use MATLAB to check that the magic function works.

Mean and standard deviation

The sample mean of a group is the average. If x is a vector with N elements, the sample mean is calculated as

$$m = \frac{\sum_{k=1}^{N} x_k}{N}$$

or in MATLAB

```
m=mean(x)
```

In statistics we frequently use the sample mean as an estimate of the underlying true mean, denoted with the Greek letter μ (mu).

The sample standard deviation is an indication of the spreading or variation in data. If x is a vector with N elements and mean, m, the sample standard deviation is calculated as

$$s = \sqrt{\frac{\sum_{k=1}^{N} (x_k - m)^2}{N-1}}$$

or in MATLAB

```
>> s=std(x)
```

In statistics we frequently use the sample standard deviation as an estimate of the underlying true standard deviation, denoted with the Greek letter σ (sigma).

The sum of squared differences can be calculated either using an array computation or a matrix computation. Let x denote a column vector. The mean is calculated as

```
N=length(x)
```

¯ m=sum(x)/N

Using array computations the sample standard deviation is calculated as

¯ sqrt(sum((x-m) .* (x-m))/(N-1))

Using matrix computations, we first observe that the definitions of a dot product between two vectors is

$$\text{dot product} = v \cdot w = \sum_{k=1}^{N} v_i w_i$$

If both vectors are column vectors, the dot product can be expressed as the matrix multiplication

$$\text{dot product} = v \cdot w = v' w$$

Thus, the sample standard deviation is calculated using matrix computation as

¯ sqrt((x-m)' * (x-m) /(N-1))

Exercise 3.9

In the following, we consider a study performed to identify factors affecting the level of phytoplankton in a bay. Phytoplankton growth may be stimulated by many factors, including nitrogen concentration, phosphate concentration, and available light. The table below shows the results from 15 samples taken.

Calculate the mean and variance for each variable using

a. the built-in functions

b. using array computations

c. using matrix computation

Phytoplankton million per L	Nitrogen uM	Phosphate uM	Light longleys/day
0.42	2.69	0.87	266
0.47	0.63	1.22	272
0.23	2.02	0.93	55
0.24	0.64	0.81	197
0.69	2.77	0.88	455
0.49	2.29	0.84	341
0.34	2.47	0.94	202
0.34	2.65	1.14	244
0.67	3.31	0.73	451
0.48	1.76	1.08	331
0.60	1.89	1.28	374
0.57	2.46	1.03	341
0.65	2.73	1.17	412
0.58	1.16	0.98	358
0.23	1.58	1.50	190

Correlation

Correlation is used to determine how much the value of one variable depends on the value of another variable. For example, consider a collection age-height-weight data as below

	Age	Height	Weight
Person 1	10	125	45
Person 2	15	170	65
Person 3	17	190	90
Person 4	18	210	120
Person 5	10	135	50

As one would expect the data indicates, that the older people are the taller and the heavier they are (at least for teenagers) and that the taller they are the heavier they are. We say that height and weight is positively correlated with the age and that weight is positively correlated with hight.

Quantitatively correlation can be expressed by a covariance matrix or a correlation coefficient matrix. Both types of matrices are square with dimension equal the number of variables (three in the above example) and symmetric (if A correlates this much with B, then B correlates the same amount with A).

When calculating the sample covariance and correlation coefficient matrices in MATLAB, the data should be organised as above, i.e. each row represents a separate observation (a new person) and each column represents a variable. Let X be such a data variable with n observations and p variables and let μ_i , i=1..p, be the mean of all observations for each variable, then the elements in the covariance matrix are calculated as

$$C_{i,j} = \frac{\sum_{k=1}^{n} \left(X_{k,i} - \mu_i\right)\left(X_{k,j} - \mu_j\right)}{n-1}, \quad i,j = 1 \ldots p$$

In MATLAB, the covariance matrix is calculated with the statement

- C=**cov(X)**

The basic algorithm for this calculations is a matrix computation

- [n,p] = size(X) % get the size of the matrix
- D = X - ones(n,1)*mean(X) % subtract the means from the observations
- C = X'*X/(n-1) % Calculate the covariance matrix

Notice how all elements in C is calculated in one statement. A similar simple calculation is not feasible using array calculations.

Using the MATLAB function to calculate the covariance for the above example, we find

```
X =
   10   125    45
   15   170    65
   17   190    90
   18   210   120
   10   135    50

   » C=cov(X)
   C =
     1.0e+003 *
      0.0145    0.1350    0.1088
      0.1350    1.2925    1.0763
      0.1088    1.0763    0.9675
```

Notice, that the diagonal elements contains the variance of the three variables. Using the covariance matrix it is difficult directly to judge the degree of correlation. The correlation coefficient matrix is more useful to judge this. The correlation coefficient matrix is defined from the covariance matrix as

$$S_{i,j} = \frac{C_{i,j}}{\sqrt{C_{i,i} C_{j,j}}}, \quad i,j = 1 \ldots p$$

Using the MATLAB function for calculating the correlation coefficient matrix we find

S=**corrcoef(X)**

```
S =
   1.0000    0.9861    0.9182
   0.9861    1.0000    0.9624
   0.9182    0.9624    1.0000
```

You notice that the diagonal contains ones (age is 100% correlated with age, etc.), while the off-diagonal elements are between -1 and 1. In this case, you could say that height is 99% correlated to age, weight is 92% correlated to age, while weight is 96% correlated to height.

Exercise 3.10

Calculate the covariance and correlation coefficient matrices for the data in exercise 3.9. What factor do these data indicate is most important in determining the phytoplankton level.

Interpolation

Given a set of observations of an unknown function in a set of points, we often wish to obtain an estimate of the function value in a point not originally in the observations. The process of generating such estimates is termed interpolation. Interpolation will not be covered in detail in this Workbook. Please refer to a numerical methods book for the details.

MATLAB provides for both one and two dimensional interpolation, i.e. interpolation for both one and two independent variables. The two key functions are **interp1** (one dimensional) and **interp2** (two dimensional).

interp1 provides a common access to linear interpolation (using **table1**), cubic interpolation (using **icubic**) and cubic spline interpolation (using **spline**; see also **ppval**, **mkpp**, and **unmkpp** for general functions for piecewise polynomials).

As a simple example of one-dimensional interpolation, try these commands that generates a coarse sine curve, then interpolates over a finer abscissa:

```
>> x = 0:10;  y = sin(x);
>> xi = 0:.25:10;
>> yi = spline(x,y,xi);
>> plot(x,y,'o',xi,yi)
```

interp2 provides a common access to linear interpolation (using **interp4**) and cubic interpolation (using **interp5**). There is also biharmonic interpolation (**interp3**) and another linear interpolation function (**table2**).

Exercise 3.11

Estimate the enthalpy of saturated steam at 252°F, when the following table values are given

Temperature (°F)	Enthalpy (BTU/LB)
220	1153.4
240	1160.6
260	1167.4
280	1173.8

Exercise 3.12

Estimate the enthalpy of superheated steam at 480 psia and 650°F, when the following table values are given

	Enthalpy (BTU/LB)	
Pressure(psia)	600°F	700°F
440	1304.2	1361.1
500	1299.1	1357.7

Least square polynomial regression

Unlike interpolation, regression is based on the assumption that the type of function is known and only the parameters of the function is unknown. In least square regression, the unknown parameters are chosen so to minimise the sum of squared differences between the observed and the predicted function values.

polyfit(x,y,n) will fit a n-th order polynomial, p(x), to the data y, and return the n+1 coefficients in descending powers of x. **polyval(coef,x)** will evaluate the polynomium defined by the coefficients in coef in the point x and return the value.

Exercise 3.13

Nitrous anhydride (N_2O_5) will decompose homogenously to dinitrogen tetraoxide (N_2O_4) and oxygen by the following reaction:

$$N_2O_5(g) \xrightarrow{r} N_2O_4(g) + \frac{1}{2}O_2(g)$$

Assuming the reaction is first order, i.e.

$$r = kC_A$$

the concentration of nitrous anhydride (C_A) would vary according to

$$C_A = C_{A0}e^{-kt}$$

where C_{A0} is the initial concentration. Taking the natural logarithm on both side, we have

$$\ln(C_A) = \ln(C_{A0}) - kt$$

Following is a series of measurements made on this system at 313.1 K

Time (sec)	$[N_2O_5]$, gmole/liter
0	0.1000
500	0.0892
1000	0.0776
1500	0.0705
2000	0.0603
2500	0.0542
3000	0.0471

Use **polyfit** to fit a line to the logarithm of the y-data. Then use **polyval** to generate a data for the fitted line for times between 0 and 3000 at intervals of 100. Finally, plot the fitted line and the observed points (as circles) on a semilog graph using the **semilogy** plot function.

Summary

This chapter has discussed the two different types of computations performed in MATLAB. Matrix computations are performed in accordance with the conventions of the linear algebra. Array computations are scalar computations performed in parallel on an element by element basis.

The calculation of the sample standard deviation illustrated how both types of computation can be used when solving some problems. The calculation of the covariance matrix illustrated how elegantly some problems can be solved using matrix computations.

This chapter also gave a brief introduction to MATLAB's interpolation and polynomial fitting functions.

A number of operators and functions have been presented in this chapter.

Special character

'	conjugate matrix transpose
+,-	matrix or element by element addition and subtraction
*,^	matrix multiplication and matrix power
.*,./,.^	element by element multiplication, division, and power

Matrix functions

det	matrix determinant
trace	trace of matrix
inv	inverse of matrix

Element by element functions

abs	absolute value
rem	remainder of x/y
sign	sign of x
sqrt	squareroot
exp, log	exponential and natural logarithm
log10	10 base logarithm
round	rounding to nearest integer
fix	rounding to nearest integer towards 0
floor	rounding to nearest integer towards -∞
ceil	rounding to nearest integer towards ∞
sin,cos,tan	sine, cosine, and tangent
asin,acos,atan	inverse sine, cosine, and tangent
sinh,cosh,tanh	hyperbolic sine, cosine, and tangent
asinh,acosh,atanh	inverse hyperbolic sine, cosine, and tangent

Column data analysis

min,max	minimum and maximum component
sort	sorting
median	computes median

sum,prod	computes sum and product
cumsum	cumulative sum
cumprod	cumulative product
mean	sample mean
std	sample standard deviation
cov	covariance matrix
corrcoef	correlation coefficient
interp1	one dimensional interpolation
interp2	two dimensional interpolation
polyfit	least square polynomial fitting
polyval	computes values of polynomials

CHAPTER 4

Using functions in MATLAB

As you use MATLAB to perform more and more computations, you will find calculations that you wish were included as functions. In MATLAB you can create functions as M-files, that behave exactly as the build-in functions. In fact, many of the "build-in" MATLAB functions are nothing but M-file function written and provided by MathWorks.

The function M-files differs from script M-files in that the input and output variables must be specified. In this chapter, you will learn how to write function M-files. A special type of M-file functions is the so-called function functions. This type of functions differs from normal functions in that the input is functions rather than matrices. In this chapter, you will also learn how to write function functions.

Writing a function

MATLAB does not provide a secant (reciprocal cosine) function. The following function M-file implements the secant function in MATLAB.

```
function y = sec(x)
%SEC        Calculate secant of x.
%           SEC calculates the secant of x using the build-in cosine function
%
%           USAGE:      y = sec(x)
%
%           INPUT:
%           x       Input matrix variable
%
%           OUTPUT:
%           y       Returned solution matrix
%
y = 1./cos(x)
```

The key difference between writing a script M-file and a function M-file is that the first line must start with the word **function**, followed by the output argument, an equal sign, the function name, and the input arguments enclosed in parentheses. This line is what MATLAB uses to identify the M-file as a function file and defines the input and output arguments.

If the function is to return more than one variable, all the variables must be listed in the output argument, as in this example, which will return three values

```
function [dist,vel,accel] = motion(x)
```

All values returned must be computed within in the function.

A function that have multiple input arguments must list the arguments in the function statement, as shown here

function x = position(t,x0,vel,acc)

The function M-file must be named with the function name, i.e. the above function must be stored as SEC.M, in order for MATLAB to find it. Line 2 and the following lines starting with a % sign will be used for the on-line help.

There is an important difference in how script M-files and function M-files works. Script M-files are executed line by line within the normal MATLAB environment. All variables previously defined within MATLAB can be used.

Function M-files on the other hand are (automatically) compiled and remain totally isolated from the rest of MATLAB except for the input and output variables. A function can not use variables defined anywhere else in MATLAB unless they are placed in the input argument. A function will not return any other variables than those specified in the output argument. This isolation is in general an advantage as it means you do not have to consider the possible conflict of names used in a function with names used elsewhere in MATLAB. It does however mean that all variables - other than the input values - used inside a function must be assigned a value within the function.

It is important to understand, that when a program calls a function it is only the values of the input variables, not the variables themselves that goes to the function. Similarly, only the values of the output variables, not the variables themselves, are returned to the program. To illustrate the separation of the function and the program further, consider a slightly modified secant function

```
function y = sec(x)
%SEC            Calculate secant of x.
%               SEC calculates the secant of x using the build-in cosine function
%
%               USAGE:      y = sec(x)
%
%               INPUT:
%               x       Input matrix variable
%
%               OUTPUT:
%               y       Returned solution matrix
%
x=x+2*pi;       % added for illustration purposes only
y = 1./cos(x)
```

where we add 2π to x before calculating y. If we call this function from MATLAB as

- x = pi/3;
- y = 0;
- q = 0
- q = sec(x);
- disp(x) % x will be pi/3, not pi/3+2*pi
- disp(y) % y is 0 not sec(x)
- disp(q) % q is sec(x) = 2

We notice two important points:

1. A function may change the local (i.e. within the function) value of the input variable (here x), but it will not change the global value
2. The names used as arguments in the program have nothing to do with the names used in the function. The function uses y as an output argument. Still, the change in the program occurs to the q variable used in the call, not to the y variable.

nargin , **nargout.** The special variables **nargin** and **nargout** can be used to determine how many input arguments were provided by the calling program and how many output arguments are expected by the calling program. Many functions use these variables to make them more flexible. The **sort** function described in last chapter, for example, can be called with the statement

```
[Y,I]=sort(X)
```

or the statement

```
Y=sort(X)
```

In the former case, sort will return both the sorted matrix and an index matrix. In the latter, only the sorted matrix is returned.

Exercise 4.1

Write a function, cot, that computes the cotangent, i.e. 1/tan.

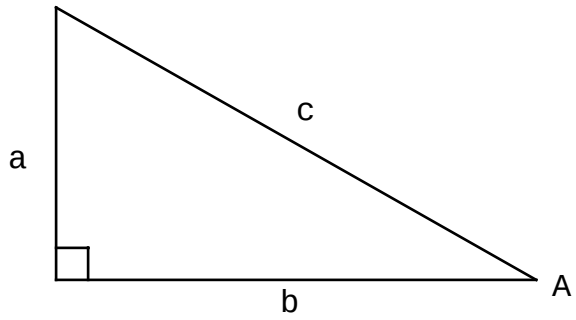
Exercise 4.2

Write a function, acsch, that calculates the inverse hyperbolic cosecant, according to

$$\text{acsch}(x) = \ln\left(\frac{1}{x} + \frac{\sqrt{1+x^2}}{|x|}\right) \quad \textit{for } x \neq 0$$

Exercise 4.3

In the right triangle



the length of the opposite side (a) and the adjacent side (b) can be calculated from the length of the hypotenuse (c) and the angle A, using

$\sin A = a/c$ and $\cos A = b/c$

Write a function that returns the lengths of both the opposite and the adjacent sides when given the hypotenuse and the angle.

function functions

Function functions differ from normal functions in that the input contains one or more functions in addition to matrices. To illustrate the use of function functions, we will use MATLAB to integrate the function humps given by the equation

$$f(x) = \frac{1}{(x-0.3)^2 + 0.01} + \frac{1}{(x-0.9)^2 + 0.04} - 6$$

This function is provided in the function M-file called HUMPS.M. To see the content of the file, we can issue the command

```
– type humps
```

giving us the result

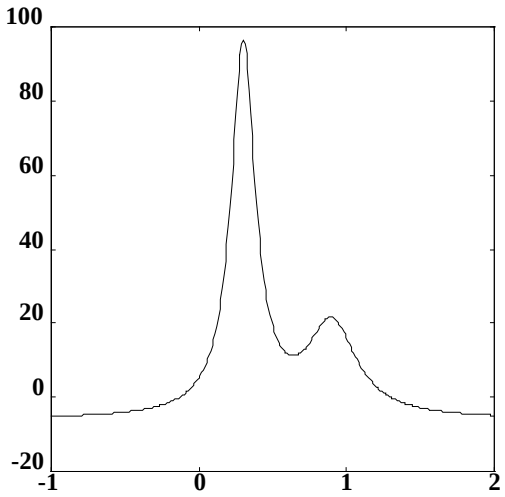
```
function y = humps(x)
%HUMPS      A function used by QUADDEMO, ZERODEMO and FPLOTDEMO.
%       HUMPS(X) is a function with strong maxima near x = .3 and x = .9.
%       See QUADDEMO, ZERODEMO and FPLOTDEMO.

%      Copyright (c) 1984-93 by The MathWorks, Inc.

y = 1 ./ ((x-.3).^2 + .01) + 1 ./ ((x-.9).^2 + .04) - 6;
```

We can plot the function using

```
– x=-1:.01:2;
– plot(x,humps(x))
```



To integrate humps from 0 to 1, we use MATLAB's numerical integration function, **quad**, as

```
– q = quad('humps',0,1)

q =
```

29.8583

Notice that the first argument to quad is a quoted string containing the name of the function, we wish to integrate (here humps).

quad, quad8. MATLAB provides two numerical integration routines. **quad** uses an adaptive Simpson's rule, while **quad8** uses an adaptive Newton Cotes 8 panel rule. The latter is better a handling certain types of singularities (please refer to Numerical methods for further details).

Writing function functions

Writing function functions is very similar to writing any other function. The following program calculates the square of any function given in the input argument

```
function f2 = squared(fun_str,x)
%SQUARED  Calculate square of f(x)
%         SQUARED calculates the square of any M-file function
%
%         USAGE:      y = squared(fun_str,x)
%
%         INPUT:
%         fun_str     string variable containing the name of the function
%         x           Point in which function is evaluated
%
%         OUTPUT:
%         y           Returned solution matrix
%
f = feval(fun_str,x)
f2 = f^2
```

Thus, computing the square of the hump function in x=0.5 would be done as

```
>> humps2 = squared('humps',0.5)
```

The only new feature is the **feval** function. The **feval** function takes the name of a function as its first argument and the input variables of this function as the following arguments. **feval** will find the function with the name given (here '**humps**'), send the input arguments to the function, accept the return values from the function, and return these values to the calling program or function.

Using this mechanism, we overcome the problem that when a function function is written we don't know what function is to be evaluated. We only have a generic string variable (fun_str in the above function), that we know eventually will contain the name of the function of interest.

Notice that the **feval** function takes a string-variable as its first argument. The reason why we do not need to put apostrophes around *fun_str* in the function function is that when calling the function function we send the string variable, '**humps**', across. Thus, inside the function fun_str will already be a string variable.

If the functions, we wish to evaluate contains more than one input argument, these arguments are just added to the **feval** input list, e.g.

feval(fun_str,x,y,z)

Exercise 4.4

The trapezoidal rule is a simple integration rule, in which the true function is replaced by a piecewise linear function and the integral calculated as the sum of (signed) areas of the trapezoids formed. If the integral of interest is

$$I = \int_a^b f(x)dx$$

and the interval [a,b] is divided into n-1 equal sections giving n points, x_i i=1:n, then the integral can be approximated by the formula

$$I \approx \frac{b-a}{n-1}\left(\frac{1}{2}f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{1}{2}f(x_n)\right)$$

Write a function, trapez, that returns the integral of a function and takes as arguments either the name of the function, the starting point, the final point, and the number of points used in the formula, or just the name of the function, starting and final points. (Hint: You will need to use **nargin** to do this.) Use this function to integrate **humps**(x) as above. Try different number of points to see how accurate the method is.

Summary

This chapter has shown how you can write your own function in MATLAB by using a special version of the M-file. A special type of functions, the so-called function functions, was also introduced. These functions take as one or more of their arguments the name of a function.

Functions and key words introduced in this chapter includes

Functions and keywords

function	keyword indicating start of a function M-file
nargin	variable containing the number input arguments specified by the calling program
nargout	variable containing the number output arguments specified by the calling program
feval	evaluates functions based on their string name
quad,quad8	numerical integration routines

CHAPTER 5

MATLAB graphics

MATLAB's graphics system provides a variety of sophisticated techniques for presenting and visualising data. This system is built upon a collection of graphics objects, such as lines and surfaces, whose appearance you can change by setting the values of object properties. It is beyond scope of this text, however, to describe the details of this system. Instead, only the high level commands will be described.

2-D Graphics

MATLAB provides a variety of functions for displaying data as 2-D graphs and for annotating these graphs. Data can be plotted on either linear and/or logarithmic axes using the plot commands

plot linear x and y axes

loglog logarithmic x and y axes

semilogx Linear y and logarithmic x axes

semilogy Linear x and logarithmic y axes

Given a single matrix as input, e.g. **plot**(Y), the **plot** commands will plot the data columnwise versus their index. Given two matrices of same size as input, e.g. **plot**(X,Y), the **plot** commands will plot the data in Y versus the data in X columnwise (if one is a vector and the other a matrix, the vector data will be used for each curve). A third string argument can be used to specify the appearance of the plot. The string can contain up to three characters, specifying line type, point type, and colour. The options are

Line	**Char**	**Point**	**Char**	**Colour**	**Char**
solid	-	point	.	red	r
dashed	- -	plus	+	green	g
dotted	:	star	*	blue	b
dashdot	-.	circle	O	white	w
		x-mark	x		

Thus, **plot**(x,y,':*b') would plot y versus x joining the points with a dotted line, show the points as a star, and use the colour blue.

Using matrices as arguments will generate multiple curves on the same graph. Another way to do this is to use multiple sets of arguments, e.g. **plot**(x1,y1,s1,x2,y2,s2). In this way, the curves can have different attributes, e.g. one curve green and the other red.

A third way is to issue the **hold** command, in which case subsequent plot commands will be added to the current axes without changing the axes' limits. Reissuing **hold** will release the hold. Alternatively, **hold on** and **hold off** can be used directly to specify the hold status.

You can add titles, axis labels, grid lines and text to your graph using the **title**, **xlabel**, **ylabel**, **grid** and **text** commands. **title**, **xlabel**, **ylabel** all take a single string argument and places the string automatically. **grid** will toggle grid lines on/off and takes no arguments. **text** takes three arguments, e.g. **text**(x,y,'text'), where x and y specifies the location using the axes from the current plot.

You can display up to four plots in the same window using the **subplot** function. The command is called with three separate 1 digit arguments, **subplot**(m,n,p). The m and n specifies that the window should be split in m by n subwindows and both can have the values 1 or 2. Thus, you can split the window two panels on top of each other (21p), two panels beside each other (12p), or four panels (22p). p specifies the current panel, i.e. the panel the next plot command will operate on. subplot(111) or subplot(1,1,1) will return the graphics windows to a single window.

To see how 2-D graphics works try the following commands

```
Y = peaks;              % generate data
size(Y)
subplot(2,1,1)
plot(Y,'r:')
title('Plot of peaks')
subplot(2,1,2)
plot(Y,rot90(Y'),'-g')
title('Plot of two matrices')
xlabel('peaks')
ylabel('rot90(peaks)')
clg     % clears the graphic screen
plot(Y,1:49,'w')
title('Plot of vector versus matrix')
clg
```

Specialised 2-D plotting functions

MATLAB includes a variety of specialised plotting in addition to the ones described above. The following list briefly describes some of the other plotting functions available in MATLAB. Please refer to the on-line help for further details.

- **bar** - creates a bar graph.
- **compass** - creates a graph of angles and magnitudes of complex numbers as arrows emanating from the origin
- **errorbar** - creates a plot with error bars
- **feather** - creates a graph of angles and magnitudes of complex numbers as arrows emanating from equally spaced points along the horizontal axis
- **fplot** - a function function that evaluates a function and plots the results
- **hist** - creates a histogram
- **polar** - creates a plot in polar coordinates of angles versus radii
- **quiver** - creates a plot of a gradient or other vector field
- **rose** - creates an angle histogram

- **stairs** - creates a graph similar to a bar graph, but without internal lines
- **fill** - draws a polygon and fills it with solid or interpolated colors

3-D graphics

mesh surfaces

A mesh surface is generated by a set of values in a matrix; each point in the matrix represents the value of a surface that corresponds to that point in the grid. For example,

- Z=peaks
- mesh(Z)

generates a surface for the peaks function. (**help peaks** will give a description of this function)

meshgrid can be used to generate X- and Y-grid matrices for use when computing the displayed Z matrix. For example,

- clg
- [Xgrid,Ygrid] = meshgrid(-0.5:0.1:0.5,-0.5:0.1:0.5)
- Z = sqrt(abs(1-Xgrid.^2-Ygrid.^2));
- mesh(Z)

can be used to create part of a sphere.

The viewing point can be changed by using a separate **view** function, for example

- view(-37.5,10)

The first argument is the azimuth (horizontal rotation in degree) and the second argument is the vertical elevation in degrees (try a couple of combinations to see the effect).

contour plots

A contour plot is an elevation map containing lines connecting points of equal elevation, similar to geographical maps.

- contour(peaks)

will generate a contour plot of the peaks function;

- contour(peaks, 10)

will generate a contour plot of the peaks function with 10 contour lines; and

- contour(peaks, -1:0.2:1)

will generate a contour plot of the peaks function with lines for each of the points in the vector, i.e. -1, -0.8, etc.

Summary

This chapter has briefly discussed the 2-D and 3-D graphical functions and options available in MATLAB. The list is far from comprehensive and the reader is advised to consult the quick reference for more functions and the on-line help for details on how to use them. The functions covered were

Functions	
plot	Generates 2-D plot with linear x and y axes
loglog	Generates 2-D plot with logarithmic x and y axes
semilogx	Generates 2-D plot with linear y and logarithmic x axes
semilogy	Generates 2-D plot with linear x and logarithmic y axes
hold	Holds current graph on screen
title, xlabel, ylabel	Adds title, x axis label, and y axis label to a plot
grid	Adds a grid to a plot
text	Adds free text to a plot
subplot	Splits graphics windows into subwindows
clg	Clear graphics
mesh	Generates a 3-D surface mesh plot
contour	Generates a contour plot
meshgrid	Generates vectors for grid computation

CHAPTER 6

Strings and formated output

MATLAB is not a general purpose programming language and MATLAB has traditionally been relatively weak in terms of dealing with strings and formatted output.

Strings

Text strings are entered into MATLAB surrounded by single quotes. For example,

```
¯ s = 'hello'
s =
        Hello
```

The text is stored in a vector, one character per element. In this case

```
¯ size(s)
ans =
        1       5
```

indicates that s has five elements. The characters are stored as their ASCII values and **abs** shows these values

```
¯ abs(s)
ans =
        72      101     108     108     111
```

In fact, a string is just a vector of integers in which a special marker (the string flag) has been set to on. Commands such as **disp** will check for this flag and - if on - they know to treat the variable as a string rather than an array of integers. With this layout, string concatenation is done straight forward by adding elements to a vector as in

```
¯ s = [s, ' World']
s =
        Hello World
```

Numeric values can be converted into strings using **num2str** (real values to string) or **int2str** (integer to strings). Strings can also be converted into numeric form using **str2num**. Combined with concatenation this can be used to display variables as part of a string as in

```
¯ f = 70; c = (f-32)/1.8;
¯ disp(['Room temperature is ', num2str(c),' degrees C'])
```

It is also possible to have a text matrix containing several strings rowwise. The only obvious condition is that the strings must be of identical length, possibly by placing blanks at the end of shorter lines.

Formatted output

The **fprintf** command gives you more control over the output format than the **disp** command. The **fprintf** command takes as arguments a format string and a list of matrices to be displayed, fprinf(format string, matrices).

In the format string is string containing ordinary characters and conversion characters. The ordinary characters include the normal alphanumeric characters and escape characters. Escape characters include

\n	new line
\t	horizontal tab
\b	backspace
\r	carriage return
\f	form feed
\\	backslash
\'	single quote

Conversion specifications involve the character %, optional width fields, and a conversion character. The conversion characters are

%e	exponential notation
%f	fixed point notation
%g	use whatever is shorter

Between the % sign and the conversion character (e,f, or g), you can add a minus sign to specify left adjustment and a field width designator, e.g. 6.2, where the first digit specifies the minimum field width and the second digit specify the precision (i.e. number of digits to the right of the decimal point).

Example

```
x = 0:.1:1;
y = [x; exp(x)];
fprintf('   x        exp(x) \n'), fprintf(' %6.2f    %12.8f \n',y)
```

Notice, how the fprintf command is repeated for each column in y.

Summary

In MATLAB, strings are vectors of integers with the string flag set, so the the integers are treated as ASCII codes. Functions described in this chapter were

Functions

num2str	Converts a numeric value to a string
int2str	Converts an integer to string
str2num	Converts a string to numeric format
disp	Displays a string or matrix
fprint	Formatted output of strings and matrices

CHAPTER 7

MATLAB programming

You should by now have a reasonable understanding of the various building stones used to write MATLAB programs: M-files, commands, graphics etc. This chapter introduces general programming strategies, programming constructs used to control program flows, and optimisation strategies.

Before starting to program you might want to keep the break command in mind. **Ctrl-C** (i.e. press down the Ctrl-key and C, simultaneously) will halt any program as soon as an interrupt is called (normally almost instantaneously). This is useful if you get stuck in a program execution (e.g. infinite loop).

How to program

Given a problem to solve, you have to decide how to write the program that will solve the problem. While it may be tempting to go directly to the computer and start hacking some code together, it is never the most efficient way. Efficient programming requires a structured approach. Otherwise you find yourself spending hours of debugging the program. For typical engineering computation problems, the 6-step problem-solving method developed below (mainly due to D. M. Etter, *"Engineering Problem Solving with Matlab"*, Prentice-Hall, 1993) is a very suitable structured approach. The six-step process is

1. State the problem clearly.
2. Describe the input and output information.
3. Work the problem by hand (or with a calculator) for a simple set of data.
4. Develop a MATLAB solution.
5. Test the solution using a variety of data sets.
6. Optimise the solution

To illustrate the technique consider the example of computing the distance between two points in the plane.

1. Problem statement

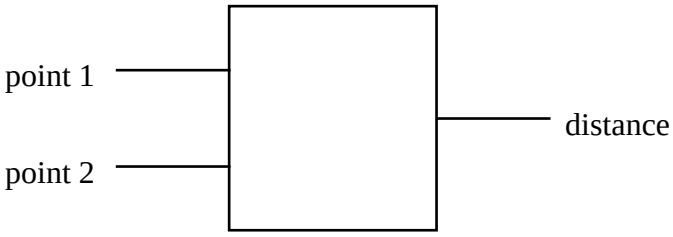
The first step is to state the problem clearly. It is extremely important to give a clear and concise problem statement to avoid any misunderstandings. Furthermore, a clear and concise statement often guides the solution. For this example, the problem statement is

> compute the straight-line distance between two points in a plane

2. Input/output description

The second step is to describe carefully the information that is given to solve the problem (input) and then to identify the values to be computed (output). A "black box" diagram is a

useful way to represent the input/output of a problem. For the example, the "black box" diagram could look something like



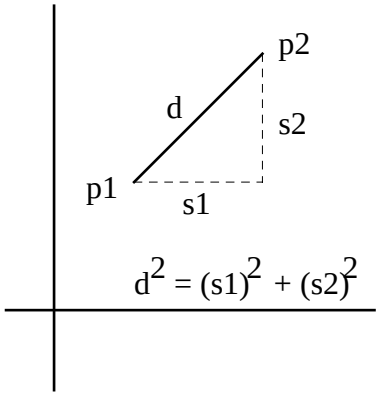
3. Hand example

The third step is to work the problem by hand or with a calculator, using a simple set of data. This is a very important step and should not be skipped even for simple problems. This is the step in which you work out the details of the problem solution. We term the step-by-step outline typical of the solution developed here the algorithm. If you can't take a simple set of numbers and compute the output by hand (or with a calculator), then you are not ready to move on to the next step. Instead you have to reread the problem and perhaps consult reference material concerned with the appropriate algorithms for the problem you wish to solve. For the specific example considered, a hand calculation would be

Let the points, p1 and p2, have the coordinates

p1 = (1,5), p2=(4,7)

We will compute the distance, d, between the two points using the Pythagorean theorem



as the basis for our algorithm. So we compute

$$d = \sqrt{s1^2 + s2^2}$$
$$= \sqrt{(4-1)^2 + (7-5)^2}$$
$$= \sqrt{13}$$
$$= 3.61$$

4. MATLAB solution

In step 4, we translate the algorithm developed in the hand example in to MATLAB code. One advantage of MATLAB is that the statements are very similar to the equations we use

for our hand solutions. The straight-forward MATLAB solution to the above problem would be

```
% This program computes the straight-line distance between two points
%
p1 = [1,5]; % initialise point 1
p2 = [4,7]; % initialise point 2
d=sqrt((p2(1)-p1(1))^2+(p2(2)-p1(2))^2) % calculate distance
```

5. Testing

The final step in the problem-solving process is to test the program. Here we should first test using the hand calculation. When the MATLAB program above is executed, the computer displays the following output:

```
d =

    3.6056
```

If the output does not match the hand calculation, we would start reviewing both calculations (the debugging process). Having reached agreement between both solutions, we should evaluate the program for a number of other data sets to be sure that the program works.

6. Optimisation

Although the above program is valid, it is not the most efficient program in MATLAB. Optimisation will be discussed later in this Chapter. In developing this program, we failed to employ the vector nature of the problem. From the linear algebra, we recall that the distance in the plane can be generalised for n-dimensional vectors as the Euclidean distance and calculated as the 2-norm of the vector joining the two points. The vector p joining the two points would be

```
p =  p2 - p1
```

and the 2-norm is calculated as

$$d = \|p\|_2 = \sqrt{p \cdot p} = \sqrt{p' p}$$

The 2-norm is the standard norm calculated for vectors by the MATLAB command **norm**, thus the MATLAB solution of the above problem could also be written

```
% This program computes the straight-line distance between two points
%
p1 = [1;5]; % initialise point 1 (notice the vector is a column vector)
p2 = [4;7]; % initialise point 2 (notice the vector is a column vector)
d=norm(p2-p1) % calculate distance
```

Program control

The MATLAB programs that we have written so far have included only sequential steps, i.e. one step was performed after another until we had completed the computation. We often need to repeat a group of statements several times and for this purpose we can use the **for** loops. Frequently, we also need a selection command that allows us to select one set of statements if a specified condition is true and another if the specified condition is false (**if** statements). Finally, we may also need a command that allows us to repeat a group of statements while a certain condition is true (**while** loops). Together these types of statements

are called *control statements* because they allow us to control the order in which statements are executed.

for loop

The **for** statement has the following general structure

```
for index = matrix
        statement group
end
```

The statement group between the **for** line and the **end** line is repeated as many times as there are columns in the matrix (the matrix can be replaced with any expression that results in a matrix). Each time through the loop, the index has the value of the corresponding column in the vector.

Most frequently, the colon operator is used to define matrix as in

```
for time=1:10
```

Some of the characteristics of the **for** loop are

- If the matrix is the empty matrix, the loop will not be calculated. Control will pass directly to the statement immediately following the end statement.
- If the matrix is scalar, the loop will be executed once, with the index containing the scalar.
- If the matrix is a row vector as in the above, 1:10, then each time through the loop, the index will contain the next element in the vector.
- If the matrix is a full matrix, then each time through the loop, the index will contain the next column in the matrix
- Upon completion of the **for** loop, the index contains the last value used

Relational and logical operators, expressions, and functions

The **if** statement and **while** statement both rely on logical variables. Thus before discussing these control statements, we will discuss how MATLAB works with logical variables.

A **logical variable** is a variable that only contains the values true or false. In MATLAB, true is represented by a non-zero element (typically 1) and false is represented by a zero. Hence, a logical matrix variable is typically a matrix in which all elements are either zero or one.

MATLAB has six **relational operators** for comparing two matrices of equal size

Relational operator	Interpretation
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
~=	not equal

Expressions involving these operators are called logical expressions and the result is a logical matrix variable. For example, if we define

```
¯ a = [2 4 6]
¯ b = [3 5 1]
```

then compute

```
¯ c = a<b
c =
        [1 1 0]
```

and

```
¯ d = (a~=b)
d =
        [1 1 1]
```

The **logical operators**

Logical operator	**Symbol**
and	&
or	|
exclusive or	xor
not	~

are used to compare logical (0-1) matrices. **and** and **or** perform element by element comparisons between two matrices, while **not** calculates the logical complement of a matrix. All the possible results are listed below

A	B	~A	xor	A|B	A&B
false	false	true	false	false	false
false	true	true	true	true	false
true	false	false	true	true	false
true	true	false	false	true	true

Logical operators can be used to join up logical expressions as in

```
¯ e = a<b & a~=b
```

An expression can contain several operators and the hierarchy, from highest to lowest is **not**, **and** , and **or**. This hierarchy can be changed by using parentheses, as in

```
¯ f = ~(a<b & a~=b)
```

in which the logical expression inside the parentheses is calculated first and then "negated", compared to

```
¯ g = ~a<b & a~=b
```

in which a<b is first "negated" (i.e. same as a>=b) before being compared to a~=b.

MATLAB also provides a set of powerful **relational and logical functions**.

- **any(**x**)** for each column of x, this function returns true if any elements in the column are non-zero
- **all(**x**)** for each column of x, this function returns true if al elements in the column are non-zero
- **find(**x**)** returns a vector containing the indices of non-zero elements in x. If x is a matrix, the indices are the indices from the column vector x(:).
- **exist('**A'**)** returns a value of 1 if A exists as a variable in the workspace, 2 if A or A.M is a file, and 0 if A does not exists. Notice, that A must be in quotes.
- **isinf(**x**), isnan(**x**), finite(**x**)** all return matrices of zeros and ones. **isinf** labels all inf elements in x as true, **isnan** labels all NaN elements as true, **finite** labels all finite elements as true.
- **isempty(**x**)** returns true if x is an empty matrix.
- **isstr(**x**)** returns true if x is a string.
- **strcmp(**string1,string2**)** compares two strings returning one if they are identical and zero otherwise. The comparison is case-sensitive and leading and trailing blanks are included.

if statements

The full **if** statement construct is

```
if  logical expression 1
        statement group A
elseif  logical expression 2
        statement group B
elseif  logical expression 3
        statement group C
----------
else
        statement group X
end
```

Maximum one statement group will be executed in an **if** statement. If the logical expression 1 is true then statement group A will be executed and all other statement groups ignored. This happens whether or not the following logical statements are true. If the logical expression 1 is false then the logical expression 2 is evaluated. If true, then statement group B will be executed and all other statement groups ignored. If false, then logical expression 3 is evaluated and so forth. If all logical expressions are false only the statement group following **else** will be executed.

The elseif and else statements are not compulsory, for example

```
if exist('my_fun')==2
        f=my_fun(x)
end
```

while loop

The general format of the while loop is

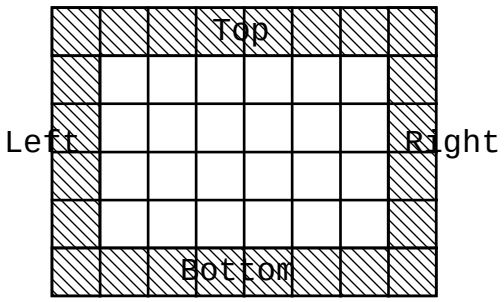
```
while logical expression
        statement group
end
```

If the logical expression is true, the statement group is executed. Then the logical expression is evaluated again. If still true, the statements are executed again, and so on until the logical expression evaluates to false. If the logical expression never becomes false, the loop becomes infinite (remember that Ctrl-C will break a loop).

It should be stressed that leaving potential infinite loops in your code is poor programming practise. We will discuss how to build in error checks in programs after studying the following example that will show a typical program containing all three types of control statements.

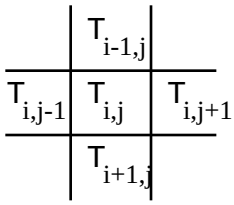
Thermal equilibrium problem

In this problem we will consider the temperature distribution in a thin metal plate as it reaches a point of thermal equilibrium. The four sides of the will be maintained at constant temperatures (not the same temperature for all four walls). The temperature at other points on the plate is a function of the temperature of the surrounding points. If we consider the plate to be similar to a grid



then a matrix can be used to store the temperatures of the corresponding points on the plate. The plate on the figure is divided into 6-by-8 temperature measurements. The shaded squares represent the sides with a known constant temperature. The problem is to determine the temperatures in the internal open squares.

In the algorithm we will use, the temperature in these interior points will be set to an arbitrary value, say zero. The temperature in each internal point will be calculated as the average of the four surrounding points as illustrated below



$$T_{i,j} = (T_{i-1,j} + T_{i+1,j} + T_{i,j-1} + T_{i,j+1})/4$$

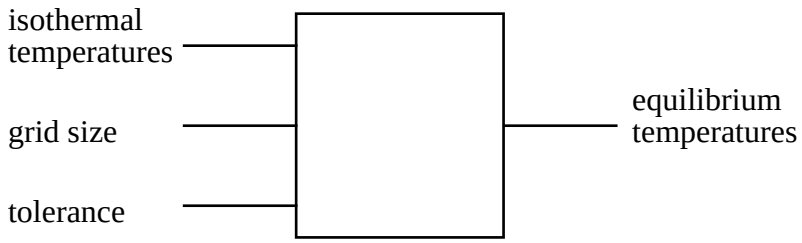
Given that this calculation will make use of the interior points and given that all the interior points were chosen arbitrarily, we will not automatically accept the first set of temperatures calculated. Instead we will compare the new temperatures calculated with the old temperatures. If the difference in any interior point is greater than some specified tolerance, we will repeat the above procedure. Ultimately, the new temperatures calculated will be within tolerance of the previous temperatures calculated and we will accept the solution.

We will now apply the problem solving strategy developed above to the problem.

1. Problem statement

Determine the equilibrium temperature values for a metal plate with isothermal sides.

2. Input/output description



3. Hand example

For our test example, we will assume the matrix contains four rows and four columns, the isothermal temperatures are 100ø on the top and sides and 50ø on the bottom, and we use a tolerance of 40ø. The iterations would look something like this

Initial temperatures

100	100	100	100
100	0	0	100
100	0	0	100
50	50	50	50

First iteration

100	100	100	100
100	50	50	100
100	37.5	37.5	100
50	50	50	50

Second iteration

100	100	100	100
100	71.875	71.875	100
100	59.375	59.375	100
50	50	50	50

Since none of the temperature changes between first and second iteration exceeds the tolerance value of 40ø, the temperatures in the second iteration is accepted as the equilibrium temperatures.

4. MATLAB solution

```
% This program initializes the temperatures in the metal plate and computes the equilibrium
% temperatures based on a tolerance value.

% Initial values
rows = 4;       % Number of rows
cols = 4;       % Number of columns
top =100;       % Top temperature
right=100;      % Right side temperature
left=100;       % Left side temperature
bottom=50;      % Bottom side temperature
tol=40;         % Tolerance temperature
% Initialize matrices
old = zeros(rows,cols);
old(1,:) = top+zeros(1,cols);           % Notice this construct create a row vectors with 'cols'
old(:,1) = left+zeros(rows,1);          % no. of columns all having the value 'top'. Same idea
old(:,cols) = right+zeros(rows,1);      % for all these four lines.
old(rows,:) = bottom+zeros(1,cols);     %
disp('Initial temperatures')
disp(old)
new =  old;
equilibrium = 0;
while ~equilibrium
        for m=2:rows-1
                for n=2:cols-1
                        new(m,n) = (old(m-1,n)+old(m+1,n)+old(m,n-1)+old(m,n+1))/4;
                end
        end
        if all(new-old<=tol)
                equilibrium = 1;        % Tolerance met
                disp('Equilibrium temperatures')
                disp(new)
        end
        old = new;
end
```

5. Testing

If we use the data from the test example, the output is

```
Initial temperatures
   100   100   100   100
   100     0     0   100
   100     0     0   100
    50    50    50    50
Equilibrium temperatures
  100.0000  100.0000  100.0000  100.0000
  100.0000   71.8750   71.8750  100.0000
  100.0000   59.3750   59.3750  100.0000
   50.0000   50.0000   50.0000   50.0000
```

6. Optimisation

The optimisation will be performed later in this chapter.

Exercise 7.1

The tolerance is an important variable in this problem solution. Try using a tolerance of 1ø.

Exercise 7.2

Another important variable in this problem solution is the grid size. Try using a grid size of 10-by-10.

Exercise 7.3

Use a contour plot to visualise the results from exercise 7.2.

return, break, and error commands

It is often desirable to be able to jump out of a loop or a function under certain conditions without properly finishing the loop or function.

return causes a normal return to the invoking function or the MATLAB prompt. **return** is frequently used to deal with special cases. For example, in a function calculating the determinant of a matrix the special case of an empty matrix may be dealt with as

```
function d = det(A)
%DET          det(A) is the determinant of A.
if isempty(A) % Special case of empty matrix
        d=1;
        return
end
% normal calculation follows
        ...
```

Obviously a similar result could be achieved using an if-statement. If the normal computation is long, however, the clarity of the program may be lost.

break terminates the execution of for and while loops, transferring control the line following the end statement. In nested loops, break exits from the innermost loop only.

error('message'**)** displays the text in the quoted string and causes an error return to the MATLAB prompt.

As discussed earlier, good programming practise calls for avoiding the potential for forming infinite loop. This is typically achieved by introducing a maximum number of iterations allowed and an iteration counter to keep track of the present number of iterations. Using a while loop the construct would look something like

```
Nmax = 100;
iterations=0;
while condition
```

```
        iterations=iterations+1
        if iterations>Nmax
                error('Maximum iterations reached')
        end
        statement group
end
disp(['solution found in ',int2str(iterations),' iterations])
```

An alternative to this approach is to use an external for-loop together with a break. This looks something like

```
Nmax = 100;
for iterations=1:Nmax
        statement group A
        if condition     % solution found
                break
        end
        statement group B
end
if (iterations==Nmax)
        error('Maximum iterations reached')
end
disp(['solution found in ',int2str(iterations),' iterations'])
```

The latter approach has the advantage that the condition statement can be placed anywhere in the loop. Hereby, one can avoid the pre-loop statements often required when using the while construct. On the other hand, the while loop statement is typically a bit more clear and whether you use the one or the other is a matter of personal taste. .

Debugging

Writing a program that works the first time is a very rare event. Typically, the coding goes through a series of iterations of running the program and fixing errors called the debugging process. This process is almost always extremely time-consuming, and it is useful to try to design any program so that debugging will be as straight-forward as possible.

A simple form of debugging involves removing semicolons at the end of statements and inserting display commands, so that you can see the result of any calculation. For the size of programs, we consider in the remainder of this book this will suffice. MATLAB includes an M-file debugger for use when writing larger programs. This is however beyond the scope of this text.

Exercise 7.4

The equilibrium temperature problem contains an open ended while loop. Rewrite the program using the for loop construct above to remove the potential risk of forming an infinite loop. At the same time, display the number of iterations required to reach a solution.

Optimising programs

When a program is working, it is worthwhile to consider whether the program performs optimally. Whether or not to try optimising the program depends very much on how often you'll need the program and how long it takes to run. If you only need the program a couple of times or if the program only takes a couple of seconds, it is probably not worthwhile spending time on optimising it.

For larger programs, it is important first to make a break down analysis of time consumption in the program. It makes little sense to spend time optimising a 2 second function block in a program that runs for half an hour. The timing functions discussed in Chapter 3 can be used to measure both true and cpu time. Another function is **flops** that will tell you the number of flops (floating point operations) performed in the session. The flops counter can be reset during a session using **flops(**0**)**.

Having identified the big time consuming blocks in a program, you would first see if these functions could be written more efficiently in MATLAB. A couple of hints are

- Try to rewrite any for-loop to matrix form. Anything that can be calculated in parallel (i.e. is not dependent on previous calculations) should be calculated in vector form.

  Consider the calculation of the sample standard deviation discussed in Chapter 3

  $$s = \sqrt{\frac{\sum_{k=1}^{N} \left(x_k - m\right)^2}{N-1}}$$

  Assuming the mean, m, is already calculated, this computation could be done using the for-loop

  ```
  ss = 0;
  for i=1:n
          ss=ss+(x(i)-m)^2;
  end
  s = sqrt(ss/(n-1))
  ```

  In MATLAB, however, it is much more efficient to use the matrix computation

  ```
  >> s= sqrt((x-m)' * (x-m) /(N-1));
  ```

- If you add elements to a matrix in a for-loop make certain to initialise the matrix. MATLAB will allow you to increase the size of a matrix interactively, but it wastes time.

  Consider calculating the cumulative product series of a vector of length n. This can be done using

  ```
  cumprod(1) = x(1)
  for i=2:n
          cumprod(i) = cumprod(i-1)*x(i)
  end
  ```

  MATLAB will automatically increase the length of the cumprod vector as needed. This however wastes time. It is more efficient to allocate the full cumprod vector by the command

  ```
  cumprod=zeros(n,1)
  ```

before doing the for loop.

- Never use repeated calls to a script M-file, always use a function M-file. Function M-files are translated once into memory and will run faster in any later calls. Script M-files are translated line by line every time they are called.

If you still need to improve speed after having optimised the MATLAB code, you'll need to use MEX-files, i.e. C or Fortran programs linked to MATLAB. This process is not trivial and is beyond the scope of this Workbook.

Exercise 7.5

Perform a break down analysis of the computing effort in the thermal equilibrium program. Use both **cputime** and **flops**. Consider ways of optimising the program.

SUMMARY

In this chapter a structured programming technique has been introduced. The technique has 6 steps

1. State the problem clearly.
2. Describe the input and output information.
3. Work the problem by hand (or with a calculator) for a simple set of data.
4. Develop a MATLAB solution.
5. Test the solution using a variety of data sets.
6. Optimise the solution

The chapter also introduced the three different program control constructs used in MATLAB. The **for** loop construct is used to repeat a statements group a given number of times. The **if** statement is used to choose between different possible statements groups. The **while** loop is used to continue performing a statement group while a certain condition remains true.

A number of special characters and functions were introduced to deal with logical variables.

relational operators

<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
~=	not equal

The **logical operators**

&	and
|	or
xor	exclusive or
~	not

relational and logical functions.

any	for each column of x, this function returns true if any elements in the column are non-zero
all	for each column of x, this function returns true if al elements in the column are non-zero
find	returns a vector containing the indices of non-zero elements in x. If x is a matrix, the indices are the indices from the column vector x(:).
exist	returns a value of 1 if A exists as a variable in the workspace, 2 if A or A.M is a file, and 0 if A does not exists. Notice, that A must be in quotes.
isinf, isnan, finite	all return matrices of zeros and ones. **isinf** labels all inf elements in x as true, **isnan** labels all NaN elements as true, **finite** labels all finite elements as true.
isempty	returns true if x is an empty matrix
isstr	returns true if x is a string.
strcmp	compares two strings returning one if they are identical and zero otherwise. The comaprison is case-sensitive and leading and trailing blanks are included.

Other functions

return	causes a normal return to the invoking function or the MATLAB prompt.
break	terminates the execution of for and while loops, transfering control the line following the end statement.
error	displays a text string and causes an error return to the MATLAB prompt.
flops	Displays and zeros the MATLAB flop counter.

CHAPTER 8

Scalar functions

In this chapter, we will look at how to solve scalar equations, i.e. find the solution to

$$f(x) = 0$$

where x is a scalar. We will also look at how we can perform a constrained minimisation of a function, i.e. find the solution to the problem

minimise f(x) with respect to the scalar x, when $x \in [x_1, x_2]$

Scalar equations

In engineering, we are frequently faced with the problem of finding the unknown in an equation. Typically, we will find the unknown by rearranging the equation. Consider solving the equation

$$F(x) = xe^x - 1 = 0$$

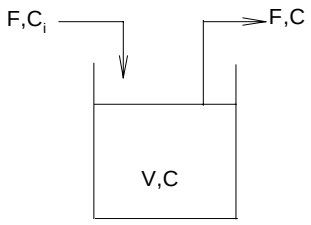
It is clear, that there is no way to rearrange the equation to an explicit form where x is isolated on the left hand side as in

$$x = ...$$

Thus, we can not find a simple analytical answer to this problem. We call this type of equations implicit in contrast to explicit. Implicit equations are very common in process engineering. In the exercises, we will consider three particular problems.

Problem 1: Steady state concentration in CSTR

Consider a well-mixed continuous stirred tank reactor (CSTR) of volume, V, to which a reactant stream of concentration C_i, is fed at a rate of F. The volume is maintained constant by harvesting at the same rate as feeding. The reactant is assumed to be converted at a rate of



$$r = k\,C^{3/2} \qquad [\text{moles dm}^{-3}\,\text{h}^{-1}]$$

We wish to determine the steady state concentration in the CSTR, C*, at a given inlet concentration, C_i. A mass balance over the reactor reads

IN:	$C_iF\Delta t$
OUT:	$C^*F\Delta t$
REACTED:	$Vr\Delta t = Vk(C^*)^{3/2}\Delta t$
ACCUMULATED:	0 (Steady state)

IN	-	OUT	=	REACTED	+	ACCUMULATED
$C_i F\Delta t$	-	$C^* F\Delta t$	=	$Vk(C^*)^{3/2}\Delta t$	+	0

Rearranging the equation, we find

$$f(C^*) = k(C^*)^{3/2} + D(C^* - C^*_i) = 0$$

where

$$D = F/V = \text{dilution rate} = \tau^{-1} = (\text{residence time})^{-1}$$

For the exercise we will assume that

$C_i = 5$ moles dm^{-3}, $F = 100\ dm^3\ h^{-1}$, $V = 1000\ dm^3$, and $k = 1\ h^{-1}\ dm^{1.5}\ moles^{-0.5}$

Problem 2: Equilibrium determination

Determine the equilibrium conversion for

$$2NO + O_2 \rightarrow 2NO_2$$

if stoichiometric amounts of NO and air are reacted at 770 K and 2 atmosphere pressure. At 770 K the standard equilibrium constant for this reaction is 3 atm^{-1}. (If y_{NO} , y_{O2} , and y_{NO2} are the mole fractions of NO, O_2 and NO_2, respectively, and the total pressure is P, then $y_{NO2}^2 / (y_{O2}\ y_{NO}^2\ P) = 3\ atm^{-1}$. As a basis, consider 2 gmoles of NO. Then there would be 1 gmole of O_2 and 3.76 gmoles of N_2. Performing a mass balance on each species and defining x as the amount of NO that reacts yields

	$2NO + O_2 \rightarrow 2NO_2$			Inerts: N_2	Total
Before	2	1	0	3.76	6.76
After	2-x	1-0.5x	x	3.76	6.76-0.5x

The mole fractions after reaction are given by

$$y_{NO} = \frac{N_{NO}}{N_T} = \frac{2 - x}{6.76 - 0.5x}$$

$$y_{O_2} = \frac{N_{O_2}}{N_T} = \frac{1 - 0.5x}{6.76 - 0.5x}$$

$$y_{NO_2} = \frac{N_{NO_2}}{N_T} = \frac{x}{6.76 - 0.5x}$$

Substitution yields,

$$3\text{atm}^{-1} = \frac{x^2(6.76 - 0.5x)}{(1 - 0.5x)(2 - x)^2}\frac{1}{2\,\text{atm}}$$

Rearranging into a normalised form

$$F(x) = \frac{x^2(6.76 - 0.5x)}{3(1 - 0.5x)(2 - x)^2 2} - 1 = 0$$

This is the scalar equation we must solve in order to determine the equilibrium conversion.

Problem 3: Equation of state

The Redlich-Kwong equation is given by

$$P = \frac{RT}{V-b} - \frac{aT^{-1/2}}{V(V+b)}$$

where P = pressure (bar), T = temperature (K), V = molar volume (cm^3/mole)
a = 0.42748 $R^2T_c^{2.5}/P_c$, b = 0.08664 RT_c/P_c,
P_c, T_c = critical pressure (bar) and temperature (K) of the fluid
R = gas constant = 83.14 cm^3 bar/mole K

The critical temperature and pressure for water are 647.3 K and 221.2 bar, respectively. We wish to find the molar volume of water at 37°C and its vapour pressure, 0.06245 bar. That is we wish to solve the equation

$$F(V) = \frac{RT}{V-b} - \frac{aT^{-1/2}}{V(V+b)} - P = 0$$

All three problems above could be rearranged to polynomial equations and hence have analytical solutions. Frequently, however, it is easier to solve the problem numerically.

General numerical algorithm for algebraic equations

Implicit equations are brought in homogenous form

f(x)=0

and solved using an iterative algorithm of the general form

1. guess an initial value of x
2. evaluate f(x)
3. decide if f(x)=0 and stop with the solution if this is the case
4. otherwise update the estimate of x and return to step 2.

Making a good initial guess is very important. To do so, we will - whenever possible - plot the function to see how many solutions the equation has and where they are located. Then we will use our insight into the problem to decide which solution(s) is (are) of interest to us. In both the problems outlined above, the functions are of third order and thus have three solutions.

Evaluating f(x) can be as easy as inserting x in a function as in the problems above. It can, however, equally well be a matter of solving a difficult numerical integration or similar.

Computers rely on discrete mathematics and we can not be certain that the computer can form an x that will make f(x) perfectly zero. Hence, our criteria for having found a solution is typically that the absolute value of f(x) is less than a certain small value, called the tolerance.

The update algorithm is what distinguish various algorithms for solving scalar equations from each other. Three important issues should be considered when choosing an update algorithm

How robust is the algorithm? No numeric method is guaranteed to find the solution from whatever initial guess we may choose. The robustness expresses how insensitive the algorithm is to how accurate we choose our initial guess. Robustness is obviously dependent on the problem. Some problems can be solved from any starting point of choice, while other problems may be impossible to solve from any starting estimates.

How fast does the algorithm converge? Convergence is typically expressed in terms of order of convergence. The order of convergence indicates the manner in which the difference between the estimate and the true value tends to zero as we near the true solution. The higher the order, the less number of iterations is required to reach a solution of a certain tolerance.

What are the computational overhead and work per iteration? The number of iterations required is not the only factor determining the computation time. Some algorithms require some overhead to start up the algorithm and some algorithms require two instead of one evaluation of the function per iteration.

In this chapter, we will consider two different algorithms: the bisection method and the Regula Falsi method. Another popular algebraic equation solver, the Newton Method, will be presented later when solving vector rather than scalar equations. Before discussing the two methods, we will first see how we use MATLABs built-in algebraic equation solver, **fzero**.

MATLABs built-in function: fzero

fzero is a function function, i.e. it expects to be given the name of the function to be solved in a string variable. **fzero** also requires a starting estimate, so a call to fzero will look like

```
x=fzero('fun_str',x0)
```

The function specified in fun_str must have only one input argument (x) and one output argument f(x) (remember to bring the function on normalised form, f(x)=0, before writing the function M-file)..

It is also possible to define the tolerance as an optional third argument, i.e.

```
x=fzero('fun_str',x0,tol)
```

Finally, it is possible to get some the intermediate calculation shown on the screen by specifying a non-zero fourth argument, as in

```
x=fzero('fun_str',x0,tol,1)
```

Exercise 8.1

Plot the function

$$f(x) = xe^x - 1$$

then use fzero to solve the equation

$$f(x) = xe^x - 1 = 0$$

to a tolerance of 10^{-8}.

Use the **flops** command to estimate the computing effort for solving this problem.

The bisection method

The bisection method involves two main steps:

1. bracketing the solution
2. reducing the interval size

Bracketing

In the bracketing phase we pick an arbitrary point, evaluate the function and then step out to point where the function takes on the opposite sign. If the function is positive in one point, negative in the other, and continuous in between, you are guaranteed that a solution to f(x)=0 exists between the two points.

Practically, bracketing is performed by choosing a starting point, then stepping out in the positive direction doubling the step length in each step, and stopping either if a sign change occurs or if a maximum number of steps have been reached. If no sign change occurs, the same is done in the negative direction. If this fails too the technique has failed to bracket the solution and another starting point must be chosen. An efficient algorithm is

choose a starting point, a
set dx = max(1/20,a/20)
set nbrack = 10
compute b = a + 2^i dx for i = 0:nbrack checking if f(a)*f(b)<0 to stop at b
if no appropiate b found try
compute b = a - 2^i dx for i = 0:nbrack checking if f(a)*f(b)<0 to stop at b

Notice, dx is set to the maximum of 1/20 and a/20, because for numerically very small starting point values (e.g. 0) a step of a/20 would be too small, while for large starting values a fixed step of 1/20 would be too small.

Bisection

Having bracketed the solution, the next step is to divide the interval [a,b] into half while maintaining a sign change between two endpoints to keep the solution bracketed. This is done by evaluating the function in the midpoint, c, and replace the interval endpoint in which the function has the same sign as in the midpoint. By repeating the bisection procedure, the interval will eventually be so small that function is guaranteed to evaluate to a value less than the tolerance.

```
set tolerance and Nmax (to avoid infinite loop)
for iteration=1:Nmax
        calculate midpoint, c = (a+b)/2
        evaluate f(c)
        if |f(c)|< tol, quit with solution
        if f(c)*f(a)> 0  set a = c
        else             set b = c
end
```

Exercise 8.2

Use the programming strategy developed in last chapter to develop a program that solves the equation

$$f(x) = xe^x - 1 = 0$$

using a bisection method.

In the hand example, choose the same starting point as in exercise 1, but a much larger tolerance. Having written and tested the program, solve the equation to the same tolerance as in exercise 1 and compare the number of iterations. Write these programs to be as general as possible, so that the function being evaluated can easily be changed if necessary.

Analyse the program using the flops command around the whole program and around the bracketing and bisection algorithm separately. Where is most of the computing effort used?

The Regula Falsi method

The bisection method is very robust: if a solution has been bracketed, the method is guaranteed to find it. Bisection is only a first order method, however, and hence relatively slow in convergence. In the bisection method, the function is evaluated in each iteration, yet the only information we use is the sign of the function. We may know that the function value in one endpoint is -1 and in the other endpoint is +1000, still we will choose the midpoint as our next estimate in the bisection method.

The Regula Falsi method employs the actual function values to form a line between the two endpoints and the new estimate is chosen as the point in which this line crosses the x-axis (linear interpolation).

If a and b are the two endpoints and fa and fb the corresponding function values, then the line going through both points has the form

$$y - fa = (x - a)\frac{fb - fa}{b - a}$$

This line intercepts the x-axis (i.e. y=0) for

$$x = c = a - fa\frac{b - a}{fb - fa}$$

which the new value we use instead of the midpoint.

Exercise 8.3

Repeat exercise 2 but this time modify the code to implement the Regula Falsi method. Do the hand example first to ensure that you have understood the algorithm.

Exercise 8.4: Steady state concentration in CSTR

Find the steady state concentration in problem 1 using the three methods discussed in this chapter.

Exercise 8.5: Equilibrium

Solve problem 2 using the three methods discussed in this chapter. Remember to plot the function first to decide which solution is of interest. How would you check your result?

Exercise 8.6: Equation of state

Before solving Problem 3, you may wish to plot the P-V curve predicted by the Redlich-Kwong equation for T=37°C and V in the range 1-10^6 cm^3/mol (make three plots in the intervals 1-100, 50-1000, and 10^3-10^6). Please consult your Thermodynamics books to recall the interpretation of this type of plot. Consider that the solution we wish to find is the intercept of the shown figure and the horizontal line P=0.06245 bar.

On the basis of the graphs you should also consider the problems that may arise when trying to solve the equation. Indicate the number of solutions, you expect to find. What is the physical interpretation of these solutions?

Now use the three methods discussed in this chapter to find the solutions. Did you experience any problems? How can you check your results? Are the results correct?

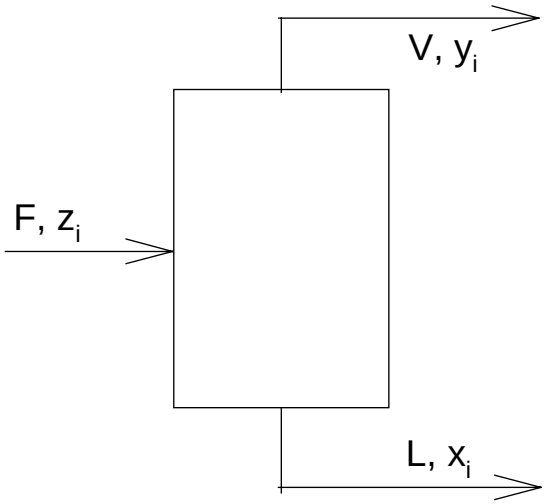
Exercise 8.7: Separation of components

Multicomponent chemical mixtures can be separated in a flash drum, a device of the general form shown below.

A mixture of components flows into the flash drum through a reducing valve. Because the pressure in the flash drum is lower than in the feed stream, the volatile components of the mixture will tend to vaporise, while the less volatile will remain as liquid. If two output streams are connected to the top and the bottom of the flash drum, then the mixture in the top stream (the vapour stream) will contain a greater percentage of the lighter components, while the bottom stream (the liquid stream) will contain a greater percentage of the liquid components.

Suppose the incoming stream contains n different components. Then we use the following notation:

- F is the flow rate of the feed stream (moles per hour)
- V is the flow rate of the vapour stream (moles per hour)
- F is the flow rate of the liquid stream (moles per hour)
- z_i is the mole fraction of component i in the incoming stream, i = 1,…,n
- y_i is the mole fraction of component i in the vapour stream, i = 1,…,n
- x_i is the mole fraction of component i in the liquid stream, i = 1,…,n



Typically we would be given the incoming flow rate and mole fractions. From these we would be interested in calculating the mole fractions and flow rates in the vapour and liquid streams. For example, if we deal with two components, and we calculate that $y_1=1$ and $x_1=0$ for the first component, while $y_2=0$ and $x_2=1$ for the second component, then we conclude that the two components are fully separated.

A mass balance over the whole drum leads to the following equation

$$F = V + L$$

$$Fz_i = Vy_i + Lx_i, \quad i = 1,\ldots,n$$

Per definition the sum of mole fractions in each stream must equal 1, thus

$$\sum z_i = 1 \qquad \sum y_i = 1 \qquad \sum x_i = 1$$

We define the distribution coefficients, K_i, as

$$K_i = y_i/x_i$$

Using these four equation we can derive

$$x_i = \frac{z_i}{1-\alpha_V+\alpha_V K_i}$$

$$y_i = \frac{z_i K_i}{1-\alpha_V+\alpha_V K_i}$$

where $\alpha_V = V/F \in [0,1]$, i.e. the fraction of total flow that vapourise.

Using the third equation we can write

$$\sum x_i - \sum y_i = 0$$

into which we substitute the derived values and find

$$\sum_{i=1}^{n} \frac{z_i (1-K_i)}{1+\alpha_V (K_i - 1)} = 0$$

This is called the *flash equation*. If the K_i's are known, the flash equation can be used to determine α_V and then using the previous two equations to determine y_i and x_i. The K_i's can be approximated using Raoult's law as

$$K_i's = p_i^*/P$$

where p_i^* is the vapour pressure of component i at temperature T and P is the total pressure. The vapour pressure of each component can be calculated using the correlation termed Antoine's equation:

$$\ln p_i^* = a_1 + \frac{a_2}{(T+a_3)} + a_4 T + a_5 \ln T + a_6 T^{a_7}$$

where the a_i's are coefficients specific for each component.

In this problem, we will consider the separation at a temperature of 372 K of a mixture of hydrocarbons with the following composition and Antoine coefficients

Hydrocarbon	z_i	a_1	a_2	a_3	a_4, 10^{-3}	a_5	a_6, 10^{-16}	a_7
C_2H_4	0.02	60.2591	-2517.89	0	12.3597	-7.00865	29.7139	6
C_2H_6	0.03	54.9873	-2636.62	0	8.50757	-5.89314	20.6619	6
C_3H_6	0.05	52.7642	-3243.44	0	4.29984	-5.13649	8.59519	6
C_3H_8	0.10	63.5869	-3550.19	0	8.18977	-7.09227	7.13799	6
i--C_2H_{10}	0.60	66.1499	-4301.38	0	6.53669	-7.24760	3.44859	6
n--C_2H_{10}	0.20	62.5348	-4038.33	0	5.35525	-6.64216	4.63607	6

For a total pressure of $2.0673 \cdot 10^6$ pascals, find the vapour fraction α_V using **fzero** to solve the flash equation. Then determine the x_i's and y_i's.

Constrained scalar function minimisation

From calculus we know that the solution to the problem

$$\min_{x_1 < x < x_2} f(x)$$

is found either in the one of the endpoints or in one (if any) of the interior stationary points of f(x), i.e. the solutions to f '(x) = 0. If f(x) is readily differentiable, this can be used as the basis for an analytic solution to the problem.

If f(x) is not readily differentiable, it is difficult to find the stationary points. Hence, we are interested in some numerical method of finding a solution. The method we will consider here follows the same principle as the methods used for solving the equations above, i.e. the solution is first bracketed and then the interval is reduced until a solution is found.

It should be stressed that minimisation routines find local minima, not the global minimum, so as for finding solutions to non-linear equations, the function should be plotted and/or several starting points attempted.

MATLAB's built-in minimiser

MATLAB has a built-in minimiser function function, **fmin**. This function takes several options, but in its simplest form it is used as

```
>> fmin('fun_str',x1,x2)
```

where fun_str specifies a function M-file with the function to be minimised. x1 and x2 species the initial interval over which the function should be minimised.

Exercise 8.8

Plot the function

$$f(x) = x^2-x$$

between $-3<x<3$. Then use **fmin** to minimise the function over this interval.

Golden section search

fmin uses two different algorithms, one of which is a golden section search. The golden section refers to a way of dividing an interval into two subintervals, so that the ratio between the two subintervals is the same as the ratio between the longest subinterval and the full interval.

$$\frac{x+y}{x} \quad \text{———————————————————————————————}$$

$$\frac{x}{y} \quad \text{———————————————————————}$$

$$x/y = y/(x+y)$$

If we set x+y=1, this equation can be solved to

$$x = (3 - \sqrt{5})/2 \approx 0.382 \qquad \text{and } y = (1-x) \approx 0.618$$

If the full interval is $[x_1,x_2]$, we will define two interior points as

$$a = x_1 + x\cdot(x_2-x_1)$$

$$b = x_1 + y\cdot(x_2-x_1) = x_2-x\cdot(x_2-x_1)$$

In the golden section search, we evaluate the function in a and b. If f(a)>f(b), we replace the left interval point, x_1, with a, otherwise we replace the right interval point, x_2, with b. Graphically, going from the k'th iteration step to the k+1'th step look something like illustrated below.

In the example, we have found the two interior points, a^k and b^k, using a golden section. Evaluating the function in these points, we find $f(a^k)>f(b^k)$ and set the new left endpoint $x_1^{k+1} = a^k$ (leaving $x_2^{k+1} = x_2^k$). In the k+1'th iteration, we perform another golden section to get the two new interior points, a^{k+1} and b^{k+1}. However, by having used golden section in the previous one point is already known, here $a^{k+1} = b^k$. More importantly the function value in the point would already have been calculated. Thus, only one point has to be updated, in this case, $b^{k+1} = x_2^{k+1} - x\cdot(x_2^{k+1} - x_1^{k+1})$.

The golden search technique is repeated until the distance between x_1 and x_2 is small enough to be ignored and the average then used as the solution. Sometimes insignificant reduction in the function values between present and last iteration is used as an additional termination criteria.

Exercise 8.9

In exercise 8.7 you solved the flash equation for given pressure. We wish to use the flash drum to remove the traces of C_2 hydrocarbons from the liquid stream. Using a low pressure in the drum would almost totally remove the traces of C_2 hydrocarbons. At low pressure, however, you will also loose a substantial amount of the C_3 and C_4 hydrocarbons. To express this mixed objective, we look at the following extreme problem

$$\max_P \left(100\alpha_V \sum_{i=1}^{2} y_i + (1-\alpha_V) \sum_{i=3}^{6} x_i \right)$$

where the first part stipulates that we wish as much of the C_2 hydrocarbons to enter the vapour stream and the second part stipulates that we wish as much of the C_3 and C_4 hydrocarbons to enter the liquid stream. The 100 is an attempt to way up the two different objectives against each other.

The maximisation problem is readily turned into a minimisation problem by using the negative of the function which can be solved using **fmin**. The function M-file to be minimised would look something like

```
function objval = object(P)
Use Antoine's equation and P to determine K using Raoult's law.
Solve flash equation to determine αV , e.g.  αV = fzero('flasheq',0.5)
Determine x and y using αV and K
objval = -100*αV*sum(y(1:2)) -(1-αV)*sum(y(3:6))
```

Before writing this function we need to look at the **global** specifier. Recall that functions are isolated from the surrounding program, i.e. only variables specified as input arguments can be used inside a function. Typically, this does not constitute a major problem; you just specify all the variables needed as input arguments. When using MATLAB's built-in function functions, however, you may have a problem. For example, fzero will send only one input argument (the variable you try to solve for) to the function you specify.

In order to solve the flash equation using **fzero** we need to have K given inside the **flasheq** function. K however changes with pressure so each time the object function is called a new

K is required in **flasheq**. **fzero** will only send the most recent guess of α_V to **flasheq**, thus K can not be specified as an argument to **flasheq**. The only way around this problem is to define K as a global variable. Global variables are defined using the specifier **global** in both the function where it is assigned a value and in the function where it used. For our problem above we would have something like

object.m

```
function objval=object(P)
global K
z =[0.02;0.03;0.05;0.1;0.6;0.2]; % Feed mole fraction
T = 372;          % Temperature in Kelvin
% Antonine coefficients
A=[     60.2591,-2517.89,0,1.23597e-2,-7.00865,2.97139e-16,6; ...
        54.9873,-2636.62,0,8.50757e-3,-5.89314,2.06619e-16,6; ...
        52.7642,-3243.44,0,4.29984e-3,-5.13649,8.59519e-17,6; ...
        63.5869,-3550.19,0,8.18977e-3,-7.09227,7.13799e-17,6; ...
        66.1499,-4301.38,0,6.53669e-3,-7.24760,3.44859e-17,6; ...
        62.5348,-4038.33,0,5.35525e-3,-6.64216,4.63607e-17,6];
% Calculate distribution constants
lnp = A(:,1)+A(:,2)./(T+A(:,3))+A(:,4)*T+A(:,5)*log(T)+A(:,6).*T.^A(:,7);
K=exp(lnp)/P;
alphaV = fzero('flasheq',0.5);  % Solve flash equation
x=z./(1+alphaV*(K-1));
y=z.*K./(1+alphaV*(K-1));
objval=-100*alphaV*sum(y(1:2))-(1-alphaV)*sum(x(3:6));
```

flasheq.m

```
function f=flasheq(alphaV)
global K
z =[0.02;0.03;0.05;0.1;0.6;0.2]; % Feed mole fraction
f = sum(z.*(1-K)./(1+alphaV*(K-1)) );
```

Plot the value of the object function for P in the interval [1.8,2.6] MPa. Then, use **fmin** to find the optimal pressure in this interval.

Summary

This chapter has presented methods to solve the scalar equation f(x)=0 and to minimise a scalar function f(x) over an interval.

The built-in MATLAB function function, **fzero**, will solve a scalar equation if given a function name and a starting guess. This is the function we will use in the rest of the course. The algorithm used by **fzero** is an improved version of the Regula Falsi method that itself is an improved version of the bisection method. All three methods employ bracketing, i.e. the first step performed is to create an interval in which a solution is guaranteed to be found. The difference is the way in which the methods reduce this interval to trap the solution.

fmin solves the scalar minimisation problem. One of two algorithms used by fmin is the golden section algorithm. Using function functions such as **fzero** and **fmin** it is sometimes necessary to use **global** variables to change the value of variables not in the argument list.

CHAPTER 9

Linear equation systems

The solution of linear equations is important in itself and forms an essential part of solving non-linear problems as we shall see later. A linear equation system

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$
$$\cdots$$
$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m$$

where x_i are the unknowns is readily written in matrix form as

Ax=**b**

where **A** is a m-by-n matrix, **x** is a n-by-1 (column vector), and **b** is a m-by-1 (column) vector.

In this chapter, we will only consider the case where **A** is square (m=n), i.e. where there are as many equations as there are unknowns and the equation system can be solved in the traditional analytical sense.

It should be noted that over-determined (m>n) and under-determined (m<n) systems can still be solved in a least square sense, i.e. we define the solution as the vector **x** that minimises

$(\mathbf{Ax}-\mathbf{b})^T(\mathbf{Ax}-\mathbf{b})$

Although not covered any further in this Workbook, this definition of a solution is used in many areas of computation and is fully supported by MATLAB.

From linear algebra, we recall that the solution to a linear equation system represents the point of intersection of two lines in the 2-dimensional case, the point of intersection of 3 planes in the three dimensional case, and - in general - the point of intersection of n n-dimensional hyperplanes.

In the two-dimensional case, there are three possible solutions: the two lines intersects in point (unique solution), the two lines are parallel and never insect (no solution), or the two lines are identical (infinite number of solutions). In the first case the square matrix **A** will be non-singular (i.e the rows in **A** are linearly independent), while in the latter two cases **A** will be singular.

This is also true for the general n-dimensional case and we can check whether the square matrix **A** is non-singular by checking if it has full rank, i.e. if rank(**A**)=n. In MATLAB this is done by

```
>> r=rank(A)
```

Gaussian elimination and back substitution

If **A** is square and of full rank, the linear equation system is typically solved by gaussian elimination and back substitution. This process is essentially the process you would have used over and over again when solving linear equations by hand (e.g. when finding where two lines cross).

To solve the following system

$$\begin{bmatrix} 3x_1 & +2x_2 & -x_3 & = 10 \\ -x_1 & +3x_2 & 2x_3 & = 5 \\ x_1 & -x_2 & -x_3 & = -1 \end{bmatrix}$$

or in matrix form

$$\begin{pmatrix} 3 & +2 & -1 \\ -1 & +3 & 2 \\ 1 & -1 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 5 \\ -1 \end{pmatrix}$$

you first remove all subdiagonal elements in **A** using row additions (Gaussian elimination)

$$\begin{matrix} \text{Row 2:Row 2} + 1/3 \text{ Row 1} \\ \text{Row 3:Row 3} - 1/3 \text{ Row 1} \end{matrix} \sim \begin{pmatrix} 3 & +2 & -1 \\ -1+1 & +3+2/3 & 2-1/3 \\ 1-1 & -1-2/3 & -1+1/3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 5+10/3 \\ -1-10/3 \end{pmatrix}$$

$$\sim \begin{pmatrix} 3 & +2 & -1 \\ 0 & +11/3 & 5/3 \\ 0 & -5/3 & -2/3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 25/3 \\ -13/3 \end{pmatrix}$$

$$\text{Row 3:Row 3} + 5/11 \text{ Row 2} \sim \begin{pmatrix} 3 & +2 & -1 \\ 0 & +11/3 & 5/3 \\ 0 & -5/3+5/3 & -2/3+25/33 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 25/3 \\ -13/3+125/33 \end{pmatrix}$$

$$\sim \begin{pmatrix} 3 & +2 & -1 \\ 0 & +11/3 & 5/3 \\ 0 & 0 & 1/11 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10 \\ 25/3 \\ -6/11 \end{pmatrix}$$

Then you would solve the resulting diagonal system can the be solved from the bottom up (back-substitution)

$$1/11 x_3 = -6/11 \Leftrightarrow x_3 = -6$$

$$11/3 x_2 + 5/3 x_3 = 25/3 \Leftrightarrow x_2 = (25/3 + 5/3 * 6) * 3/11 = 5$$

$$3x_1 + 2x_2 - x_3 = 10 \Leftrightarrow x_1 = (10 - 2*5 - 6)/3 = -2$$

The only modification to this strategy used when computing the solution numerically is that the pivoting element (i.e. the element used as the base row to add to other rows) is chosen carefully in each step of the gaussian elimination. Typically, rows are swapped in order to make the pivoting element the largest element in the column presently being eliminated. This is done to improve the numerical accuracy (if the pivoting element was 0.001 and one of the

elements to be removed was 1000, then we would have to add 1000000 times the pivoting row to this row; this can cause large errors).

Left and right division

Linear equation are solved in MATLAB using the so-called left division denoted by a backslash as in

```
>> x = A\b % the solution to Ax=b
```

A\b is defined whenever b has as many rows as A. If A is square, gaussian elimination and back substitution is used to calculate x (if A is not square, the equation will be solved in a least square sense). If b is a matrix, x will be a matrix of the same dimension, in which column j corresponds to the solution to

```
AX(:,j)=B(:,j)
```

i.e. you can solve several sets of linear equation with identical A matrix simultaneously. If the A matrix is close to singular (see below), a warning message will be displayed.

Right division is defined in terms of left division by

```
b/A = (A'\B')'
```

and is a solution to the equation xA=b.

Singularity and ill-conditioning

As mentioned above, **A** being singular corresponds to two lines (or hyperplanes, in general) being parallel or identical and this results in either no or an infinite number of solutions, respectively.

In an analytical sense, singularity is a case of either or. In a numerical sense, however, the chance is gradual as the accuracy of the numerical solution deteriorates due to round of errors the closer the hyperplanes nears to parallel. To illustrate this point, consider the following equations

$$49x_1 - 5x_2 = 39$$
$$x_1 - 0.10x_2 = 0.80$$

with the solution x_1 =1 and x_2 =2. If the first line was scaled by dividing by 49 and using only 2 significant figures, it would read

$$x_1 - 0.10x_2 = 0.80$$

This equation is now identical to the second line indicating singularity.

Nearly singular matrices are termed ill-conditioned and the corresponding linear equation system is typically solved using singular value decomposition.

Singular value decomposition

Consider a set of linear equations in matrix form

$$\mathbf{Ax} = \mathbf{b}$$

Using singular value decomposition, the coefficient matrix **A** can always be transformed into the product of three matrices

$$\mathbf{A}=\mathbf{U}\mathbf{W}\mathbf{V}^T$$

where if **A** is m-by-n, **U** is m-by-n, **W** is n-by-n, and **V** is n-by-n. In addition, **U** and **V** are each orthogonal, i.e.

$$\mathbf{U}^T\mathbf{U} = \mathbf{I} \qquad \mathbf{V}^T\mathbf{V} = \mathbf{I}$$

Also, **W** is a diagonal matrix (the diagonal elements are the only non-zero elements) can only be and the diagonal elements, w_i's, are termed the singular values of the matrix. If any of the singular values are zero the matrix is singular and the rank of A equals the number of non-zero singular values.

Ill-conditioning (as discussed above) is characterised by a large difference between the numerically smallest and largest singular value. We define the condition number as the ratio between these extreme values, i.e.

$$\text{condition number} = \frac{|w_i|_{max}}{|w_i|_{min}}$$

In numerical routines, it is more convenient to use the reciprocal condition number. The reciprocal condition number has values between 0 and 1 with values close zero indicating ill-conditioning.

Solving ill-conditioned equation systems using singular value decomposition involves setting the small singular values to zero. In effect, this corresponds to combining one or more linear combinations in the original system. The resultant system is thus under-determined and can only be solved in a least square sense. The benefit is that the final solution is much less sensitive to the ill-conditioning, i.e. less sensitive to the inevitable round-off error of the computations.

In MATLAB, singular value decomposition is performed by the function

>> [U,S,V] = **svd(A)**

This function is used when MATLAB calculates the condition number **cond(A)** or the reciprocal condition number **rcond(A)**.

Exercise 9.1

Rewrite each of the following linear equation systems to matrix form, **A**x = **b**. If possible, solve the equation by hand using Gaussian Elimination and back substitution. Use **rank**, **cond**, and **rcond** to gain information of the potential difficulties in solving the equation system. Finally, solve the equation using a left division.

1. $$\begin{aligned} -2x_1 + x_2 &= -3 \\ x_1 + x_2 &= 3 \end{aligned}$$

2. $$\begin{aligned} -2x_1 + x_2 &= -3 \\ -2x_1 + x_2 &= 1 \end{aligned}$$

3. $$\begin{aligned} -2x_1 + x_2 &= -3 \\ -6x_1 + 3x_2 &= -9 \end{aligned}$$

4.
$$\begin{aligned} -2x_1 + x_2 &= -3 \\ -2x_1 + x_2 &= 3.00001 \end{aligned}$$

5.
$$\begin{aligned} 3x_1 + x_2 - x_3 &= 10 \\ -x_1 + 3x_2 + 2x_3 &= 5 \\ x_1 - x_2 - x_3 &= -1 \end{aligned}$$

6.
$$\begin{aligned} 3x_1 + x_2 - x_3 &= 1 \\ -x_1 + 3x_2 + 2x_3 &= 1 \\ x_1 - x_2 - x_3 &= 1 \end{aligned}$$

Flow sheeting problems

We will be solving linear equation systems later in this book as part of other solutions. In the remainder of this chapter, however, we will illustrate how linear equations can arise when solving flowsheeting problems.

The flowsheet below illustrate the various components of a simple flowsheet. A flowsheet is composed of several unit operations (mixer, reactor, separator, and divider) linked together with streams that indicate the flow rates of different component (either distinct chemical species or distinct phases).

In a mathematical sense, the streams constitutes a set of variables. We will use moles/h as the standard unit for streams. The unit operations process the streams in some way and are - in a mathematical sense - relational equations. We normally use flow sheeting to determine steady state flows, i.e. when formulating mass balances over the unit operations we can ignore the accumulation term.

Mixer

A mixer takes two or more input streams and join them to one output stream. For each

component i, a mass balance over the mixer yields

IN	-	OUT	=	0
$F_{1i} + F_{2i}$	-	F_{3i}	=	0

the relational equations for the mixer. F_{kj} is the flowrate of component j in stream k.

Reactor

The reactor converts components of the incoming stream into other components. The degree of reaction is typically expressed in terms of how much of one of the reactant streams has been converted, also termed the extent of reaction. Let the reaction be

$$A + \upsilon_B B \rightarrow \upsilon_C C + \upsilon_D D$$

or

$$A + \upsilon_B B - \upsilon_C C - \upsilon_D D = 0$$

If we use A to specify the extent of reaction, we will define the extent for each component as

$e_i = \upsilon_i e_A$ (for reactants) or $e_i = -\upsilon_i e_A$ (for products)

Then the mass balance over the reactor for each component will be

IN	-	OUT	=	REACTED
F_{1i}	-	F_{2i}	=	$e_i F_{1A}$

or

$$F_{1i} - F_{2i} - e_i F_{1A} = 0$$

Notice. All reactions are specified relative to the flow of component A.

Separator

The separator separates components into different streams. It is common practise to define one of the outgoing streams as tops and the other as bottoms. Defined in this way, we can define a separation ratio as

$$S_i = F_{ti}/F_{1i}$$

or

$$S_i F_{1i} - F_{ti} = 0$$

A mass balance for each component yields

IN	-	OUT	=	REACTED
F_{1i}	-	$F_{ti} - F_{bi}$	=	0

These two sets of equations defines the separator.

Divider

The divider is like a separator with all separation ratios identical, i.e. the relative composition of all outgoing streams remains constant. If S is the separation ratio we have

$$SF_{1i} - F_{ti} = 0$$

$$(1-S)F_{1i} - F_{bi} = 0$$

Solving flowsheet problems

Consider a two component (A & B) flowsheeting problem with a mixer, a reactor, and a separator.

The typical problem is to determine some of the 10 flows (2 components x 5 streams) and 4 design parameters (e_A, e_B, S_A, and S_B). The relations specified by the unit operations can be written in tabular form with unit operations forming the rows and the flows forming the columns.

	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	RHS
Mixer	1	0	1	0	-1	0	0	0	0	0	0
	0	1	0	1	0	-1	0	0	0	0	0
Reactor	0	0	0	0	$1-e_A$	0	-1	0	0	0	0
	0	0	0	0	$-e_B$	1	0	-1	0	0	0
Separator	0	0	-1	0	0	0	1	0	-1	0	0
	0	0	0	-1	0	0	0	1	0	-1	0
	0	0	0	0	0	0	S_A	0	-1	0	0
	0	0	0	0	0	0	0	S_B	0	-1	0

So far this system has eight equations to find 10 unknown flows and 4 design parameters. Here, we will specify the four design parameters as $e_A = - e_B = 0.7$ (A→B) , $S_A = 0.2$, $S_B = 0.9$. Hereby, all internal elements in the table are defined. Typically either the actual

input flow or the desired output flows can also be specified. Here let us assume that the input flow 1.0 moles/h for A and 0.3 moles/h for B. Then we can add the following two rows to the table.

	1A	1B	2A	2B	3A	3B	4A	4B	5A	5B	RHS
Input	1	0	0	0	0	0	0	0	0	0	1.0
	0	1	0	0	0	0	0	0	0	0	0.3

The interior of the matrix can be used to form a 10-by-10 matrix, **A**, and the RHS column to form a vector **b**. Then the unknown vector of flows, x, can be determined as the solution to the linear equation **Ax** = **b**. It should be stressed, that rank deficiency is a common problem when specifying design parameters, i.e. sometimes some of the equations specified will be linearly dependent. Hence, a test for rank deficiency is always advisable. In the above case, however, the system has full rank.

Using MATLAB, we would find

```
¯ x = A\b
x =
   1.0000
   0.3000
   0.3158
   0.1357
   1.3158
   0.4357
   0.3947
   1.3567
   0.0789
   1.2211
```

Exercise 9.2

How would the flows change, if the reaction was

a. $A \rightarrow 2B$ instead of $A \rightarrow B$

b. $2A \rightarrow B$ instead of $A \rightarrow B$

e_A remaining constant.

Exercise 9.3

What should the input streams be if the output stream in the above flowsheet (stream 5) is 0.08 mole/h A and 1.00 mole/h B.

Exercise 9.4: gas scrubber

Consider the above flowsheet with two mixers and two separators. A stream of 10 kg/h of dry air containing 2% solvent is mixed with a recycle stream of solvent and water and fed to a separator. The top stream is released to the atmosphere and the bottom stream is processed in a second separator. The top stream from the second separator is recovered solvent which must contain 5% water. The bottom stream is mixed with pure makeup water before being recycled to mix with the incoming air stream.

Write a program that sets up the linear problem, test for rank deficiency, and computes the flows. What fraction of solvent is recovered from the air stream?

Exercise 9.5: Pasteboard production

A process for producing pasteboard from waste wood by removing some water and adding some glue is illustrated above. The flowsheet contains a divider, a separator, two mixers, and four components (wood, water, salt, and glue). A feed stream of indicated composition is split in the divider, one stream going to a separator (an evaporator in this case), the other to the final mixer. The evaporator will extract 2/3 of the water in the stream. The remainder enters a mixer where 0.5 kg of glue is added per kg of water. The final product must contain 3% glue.

Write a program that sets up the linear problem, test for rank deficiency, and computes the flows. What is the composition of the product?

Design parameters in flowsheets

In order to achieve a linear flowsheeting problem as above, we must generally

- fix all design parameters
- use only mass balances

In designing processes, however, we typically do not know the design parameters. Instead we might have some design objectives. For example, in the reaction example we may wish to convert 95% of component A in the feed stream and want to find out what extent, e_A, of the reaction is required to achieve this.

There are several ways of addressing this problem:

1. We could add an extra equation to the system

   $0.05F_{1A} - F_{5A} = 0$

   and treat e_A as an unknown variable. In this case, we would end up with 11 unknowns and 11 equations some of which are non-linear (where e_A or e_B is used in the A matrix). In the next chapter, we will discuss how to solve non-linear vector equations.

2. We could also treat the problem as a non-linear scalar equation

   $F(e_A) = 0.05F_{1A} - F_{5A} = 0$

   In this case the existing linear set of equations would be solved as an intermediate calculation in order to determine F_{1A} and F_{5A}. The non-linear equation could be solved using **fzero**.

3. Finally, we could solve the problem as an optimisation problem with the objective of minimising the function

   $F(e_A) = |0.05F_{1A} - F_{5A}|$ (Notice. $F(e_A) \geq 0$)

   Again the existing linear set of equations would be solved as an intermediate calculation in order to determine F_{1A} and F_{5A}. The minimisation problem could be solved using **fmin**.

Exercise 9.6

Solve the problem using both **fzero** and **fmin**.

It is rare that we have a hard objective of for example 95% conversion. More typically we have an objective of at least 95% conversion combined with some cost function indicating the price of various options. Hence, in general it makes more sense to treat the design problem as a minimisation problem. General constrained optimisation, however, is beyond the scope of this subject.

We will, however, consider a simple multivariable optimisation example. In this example, we wish to manipulate the two design parameters, e_A and S_B, to satisfy two hard objectives

1. the conversion of A is 85%

2. the concentration of B in the feed to the reactor is 35%

Multiple criteria can be handled by adding together the criteria, so meet both criteria by minimising the function

$$F(e_A,S_B) = |0.15F_{1A} - F_{5A}|+|F_{3B}/(F_{3A}+F_{3B}) - 0.35|$$

Exercise 9.7

Given that there are two manipulated variables, the **fmin** function can not be used to solve the above problem. Use the on-line help to check up on the multivariable unconstrained minimiser **fmins**. Use **fmins** in its simples form, **fmins**('flow3',startvector), to solve the problem.

Summary

In this chapter, we have studied the solution to the linear algebraic equation system

$$\mathbf{Ax=b}$$

when A is a square matrix. A unique solution can be found to this system using gaussian elimination and back substitution, if the matrix A has full rank. In MATLAB, the rank of a matrix can be computed using the **rank** command and gaussian elimination/back-substitution is performed using the backslash operator, so

```
x = A\b
```

computes the solution to the linear system. Similarly, the front slash operator

```
x = b/A
```

will compute the solution to

$$\mathbf{xA = b}$$

Analytically, a linear equation either has or has not a unique solution. Numerically, solving linear systems where the set of equations are close to showing linear dependency is very erroneous. We term such systems ill-conditioned and use the condition number (**cond** in MATLAB) or the inverse condition number (**rcond** in MATLAB) as a measure of ill-conditioning. Ill-conditioned full rank problems, under-determined, and over-determined problems can be solved in a least square sense using singular value decomposition (**svd** in MATLAB).

Flowsheeting problems were presented as problems that sometimes can be formulated a linear algebraic equation systems. In order for the resulting equation system to become linear, however, it was necessary to assume the design parameters fixed. When designing processes we typically use flowsheeting to determine these parameters based on some objectives for the process. Problems with "hard objectives" such as flow specification could be solved using either a non-linear equation solver or a minimisation routine. The two minimisation routines, **fmin** and **fmins**, solve single variable and multi variable optimisation problems, respectively.

CHAPTER 10

Non-linear equation systems

Most practical problems are of non-linear nature. In Chapter 8, you learned how to solve non-linear scalar equations. In this chapter, you will learn how to solve systems of non-linear equations. A set of non-linear equations

$$f_1(x_1, x_2, \ldots, x_n) = 0$$
$$f_2(x_1, x_2, \ldots, x_n) = 0$$
$$\ldots$$
$$f_n(x_1, x_2, \ldots, x_n) = 0$$

is written in vector form as

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \ldots \\ f_n(x_1, x_2, \ldots, x_n) \end{pmatrix} = \mathbf{0}$$

The Newton method presented in this chapter is based on repeatedly linearising the non-linear function and solving a linear system of equations as outlined in Chapter 9. Thus, we will start with a discussion on how we linearise an vector function.

Linearising a vector function

You should be familiar with the Taylor series expansion of a scalar function around the point x_0, looking something like

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

A linear approximation to the scalar function includes only the first differential term in the Taylor series

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) = (f(x_0) - f'(x_0)x_0) + f'(x_0)x = a + bx$$

Vector functions can be linearised in a similar way, yielding

$$\mathbf{f}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + \mathbf{J}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$$

where **J** is the so-called Jacobian of **f**. The Jacobian is a matrix with the same number of rows as **f** and with the same number columns as there are rows in **x** (**x** is a column vector). Each element (i,j) is calculated as

$$J_{i,j} = \frac{\partial F_i}{\partial x_j}$$

i.e. the i'th function differentiated with respect to the j'th variable. To evaluate the Jacobian in a given point, $\mathbf{x}_0$, you simple insert the values of x, in the analytical Jacobian. For example, if

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{pmatrix} = \begin{pmatrix} x_1 + x_2 - 1 \\ x_1 x_2 - 0.25 \end{pmatrix}$$

then the Jacobian is

$$J_{1,1} = \frac{\partial f_1}{\partial x_1} = 1; \quad J_{1,2} = \frac{\partial f_1}{\partial x_2} = 1; \quad J_{2,1} = \frac{\partial f_2}{\partial x_1} = x_2; \quad J_{2,2} = \frac{\partial f_2}{\partial x_2} = x_1$$

$$\mathbf{J} = \begin{pmatrix} 1 & 1 \\ x_2 & x_1 \end{pmatrix}$$

In particular, the Jacobian in the point $\mathbf{x}_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ is

$$\mathbf{J}(\mathbf{x}_0) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

So a linear approximation to $\mathbf{f}(\mathbf{x})$ in the point $\mathbf{x}_0$ is

$$\mathbf{f}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + \mathbf{J}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) = \begin{pmatrix} 0 \\ -0.25 \end{pmatrix} + \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \left(\mathbf{x} - \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right)$$

Linearisation of vector functions is a very common technique and you will be using it several times later in this book as well as in other engineering courses. Use the following exercise to make certain that you have understood the principle.

Exercise 10.1

Linearise the following functions around the point $\mathbf{x}_0 = (1\ 2)^T$ or $\mathbf{x}_0 = (1\ 2\ 3)^T$

1. $\mathbf{f}(\mathbf{x}) = \begin{pmatrix} 2x_1^2 + 3x_2^2 - 50 \\ 2x_1^2 - x_2 - 9 \end{pmatrix}$

2. $\mathbf{f}(\mathbf{x}) = \begin{pmatrix} 5(x_2 - x_1^2)^4 \\ (1 - x_1)^2 \end{pmatrix}$

3. $\mathbf{f}(\mathbf{x}) = \begin{pmatrix} x_1^3 - e^{x_2} + \sin x_3 - 3.5 \\ x_1^2 x_3 + (x_2^2 - x_3)^2 - 4 \\ x_1 x_2 x_3 - x_3 + x_1 x_2 - 5 \end{pmatrix}$

Numerical differentiation

Computing $\mathbf{J}(\mathbf{x})$ may represent somewhat of a problem. For small and/or simple equation systems, it may be possible to write out the analytical Jacobian of $\mathbf{f}(\mathbf{x})$ as done above. For even moderately sized problems, however, it may be next to impossible (e.g. for n=10, the

full Jacobian contains 100 elements). In this case, we will need to compute the Jacobian numerically, i.e. we will perform a numerical differentiation.

As an aside, it should be mentioned that large scale analytical differentiation can be performed using symbolic math packages such as MAPLE or MATHEMATICA. It is rarely worthwhile, however, to go this effort and using the analytical Jacobian can often be slower and even less accurate than using a numerical Jacobian.

Numerical differentiation can be done by finite differencing, i.e. we make a small perturbation to one of the x elements (keeping the other constant), evaluate the function, and use the difference in function value over the perturbation as a measure of the differential with respect to this x element. If h_j is the size of the j'th perturbation and $\mathbf{e}_j$ is the unit vector with one in the j'th element and zero elsewhere, then the Jacobian elements in the point $\mathbf{x}_0$ are calculated as

$$\mathbf{J}_{i,j} \approx \mathbf{D}_{i,j} = \frac{f_i(\mathbf{x}_0 + h_j\mathbf{e}_j) - f_i(\mathbf{x}_0)}{h_j}, \quad i, j = 1, \ldots, n$$

If the function returns a column vector, the Jacobian can be computed columnwise as

$$\mathbf{J}_{:,j} \approx \mathbf{D}_{:,j} = \frac{\mathbf{f}(\mathbf{x}_0 + h_j\mathbf{e}_j) - \mathbf{f}(\mathbf{x}_0)}{h_j}, \quad j = 1, \ldots, n$$

To illustrate how we generate a numerical estimate of the Jacobian consider finding the Jacobian of the vector equation

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} x_1^2 - 2x_2 \\ 3x_1x_2 + x_2^3 \end{pmatrix}$$

in the point $\mathbf{x}_0 = (1\ 1)^T$. The function value in this point is

$$\mathbf{f}(\mathbf{x}_0) = \begin{pmatrix} -1 \\ 4 \end{pmatrix}$$

To estimate the Jacobian we will perturb both elements of $\mathbf{x}$ with h=0.01. For the first element we find

$$\mathbf{x}_h = \mathbf{x}_0 + h_1\mathbf{e}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} + 0.01\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1.01 \\ 1.00 \end{pmatrix} \quad \mathbf{f}(\mathbf{x}_h) = \begin{pmatrix} -0.9799 \\ 4.03 \end{pmatrix}$$

$$\mathbf{J}_{:,1}(\mathbf{x}_0) \approx \mathbf{D}_{:,1} = \frac{\mathbf{f}(\mathbf{x}_h) - \mathbf{f}(\mathbf{x}_0)}{h_j} = \frac{\begin{pmatrix} -0.9799 \\ 4.03 \end{pmatrix} - \begin{pmatrix} -1 \\ 4 \end{pmatrix}}{0.01} = \begin{pmatrix} 2.01 \\ 3.00 \end{pmatrix}$$

For the second element of $\mathbf{x}$, we find

$$\mathbf{x}_h = \mathbf{x}_0 + h_2\mathbf{e}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} + 0.01\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1.00 \\ 1.01 \end{pmatrix} \quad \mathbf{f}(\mathbf{x}_h) = \begin{pmatrix} -1.02 \\ 4.06 \end{pmatrix}$$

$$\mathbf{J}_{:,2}(\mathbf{x}_0) \approx \mathbf{D}_{:,2} = \frac{\mathbf{f}(\mathbf{x}_h) - \mathbf{f}(\mathbf{x}_0)}{h_2} = \frac{\begin{pmatrix} -1.02 \\ 4.06 \end{pmatrix} - \begin{pmatrix} -1 \\ 4 \end{pmatrix}}{0.01} = \begin{pmatrix} -2.0 \\ 6.00 \end{pmatrix}$$

Thus, the estimated Jacobian of **f** in $\mathbf{x}_0$ is

$$\mathbf{J}(\mathbf{x}_0) \approx \mathbf{D} = \begin{pmatrix} 2.01 & -2.0 \\ 3.0 & 6.0 \end{pmatrix}$$

Size of perturbation. The size of the perturbation is obviously important for the accuracy of the approximation. The choice of perturbation is a trade off between making the perturbation as small as possible to improve the theoretical accuracy, while at the same time not making it so small that the accuracy of the computer becomes limiting. A good choice is to make the perturbation

$$h_j = (1 + \text{abs}(x_j)) \cdot 10^{-7}$$

where the "1" ensures that the value does not get too close to zero.

Numerical differentiation algorithm

In MATLAB, a general algorithm for performing a numerical differentiation would look something like

```
%
%       This algorithm performs a numerical differentiation.
%       The name of the vector function to be evaluated is assumed to be specified
%       in the string  variable fun_str.
%       The function is here assumed to take one column vector argument only
%       and to return a column vector.
%
x0 = ... ;          % Column vector point in which to evaluate Jacobian
dim = length(x0); % Determine size of vector
jac = zeros(dim,dim)
h = (1+abs(x0))*1e-7; % Set size of perturbation for each variable
f0 = feval(fun_str,x0);
for jcol=1:dim
        xh = x0; % Set all elements to x0 values
        xh(jcol) = x0(jcol)+h(jcol); % Perturb jcol element
        fh = feval(fun_str,xh); % Calculate function for perturbed value
        jac(:,jcol) = (fh-f0)/h(jcol); % Form column of Jacobian by finite difference
end
```

Exercise 10.2: A generic function differentiator

Using the above algorithm, write a function function, **numjac**, that given the name of a function and a vector will return the Jacobian of the function in this point. Use the following function for your hand example

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{pmatrix} = \begin{pmatrix} x_1 + x_2 - 1 \\ x_1 x_2 - 0.25 \end{pmatrix}$$

When the program is finished, test the algorithm on the functions in exercise 10.1 and see if you get the same result as then.

Newton's method for vector equations

Newton's method follow the general iterative algorithm presented for scalar equations, i.e.

1. guess an initial value of x
2. evaluate **f**(**x**)
3. decide if **f**(**x**)=0 and stop with the solution if this is the case
4. otherwise update the estimate of x and return to 2.

In the vector case, guessing an appropriate initial value can be very difficult as it may be difficult to visualise the function using plots. In addition to using our insight into the problem, we frequently have to commence the iteration from a set of different points to see if there are more than one acceptable solution.

In the scalar case, we used the absolute value as a measure of how close f(x) was to zero. In the vector case, we will typically use the **norm.** The norm of a matrix is a scalar that gives some measure of the magnitude of the elements of the matrix. Here, we will only consider the 2-norm of a vector (For further details on other vector norms or the norm of a matrix, refer to the on-line help.) The 2-norm defines the Euclidean length of the vector, i.e.

$$norm(\mathbf{v}) = \sqrt{v_1^2 + v_2^2 + v_3^2 + \cdots}$$

By insisting that the norm of **f**(**x**) should be smaller than some small tolerance value, we ensure that all the elements in **f**(**x**) are numerically smaller than the tolerance.

In order to update the estimate, Newton's method linearises the function around the last estimate, i.e. if $\mathbf{x}_k$ is the last estimate we have

$$\mathbf{f}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_k) + \mathbf{J}(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k) = \mathbf{P}(\mathbf{x})$$

We then choose the new estimate, $\mathbf{x}_{k+1}$, so that $\mathbf{P}(\mathbf{x}_{k+1})=\mathbf{0}$, i.e.

$$\mathbf{f}(\mathbf{x}_k) + \mathbf{J}(\mathbf{x}_k)(\mathbf{x}_{k+1} - \mathbf{x}_k) = \mathbf{0}$$

or when introducing $\delta = \mathbf{x}_{k+1}-\mathbf{x}_k$ and rearranging

$$\mathbf{J}(\mathbf{x}_k)\delta = -\mathbf{f}(\mathbf{x}_k)$$

This is obviously a linear equation with δ the unknown "**x**", $\mathbf{J}(\mathbf{x}_k)$ the known **A**, and $-\mathbf{f}(\mathbf{x}_k)$ the known **b**, when comparing to last chapter. In MATLAB, this equation is solved as

δ = **-J\f**

and we find $\mathbf{x}_{k+1}$ as

$\mathbf{x}_{k+1} = \mathbf{x}_k+\delta$

The figure below illustrates the Newton algorithm applied to a scalar problem. In each step, the tangent taken in the present estimate is extrapolated until it reaches the x-axis. The intercept with the x-axis is taken as the new estimate.

The Newton algorithm has a second order rate of convergence and requires less iterations than the algorithms presented in Chapter 8. On the other hand, the Newton algorithm requires many additional function calls in order to obtain both the function value and the Jacobian.

The general form of the Newton algorithm is listed below

```
% NEWTON. This is the general form of the Newton algorithm
%
% Newton's method solves the vector equation of the form
%                   f(x)=0
%
Nmax = ...; % Set limit on maximum number of iterations
tol = ... % Specify a tolerance for the norm
x =x0 ... % assign x the initial value in column vector
for iterations=1:Nmax
          f = ... % compute f(x)
          if norm(f)<tol
                    break
          end
          J=  ... % compute J(x), e.g. using the numerical differentiation algorithm
          x = x -J\f;
end
if (iterations==Nmax)
          error('Maximum iterations reached')
end
disp(['Solution found in ',int2str(iterations),' iterations'])
disp(x)
```

Exercise 10.3: Newton's method algorithm for test example

Develop a program that uses the Newton algorithm to solve the following equation

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} F_1(x_1, x_2) \\ F_2(x_1, x_2) \end{pmatrix} = \begin{pmatrix} x_1 + x_2 - 1 \\ x_1 x_2 - 0.25 \end{pmatrix} = \mathbf{0}$$

starting at $\mathbf{x} = (1\ 0)^T$.

Exercise 10.4: Generic Newton's method algorithm

Modify the above code to form a generic Newton algorithm, i.e. a function function that given a function name, a starting point, a tolerance, and a maximum number of iterations will compute a solution to **f**=**0**. Use this program to solve the following equations

For each function, make a break down analysis of where the computation time is consumed, i.e. measure the CPU time or flops used for calculating the function, calculating the Jacobian, solving the linear equation system.

Summary

This chapter has presented Newton's method for solving non-linear equation systems. In Newton's method, the function is repeatedly linearised using the vector form of the Taylor approximation. In order to perform the linearisation, it is necessary to know the Jacobian of the vector equation (equivalent to the slope in the scalar case). The Jacobian is frequently too large to compute analytically and a numerical method for differentiation was presented.

Exercise 10.5: Non-linear flowsheet problem

In last chapter, it was mentioned that when design parameters were chosen based on "hard objectives", one way of solving the flowsheeting problem was to treat the whole equation system as a non-linear system. Solve exercise 9.6 and 9.7 again, this time using your Newton algorithm.

Exercise 10.6: CSTR 1

Consider the following hypothetical reaction scheme for a liquid phase system:

$$A \xrightarrow{r_1} 2B$$

$$A \underset{r_3}{\overset{r_2}{\rightleftharpoons}} C$$

$$B \xrightarrow{r_4} D + C$$

where

$r_1 = k_1 C_A$	[gmolesú litre^{-1}ú sec^{-1}]	$k_1 = 1.00$ sec^{-1}
$r_2 = k_2 (C_A)^{1.5}$	[gmolesú litre^{-1}ú sec^{-1}]	$k_2 = 0.20$ litre$^{0.5}$úsec^{-1}úgmoles$^{-0.5}$
$r_3 = k_3 (C_C)^2$	[gmolesú litre^{-1}ú sec^{-1}]	$k_3 = 0.05$ litreúsec^{-1}úgmoles^{-1}

$r_4 = k_4(C_B)^2$ [gmolesú litre^{-1}ú sec^{-1}] $k_4 = 0.04$ litreúsec^{-1}úgmoles^{-1}

A continuous stirred tank reactor (CSTR) is used for this reaction system.

The reaction volume, V, is 100 litres and the volumetric feed rate to the reactor, F, is 50 litres per second at a concentration, C_{Ai}, of 1.0 mole/litre of component A.

Since a CSTR is designed to operate at steady state and this system is assumed to be operated under isothermal conditions, steady state mole balances define the performance of this system (i.e., concentrations coming out of the reactor). The following mole balances can be constructed:

	IN	-	OUT	+	GENERATED	= 0
Component A	$C_{Ai}F$	-	C_AF	+	$V(r_3-r_1-r_2)$	= 0
Component B	0	-	C_BF	+	$V(2r_1-r_4)$	= 0
Component C	0	-	C_CF	+	$V(r_2+r_4-r_3)$	= 0
Component D	0	-	C_DF	=	Vr_4	= 0

Substituting in the rate expressions and introducing the dilution rate, D=F/V (inverse residence time), results in the following set of non-linear equations:

$$F1 = D(C_{Ai}-C_A) + (k_3(C_C)^2-k_1C_A-k_2(C_A)^{1.5}) = 0$$

$$F2 = D(0-C_B) + (2k_1C_A-k_4(C_B)^2) = 0$$

$$F3 = D(0-C_C) + (k_4(C_B)^2+k_2(C_A)^{1.5}-k_3(C_C)^2) = 0$$

$$F4 = D(0-C_D) + k_4(C_B)^2 = 0$$

Solve these equations.

Exercise 10.7: CSTR 2

Consider a chemical reaction system involving three species, A, B, and C, in a CSTR. The concentrations of the species are C_A, C_B, and C_C, respectively. The three species reacts according to three reactions with known kinetics

$A \rightarrow B$	$r_1 = k_1 C_A$	k_1=0.04 sec^{-1}
$B \rightarrow C$	$r_3 = k_3 C_B^2$	k_3=3x10^7 literúsec^{-1}úgmoles^{-1}
$B \rightarrow A$	$r_2 = k_2 C_B C_C$	k_2=10^4 literúsec^{-1}úgmoles^{-1}

i.e. A will react to form B in a relatively slow first order reaction, B will react to form C in a very fast second order reaction, and B can react back to A in presence of C.

The reaction volume, V, is 1000 litres and the volumetric feed rate to the reactor, F, is 1 litres per second at a concentration, C_{Ai}, of 1.0 mole/litre of component A.

Find the steady state concentrations of A,B, and C.

Exercise 10.8: Catalyst pellet problem

Many advanced numerical techniques convert complex problems (such as solving partial differential equations) into a problem of solving a set of non-linear equations. In this example, orthogonal collocation was used to convert a boundary value problem describing the steady state concentration profile of a chemical species in a catalytic pellet into the following three non-linear equations

$$-13.59530877x_1 + 20.42831009x_2 - 6.833001321x_3 = \phi^2 f(x_1)$$
$$14.57168991x_1 - 91.404669119x_2 + 76.83300129x_3 = \phi^2 f(x_2)$$
$$0.9482702526x_1 - 14.948270256x_2 + 14x_3 = Bi_m(1 - x_3)$$

where x_1, x_2 represents the concentration in two points inside the pellet and x_3 represents the concentration on the surface. ϕ is the Thiele modulus, Bi_m is the Biot number for mass, and f() is an expression for the reaction. We will consider two cases

Case 1: $f(x) = x^2$, ϕ = 1, Bi_m= 100

Case 2: $f(x) = \dfrac{x}{(1+\alpha x)^2}$, α=20, ϕ = 32, Bi_m= 100

Solve the equation system for both cases [Be warned that case 2 has three solutions. Are you able to find all three?].

CHAPTER 11

Ordinary differential equations - The initial value problem

An introductory example

In exercise 10.6 we considered a hypothetical reaction scheme involving four components - A, B, C, and D - and found the steady state concentrations of these. In order to describe the dynamics of the system, we will redo the mass balances this time taking accumulation into account. Because the system is dynamic and the behaviour of the system will change over time, it is necessary to consider a mass balance over only a short period of time, from t to $t+\Delta t$. The accumulation term is the change in total mole content over this period and we will write this

$$\Delta(C_i V) = C_i(t+\Delta t)V(t+\Delta t) - C_i(t)V(t)$$

In absolute molar amounts, the mass balance now becomes

Component	IN	-	OUT	+	GENERATED	=	ACCUMULATED
A	$C_{Ai}F\Delta t$	-	$C_A F\Delta t$	+	$V(r_3-r_1-r_2)\Delta t$	=	$\Delta(C_A V)$
B	0	-	$C_B F\Delta t$	+	$V(2r_1-r_4)\Delta t$	=	$\Delta(C_B V)$
C	0	-	$C_C F\Delta t$	+	$V(r_2+r_4-r_3)\Delta t$	=	$\Delta(C_C V)$
D	0	-	$C_D F\Delta t$	+	$Vr_4\Delta t$	=	$\Delta(C_D V)$

These equations can be rearranged to

$$\frac{\Delta(C_A V)}{\Delta t} = F(C_{Ai} - C_A) + V(r_3 - r_1 - r_2)$$

$$\frac{\Delta(C_B V)}{\Delta t} = F(0 - C_B) + V(2r_1 - r_4)$$

$$\frac{\Delta(C_C V)}{\Delta t} = F(0 - C_C) + V(r_2 + r_4 - r_3)$$

$$\frac{\Delta(C_D V)}{\Delta t} = F(0 - C_D) + Vr_4$$

We will now make Δt infinitely small, by forming the limit as $\Delta t \to 0$. For the left hand sides, we find

$$\lim_{\Delta t \to 0} \frac{\Delta(C_i V)}{\Delta t} = \frac{d}{dt}(C_i V) = V\frac{dC_i}{dt} + C_i\frac{dV}{dt} = V\frac{dC_i}{dt}$$

that is the delta-changes has been replaced with a differential, then we have differentiated by parts, and finally used that dV/dt =0 as as the volume is assumed constant in the CSTR. The right hand sides do not contain terms with Δt, thus they remain unaltered when the limit is taken. Dividing with V on both sides and introducing D = F/V, this finally yield

$$\frac{dC_A}{dt} = D(C_{Ai} - C_A) + (r_3 - r_1 - r_2)$$

$$\frac{dC_B}{dt} = D(0 - C_B) + (2r_1 - r_4)$$

$$\frac{dC_C}{dt} = D(0 - C_C) + (r_2 + r_4 - r_3)$$

$$\frac{dC_D}{dt} = D(0 - C_D) + r_4$$

which are the set of four ordinary differential equations that describes the dynamics of the CSTR. Notice, that the steady state solution can be derived from the dynamic model by setting the differential terms to zero. This reduces the ODEs to the four algebraic equations we solved in Exercise 10.6.

Exercise 11.1

Develop the dynamic model for the CSTR described in Exercise 10.7.

We could use the two dynamic models developed above, for example, to investigate the behavior of the two CSTRs as they are started up and before they reach steady state. In both cases, we may quickly have filled the reactor with the feed stream (i.e. pure A), so at time zero we would have pure A in the reactor. In an alternative situation, we may be interested in predicting the transient behavior of the CSTR when we shift from one feed concentration or feed rate to another (and thus changes the steady state solution).

The type of solution we are looking for is a concentration profile for each component over time. If the initial value problem is solved analytically, the result is four (or three for CSTR 2) explicit functions in time, i.e.

$$C_A = f_1(t);\ C_B = f_2(t);\ C_C = f_3(t);\ C_D = f_4(t)$$

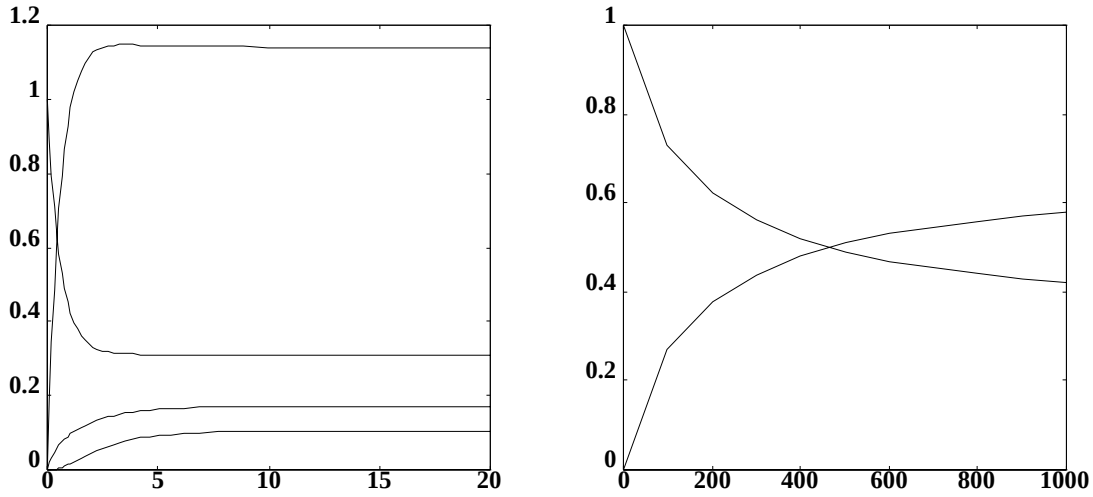
These four functions can be plotted versus t forming the concentration profiles for the batch reaction. When solved numerically, the result is a series of discrete data points

$$(t_1, C_{A1}, C_{B1}, \ldots), (t_2, C_{A2}, C_{B2}, \ldots), \ldots, (t_n, C_{An}, C_{Bn}, \ldots)$$

each composed of a time point and the corresponding estimates of the y-values.

The figures below show plots of the dynamic behavior of the two CSTRs discussed above. Notice, how in CSTR 1 the steady state is reached in as little as 5 seconds, while in CSTR 2 the system still hasn't reached steady state after 1000 seconds (NB: the concentration of component B is too small to be seen in the CSTR 2 plot).

Conc., mM CSTR 1 Conc., mM CSTR 2



Time, seconds Time, seconds

The initial value problem

A general formulation of the inital value problem is to find the explicit vector function

$$\mathbf{x} = \mathbf{f}(t), \quad t \in [a,b]$$

which forms a solution to the set of ordinary differential equations

$$\mathbf{x}' = \mathbf{g}(t,\mathbf{x}), \quad t \in [a,b]$$

with initial value

$$\mathbf{x}(a) = \mathbf{x}_0$$

Notice, that the right hand side of **g** in general contains an expression in both t and **x**.

Built-in ODE solvers: ode23 and ode45

MATLAB contains two functions for computing numerical solutions to the initial value problem - **ode23** and **ode45**. **ode23** uses second and third order Runge-Kutta integration equations, while **ode45** uses fourth and fifth order Runge-Kutta integration equations and is thus more accurate (we discuss Runge-Kutta method in detail later). The two functions are used in an identical manner and we shall only consider **ode45** here.

In order to make a generic ode solver that can solve any set of ODEs, **ode45** is implemented as function function, i.e. it takes the name of a function as an argument. It need at least three more arguments: the starting point of integration, the end point of integration, and vector containing the initial values of the x's. A further two arguments are optional: the accuracy required and a trace value to make the function display intermediate results (see on-line help for details).

ode45 returns a row vector, t, containing the time points where an estimate has been established and a matrix, **x**, in which each column contains the corresponding estimates for one of the **x** values.

Example 11.1

The function required by **ode45** is the right hand side of the ODE, i.e. the **g**(t,**x**) function. So to solve the IVP

$x' = 3t^2 \qquad x(2) = 0.5$

in the interval t=[2,4], we would first create a function

```
function g=ex11_1(t,x)
g=3*t^2;
```

Notice, in this case x is not part of the g-function. We still have to include x, because ode45 will send the value across (it doesn't know the nature of the function).

Then solve the the IVP using

```
[t,x_num]=ode45('ex11_1',2,4,0.5);
x_ana = t.^3 -7.5; % Calculate analytical solution for illustration purposes
plot(t,x_num,t,x_ana,'o'), ...
title('Solution to Example 11.1'),...
xlabel('t'),ylabel('x=f(t)'),grid
```

This resulting figure shows, as expected, that the true values ('o') lie on the estimated function line. Notice, how the distance between points is not constant. **ode45** makes use of automatic step control, in which each step is chosen to ensure that the required accuracy is maintained. Thus, where possible longer steps will be made to reduce the time required to compute the results.

Exercise 11.2

Solve the initial value problem

$x' = g(t,x) = -x \qquad x(0) = -3$

over the interval t = [0,2]. Compare the result to the analytical solution

$x = -3e^{-t}$

Exercise 11.3

Solve the initial value problem

$$x' = g(t,x) = \frac{-t - e^t}{3x^2} \quad x(0) = 3$$

over the interval [0,2]. Compare the result to the analytical solution

$$x = \sqrt[3]{28 - 0.5t^2 - e^t}$$

Exercise 11.4: CSTR 1 startup

Solve the set of ordinary differential equations for the CSTR 1 for the time period 0 to 20 seconds assuming the reactor is initially filled with 1 moles/litre component A.

Exercise 11.5: CSTR 1 feed change

Simulate the change in concentrations if the feed concentration is lowered from 1 moles/litre to 0.5 moles/litre.

Exercise 11.6: CSTR 2 - Limitations

ode45 is a very accurate and relatively fast IVP solver when it works. Unfortunately, there are many cases where it does not work.

Consider solving the CSTR 2 problem. Initially, use **ode45** to solve the problem for the small time interval, t=[0,0.2], assuming the reactor is initially filled with 1 moles/litre component A.

How many steps were required to solve the IVP for this time interval? Assuming the problem showed constant behaviour, how many steps would be required to solve the IVP for a time interval of [0,1000]?

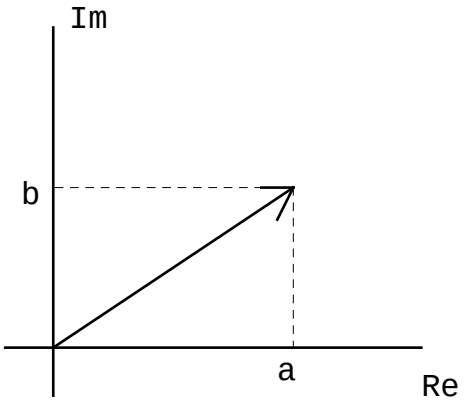
In this case, the problem actually becomes more difficult to solve and a smaller stepsize is needed as we move away from t=0. The reason for the difficulties is that this particular IVP is a so-called stiff problem. Stiff problems are common in process engineering and this chapter is dedicated to a discussion this and other possible characteristic of IVPs. In the next chapter you will to learn about a large family of IVP solver algorithms - the Runge-Kutta family - of which the **ode23** and **ode45** are just two members. Fortunately, other members of this family are much more suited to solve the chemical reaction system problem. In fact, a reasonable solution to the problem can be found using as little as 10 steps!!!

The theory of IVPs will be developed gradually and relevant functions in MATLAB will be discussed as we go along. The first two sections consider two mathematical tools required later: complex number theory and eigenvalue decomposition. These tools are then used to find the analytical solution of linear IVPs which is used to discuss the stability of IVPs.

A general discussion of numerical method for solving IVPs follows. In this context, linear IVPs are important to develop the concepts of stability and accuracy of numerical methods. Finally, in the next chapter we will then return to the general problem of solving non-linear ODEs using Runge-Kutta methods.

Complex numbers

Complex numbers are most easily understood, if interpreted as vectors in an orientated plane, where the real component of the number is plotted along the x-axis and the imaginary part of the number is plotted along the y-axis.



Complex numbers - just as vectors - can be expressed in either cartesian or polar coordinates. In cartesian coordinates, the shown complex number would be written as z=(a,b) or using the standard complex number notation:

$$z = a+ib$$

where i is the imaginary unit with the special property, that

$i^2 = -1$

In polar coordinates, the vector is expressed in terms of its length, r, and its angle with the real axis, θ, i.e. z= (r,θ) or using the common notation

$$z= r(\cos\theta + i \sin\theta) = r\, e^{i\theta}$$

The last form may seem a bit odd, but reflects the way a complex number behaves in multiplications and division as will be shown below. The length of the vector, r, is often called the modulus and θ the amplitude of the complex number, z.

From above, it is easily seen that the relationship between cartesian and polar coordinates is

$$a=r\cos\theta \; ; \; b = r\sin\theta \text{ and } r = \sqrt{a^2 + b^2}$$

The value of having both formats is that addition and subtraction is most readily performed in cartesian coordinates, while multiplication and division is most readily performed in polar coordinates. In the following, let

$$z_1 = a_1+ib_1=r_1e^{i\theta_1} \qquad z_2 = a_2+ib_2=r_2e^{i\theta_2}$$

Addition and subtraction (cartesian coordinates)

$$z = z_1 + z_2 =(a_1+ib_1) + (a_2+ib_2) = (a_1+a_2) + i(b_1+b_2)$$

$$z = z_1 - z_2 =(a_1+ib_1) - (a_2+ib_2) = (a_1-a_2) + i(b_1-b_2)$$

Notice! The imaginary unit, i, is just treated as a constant.

Multiplication (cartesian coordinates)

$$z= z_1z_2 = (a_1+ib_1)(a_2+ib_2) = a_1a_2 + a_1ib_2 + ib_1a_2 + ib_1ib_2$$
$$= (a_1a_2 - b_1b_2) + i(a_1b_2 + b_1a_2)$$

Notice! Values are multiplied together with i treated as a constant. The special property of the imaginary unit is used to replace $i*i = i^2 = -1$.

Division (cartesian coordinates)

$$\frac{z_1}{z_2} = \frac{a_1 + ib_1}{a_2 + ib_2} = \frac{(a_1 + ib_1)(a_2 - ib_2)}{(a_2 + ib_2)(a_2 - ib_2)} = \frac{(a_1 + ib_1)(a_2 - ib_2)}{a_2^2 - (ib_2)^2}$$
$$= \frac{a_1a_2 - a_1ib_2 + ib_1a_2 - ib_1ib_2}{a_2^2 + b_2^2} = \frac{(a_1a_2 + b_1b_2) + i(b_1a_2 - a_1b_2)}{a_2^2 + b_2^2}$$
$$= \frac{a_1a_2 + b_1b_2}{a_2^2 + b_2^2} + i\frac{b_1a_2 - a_1b_2}{a_2^2 + b_2^2}$$

Notice! The first step consists of multiplying in nominator and denominator with the so-called complex conjugate of the denominator (a_2 - ib_2). Then the denominator is calculated as *the product of the sum and the difference of two elements, is the square of the first element minus the square of the second element*. The imaginary unit, i, is then removed from the denominator using $i^2 = -1$ leaving a real number only. The remainder is standard multiplications and rearrangement.

Multiplication and division (polar coordinates)

$$z_1 z_2 = r_1 e^{i\theta_1} r_2 e^{i\theta_2} = r_1 r_2 e^{i(\theta_1+\theta_2)}$$

$$\frac{z_1}{z_2} = \frac{r_1 e^{i\theta_1}}{r_2 e^{i\theta_2}} = \frac{r_1}{r_2} e^{i(\theta_1-\theta_2)}$$

Notice! The imaginary unit, i, is again treated as a constant, and we use standard rules for multiplying and dividing the exponential terms. Addition and subtraction has no simple solution in polar coordinates.

Examples

For the purpose of studying stability later, we are interested in determing the modulus of three complex number functions, R(z), expressed in a complex variable, z= hλ. We define

$$z = h\lambda = a+ib$$

a. stability function for Euler's method

$$R(z) = 1+z = (1+a)+ib \Rightarrow |R(z)| = \sqrt{(1+a)^2+b^2}$$

Here, |R(z)| is just another way of writing the modulus of a complex variable.

b. stability function for Backward Euler's method

$$R(z) = \frac{1}{1-z} = \frac{1}{(1-a)-ib}$$

The denominator can be expressed in polar coordinates as $re^{i\theta}$, where r is found as

$$r = \sqrt{(1-a)^2+(-b)^2}$$

and we will leave the amplitude, θ, as unknown. Inserting the denominator into the equation we find

$$R(z) = \frac{1}{re^{i\theta}} = \frac{1}{r} e^{i(0-\theta)}$$

thus the modulus of the stability function is

$$|R(z)| = \frac{1}{r} = \frac{1}{\sqrt{(1-a)^2+b^2}}$$

the amplitude of R(z) apparently is -θ, but we won't need that.

c. stability function for Trapezoidal method

$$R(z) = \frac{1+0.5z}{1-0.5z} = \frac{z_1}{z_2} = \frac{r_1 e^{i\theta_1}}{r_2 e^{i\theta_2}}$$

The moduli for the nominator and denominator are

$$r_1 = \sqrt{(1+0.5a)^2 + (0.5b)^2} \quad r_2 = \sqrt{(1-0.5a)^2 + (-0.5b)^2}$$

and again we don't need to know the amplitudes, θ_1 and θ_2. Using the division formula, the modulus of the stability function is found as

$$|R(z)| = \frac{r_1}{r_2} = \frac{\sqrt{(1+0.5a)^2 + (0.5b)^2}}{\sqrt{(1-0.5a)^2 + (-0.5b)^2}} = \sqrt{\frac{(2+a)^2 + b^2}{(2-a)^2 + b^2}}$$

the amplitude of R(z) is θ_1-θ_2, but we won't need that.

Complex number in MATLAB

In MATLAB, complex numbers are defined in a most straightforward way. Unless overwritten by an assignment, the built-in variables i and j can be used to form complex numbers as

```
>> z = 3 +4*i;
>> z = 3+4*j;
```

Alternatively, polar coordinates can be used as in

```
>> z = r*exp(i*theta)
```

A matrix of complex elements can be defined using

```
>> Z=A+i*B
```

where A and be are real number matrices.

The arithmetic functions, +, -,*,and /, work according to the rules for complex numbers. The normal MATLAB transpose, ', is the generalised conjugated transpose, i.e. all elements are conjugated before being transposed.

MATLAB also has 5 functions specific to complex numbers

real(z), **imag(z).** Computes the real and imaginary portion of complex numbers.

conj(z). Computes the conjugate of a complex number.

abs(z), angle(z). Computes the modulus and the amplitude (between [-π,π]) of a complex number.

Eigenvalue decomposition

A square n-by-n matrix, A, has n eigenvalues, i.e. the values of λ for which the equation

$\mathbf{Ax}=\lambda\mathbf{x}$ (**x** is a column vector)

has non-trivial solutions. For a given λ, the non-trivial solution **x** is termed the corresponding eigenvector.

The eigenvalues can be found as the roots in the characteristic polynomium of A. The characteristic polynomium of a square matrix is

$P=\det(A-\lambda I)$

and the roots are the solution to

$P=0$

The eigenvalues to the matrix $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, for example, are found as

$$\det(A - \lambda I) = \begin{vmatrix} 1-\lambda & 2 \\ 3 & 4-\lambda \end{vmatrix} = (1-\lambda)(4-\lambda) - 3 \cdot 2 = \lambda^2 - 5\lambda - 2 = 0$$

$$\lambda_1 = \frac{5+\sqrt{33}}{2} \quad \lambda_2 = \frac{5-\sqrt{33}}{2}$$

Remember from solving the general n'th order equation, that some of the eigenvalues may be multiple roots in the polynomial and some may form complex conjugated pairs.

From the linear algebra, we further know that any square matrix A can be decomposed as

$$\mathbf{A}=\mathbf{V}\mathbf{D}\mathbf{V}^T$$

where **D** is a diagonal matrix containing the eigenvalues of **A** and **V** is an orthogonal matrix containing the orthonormal eigenvectors to **A**. Since V contains orthonormal eigenvectors, we have

$$\mathbf{V}\mathbf{V}^T = \mathbf{V}^T\mathbf{V} = \mathbf{I}$$

In MATLAB, the eigenvalues can be found directly in MATLAB using

```
d = eig(A)
```

where d is a column vector.

This command can also be used with two output arguments as

```
[D,V] = eig(A)
```

In this case, D and V are as defined above.

It is also possible in MATLAB to obtain the characteristic polynomium of A using the statement

```
c = poly(A)
```

here c will contain the coefficients of the polynomium in descending order. The roots of the characteristic polynomium (and any other polynomium) is found using

```
r = roots(c)
```

We can also generate the coeeficients of a polynomium, if the roots are known using

```
c = poly(r)
```

where r is a vector.

Exercise 11.7

Find the eigenvalues of the matrix

$$\mathbf{A} = \begin{pmatrix} 0.50 & 0.25 \\ 0.25 & 0.50 \end{pmatrix}$$

a. by hand

b. using **eig**

c. using **poly** and **roots**

Then use **eig** to perform an eigenvalue decomposition and show that

$$\mathbf{A}=\mathbf{V}\mathbf{D}\mathbf{V}^{T}$$

and that

$$\mathbf{V}\mathbf{V}^{T} = \mathbf{V}^{T}\mathbf{V} = \mathbf{I}$$

Exercise 11.8

If λ is an eigenvalue of **A** with V as a corresponding eigenvector, then for any positive integer, k, λ^k is an eigenvalue of A^k again with V as a corresponding eigenvector. Write a program to demonstrate this property for k = 1 to 5, using the following matrix A:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ -8 & 4 & -6 \\ 8 & 1 & 9 \end{pmatrix}$$

Exercise 11.9

If λ is an eigenvalue of **A** (for which an inverse exists) with V as a corresponding eigenvector, then λ-1 is an egenvector of A-1, again with V as a corresponding eigenvector. Write a program to demonstrate this proporty, using the following matrix A:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ -8 & 4 & -6 \\ 8 & 1 & 9 \end{pmatrix}$$

The scalar linear IVP

The behaviour of the linear inital value problem will be used to investigate the behavior of the general non-linear IVP and to investigate the behavior of methods used to solve IVPs. Firstly, consider the scalar linear IVP

$$\frac{dx}{dt} = \lambda x \quad x(0) = x_0$$

This problem has the solution

$$x(t) = e^{\lambda t} x_0$$

We will define stability of the mathematical problem based on its behaviour as $t\rightarrow\infty$. A stable mathematical problem is a problem for which the solution is bounded as $t\rightarrow\infty$, i.e. there exist a number k, so that $|x(t)|<k$ for all $t>0$.

For λ a real number, we will consider three cases: $\lambda>0$, $\lambda<0$, and $\lambda<<0$ for the scalar linear IVP.

$\lambda>0$: For $\lambda>0$ the solution either increases or decreases exponentially - depending on the sign of x_0 - and is thus unbounded. Therefore, $\lambda>0$ corresponds to an unstable mathematical problem. Unstable problems may be encountered in process risk analysis and control system analysis (as something to avoid).

$\lambda<0$: For $\lambda<0$ the initial value will exponentially decay to zero and is thus bounded. Hence, $\lambda<0$ corresponds to a stable mathematical problem.

$\lambda<<0$: For $\lambda<<0$ the initial value will *immediately* decay exponentially to zero. Again, this corresponds to a stable mathematical problem. The only reason why we distinguish this case from λ just negative, is that for large negative values the problem becomes more difficult to solve numerically (as we will see later). We term this type of problems ultrastable of stiff. As stressed earlier this type of problems is quite common in process engineering. A typical case is when part of a process shows very fast dynamics while another part shows slow dynamics. Then the fast dynamic process will be stiff in relation to the dominant slow process.

We will also need to consider the case where λ is a complex number. If we define

$$\lambda = a+ib$$

the solution can be rewritten as

$$y(t) = e^{\lambda t}y_0 = e^{(a+ib)t}y_0 = e^{at}e^{ibt}y_0 = e^{a}(\cos(bt) + i\sin(bt))y_0$$

It is clear that the sin and cos elements will introduce an occilation in the solution. We see that the sign of the real part will define the stability of the problem, while the size of the imaginary part will determine the rate of oscillation of the solution.

Exercise 11.10

Consider the solution to the linear IVP

$$\frac{dx}{dt} = \lambda x \quad x(0) = 1$$

For each of the following λ values try to predict the nature of the solution in terms of stability (unstable, stable, and stiff) and oscillation (high or low frequency). Then plot both the real and imaginary part of the solution for t = 0 to 5 at intervals of 0.05.

a. -100	b. -10	c. -1	d. 0.1	e. 0.5	f. 5
g. i	h. 10i	i. -1+i	j. -1-i	l. -100+100i	k. -10+10i

The linear IVP

The vector linear IVP problem reads

$$\frac{d\mathbf{x}}{dt} = \mathbf{A}\mathbf{x} \quad \mathbf{x}(0) = \mathbf{x}_0$$

Using eigenvalue decomposition we find

$$\frac{d\mathbf{x}}{dt} = \mathbf{V}\mathbf{D}\mathbf{V}^{\mathrm{T}}\mathbf{x} \quad \mathbf{x}(0) = \mathbf{x}_0 \quad \Rightarrow$$

$$\mathbf{V}^{\mathrm{T}}\frac{d\mathbf{x}}{dt} = \frac{d(\mathbf{V}^{\mathrm{T}}\mathbf{x})}{dt} = \mathbf{V}^{\mathrm{T}}\mathbf{V}\mathbf{D}\mathbf{V}^{\mathrm{T}}\mathbf{x} = \mathbf{D}\mathbf{V}^{\mathrm{T}}\mathbf{x} \quad \mathbf{V}^{\mathrm{T}}\mathbf{x}(0) = \mathbf{V}^{\mathrm{T}}\mathbf{x}_0$$

Introducing the new variable, $\mathbf{z} = \mathbf{V}^{\mathrm{T}}\mathbf{x}$, we find

$$\frac{d\mathbf{z}}{dt} = \mathbf{D}\mathbf{z} \quad \mathbf{z}(0) = \mathbf{z}_0$$

D being a diagonal matrix, the equation in **z** consists of n independent scalar equations in the form

$$\frac{dz_i}{dt} = \lambda_i z_i \quad z_i(0) = z_{i0}$$

The solution of these scalar equations was discussed above and we find

$$z_i(t) = e^{\lambda_i t} z_{i0}$$

We then find **x** as

$$\mathbf{x}(t) = \mathbf{V}\mathbf{z}(t) = \mathbf{V}\begin{pmatrix} e^{\lambda_1 t} & 0 & 0 & 0 \\ 0 & e^{\lambda_2 t} & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & e^{\lambda_n t} \end{pmatrix}\mathbf{V}^T\mathbf{x}_0$$

If multiplied out, we would find that each element of **x** is a linear combination of all the exponential terms, i.e.

$$x_i(t) = \sum_{j=1}^{n} c_{i,j} e^{\lambda_i t}$$

That is, the problem nature corresponding to each eigenvalue will enter each element of the solution in an additive manner. Thus, if the **A** matrix has three eigenvalues: one positive (unstable), one small negative (stable), and one large negative (stiff), the solution would show a combined behaviour of unstable, stable and stiff. In this case, the unstable element would take over as time goes to infinity.

The solution to the linear IVP can also be found using the matrix exponential. In this case, the solution looks very similar to the scalar solution, namely

$$\mathbf{x}(t) = e^{\mathbf{A}t}\mathbf{x}_0$$

It must be stressed that the matrix exponential is not the exponential of each element of the matrix. By comparison to the above it is clear that

$$e^{\mathbf{A}} = \mathbf{V}e^{\mathbf{D}}\mathbf{V}^T = \mathbf{V}\begin{pmatrix} e^{\lambda_1} & 0 & 0 & 0 \\ 0 & e^{\lambda_2} & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & e^{\lambda_n} \end{pmatrix}\mathbf{V}^T$$

In MATLAB, the matrix exponential function is called **expm** and used as

```
>> eA = expm(A)
```

The other exponential function **exp** is an element by element exponential.

Similar to **expm,** the **log** and **sqrt** functions also have special matrix forms termed **logm** and **sqrtm**, which sometimes are used to expand scalar solutions into vector solutions. MATLAB also have a general matrix function evaluator - **funm** - which uses Partletts algorithm.

Exercise 11.11

Determine the eigenvalues of

a. A = [-5,2;8,-5]

b. A = [0,5,-5,0]

c. A = [1,4;-4,1]

d. A = [-4,-3,3,-4]

e. A = [3,2;-3,8]

Based on the eigenvalues predict the behaviour of the solution to

$$\frac{d\mathbf{x}}{dt} = \mathbf{A}\mathbf{x} \quad \mathbf{x}(0) = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Test your predictions by developing a program that computes the solution of the the IVP for t=0 to 5 at 0.05 intervals using both the eigenvalue decomposition path and the matrix exponential path.

Non-linear IVP

Since we already can solve the linear IVP analytically, our ultimate aim is to develop numerical methods to solve non-linear IVP. Hence, it is important to consider how much the behaviour of the linear IVP relates to the behaviour of the general non-linear IVP. For the general non-linear IVP problem it is possible to linearise the right hand side, $\mathbf{g}(t,\mathbf{x})$, around the present value and use the eigenvalues of the Jacobian of the system to define the behaviour of the problem locally (and thus consider the local behavior of a given numerical method applied to the problem). It must be stressed, however, that the set of eigenvalues changes as we move away from the point of linearisation. A classic example of this is flow in a pipe, where the solution is linearly stable at a Reynolds number of 2000 but becomes unstable (turbulent) to large perturbations in the transition region between laminar and turbulent flow.

The numerical solution

General algorithm

Let's restate the IVP. We wish to find the explicit vector function

$$\mathbf{x} = \mathbf{f}(t), \quad t \in [a,b]$$

which forms a solution to the set of ordinary differential equations

$$\mathbf{x}' = \mathbf{g}(t,\mathbf{x}), \quad t \in [a,b]$$

with initial value

$$\mathbf{x}(a) = \mathbf{x}_0$$

IVPs are numerically solved by discretisation, i.e. instead of finding the continuous analytical solution, $\mathbf{f}(t)$, we find an estimate of the solution in a set of descrete time points. We term the interval between two descrete time points the steplength, h, and for simplicity we will here assume that the step length is constant. For the remainder of this chapter we use the following notation

k the discretisation index, k=1:n

t_k the descretisation time points

$\mathbf{x}_k$ the numerical solution in the descrete point

$\mathbf{x}(t_k)$ the true solution in the descrete point

Notice that we have chosen to start the counter at 1 not 0. This is done because MATLAB's counters always start with 1.

The Runge-Kutta methods we will discuss in this chapter are all so-called one step methods, i.e. the numerical solution $\mathbf{x}_{k+1}$ is found from the numerical solution in the previous point, $\mathbf{x}_k$, and the function $\mathbf{g}(t,\mathbf{x})$. In contrast, multipoints methods use several previous points to derive a solution.

To understand how we find the solution from the previous point recall the *mean value theorem* for scalar functions

If f is differentiable on an interval [a,b], then a point $c \in [a,b]$ exists so that

$$f'(c) = \frac{f(b) - f(a)}{b - a}$$

Specifically for our problem, we have the differential specified in the implicit function $\mathbf{g}(t,\mathbf{x})$. We can restate the theorem as follows.

On the interval $t=[t_k,t_{k+1}]$ exists a point, t*, so

$$\mathbf{g}(t^*,\mathbf{x}(t^*)) = \frac{\mathbf{x}(t_{k+1}) - \mathbf{x}(t_k)}{t_{k+1} - t_k} = \frac{\mathbf{x}(t_{k+1}) - \mathbf{x}(t_k)}{h}$$

This equation can be rearranged to yield

$$\mathbf{x}(t_{k+1}) = \mathbf{x}(t_k) + h\mathbf{g}(t^*,\mathbf{x}(t^*))$$

In descretised form, this equation will read

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{g}_k^*$$

which is the fundemental step function used in all algorithms. Thus, the general algorithm of a one-step method is sequential (one point at a time, starting with the initial value) and looks something like

$\mathbf{x}_1 = \mathbf{x}(t_1)$ % Assign known initial value to first numerical solution point

for k=1:n-1

determine $\mathbf{g}_k^*$

$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{g}_k^*$

end

The remaining problem is how to determine $\mathbf{g}_k^*$. It must be stressed that there is no general way to determine the true $\mathbf{g}(t^*,\mathbf{x}(t^*))$. Different algorithms use different estimates of this value, $\mathbf{g}_k^*$, but none will be able to determine the true value. Hence, a numerical routine will always introduce to some level of inaccuracy in the solution.

Furthermore, given that a numerical routine operates in a stepwise or sequential manner and given that errors are introduced in each step, the inaccuracies can compound and the numerical solution end up becoming totally dissimilar to the true solution. Consider the general algorithm above. The first value of the numerical solution, $\mathbf{x}_1$, will be perfectly accurate as it is based on the known true value, $\mathbf{x}(t_1)$. In the second value of the numerical solution, $\mathbf{x}_2$, an error will occur because $\mathbf{g}_1^*$ is not the true $\mathbf{g}(t^*,\mathbf{x}(t^*))$. In the third value value of the numerical solution, $\mathbf{x}_3$, there will be two sources of error. First of all, there is the error because $\mathbf{g}_2^*$ is not the true $\mathbf{g}(t^*,\mathbf{x}(t^*))$. Secondly, there is the error because there was an error in the starting point, i.e. because $\mathbf{x}_2 \neq \mathbf{x}(t_2)$. These two sources of error will occur in all future estimates. Depending on the numerical algorithm, these errors may dampen out giving a stable solution or become amplified resulting in an unstable solution.

Euler, backward euler, and trapezoidal

The euler, backward euler, and the trapezoidal methods represents three different choices of $\mathbf{g}_k^*$.

The **euler method** uses the differential taken in the initial point as an estimate of the true

$$\mathbf{g}(t^*,\mathbf{x}(t^*)), \text{ i.e. } \mathbf{g}_k^* = \mathbf{g}(t_k,\mathbf{x}_k).$$

Thus, the step equation is

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{g}(t_k,\mathbf{x}_k)$$

This equation is explicit, i.e. we know all the elements on the right hand side and can find $\mathbf{x}_{k+1}$ simply insertion of these.

The **backward euler method** uses the differential taken in the final point as an estimate of the true $\mathbf{g}(t^*,\mathbf{x}(t^*))$, i.e. $\mathbf{g}_k^* = \mathbf{g}(t_{k+1},\mathbf{x}_{k+1})$. Thus, the step equation is

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{g}(t_{k+1},\mathbf{x}_{k+1})$$

This equation is implicit. The unknown $\mathbf{x}_{k+1}$ is on both side of the equation. In the figure, we see that we need to evaluate the function in the unknown endpoint. In general, we must solve this equation using an algebraic equation solver, e.g. Newton's method. This aspect will be discussed in detail later.

The **trapezoidal method** uses a comprimise of taken half the differential in the initial point and half in the final point as an estimate of the true $\mathbf{g}(t^*,\mathbf{x}(t^*))$, i.e.

$$\mathbf{g}_k^* = 0.5(\mathbf{g}(t_k,\mathbf{x}_k) + \mathbf{g}(t_{k+1},\mathbf{x}_{k+1}))$$

Thus, the step equation is

$$\mathbf{x}_{k+1} = \mathbf{x}_k + 0.5h(\mathbf{g}(t_k,\mathbf{x}_k) + \mathbf{g}(t_{k+1},\mathbf{x}_{k+1}))$$

This equation is also implicit. We still have the unknown, $\mathbf{x}_{k+1}$, on both side of the equation. Again, we must solve this equation using an algebraic equation solver, e.g. Newton's method.

To discuss the difference between these three methods, we first have discuss errors and stability of numerical methods.

Numerical solution: global error, local error, and stability

Recall that the analytical solution to the scalar IVP

$$y' = \lambda x \qquad x(t_0) \text{ given}$$

was

$$x(t) = e^{\lambda t}x(t_0)$$

For a single step, we could write the true solution

$$x(t_{k+1}) = e^{h\lambda}x(t_k)$$

As we shall see in the next section, a numerical solution can be regarded as a rational approximation to the exponential function. Defining the rational approximation as $R(\lambda h)$, we can write the numerical solution as

$$x_{k+1} = R(h\lambda)x_k$$

Global error

The global error at time point k+1, e_{k+1}, is the difference between the true and the numerical solution at this time point, so

$$e_{k+1} = x(t_{k+1}) - x_{k+1} = e^{h\lambda}x(t_k) - R(h\lambda)x_k$$

If we add and subtract $R(h\lambda)x(t_k)$ and rearrange we find

$$e_{k+1} = e^{h\lambda}x(t_k) - R(h\lambda)x(t_k) - R(h\lambda)x_k + R(h\lambda)x(t_k)$$

$$e_{k+1} = [e^{h\lambda} - R(h\lambda)]x(t_k) + R(h\lambda)[x(t_k)-x_k]$$

$$e_{k+1} = T(\lambda h)x(t_k) + R(h\lambda)e_k$$

where $T(\lambda h) = e^{h\lambda} - R(h\lambda)$ is the so-called truncation error. Thus, the global error at time k+1 is composed of the truncation error for the last step and the previous error weighted with $R(h\lambda)$.

Stability and accuracy

If $|R(h\lambda)|>1$ the error will eventually blow up. Thus, a criterium for gaining a stable numerical solution is that $|R(h\lambda)|\leq 1$. Stability will be discussed in more detail in the following sections.

The truncation error is an indication of how good our rational approximation fits the exponential function. Refering back to last section, it is an indication of how well we have chosen $\mathbf{g}_k$*. The rational approximation will improve when we reduce the step length. We say that a method is accurate of order p, if

$$|T(h\lambda)x(t_k)| = O(h^{p+1})|x|$$

i.e. if the improvement in accuracy as we reduce the steplength is of order p+1. Thus, if the accuracy of the rational approximation improves quadratically we call the method linear. The apparant decrepency is due to the fact that order 2 accuracy for the single step (local behavior) result in only order 1 accuracy for the full solution (global behavior).

If the method is order p and is stable, then the global error is bounded according to

$$|e_k| = O(h^p)$$

If the method is not stable, the global error will not automatically be bounded.

Euler, backward euler, and trapezoidal methods

In order to illustrate the stability concepts developed above we will consider the three methods again. First, we will determine the rational approximations corresponding to the three methods.

For the scalar linear IVP, we have

$$g(t,x) = \lambda y$$

Thus, the **euler method** reads

$$x_{k+1} = x_k + hg(t_k,x_k) = x_k + h\lambda x_k$$

or

$$x_{k+1} = (1+ h\lambda)x_k$$

and the corresponding rational approximation is

$$R(h\lambda) = 1+ h\lambda$$

The **backward euler method** reads

$$x_{k+1} = x_k + hg(t_{k+1},x_{k+1}) = y_k + h\lambda x_{k+1}$$

or

$$(1- h\lambda)x_{k+1} = x_k$$

and the corresponding rational approximation is

$$R(h\lambda) = 1/(1- h\lambda)$$

The **trapezoidal method** reads

$$x_{k+1} = x_k + 0.5h(g(t_k,x_k) + g(t_{k+1},x_{k+1})) = x_k + 0.5h\lambda x_k + 0.5h\lambda x_{k+1}$$

or

$$(1-0.5h\lambda)x_{k+1} = (1+0.5h\lambda)x_k$$

and the corresponding rational approximation is

$$R(h\lambda) = (1+ 0.5h\lambda)/(1- 0.5h\lambda)$$

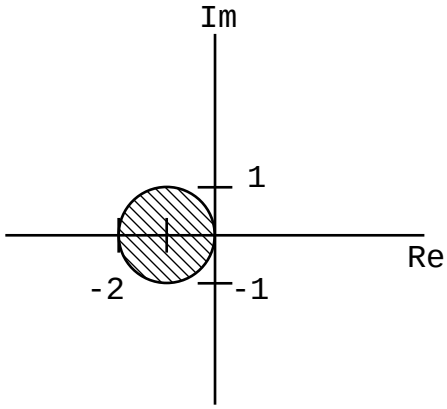
Above the criterium for a numerical method to be stable was found to be

$$|R(h\lambda)|\leq 1$$

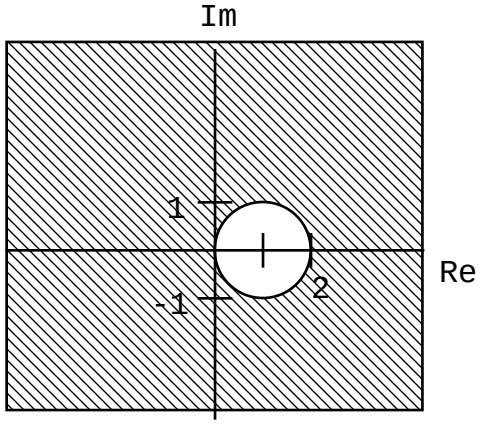
Defining $z = h\lambda$, the stability region of a numerical method is defined as the region in complex plane for which $|R(z)|\leq 1$. In the complex number section earlier, we developed the modulus for the above three rational equations. Based on this, we find for the **euler method**

$$|R(z)| = \sqrt{(1+a)^2 + b^2} \leq 1 \text{ or } (1+a)^2 + b^2 \leq 1$$

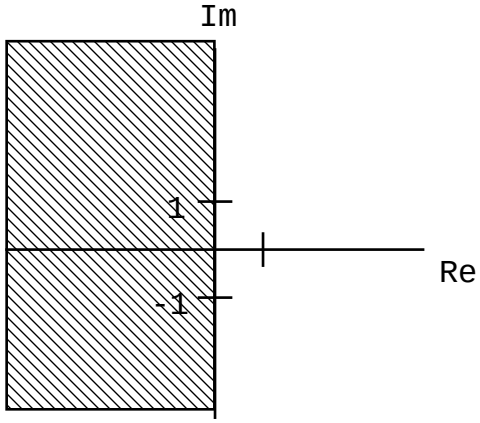
i.e. the stable region of the euler method is inside the circle with centre (-1,0) and radius 1



Similarly (try it yourself), we find for the **backward euler method** that the stable region is outside a circle with centre (1,0) and radius 1.

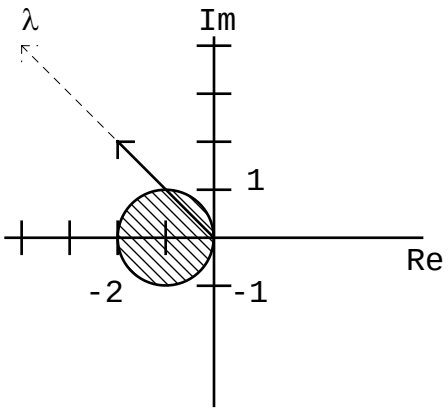


Finally, for the **trapezoidal method** we find the stable region is the whole left hand plane.

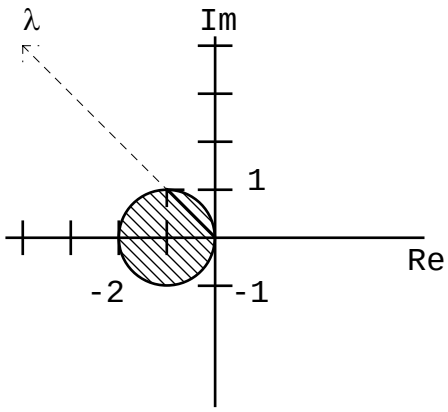


Given that we defined $z = h\lambda$, it is clear that the stability of a numerical solution depends both on the mathematical problem (via λ) and the step size h used. Graphically, we can treat λ as a vector in the complex plane and h as a scaling factor. The numerical solution will be stable if the scaled vector, $h\lambda$, lies inside the stable region. For example, if we wish to assess the stability of the euler method when solving a problem with $\lambda=-4+4i$ at two steplength: h=0.5 and h=0.25, we can graph this as

h=0.5



h=0.25



From this it is clear, that the solution is just stable for h=0.25 but not for h=0.5.

Exercise 11.12

Consider solving the linear IVP

$$\frac{dx}{dt} = \lambda x \quad y(0) = 1$$

using the euler, backward euler and the trapezoidal method. For the two steplengths h=1 and h=0.1, try to predict the behaviour of the numerical solution for each of the following λ values.

a. -100	b. -10	c. -1	d. 0.1	e. 0.5	f. 5
g. i	h. 10i	i. -1+i	j. -1-i	l. -100+100i	k. -10+10i

Now, implement the euler, backward euler and the trapezoidal methods using the rational approximation approach. It must be stressed that this approach is only valid for linear equations. For non-linear equations you must use the original equations, which in case of the backward euler and trapezoidal methods means solving implicit equations.

Solve the problems observing the stability behaviour.

Unstable problem, stable solution

Stability is not necessarily a good feature. When solving problem f. in the above exercise using the backward euler method and a steplength of 1, you would have seen a stable numerical solution. The real solution, however, is very unstable. If you were trying to determine whether a chemical plant would become unstable, you might wrongly conclude that this was not the case.

Limited, A or L stability

We know that the mathematical problem is stable for λ in the negative half-plane and we would like our numerical methods to reflect this fact. Some methods, like the euler method, only has a limited stability region in the negative halfplane. Other methods, like the backward euler and trapezoidal method, have the whole negative halfplane as stability regions. We term such methods A-stable.

In some of the cases in the above exercise, you would have observed that the numerical solution obtained using the trapezoidal method was oscillatory, while the numerical solution obtained using the backward euler method showed the true stiff nature.

The oscillatory solution is still stable (it is bounded), but it is obviously incorrect. The reason for this behavior is that the absolut value of the rational approximation used by the trapezoidal method approaches 1 for large negative z values, thus the method's ability to dampen out previous errors is relatively limited.

The rational approximation of the backward euler method tends to zero for large negative z values. Hence, the dampening effect is very good. Methods showing this strong dampening effect are termed L-stable methods (in addition to being A stable). Such methods are obviously well suited to solve stiff problems, i.e. problems with large negative eigenvalues.

The vector IVP

As was the case for the true solution, the stability of numerical solution depends on all the eigenvalues of the problem. If all the eigenvalues are scaled into a stable region, then the numerical solution is stable. If one or more value lies outside the stability region, the numerical solution will be unstable.

For the linear vector IVP (and only for a linear IVP)

$$\frac{d\mathbf{x}}{dt} = \mathbf{A}\mathbf{x} \quad \mathbf{x}(0) \text{ given}$$

it is possible to use the corresponding rational approximation as a basis for a numerical solution (just as it was done for the scalar case above). The **euler method**, for example, reads

$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{g}(t_k,\mathbf{x}_k) = \mathbf{y}_k + h\mathbf{A}\mathbf{x}_k$$

or

$$\mathbf{x}_{k+1} = = (\mathbf{I} + h\mathbf{A})\mathbf{x}_k$$

hence the corresponding rational approximation is

$$R(h\lambda) = \mathbf{I} + h\mathbf{A}$$

Exercise 11.13

Develop the rational approximations for the backward euler and trapezoidal methods. Notice: in both cases the resultant problem is a linear equation (you can not simply divide with matrices!).

Exercise 11.14

Consider solving the IVP

$$\frac{d\mathbf{x}}{dt} = \mathbf{A}\mathbf{x} \quad \mathbf{x}(0) = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Based on the eigenvalues determined earlier predict the stability when

a. A = [-5,2;8,-5]

b. A = [0,5,-5,0]

c. A = [1,4;-4,1]

d. A = [-4,-3,3,-4]

e. A = [3,2;-3,8]

and h = 0.05, 0.1, 0.2, 0.5,1.

Test your predictions by developing a program that computes and plots the solution of the the IVP for t= 0 to 5 using the analytical solution and using the euler, backward euler, and trapezoidal methods.

Summary

In this chapter, you have seen how ordinary differential equations arise when the dynamics of a system is being considered. When the initial values of the variables described by the ODEs are known, we term the problem an initial value problem. MATLAB provides two functions for solving IVPs: **ode23** and **ode45**. These functions are good for solving many IVPs, but not stiff problems.

In order to analyse IVPs in more detail, we discussed complex numbers, eigenvalue decomposition, and the functions available for this purpose in MATLAB. Using eigenvalue decomposition, the general solution to the linear IVP was developed and the eigenvalues of the Jacobian of the system were shown to determine the stability behaviour of the mathematical problem.

Numerically, IVPs are solved by discretisation and in sequential steps. Three methods - the Euler, the Backward Euler, and the Trapezoidal methods - were introduced. For linear IVPs, the global error of the numerical solution was found to depend on

- the truncation error (how well the rational approximation used by the numerical routine fits the true exponential function), and
- whether or not the error would be sequentially amplified

When the error is amplified rather than dampened, the numerical method is unstable. Stability was found to depend on the step length, the mathematical problem, and the numerical method used. Stability of a numerical method could be predicted by plotting the eigenvalues in the complex plane.

The Euler method was shown to have a limited stability region, the Backward Euler to have a large stability region including the whole negative halfplane (A-stable), and the Trapezoidal methods to have the whole negative halfplane as stable (A-stable). In addition, the Backward Euler method was L-stable, i.e. capable of dampen out possible oscilation in the numerical solution even for very large negative eigenvalues.

In the next chapter, we will look at solving the general non-linear IVP using members of the so-called Runge-Kutta family, of which the three algorithm discussed so far are members.

CHAPTER 12

Ordinary differential equations - The Runge-Kutta family

In the last chapter, we looked at the Euler, Backward Euler, and Trapezoidal methods for solving IVPs. All three methods are members of the Runge-Kutta family of methods. In this section, we will discuss the family in more detail.

Why considering other methods?

There are two factors affecting the accuracy of a numerical method for solving ODEs: the step length and the order of the method. Thus, when requiring a given accuracy we can choose between using a low order method with small steps or a higher order method with longer steps. Higher order methods require more work per step, but since they can take longer steps the final computational effort may be less. The higher the accuracy required is the greater the advantage of having a higher order method will be. The three methods considered so far are all methods of low order - order 1 for the Euler and Backward Euler methods and order 2 for the Trapezoidal method. Hence, we are interested in developing methods of higher order for when they are better at doing the job.

The Runge-Kutta algorithm

Recall that the general algorithm for a one step method looked something like

$\mathbf{x}_1 = \mathbf{x}(t_1)$ % Assign known initial value to first numerical solution point

for k=1:n-1

estimate $\mathbf{g}_k^*$

$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{g}_k^*$

end

and the problem was how to estimate $\mathbf{g}_k^*$. The Runge-Kutta methods estimate by first estimating the value of $\mathbf{x}$ in a number so-called stage points and then forming an estimate as a weighted average of $\mathbf{g}$ evaluated in these stage points. If we have s stage points and call the stage times, $t_{k,1}, t_{k,2}, \ldots, t_{k,s}$, and the corresponding estimates of $\mathbf{x}$, $\mathbf{x}_{k,1}, \mathbf{x}_{k,2}, \ldots, \mathbf{x}_{k,s}$, then the general algorithm expands to

$\mathbf{x}_1 = \mathbf{x}(t_1)$ % Assign known initial value to first numerical solution point

for k=1:n-1

define stage times: $t_{k,1}, t_{k,2}, \ldots, t_{k,s}$

estimate stage values: $\mathbf{x}_{k,1}, \mathbf{x}_{k,2}, \ldots, \mathbf{x}_{k,s}$

$$\mathbf{g}_k^* = \sum_{i=1}^{s} b_i \mathbf{g}(t_{k,i}, \mathbf{x}_{k,i})$$

$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{g}_k^*$

end

where b_i are the weights used to form the weighted average and hence should add up to one.

Estimating the stage values

The stage values are formed by making a number of steps similar to the full step, i.e. if the stage times are given as

$$t_{k,i} = t_k + c_i h \quad i = 1,\ldots,s$$

then the stage values are calculated as

$$\mathbf{x}_{k,i} = \mathbf{x}_k + c_i h \mathbf{g}_i^*$$

where $\mathbf{g}_i$' is a slope estimate for the particular stage estimate and this again is estimated as a weighted average of **g** evaluated in the stage points, so

$$\mathbf{g}_i^* = \sum_{j=1}^{s} w_{ij} \mathbf{g}(t_{k,j}, \mathbf{x}_{k,j}) \text{ and } \quad \sum_{j=1}^{s} w_{ij} = 1$$

An imporatant difference is that the stage estimates are calculated from each other, i.e. in general we need to solve s equations simultaneously!!!

We normally do not use the w-parameter form, but reparameterise introducing

$$a_{ij} = c_i w_{ij} \quad \sum_{j=1}^{Ns} a_{ij} = c_i$$

and write the stage equation as

$$\mathbf{x}_{k,i} = \mathbf{x}_k + h \sum_{j=1}^{s} a_{ij} \mathbf{g}(t_{k,j}, \mathbf{x}_{k,j})$$

The Butcher block

Each member of the Runge-Kutta family is distinguished by its set of parameters: a_{ij}, b_i, and c_i. These can be written in matrix form in what is termed the Butcher block

c_1	a_{11}	a_{12}	$\cdots$	a_{1s}
c_2	a_{21}	a_{22}	$\cdots$	a_{2s}
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
c_s	a_{s1}	a_{s2}	$\cdots$	a_{ss}
1	b_1	b_2	$\cdots$	b_s

One characteristic of the Butcher block that the first column contains the row sums of the a and b elements.

It is beyond the scope of this workbook to discuss how the actual parameters are chosen when one developes Runge-Kutta methods. Here, we will only look at methods that have already been developed, i.e. methods with a known Butcher block.

The Runge-Kutta algorithm

Given the Butcher block of a Runge-Kutta method, the computer implementation of the method would look something like this

$\mathbf{x}_1 = \mathbf{x}(t_1)$ % Assign known initial value to first numerical solution point

for k=1:n-1

$t_{k,i} = t_k + c_i h$ i = 1,…,s %define stage times

$\mathbf{x}_{k,i} = \mathbf{x}_k + h\sum_{j=1}^{s} a_{ij}\mathbf{g}(t_{k,j}, \mathbf{x}_{k,j})$, for i=1,…,s % solve stage equation system

$\mathbf{g}_k^* = \sum_{i=1}^{s} b_i \mathbf{g}(t_{k,i}, \mathbf{x}_{k,i})$ % calculate overall slope estimate

$\mathbf{x}_{k+1} = \mathbf{x}_k + h\mathbf{g}_k'$ %Calculate new point

end

We divide the methods into three subgroups - explicit, semi-implicit, and implicit - depending on how the stage equation system is solved.

Explicit Runge-Kutta methods

Explicit Runge-Kutta methods are characterised by having zeros in and above the diagonal in their Butcher block

$$\begin{array}{c|cccc} 0 & 0 & 0 & \cdots & 0 \\ c_2 & a_{21} & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ c_s & a_{s1} & a_{s2} & \cdots & 0 \\ \hline 1 & b_1 & b_2 & \cdots & b_s \end{array}$$

This make the stage equation system particularly easy to solve. The system reduces to sequential series of explicit equations. The first stage value is calculated as

$$\mathbf{x}_{k,1} = \mathbf{x}_k + h\sum_{j=1}^{s} a_{1j}\mathbf{g}(t_{k,j}, \mathbf{x}_{k,j}) = \mathbf{x}_k$$

the second stage value as

$$\mathbf{x}_{k,2} = \mathbf{x}_k + h\sum_{j=1}^{s} a_{2j}\mathbf{g}(t_{k,j}, \mathbf{x}_{k,j}) = \mathbf{x}_k + ha_{21}\mathbf{g}(t_{k,1}, \mathbf{x}_{k,1})$$

etc. Notice, how the right hand side only contains previously determined values. The first stage estimate is given by the starting point. The second stage estimate is a function of the starting point and the first stage value etc, etc. Thus, calculating the stage estimate is a mere matter of inserting known values in the right hand side.

Example 12.1 Euler's method

Euler's method is the simplest explicit method. It has the Butcher block

0	0
1	1

A function implementing the Euler method could look something like

```
function [t,x] = euler(g_str,t0,tf,h,x0)
% EULER. Solves intial value problems using Euler's method.
%
% USAGE     [t,x] = euler('g_str',t0,tf,h,x0)
%
% INPUT
%       g_str   Name of right hand side function, i.e. the g-function in the IVP
%                       x' = g(t,x)
%               Notice. Function must have two arguments: t and x.
%       t0      Starting time
%       tf      Final time
%       h       Step length
%       x0      Initial value. MUST BE COLUMN VECTOR
%
% OUTPUT
%       t       Column vector with solution time points
%       x       Solution matrix with each variable columnwise
%
% The solution can be plotted using plot(t,x)
t=t0:h:tf;
dim = length(x0); % determine dimension of problem
n = length(t);
x=zeros(dim,n);
x(:,1)=x0;
for k=1:n-1
        t1 = t(k);                      %define stage times
        x1 = x(:,k);                    % solve stage equation system
        gk = feval(g_str,t1,x1);        % calculate overall slope estimate
        x(:,k+1) = x(:,k)+h*gk;         %Calculate new point
end
x = x'; % Transpose matrix to generate a plot matrix
```

Example 12.2 Improved Euler method

The improved Euler method has order 2 rather than order 1. It has the Butcher block

0	0	0
1	1	0
1	0.5	0.5

A function implementing the Improved Euler method could look something like

```
function [t,x] = Imeuler(g_str,t0,tf,h,x0)
% HEADER INFO
t=t0:h:tf;
dim = length(x0); % determine dimension of problem
n = length(t);
x=zeros(dim,n);
```

```
x(:,1)=x0;
for k=1:n-1
%define stage times
        t1 = t(k);
        t2 = t(k)+h;
% solve stage equation system
        x1 = x(:,k);
        x2 = x(:,k)+h*feval(g_str,t1,x1);
% calculate overall slope estimate
        gk = 0.5*feval(g_str,t1,x1)+0.5*feval(g_str,t2,x2);
%calculate new point
        x(:,k+1) = x(:,k)+h*gk;
end
x = x'; % Transpose matrix to generate a plot matrix
```

Exercise 12.1 The "classical" 4th order Runge-Kutta method

One of the most frequently used IVP solver is a fourth order Runge-Kutta method with the following Butcher block

0	0	0	0	0
0.5	0.5	0	0	0
0.5	0	0.5	0	0
1	0	0	1	0
1	1/6	1/3	1/3	1/6

a. Using the general outline developed above, write a function that implement this method.

b. Use the function to solve Exercise 11.4 and compare the result with that found using **ode45**.

b. Your function would make use of 7 function calls. The algorithm can be optimised to make use of only 4 function calls. Try to optimise the code.

The classical 4th order Runge-Kutta method is the basis for the **ode45** function in MATLAB. The last **5** reflects that this routine also runs a fifth order algortihm in parallel. Using the two algorithms in parallel makes it possible to adjust the stepsize on the run to achieve the desired accuracy in as few step as possible.

Characteristics of explicit Runge-Kutta methods

The explicit Runge-Kutta methods are characterised by

- being easy to implement and have a relatively low computational effort per iteration
- having the same order of accuracy as there are number of stages
- having a relatively small stability region, similar to the Euler method i.e. [-2,0] on the real axis.

Implicit Runge-Kutta methods

Implicit Runge-Kutta methods are characterised by having zeros above the diagonal in their Butcher block

$$\begin{array}{c|cccc} 0 & a_{11} & 0 & \cdots & 0 \\ c_2 & a_{21} & a_{22} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline 1 & b_1 & b_2 & \cdots & b_s \end{array}$$

This result in a sequential series of implicit equations. The first stage equation is

$$\mathbf{x}_{k,1} = \mathbf{x}_k + h\sum_{j=1}^{Ns} a_{1j}\mathbf{g}(t_{k,j},\mathbf{x}_{k,j}) = \mathbf{x}_k + ha_{11}\mathbf{g}(t_{k,1},\mathbf{x}_{k,1})$$

Notice, how the unknown $\mathbf{x}_{k,1}$ is on both sides of the equation, i.e. $\mathbf{x}_{k,1}$ must be found using an implicit equation solver. The second stage equation is

$$\mathbf{x}_{k,2} = \mathbf{x}_k + h\sum_{j=1}^{Ns} a_{2j}\mathbf{g}(t_{k,j},\mathbf{x}_{k,j}) = \mathbf{x}_k + ha_{21}\mathbf{g}(t_{k,1},\mathbf{x}_{k,1}) + ha_{22}\mathbf{g}(t_{k,2},\mathbf{x}_{k,2})$$

Here, $\mathbf{x}_{k,1}$ is known from the previous equation, but the unknown $\mathbf{x}_{k,2}$ is placed on both sides of the equal sign and must be found using an implicit equation solver. This repeats itself for all stages.

In the program examples below, we will use Newton's method to solve each stage equation. On normalised form, the stage equations for the semi-implicit methods can be written

$$f(\mathbf{x}_{k,i}) = \mathbf{x}_{k,i} - \mathbf{x}_k - h\sum_{j=1}^{i-1} a_{ij}\mathbf{g}(t_{k,j},\mathbf{x}_{k,j}) - ha_{ii}\mathbf{g}(t_{k,i},\mathbf{x}_{k,i}) = 0$$

where the summation only contains previously estimated (i.e. known) stage values. For the Newton update we need the Jacobian of f, which is calculated as

$$J(\mathbf{x}_{k,i}) = \mathbf{I} - ha_{ii}\frac{\partial \mathbf{g}(t_{k,i},\mathbf{x}_{k,i})}{\partial \mathbf{x}_{k,i}}$$

The last term is the Jacobian of g with respect to $\mathbf{x}_{k,1}$, which will be found using numerical differentiation.

Example 12.3 Backward Euler's method

The backward Euler's method is the simplest semi-implicit method. It has the Butcher block

$$\begin{array}{c|c} 1 & 1 \\ \hline 1 & 1 \end{array}$$

A function implementing the Backward Euler method could look something like

```
function [t,x] = beuler(g_str,t0,tf,h,x0)

Nmax = 100; % Maximum iterations in the Newton algorithm
tol = 1e-8; % Tolerance for the Newton algorithm
```

```
t=t0:h:tf;
dim = length(x0); % determine dimension of problem
n = length(t);
x=zeros(dim,n);
jac = zeros(dim,dim); % Jacobian of g function!!!
x(:,1)=x0;
for k=1:n-1
% define stage times
        t1 = t(k)+h;
% solve stage equation: f(x1) = x1-xk-g(t1,x1) = 0 using Newton's method
        x1 = x(:,k); % Initial guess for Newton algorithm!!!
        for iterations = 1:Nmax
                g = feval('g_str',t1,x1);
                f = x1 - x(:,k) - g;
                if norm(f)<tol, break, end
        % Calculate jacobian of g numerically
                stepjac = (1+ x1)*1e-7;
                for jcol=1:dim
                        xh=x1;
                        xh(jcol)=x1(jcol)+stepjac(jcol);
                        gh=feval(g_str,t1,xh);
                        jac(:,jcol) =(gh-g)/stepjac(jcol);
                end
        % Calculate jacobian of f
                J = eye(dim) - jac;
        % Newton estimate update
                x1 = x1 - J\f;
        end
        if (iterations == Nmax), error('Maximum iterations reached'), end
% calculate overall slope estimate
        gk = feval(g_str,t1,x1); % or just gk = g in this case
%Calculate new point
        x(:,k+1) = x(:,k)+h*gk;
end
x = x'; % Transpose matrix to generate a plot matrix
```

Example 12.4 Two stage semi-implicit method

Consider the two stage, third order method with Butcher block

$$\begin{array}{c|cc} \gamma & \gamma & 0 \\ 1-\gamma & 1-2\gamma & \gamma \\ \hline 1 & 0.5 & 0.5 \end{array}$$

where $\gamma=(3+\sqrt{3})/6$. A function implementing this method could look something like

```
function [t,x] = semi2(g_str,t0,tf,h,x0)

Nmax = 100; % Maximum iterations in the Newton algorithm
tol = 1e-8; % Tolerance for the Newton algorithm
```

```
t=t0:h:tf;
dim = length(x0); % determine dimension of problem
n = length(t);
x=zeros(dim,n);
jac = zeros(dim,dim); % Jacobian of g function!!!
x(:,1)=x0;
gamma = (3+sqrt(3))/6;
for k=1:n-1
% define stage times
        t1 = t(k)+gamma*h;
        t2 = t(k)+(1-gamma)*h;
% solve first stage equation: f(x1) = x1-xk-gamma*g(t1,x1) = 0 using Newton's method
        x1 = x(:,k); % Initial guess for Newton algorithm!!!
        for iterations = 1:Nmax
                g1 = feval('g_str',t1,x1);
                f = x1 - x(:,k) - gamma*g;
                if norm(f)<tol, break, end
        % Calculate jacobian of g numerically
                stepjac = (1+ x1)*1e-7;
                for jcol=1:dim
                        xh=x1;
                        xh(jcol)=x1(jcol)+stepjac(jcol);
                        gh=feval(g_str,t1,xh);
                        jac(:,jcol) =(gh-g1)/stepjac(jcol);
                end
        % Calculate jacobian of f
                J = eye(dim) - gamma*jac;
        % Newton estimate update
                x1 = x1 - J\f;
        end
        if (iterations == Nmax), error('Maximum iterations reached'), end
% solve second stage equation:
% f(x2) = x2-xk-(1-2*gamma)*g(t1,x1)-gamma*g(t2,x2) = 0
% using Newton's method
        x2 = x(:,k); % Initial guess for Newton algorithm!!!
        for iterations = 1:Nmax
                g2 = feval('g_str',t2,x2);
                f = x2-x(:,k)-(1-2*gamma)*g1-gamma*g2;
                if norm(f)<tol, break, end
        % Calculate jacobian of g numerically
                stepjac = (1+ x2)*1e-7;
                for jcol=1:dim
                        xh=x2;
                        xh(jcol)=x2(jcol)+stepjac(jcol);
                        gh=feval(g_str,t2,xh);
                        jac(:,jcol) =(gh-g2)/stepjac(jcol);
                end
        % Calculate jacobian of f
                J = eye(dim) - gamma*jac;
```

```
        % Newton estimate update
                x2 = x2 - J\f;
        end
        if (iterations == Nmax), error('Maximum iterations reached'), end
% Calculate overall slope estimate
        gk = 0.5*g1+0.5*g2;
% Calculate new point
        x(:,k+1) = x(:,k)+h*gk;
end
x = x'; % Transpose matrix to generate a plot matrix
```

Exercise 12.2 The Alexander implicit method

a. Using the general outline developed above, write a function that implement the Runge-Kutta method with the following Butcher block

γ	γ	0	0
c	$c-\gamma$	γ	0
1	b_1	b_2	γ
1	b_1	b_2	γ

where $\gamma = 0.4358665$, $c=(1+\gamma)/2$, $b_1 = -(6\gamma^2-16\gamma+1)/4$, $b_2 = (6\gamma^2-20\gamma+5)/4$.

b. Test your algorithm solving Exercise 11.6.

The Alexander method is L-stable and has an order of 3. The good stability of the method makes it particularly well-suited for stiff problems.

Characteristics of semi-implicit Runge-Kutta methods

The semi-implicit Runge-Kutta methods are characterised by requiring s implicit equation to be solved in series. They can be formulated to the same order as the number of stages, in which case the methods are L-stable (as is the case for the Backward Euler method and the Alexander method). Alternatively, they can be formulated to have an order one greater than the number of stages, but then they will only be A-stable (as is the case for the two stage method developed in Example 12.4).

Implicit Runge-Kutta methods

Implicit are characterised by requiring all stage equations to be solved simultaneously using an implicit equation solver. In general, the computational effort of doing this is too great compared to the benefit of higher accuracy and good stability. Hence, we will not discuss these methods any further.

Summary

In this chapter, you have learned about the Runge-Kutta methods and how they can be implemented in MATLAB from their Butcher block. The explicit methods are easy to implement, but have a relatively limited stability range. Hence, these methods are not suitable for stiff problems. The semi-implicit methods are solved by solving a sequence of

implicit equations. The parameters for the semi-implicit methods can be chosen, so the method is L-stable and thus very suitable for stiff problems.