This manual provides an introduction to MATLAB with exercises which are specifically oriented to the MATLAB service provided by Bath University Computing Service (BUCS). However much of the information provided here is applicable to any MATLAB installation, PC, Mac or UNIX.

This manual has a dual role: it serves first as a set of directed tutorials to be carried out in the laboratory and second as a general reference manual for MATLAB.

Each chapter of the manual represents one tutorial, and includes exercises to be done during private study time. For each tutorial you should read through the relevant chapter, trying out the various features of MATLAB which are described, and then you should do the exercises.

You may extend the chapter by doing your own experiments with the system. Responsible experimentation is essential when learning computing.
# Contents

0 **Introduction: What is MATLAB?**  

1 **First Tutorial: Simple Calculations and File Management**  
   1.1 Starting MATLAB ........................................... 1  
   1.2 Simple arithmetic in MATLAB ............................ 2  
   1.3 The MATLAB Development Environment ................. 4  
   1.4 The MATLAB demo ......................................... 5  
   1.5 Recording a MATLAB session ............................ 5  
   1.6 Printing out from MATLAB .............................. 5  
   1.7 Further information on MATLAB ......................... 5  
   1.8 Exercises for Chapter 1 .................................. 6  

2 **Second Tutorial: Variables and Graphics**  
   2.1 Arrays ...................................................... 6  
   2.2 Simple Functions ......................................... 8  
   2.3 Graphics .................................................... 9  
   2.4 Exercises for Chapter 2 .................................. 10  

3 **Third Tutorial: Loops and Logical Branching**  
   3.1 The for loop ............................................... 11  
   3.2 The while loop ........................................... 13  
   3.3 The if-elseif-else statement ........................... 13  
   3.4 Exercises for Chapter 3 .................................. 15  

4 **Fourth Tutorial: scripts and functions**  
   4.1 The MATLAB workspace ................................... 17  
   4.2 Script files ............................................... 17  
   4.3 Function files ............................................ 17  
   4.4 The general form of a function file ..................... 19  
   4.5 Exercises for Chapter 4 .................................. 19  

5 **Fifth Tutorial: A Population Dynamics Example**  
   5.1 Cellular Automata ......................................... 20  
   5.2 A simple example ......................................... 20  
   5.3 A programming exercise .................................. 21  
   5.4 Exercises for Chapter 5 .................................. 22  

6 **Sixth Tutorial: Further Experiments on Population Dynamics**  

7 **The Matlab Symbolic Math Toolbox**  
   7.1 Exercises on Chapter 7 ................................... 24  

A **Appendix: Solutions to selected exercises**  

B **Appendix: Glossary of UNIX commands**  

C **Appendix: Some useful MATLAB commands**  

D **Appendix: Advanced Programming Style in MATLAB**  


0 Introduction: What is MATLAB?

MATLAB is a package that has been purpose-designed to make computations easy, fast and reliable. It is installed on machines run by Bath University Computing Services (BUCS), which can be accessed in the BUCS PC Labs such as those in 1 East 3.9, 1 West 2.25 or 3 East 3.1, as well as from any of the PCs in the Library. The machines which you will use for running MATLAB are SUN computers that run on the UNIX operating system. (If you are a PC or Mac fan, note that this is a quite different environment from what you are used to. However you will need only a small number of UNIX commands when you are working with MATLAB. There is a glossary of common UNIX commands in Appendix B.)

MATLAB started life in the 1970s as a user-friendly interface to certain clever but complicated programs for solving large systems of equations. The idea behind MATLAB was to provide a simple way of using these programs that hid many of the complications. The idea was appealing to scientists who needed to use high performance software but had neither the time nor the inclination (nor in some cases the ability) to write it from scratch. Since its introduction, MATLAB has expanded to cover a very wide range of applications and can now be used as a very simple and transparent programming language where each line of code looks very much like the mathematical statement it is designed to implement.

Basic MATLAB is good for the following.

- Computations, including linear algebra, data analysis, signal processing, polynomials and interpolation, numerical integration (quadrature), and numerical solution of differential equations.
- Graphics, in 2-D and 3-D, including colour, lighting, and animation.
It also has collections of specialised functions, called toolboxes, that extend its functionality. In particular, it can do symbolic algebra, e.g. it can tell you that \((x+y)^2\) is equal to \(x^2+2*x*y+y^2\).

It is important not to confuse the type of programming that we shall do in this course with fundamental programming in an established high-level language like C, JAVA or FORTRAN. In this course we will take advantage of many of the built-in features of MATLAB to do quite complicated tasks but, in contrast to programming in a conventional high-level language, we shall have relatively little control over exactly how the instructions which we write are carried out on the machine. As a result, MATLAB programs for complicated tasks may be somewhat slower to run than programs written in languages such as C. However the MATLAB programs are very easy to write, a fact which we shall emphasise here. We shall use MATLAB as a vehicle for learning elementary programming skills and applications. These are skills which will be useful independently of the language you choose to work in. Some students in First Year will already know some of these skills, but we shall not assume any prior knowledge.

You will meet a course in JAVA programming in the second semester of the First Year.

1 First Tutorial: Simple Calculations and File Management

In this tutorial, you will start to become familiar with the MATLAB development environment and some of its facilities. You will learn

- how to start and quit MATLAB,
- how to do simple arithmetic calculations,
- how to assign values to variables,
- how to use some of MATLAB's built-in functions,
- how to copy files from my filespace to yours, and
- what a script is and how to write, save and run one.

1.1 Starting MATLAB

1.1.1. Starting a MATLAB session (in WindowsXP). Here is one way to do it.

- Log on to Windows.
- From the Desktop, click Start, then Amos, mary, midge from the pop-up menu.
In the BUCS XClient pop-up window, change host from bucs to amos or midge and log on.

In the blue UNIX window, type `xterm &` (where & represents a space) to the $ prompt for a second UNIX window, which you may need later. (See e.g. the last paragraph of section 1.2.)

In either one of these UNIX windows, type `matlab`.

(It is highly advisable to run only one MATLAB session at any one time. The total number of MATLAB sessions that may be run on campus is limited by the licence. You need only one session to do everything you need to do. However, there is no limit to how many UNIX windows you can have.)

**1.1.2. Running MATLAB.** After you have typed `matlab`, a MATLAB logo will come up and then a MATLAB command window with a prompt `>>`. Now you are ready to use MATLAB. When MATLAB is starting you should get a flash of the MATLAB graphics window on your screen. If this did not happen, something is wrong with your set-up and you will not get graphics when you want them. ASK YOUR TUTOR FOR HELP (and/or go to the BUCS help desk in the library after your first tutorial) if this happens.

**1.1.3. Terminating your session.** To finish a MATLAB session, type `exit` to the `>>` prompt. Then type `exit` to the $ prompts. Then log out from your Windows session. *It is essential that you log out completely after you have finished with any machine.*

From now on, when you are instructed to type various commands, unless you are explicitly told to type them elsewhere, they should be typed to the prompt `>>` in the MATLAB command window.

**1.2 Simple arithmetic in MATLAB**

The basic operations are `+`, `-`, `*`, `/`, and `^`, which stand for add, subtract, multiply, divide and exponentiate, or “raise to the power of”. Work through the following example, which shows the results of typing some simple arithmetic commands to the MATLAB prompt. The commands typed by the user are those immediately following the `>>` prompt. A `%` symbol means a comment: the rest of this line is ignored by MATLAB. When a computation produces a response, this is displayed immediately below the command that produced it.

```
% script tutorial1.m Lines beginning % are comments

>> clear % removes any old variables from the workspace
     % (There are none if you’ve just started a session,
     % but this could be important later.)

>> format short % outputs results in decimal form
     % with 5 digit accuracy

>> 2+2 % trivial computation, result displayed on screen
     ans =
        4

>> x = 2+2 % same computation, result loaded into new
     % variable x and displayed
     x =
        4

>> y = 2^2 + log(pi)*sin(x); % more complicated computation
     % illustrates built-in functions
```
%% log and sin and built-in constant pi
%% The ; at the end suppresses output

>> y % prints the contents of y

y = 3.1337

>> format short e % changes format of output
>> y

y = 3.1337e+00

>> format long % changes format of output again
>> y

y = 3.13366556593561

more on % This allows text to be presented page by page
% on the screen. Without it the text just scrolls
% around very quickly and is impossible to read

>> help format % prints the contents of the manual
% for the command ‘format’

FORMAT Set output format.

All computations in MATLAB are done in double precision.
FORMAT may be used to switch between different output
display formats as follows:

- FORMAT Default. Same as SHORT.
- FORMAT SHORT Scaled fixed point format with 5 digits.
- FORMAT LONG Scaled fixed point format with 15 digits.
- FORMAT SHORT E Floating point format with 5 digits.
- FORMAT LONG E Floating point format with 15 digits.
- FORMAT SHORT G Best of fixed or floating point format with 5 digits.
- FORMAT LONG G Best of fixed or floating point format with 15 digits.
- FORMAT HEX Hexadecimal format.
- FORMAT + The symbols +, - and blank are printed
  for positive, negative and zero elements.
  Imaginary parts are ignored.
- FORMAT BANK Fixed format for dollars and cents.
- FORMAT RAT Approximation by ratio of small integers.

Spacing:

- FORMAT COMPACT Suppress extra line-feeds.
- FORMAT LOOSE Puts the extra line-feeds back in.

>> who % Displays the current variables in
% the workspace

Your variables are:

3
This sequence of instructions has been saved to the file \texttt{~masigg/ma20014/matlab/manual/tutorial1.m}. This file is a MATLAB program or \emph{M-file}. In fact it is a particular kind of M-file called a \emph{script}, which is simply a sequence of MATLAB instructions in a file. (We shall learn about other kinds of programs in Tutorial 4 below.)

Copy the file \texttt{~masigg/ma20014/matlab/manual/tutorial1.m} over to your own filespace. \emph{In your second UNIX window}, type \texttt{cp ~masigg/ma20014/matlab/manual/tutorial1.m tutorial1.m} (where \texttt{\_} represents a space) to copy it to the directory you are working in and give it the same name \texttt{tutorial1.m}. (See the glossary of UNIX commands in Appendix B below.) Bring up an editor/debugger window by typing \texttt{edit}. (You may have to type this twice, as the first time may implement an initialisation process.) Now you can

\begin{itemize}
  \item see what is in the file by clicking on \texttt{File --> Open} in the editor/debugger window and then selecting the file name from the drop-down menu, and
  \item run it by typing \texttt{tutorial1} in the MATLAB command window.
\end{itemize}

Do so now. Note that the actual script file contains a number of \texttt{pause} statements that cause execution to stop (so that you can read the result). It starts again when any key is pressed on the keyboard.

### 1.3 The MATLAB Development Environment

This consists of

\begin{itemize}
  \item the MATLAB command window, created when you start a MATLAB session, possibly supplemented with
  \item a figure window which appears when you do any plotting,
  \item an editor/debugger window, used for developing and checking MATLAB programs, and
  \item a help browser.
\end{itemize}

The command window is the one with the > prompt. In version 7 it has three subwindows,

\begin{itemize}
  \item the current directory window, listing all the files in your current directory,
  \item the workspace window (click on the tab), listing all the variables in your workspace, and
  \item the command history window, containing all your commands going back over multiple sessions.
\end{itemize}

You can do things like running M-files from your directory, plotting variables from your workspace, and running commands from your command history or transferring them to a new M-file, by clicking (or right-clicking or double-clicking) on them.

To see the figure window in action, type

\begin{verbatim}
Z = peaks; surf(Z)
\end{verbatim}

and then hit RETURN. (Note that it is crucial to get the punctuation in the above command correct.) You should get a 3D plot of a surface in the Figure window called Figure 1. (Note that \texttt{peaks} is a built-in function that produces the data for this plot, and \texttt{surf} is a built-in function that does the plotting. When you come to do your own plots you will be creating your own data of course.) You can play with the buttons at the top of the Figure window (for example to rotate the figure and see it from different perspectives).

To obtain the editor/debugger, type \texttt{edit}. You can then create files by typing them into this window, and save them by clicking on \texttt{File --> Save As}. You must call the file something like \texttt{file1.m}, i.e. save it with a .m extension to its name, so that MATLAB recognises it as an M-file. If the file is a script, you can run it by typing its name (without the .m). Create a script file containing the single line \texttt{Z = peaks; surf(Z)}, save it as \texttt{file1.m}, and run it.

To obtain help you can type \texttt{helpwin}, which produces a simple browser containing the MATLAB manual. A much more sophisticated web-based help facility is obtained by typing \texttt{helpdesk}, but at peak times this may be slow in starting up. If you have MATLAB on your home PC you may well prefer the \texttt{helpdesk}.

Finally, if you know the name of the command you are interested in, you can simply ask for help on it at the MATLAB prompt. For example to find out about the \texttt{for} command, type \texttt{help for} or \texttt{doc for}. Do each
of these now: for is one of the loop-constructing commands that will be used in the programs you write in this course.

Use any help facility to find the manual page for the command plot, which resides in the directory graph2d. Read the manual page for plot.

1.4 The MATLAB demo

Type demo. A MATLAB demo window comes up. This contains examples of the use of many MATLAB features listed on the left-hand side of the window. You should work through the demos on Graphics, Language and Gallery. Some of the items there are quite advanced: you should not feel that you need to understand everything you see, but try to get a feel for a range of available commands. When you have completed this you may find it entertaining to look at some of the other features in More Examples.

1.5 Recording a MATLAB session

Often you may need to make a permanent record of parts of your MATLAB sessions. Any MATLAB session may be recorded by typing diary filename, where filename is any name you care to choose. Everything that appears in your MATLAB window is then recorded in the file filename until you type diary off. Type help diary to find out about this. After you have finished recording you may choose simply to look at the file filename, which can be done for example by reading it into your editor/debugger window. You may also wish to print it out. To do this, see the section below on printing.

1.6 Printing out from MATLAB

Anything printed from the BUCS machines will come out on the laser printer in the reprographics section of the library and you will have to pay a fee per page. To find out about printing in general from the UNIX machines, look at the BUCS web site at http://www.bath.ac.uk/BUCS/print.html.

Inside MATLAB, if you have created a graph and you want to print it out, make sure that the current graph window contains what you want to print and then click on File --> Print. Clicking Print in the dialog box will send the figure to the library laser printer. To check if your job went into the queue, type lpq -Pll -m****, (where ll stands for library laser and ma**** is your userid), to a $ prompt. (This again illustrates the usefulness of a second UNIX window.) If you have any trouble with this, ask the BUCS helpdesk.

1.7 Further information on MATLAB

Look in the library catalogue for titles containing the keyword MATLAB. In particular, the following are useful.

- Getting started with MATLAB 6: a quick introduction for scientists and engineers, by Rudra Pratap. This is a user-friendly book that you might wish to buy.
- Essential MATLAB for scientists and engineers, by Brian D. Hahn
- The MATLAB 5 handbook, by Eva Part-Enander and Anders Sjoberg
- The MATLAB handbook, by Darren Redfern and Colin Campbell
- MATLAB Guide, by D.J. Higham and N.J. Higham

The MATLAB web-site http://www.mathworks.com is maintained by the company The MathWorks which created and markets MATLAB.
1.8 Exercises for Chapter 1

1. Using MATLAB, do the following calculations:
   
   (i) \( 1 + 2 + 3 \),  
   (ii) \( \sqrt{2} \),  
   (iii) \( \cos(\pi/6) \).

   Note that MATLAB has \texttt{pi} as a built-in approximation for \( \pi \) (accurate to about 13 decimal places), and built-in functions \texttt{sqrt} and \texttt{cos}. Look up the manual pages for these if necessary.

2. After MATLAB has done a calculation the result is stored in the variable \texttt{ans}. After you have done (iii) above, type \texttt{ans^2} to find \( \cos^2(\pi/6) \). The true value of this is \( \frac{3}{4} \). (Check that you know why!) Print out \texttt{ans} using both \texttt{format short} and \texttt{format long} and compare with the true value. On the other hand use MATLAB to compute \texttt{ans-3/4}. This illustrates rounding error: the fact that the machine is finite means that calculations not involving integers are done inexactly with a small error (typically of size around \( 10^{-15} \) or \( 10^{-16} \)). The analysis of the effect of round-off error in calculations is a very interesting and technical subject, but it is not one of the things that we shall do in this course.

3. The quantities \( e^x \) and \( \log(x) \) (log to base \( e \)) are computed using the MATLAB commands \texttt{exp} and \texttt{log}.
   Look up the manual pages for each of these and then compute \( e^3 \), \( \log(e^3) \) and \( e^{\pi\sqrt{163}} \).

4. Compute \( \sin^2(\pi/6) + \cos^2(\pi/6) \). (Note that typing \texttt{sin^2(x)} for \( \sin^2 x \) will produce an error!)

2 Second Tutorial: Variables and Graphics

In this tutorial, you will learn

- how to create row and column vectors (one-dimensional arrays) and matrices (two-dimensional arrays),
- how to do simple arithmetic operations (addition, subtraction, and multiplication by a scalar) on arrays,
- how to do array operations, i.e. term-by-term multiplication (\( \cdot \)), division (\( \cdot / \)), and exponentiation e.g. by a scalar (\( \cdot ^ \)),
- how to use built-in functions with array arguments,
- how to set up and plot two-dimensional line plots, and
- how to produce animated plots.

2.1 Arrays

The basic data type in MATLAB is the rectangular array, which contains numbers (or even \texttt{strings} of symbols). MATLAB allows the user to create quite advanced data structures, but for many scientific applications it is sufficient to work with matrices. These are two-dimensional arrays of data in the form

\[
A = \begin{bmatrix}
a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\
a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n,1} & a_{n,2} & \cdots & a_{n,m}
\end{bmatrix}.
\]

Note that the notation implies that \( a_{i,j} \) is the element of \( A \) which is stored in the \( i \)th row and \( j \)th column. The number of rows \( n \) need not be the same as the number of columns \( m \). We say that such a matrix \( A \) is of dimension \( n \times m \). When \( n = 1 \) we have a \texttt{column vector} and when \( m = 1 \) we have a \texttt{row vector}. When \( n = m = 1 \), \( A \) has a single entry, and is commonly known as a \texttt{scalar}.

You do not need any prior knowledge of matrices to follow this part of these notes.

A matrix can be created in MATLAB using any name starting with a letter and containing only letters, numbers and underscores (_). A major distinction between MATLAB and many other programming languages
is that in MATLAB one does not have to declare in advance the size of an array. Storage of arrays is dynamic in the sense that an array with a given name can change size during a MATLAB session.

The MATLAB command

\[ x = \exp(1); \]

creates a single number whose value is \( e \) and stores it in the variable whose name is \( x \). This is in fact a \( 1 \times 1 \) array. The command

\[ y = [0,1,2,3,4,5]; \]

creates the \( 1 \times 6 \) array whose entries are equally spaced numbers between 0 and 5 and stores it in the variable \( y \). A quicker way of typing this is \( y = [0:5] \). (More generally, the command \([a:b:c]\) gives a row vector of numbers from \( a \) to \( c \) with spacing \( b \). If the spacing \( b \) is required to be 1, then it can be left out of the specification of the vector.) Matrices can change dimension, for example if \( x \) and \( y \) are created as above, then the command

\[ y = [y,x] \]

produces the response

\[ y = \]

\begin{verbatim}
Columns 1 through 7
0 1.0000 2.0000 3.0000 4.0000 5.0000 2.7183
\end{verbatim}

In the example above we created a row vector by separating the numbers by commas. To get a column vector of numbers you separate them by semi-colons, e.g. the command \( z = [1;2;3;4;5] \); creates a \( 5 \times 1 \) array of equally spaced numbers between 1 and 5.

Matrices of higher dimension are created by analogous arguments. For example, the command

\[ A = [[1,2,3];[4,5,6];[7,8,10]]; \]

puts the matrix

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 10
\end{bmatrix}
\]

into the variable \( A \). Complicated matrices can be built up from simpler ones, for example four matrices \( A1, A2, A3, A4 \) of size \( n \times m \) can be combined to produce a \( 2n \times 2m \) matrix \( A = [[A1,A2];[A3,A4]] \).

On the other hand, if an \( n \times m \) matrix \( A \) has been created then we can easily extract various parts of it. To extract the second column of \( A \) type \( A(:,2) \) and to extract the third row type \( A(3,:) \). Here is an extract of a MATLAB session in which various parts of the matrix \( A \) created above are extracted.

\[ \texttt{>>A(:,2)} \% \text{ all rows, column 2} \]

\[ \texttt{ans} = \]

\[ \begin{array}{c}
2 \\
5 \\
8 \\
\end{array} \]

\[ \texttt{>>A([1,2],[2,3])} \% \text{ rows 1 and 2, columns 2 and 3} \]

\[ \texttt{ans} = \]
Vectors and matrices can be created in your MATLAB workspace by various methods. For example there are many built-in commands: `zeros(n,m)` and `ones(n,m)` produces $n \times m$ matrices of zeros and ones. The command `rand(n,m)` produces a random $n \times m$ matrix. The command `eye(n,n)` produces the $n \times n$ identity matrix with ones on its diagonal (from top left to bottom right) and zeros elsewhere. Try these out for some values of $n$ and $m$. Type `help command` for each of these commands and read the MATLAB manual page for each of them. Don’t worry if you don’t understand all about random numbers.

If the entries of a vector or matrix are given by a formula then there is a simple way to create them, called a `for` loop. For example, if we wish to create a vector $x$ whose entries are the squares of the natural numbers from 1 to 10 we can use the command `for i=1:10; x(i) = i^2; end`. We shall return to `for` loops in tutorial 3.

### 2.2 Simple Functions

If $A$ and $B$ are two matrices (which must be of the same size) then $A+B$ gives a new matrix, each element of which is the sum of the corresponding elements of $A$ and $B$. $A-B$ is defined similarly. $5*A$ multiplies each element of $A$ by 5. The pointwise product $A.*B$ produces a new matrix, the entries of which are the products of the corresponding entries of $A$ and $B$. Note that the . before the * indicates that the multiplication is done pointwise (or elementwise). Similarly the pointwise quotient is $A./B$.

Some of you may also know about the matrix product $A*B$ which is defined even when $A$ and $B$ are not of the same size, as long as the number of columns of $A$ is the same as the number of rows of $B$. This (quite different) product is obtained in MATLAB with the command $A*B$, but we will not discuss it further here. Please remember, therefore, that if you really want to compute the pointwise product or quotient, leaving out the . could be catastrophic, as the machine may well compute a wrong answer without giving you a warning.

Every function can be applied to a whole matrix (or more generally array) of numbers. The result will be an array of the same size, each element of which is the the result of the function in question applied to the corresponding element of the array. For example, with $y$ as created in the previous subsection, the command

```matlab
z = log(y)
```

produces the response

```
Warning: Log of zero.
```

```
z =

Columns 1 through 7

-Inf 0 0.6931 1.0986 1.3863 1.6094 1.0000
```

Note that `log` is the built-in function which computes $\log_e$ (or ln) and that the resulting vector $y$ above is the result of applying $\log_e$ to each entry of the previous vector $y$. MATLAB handles an infinite answer by recording it as Inf and producing a warning. (Why is it -Inf?) Note that these calculations verify that $\log_e(1) = 0$ and $\log_e(e) = 1$.

At `~masigg/ma20014/matlab/manual/tutorial2a.m` is a short MATLAB script illustrating some elementary arithmetic. You should copy it and run it.
2.3 Graphics

The simplest plotting command is `plot`. If $x$ and $y$ are any two vectors of the same size, then the command

```
plot(x,y,'r*')
```

plots the components of $y$ in turn against the components of $x$, using red asterisks. There are several variations of this command. Type `help plot` or `doc plot` or run `demo --> Graphics --> Line plotting` to find out more about this command. Other simple plotting commands, for which you should consult the help pages or see `demo --> Graphics --> 2-D Plots`, include `fplot`, `bar`, `hist`, `stem`, `stairs` and `pie`.

The following script file (available at `~masigg/ma20014/matlab/manual/tutorial2b.m`) produces three simple plots. The first is a static plot comparing the function $\sin(x)$ with its approximation $x - x^3/3!$. The second is a static plot producing a helix (spiral). The third plots this dynamically using the `comet` command. The plots come up in the Figure window. They can be printed or copied and pasted into another document if required.

```matlab
% script file tutorial2b.m
% program to explore the use of some simple plotting commands

% First example discusses a cubic approximation to \( \sin(x) \)

x = pi*(0:100)/100; % row vector of 101 equally spaced
% points between 0 and pi
hold off % means that the next plot will be
% drawn on a clean set of axes
plot(x,sin(x),'r*') % plots \( \sin(x) \) against \( x \) using
% red *'s
hold on % next plot will be drawn on top
% of the present one
plot(x,(x-x.^3/factorial(3)),'bo') % plots values of \( x - (x^3)/3! \)
% against \( x \) using blue o's
pause % causes execution to stop until you next hit
% any key (useful for admiring intermediate plots)

% second example draws a 3D helix composed of the points
% \((\sin(t),\cos(t),t)\) for \(t\) ranging over
% 301 equally spaced points between 0 and 100

t = [0:300]/3;
hold off
plot3(sin(t),cos(t),t,'r*',sin(t),cos(t),t,'g-')
```

9
% now lets do an animated plot of this set of values
% using the command 'comet'
comet3(sin(t),cos(t),t)

Finally the script tutorial2c.m demonstrates another animated plot. It plots the waves given by the functions
\[ \sin(x - t), \cos(x - t) \]
for values of \( x \) between 0 and \( 8\pi \) and for time \( t = 0, 1, 2, \ldots, 100 \). Here a for loop is used to advance time in an obvious way. Note the very important \texttt{drawnow} command, which causes the plot to be put on the screen at each time step, thus allowing the animation to be seen.

% script file tutorial2c.m
% animated plot showing the behaviour of the
% waves \( \sin(x-t) \) and \( \cos(x-t) \) for 201 equally spaced points in the range
% \( 0 \leq x \leq 2\pi \)
% \( 0 \leq t \leq 100 \)
% and \( t \) ranging from 0 to 10 in steps of 0.1

\[ x = 8\pi*[0:200]/200; \] % 201 equally spaced points between
\[ 0 \] and \( 8\pi \)

\[ \text{for } t = 0:1:100 \] % start of for loop
\[ \% plot will be drawn for \( t=0,1,2,\ldots,100 \)\]
\[ \text{drawnow} \] % is needed to ensure that animated plots
\[ \% are drawn \]
\[ \text{plot}(x,\sin(x-t),'b-') \] % plot sin wave
\[ \text{hold on} \] % ensures next plot drawn on the same axis
\[ \% as first \]
\[ \text{plot}(x,\cos(x-t),'r--') \] % plot cosine wave
\[ \text{hold off} \] % ensures next plot is drawn on fresh
\[ \% axes \]

end % end of for t loop

2.4 Exercises for Chapter 2

1. Create the \( 3 \times 3 \) matrix whose entries in the first row and first column are all equal to 1 and all the other entries are equal to \( e \).

2. Plot the function \( \exp(-x) \sin(8x) \) for \( 0 \leq x \leq 2\pi \).

3. Generate a row vector \( x \) of even integers, each component \( x(i) \) of which lies in the range \( 0 \leq x(i) \leq 10 \). Compare the effects of the operations (i) \( \sin(x) \), (ii) \( x.\ ^{2} \), (iii) \( x.\ ^{(-1)} \).
4. Generate the $1 \times 10$ row vector $v$ whose $i$-th component is $\cos(i\pi/4)$.

5. Modify the script file tutorial2c.m in the editor/debugger window to do an animated plot of the functions $\sin(4(x - t))$ and $4\cos(x - t)$ for $x$ between 0 and $8\pi$ and $t$ from 0 to 10. Experiment with different line colours and types.

3 Third Tutorial: Loops and Logical Branching

In most of the scripts we have seen so far the commands are executed consecutively, working down the script from the top to the bottom. In real programs you may want to repeat the same command several times, or perform different calculations in different situations. Statements that tell the computer which command is to be executed next are fundamental building blocks for computer programs, and are called control-flow statements. In this tutorial you will learn how to use three control-flow statements for MATLAB programs,

- the **for** loop, for repeating a statement or group of statements a fixed number of times,
- the **while** loop, for repeating a statement or group of statements an indefinite number of times while a condition remains true, and
- the **if-elseif-else** statement, which tells the computer to perform different calculations in different situations.

3.1 The **for** loop

A simple form of such a loop is

```matlab
for index = 1:n
    statements
end
```

The statements are executed $n$ times with $\text{index} = 1, 2, 3, \ldots, n$. Here $n$ is a number that must be fixed before we enter the loop. More generally, the loop may start with the line `for index = j:m:k`. Usually $j, m$ and $k$ are integers and $k-j$ is divisible by $m$. The statements are executed repeatedly starting from $\text{index} = j$ with $\text{index}$ incremented by $m$ each time and continuing until $\text{index} = k$ (which is the last time round the loop). If $m = 1$, we can replace $j:m:k$ in the first line of the loop by $j:k$. Even more generally, we can begin with the line

```matlab
for index = v
```

where $v$ is a vector. The statements inside the loop are repeatedly executed with $\text{index}$ set to each of the elements of $v$ in turn. Type `help for` and read the manual page for the **for** loop.

As an example, consider the computation of the series

$$1 - 1/2 + 1/3 - 1/4 + 1/5 - \ldots$$

(3.1)

This is implemented in the script file tutorial3a.m:

```matlab
% script tutorial3a.m
% computes the sum of the series
% 1 - 1/2 + 1/3 - 1/4 + \ldots
% N = input('type the number of terms to be added up: ')

% asks the user to provide a value for N

sign = 1; % initialise the sign for the term
```
sum_series = 0; % initialise the sum of the series

for n = 1:N
    sum_series = sum_series + sign/n;
    sign = -sign; % changes sign for alternating series
end

sum_series % prints out the sum of the series to N terms

When you have worked with MATLAB for a while you will find that program speeds can be improved by using (if possible) vector or matrix operations instead of loops. For example, to sum the series (3.1) to an even number of terms \( N \), we can use the following very short script.

% script tutorial3b.m
% computes the sum of the series
% % 1 - 1/2 + 1/3 - 1/4 + ... %
% % to an even number of terms
% % using vector operations

N = input('type the number of terms N (with N even): ')
sum_series = sum(1./(1:2:N-1) - 1./(2:2:N))

Exercise: Make sure you understand how this program works. Use help sum.

To compare the speed of the programs tutorial3a.m and tutorial3b.m, we can add the commands tic and toc (see the manual pages) at the beginning and end of each of the scripts. I did this and found the following elapsed times (in seconds), which illustrates the benefit in terms of speed of avoiding the loops if possible.

<table>
<thead>
<tr>
<th></th>
<th>tutorial3a</th>
<th>tutorial3b</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N = 10^4 )</td>
<td>4.4</td>
<td>2.9</td>
</tr>
<tr>
<td>( N = 10^6 )</td>
<td>83.3</td>
<td>8.1</td>
</tr>
</tbody>
</table>

We have seen how to use a for loop to create a vector whose entries are given by a formula. If the entries of a matrix are given by a formula then we can use nested for loops to create it. For example the Hilbert matrix \( H \) is an \( n \times n \) matrix whose entry in the \( i \)th row and \( j \)th column is \( 1/(i+j-1) \). If a value has been assigned to \( n \), we can write

\[
\begin{align*}
\text{for } i = 1:n \\
\text{for } j = 1:n \\
H(i,j) = 1/(i+j-1); \\
\text{end } % \text{ for } j \\
\text{end } % \text{ for } i
\end{align*}
\]

I have added the comment after the end statements to show which loop is ended. This is useful when there are several nested loops.
For the MATLAB beginner, double for loops are the easiest way to create the matrix $H$. However they are actually very inefficient in terms of computing time. The reason for this is quite technical; it’s to do with the way each MATLAB command is interpreted into machine instructions. As you get better at using MATLAB you will find faster ways of creating matrices which are given by a formula.

Nevertheless it is very important for you to gain experience with nested loops here, since in other languages they are not slow and are the preferred method for doing many array operations.

3.2 The while loop

The general form of the while statement is

```matlab
while (condition) 
    statements 
end
```

The condition is a logical relation and the statements are executed repeatedly while the condition remains true. The condition is tested each time before the statements are repeated. It must eventually become false after a finite number of steps, or the program will never terminate.

Example. Suppose we have invested some money in a fund which pays 5% (compound) interest per year, and we would like to know how long it takes for the value of the investment to double. Indeed we would like to obtain a statement of the account for each year until the balance is doubled. We cannot use a for loop in this case, because we do not know beforehand how long this will take, so we cannot assign a value for the number of iterations on entering the loop. Instead, we must use a while loop.

```matlab
% script tutorial3c.m

format bank % output with 2 dec places
invest = input('type initial investment: ') %

r = 0.05; % rate of interest
bal = invest; % initial balance
year = 0; % initial year
disp(' Year Balance') % header for output

while (bal < 2*invest) % repeat while balance is
    bal = bal + r*bal; % update bal
    year = year + 1; % update year
    disp([year,bal])
end
```

3.3 The if-elseif-else statement

A simple form of the if statement is

```matlab
if (condition) 
    statements 
end
```
Here condition and statements are the same as in the while loop, but in this case the statements are executed only once if condition is true and are not executed at all if condition is false. For example the following script divides 1 by i, provided i is non-zero; otherwise, j is not assigned a value.

```matlab
if (i ~= 0)
    j=1/i;
end
```

The symbol ~= is a relational operator and stands for is not equal to. Other relational operators include ==, <=, >=, etc. Type help ops to find out about these. Note the difference between the relational operator == and the usual use of the symbol =, which assigns a value to a variable.

The if-else statement allows us to choose between two courses of action. For example the following script reads in a number and prints out a message to say if it is negative or non-negative.

```matlab
x = input(’ Type x : ’)
if (x<0)
    disp(’x is negative’)
else
    disp(’x is non-negative’)
end
```

Note that indenting of statements inside loops and if statements helps make your program more readable.

Going further, adding elseif allows us to choose between a number of possible courses of action.

```matlab
x = input(’ Type x : ’)
if (x<0)
    disp(’x is negative’)
elseif (x>0)
    disp(’x is positive’)
else
    disp(’x is zero’)
end
```

A more general form is

```matlab
if (condition1)
    statementsA
elseif (condition2)
    statementsB
elseif (condition3)
    statementsC
...
else
    statementsE
end
```

This is sometimes called an elseif ladder. Its effect is the following.

- First condition1 is tested. If it is true then statementsA are executed and execution then skips to the next statement after end.
• If condition1 is false, then condition2 is tested. If it is true, then statementsB are executed and execution skips to the next statement after end.

• Continuing in this way, all the conditions appearing in elseif lines are tested until one is true. If none is true, then statementsE are executed.

There can be any number of elseifs but only one else.

**Example.** Suppose a bank offers annual interest of 3% on balances of less than £5,000, 3.25% on balances of £5,000 or more but less than £10,000, and 3.5% for balances of £10,000 or more. The following program calculates an investor’s new balance after one year.

\[
\% \text{ script tutorial3d.m}
\]

```matlab
bal = input('type balance: ')
if (bal < 5000)
    rate = 0.03;
elseif (bal < 10000)
    rate = 0.0325;
else
    rate = 0.035;
end

disp(['new balance is : ',num2str((1+rate)*bal)])
```

The logical relations that make up the condition in a while or if statement can quite complicated. Simple relations can be converted into more complex ones using the three logical operators & (and), | (or) and ~ (not). For example the quadratic equation \(ax^2 + bx + c = 0\) has two equal roots, \(-b/(2a)\), provided that \(b^2 - 4ac = 0\) and \(a \neq 0\). This can be programmed as:

```matlab
if((b^2 - 4*a*c == 0)&(a~=0))
    x = - b/(2*a);
end
```

### 3.4 Exercises for Chapter 3

In the following exercises you should write script programs for each of the tasks, i.e. you should create files (using the MATLAB editor/debugger) which are saved (by clicking on Files --> Save As) in M-files (with the .m extension). These should contain the sequence of MATLAB commands necessary to do the tasks required.

1. Write a simple loop to list the squares of the first 10 integers.

2. Compute 20!. Devise two different ways, one with a for loop and one without. Find out about the MATLAB built-in function factorial and use it to check your answers. You might wish to make use of the MATLAB command prod.

3. Using the script tutorial3a.m, show by experiment that the series (3.1) sums to \(\log_e(2)\).
4. For a series of terms with alternating sign of the form \( \sum_{i=1}^{\infty} (-1)^{n+1}a_n \), where \( a_n \geq 0 \), let

\[
L = \sum_{n=1}^{\infty} (-1)^{n+1}a_n, \quad S_N = \sum_{n=1}^{N} (-1)^{n+1}a_n.
\]

There is a theorem that says that (under certain additional conditions) the absolute value of the error incurred in summing the series up to a finite number of terms is no more than the absolute value of the first term not included in the sum. With the above notation this means that, for any integer \( N \),

\[ |L - S_N| \leq a_{N+1}. \]

Investigate the correctness of this theorem for the series (3.1) by checking it for a number of (large) values of \( N \). (You will also need the MATLAB command \texttt{abs}.)

5. Using a simple \texttt{while} loop, write a script to sum the series \( 1 + 2 + 3 + \ldots \) such that the sum is as large as possible without exceeding 100. The program should display how many terms are used in the sum.

6. Write a script that takes as input an integer \( n \) and creates the \( n \times n \) matrix \( A \) with \( (i,j) \)th component given by \( A(i,j) = \sin(1/(i+j-1)) \).

7. Here is a guessing game. Use MATLAB to generate a random integer between 1 and 10. This can be done with the command:

\[
\text{matnum} = \text{floor}(10 \times \text{rand} + 1);
\]

(Use the manual pages to make sure you understand this command.)

You have to guess the random integer without printing it out. Write a script that computes the random integer as above and that also takes as input your guess and produces as output a message which tells you if your guess was correct or not, and if not whether your guess was too low or too high.

8. Modify the script \texttt{tutorial3b.m} to compute the sum of the series (3.1) for any number of terms (even or odd). Use an \texttt{if} statement.

9. Write a script that takes as input three numbers \( a, b \) and \( c \) and prints out either the solutions of the quadratic equation \( ax^2 + bx + c = 0 \), when these solutions are real, or a message indicating that the solutions are not real.

4 Fourth Tutorial: scripts and functions

As you have seen, there are two ways of using MATLAB: (i) \textit{interactively}, where you type a series of commands at the command line, and (ii) by writing programs in which instructions are saved in an \textit{M}-file, a file with the \textit{.m} extension. There are two types of MATLAB programs: \textit{scripts} and \textit{functions}. A script file is run by typing its name (without the \textit{.m} extension) to the MATLAB prompt. We have called (run) built-in functions like \texttt{exp} by writing e.g. \( y = \text{exp}(1) \). In this tutorial you will learn

- how to write new functions,
- how to call (or run) them,
- how scripts and functions differ in their construction, and
- how they differ in their handling of variables.
4.1 The MATLAB workspace

You have seen by now that when you use MATLAB you create variables in your workspace. At any stage of a MATLAB session you can type

```
who
```
to find out what variables are in your current workspace. A variation on this is the command `whos` which gives more information. Look up the help pages for these commands.

Variables that are lying around in your current workspace are typically called global variables. When you type a command to the `>>` prompt that involves any symbol, MATLAB first looks to see whether this symbol is the name of a variable in your current workspace. If so, it uses the current value of that variable in the computation. If not, it looks to see whether the symbol is the name of a built-in constant, in which case it uses the built-in value. If not, it looks to see whether the symbol is a script or function name, in which case it runs it. Finally, if the symbol is none of these things, an error message will usually result.

For example, if we start a new MATLAB session and to the prompt we type `pi`, the response is

```
ans = 3.1416
```

and in response to the command `3*pi`, we get

```
an = 9.4248
```

However, if we are feeling perverse, we may choose to use the symbol `pi` as the name of a new variable, with a different value. For example we may type to the prompt: `pi = 3`. After doing this the response to the command `3*pi` will be

```
ans = 9
```

On the other hand if we type (for example by accident) `pii`, we get the response:

```
?? Undefined function or variable 'pii'.
```

This illustrates that whenever you type a MATLAB command interactively, all your current variables are visible and may be operated on by the command that you type.

4.2 Script files

All the MATLAB programs that I have written for you so far have been scripts. These consist simply of a sequence of MATLAB commands that are saved to a file. When the script is run the effect is exactly the same as is achieved by simply typing the commands interactively to the MATLAB prompt.

These commands operate on the variables in your current workspace in the same way as they would if you typed them interactively.

Script files can also create new variables which may be left lying around in your workspace after the script has finished running.

4.3 Function files

By using functions you can create much more flexible MATLAB programs than would be possible just with scripts. In a function file you specify an input or inputs and generate an output or outputs. If you call a function from the MATLAB prompt then the input(s) and output(s) are part of your MATLAB workspace, but any other variables that are created inside the function itself are local to the function and will not be seen
from your MATLAB workspace. This is useful in many ways; in particular it reduces the build-up of variables in your workspace and aids in splitting up a complex task into simpler independent tasks.

All programming in high-level languages makes use of such splitting techniques.

Here is an example of a function file called `bell.m`, which calculates the bell-shaped function \( f = \exp(-ax^2/2)/\sqrt{2\pi} \) for given scalar inputs \( x \) and \( a \).

```matlab
% function bell.m

function f = bell(x,a)
    f = exp(-0.5*a*(x^2))/sqrt(2*pi);
```

One of the important rules for functions is that they must be saved to a file which has the same name as the name of the function. Thus the above function should be saved to the file `bell.m`.

To run this, for example, type

```
bell(1,2)
```

The response is

```
ans =
     0.1468
```

This is less useful than it might be. If, for example, we want to draw a plot of this function it would be much better if we could feed a vector of \( x \) values to the function `bell` and get out a whole vector of function values. A small change to the definition allows this:

```matlab
% function vbell.m

function f = vbell(x,a)
    f = exp(-0.5*a*(x.^2))/sqrt(2*pi);
```

The only difference between this and `bell.m` is that by employing the elementwise exponentiation `.^`, the variable \( x \) can be a vector and `vbell` produces a vector output. The output is obtained by typing (either interactively or inside a script):

```
vbell(x,a)
```

which gives a vector of values of the bell function evaluated at each of the entries of \( x \). An application of this to plot the bell function is contained in the following script:

```matlab
%script tutorial4a.m

a = 1;

x = (0:1000)/100;  \% equally spaced points between 0 and 10
plot(x,vbell(x,a),'b*')
```

The effect will be to plot the function `bell` at 1001 equally spaced points between 0 and 10 and with \( a \) set to be 1.

Copy `bell.m`, `vbell.m` and `tutorial4a.m`. Run `tutorial4a` and afterwards type `whos`. Note that the variable \( f \) which appears in the function `vbell` is local to that function and does not appear in your workspace.

What happens if you replace the call to `vbell` inside the plot command in `tutorial4a.m` with a call to `bell`?
4.4 The general form of a function file

Functions are incredibly useful and need considerable care when they are implemented.
A function file always begins with a function definition line of the form

\[
\text{function [output\_variables] = functionname(input\_variables)}
\]

Here \text{functionname} must be the same as \text{filename}, where \text{filename.m} is the name of the file in which the function is stored. The \text{output\_variables} are variables whose values may be determined by the function, and which may depend on the variables in the list of \text{input\_variables}.

To use the above function (e.g. in a script or interactively) you would type

\[
\text{[my\_output\_variables] = functionname(my\_input\_variables)}
\]

where \text{my\_input\_variables} are the actual input variables which you want to apply the function to and \text{my\_output\_variables} are the names of the variables where you would like to put the result.

There are a number of rules about constructing functions that I shall not attempt to reproduce here, but authors of advanced functions need to be aware of them. Type \text{help function} or \text{doc function} for more information.

Perhaps the best way to learn about functions is to look at functions that other people have written. Many (but not all) of the functions provided in \text{MATLAB} are given as \text{M}-files that you can copy and use for your own purposes. Try typing the command

\text{type factorial}

to see the listing of the \text{MATLAB}-provided function that produces the factorial of an integer. Repeat this with the functions \text{why}, \text{pascal} and \text{roots}. There are many others that you can look at.

Many contributed functions that can be freely copied are also to be found at the \text{MATLAB} website \text{http://www.mathworks.com}.

4.5 Exercises for Chapter 4

1. Write and test a function \text{twice(x)} that doubles the value of its input variable (i.e. the statement \(x = \text{twice(x)}\) should double the value of \(x\)).

2. Write and test a function \text{swop(x,y)} that swops the values of its input variables, (i.e. the statement \([x,y] = \text{swop(x,y)}\) should swop the values of \(x\) and \(y\)).

3. Write your own \text{MATLAB} function to compute the exponential function using the \text{Taylor series} expansion:

\[
e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots
\]

Remember to give your function a name that is \text{different} from the \text{MATLAB} built-in function \text{exp}. Make your own decision about when to stop adding more terms from the expansion. Compare the performance of the function that you have written with the built-in function \text{exp}.

4. If \(a\) is a positive number, then it is known that the sequence \(y_k, \ k = 0, 1, 2, \ldots\), generated iteratively by

\[
y_0 = a, \quad y_{k+1} = \frac{(y_k + a/y_k)/2}{2}, \quad k = 0, 1, \ldots,
\]

has the property that \(y_k\) converges to (i.e. gets closer to) \(\sqrt{a}\) as \(k\) increases. (In fact this is Newton’s method for finding the square root of a number, which you may have seen before.)

Use this given iteration to design your own function for computing the square root of any number and test it against the \text{MATLAB} built-in function \text{sqrt}.

(In fact many computers and calculators use such an iteration in their built-in \text{sqrt} functions.)
5 Fifth Tutorial: A Population Dynamics Example

In this tutorial you will use the skills you have developed to investigate a program on Cellular Automata.

5.1 Cellular Automata

Cellular automata (CA) provide a simplified way of modelling physical, biological or social phenomena. In a CA model the universe is divided up into small cells. Each cell can take any one of a finite number of states (e.g. live or dead, or perhaps something more complicated). A single generation is determined by assigning a single state to each of the cells. Then a strict (and often quite simple) set of rules is used to determine how a given generation evolves into the next one. The model then consists of repeated iterative application of the rules to obtain each successive generation from the previous one. In principle this iteration can go on forever; in computer implementations it goes on for a finite (possibly quite large) number of generations. There is much interest in CA models among mathematicians and applied scientists such as ecologists or biologists. This arises from the often observed fact that a simple model with relatively few states and relatively simple rules can often produce very realistic and complicated behaviour. There is also a huge interest amongst computer scientists, especially those concerned with the theory of artificial intelligence. Let us consider a flat square universe, divided up into an \( n \times n \) array of small cells, denoted \((i,j)\), for \(i,j = 1,\ldots,n\). The states of these cells may be represented as the elements \(A(i,j)\) of a matrix \(A\), for \(i,j = 1,\ldots,n\). The immediate neighbours of cell \((i,j)\) are defined to be those cells that touch it (excluding cell \((i,j)\) itself). So an interior cell \((i,j)\) (where \(i = 2,\ldots,n-1\) and \(j = 2,\ldots,n-1\)) has immediate neighbours \((i-1,j-1), (i-1,j), (i-1,j+1), (i,j-1), (i,j), (i,j+1), (i+1,j-1), (i+1,j)\) and \((i+1,j+1)\) (see Fig. 1). With this universe we can consider different types of states for the cells and different sets of rules for determining the model.

![Figure 1: Immediate neighbours of an interior cell \((i,j)\)](image)

5.2 A simple example

Let us suppose that each cell can occupy only two states (we shall say alive or dead, but other interpretations are of course possible). We may represent these by setting \(A(i,j) = 1\) when the cell \((i,j)\) is alive and \(A(i,j) = 0\) when it is dead. To decide how each generation is determined from the previous one, let us take the simple example of majority vote:

- A live cell dies in the next generation if more than 4 of its immediate neighbours at the present generation are dead; otherwise it stays alive.
- A dead cell comes to life in the next generation if more than 4 of its immediate neighbours at the present generation are alive; otherwise it stays dead.

Since the computer is finite, our universe has to end somewhere. To apply these rules consistently, we need to define the states of the cells that lie just outside the boundary of the universe. For the purpose of this example, we shall say that
• All cells outside the boundary of our universe are assumed to be dead.

This means that we can effectively extend the boundary of our universe to include cells \((0, j), (n + 1, j), j = 0, \ldots, n + 1\), and \((i, 0), (i, n + 1), i = 0, \ldots, n + 1\), all of which are defined to be dead. To compute the next generation from any given one, we can then proceed as follows:

• For each cell \((i, j), i, j = 1, \ldots, n\), compute the number of live neighbours and store these in another matrix, say \(L(i, j)\). Take particular care to get the computation correct at the edges of the universe.

• Then, using the information in the matrices \(A\) and \(L\), the next generation of live and dead cells can be computed by applying the two rules above. Since each generation is computed only from the previous one, the new generation should be computed in a separate matrix, say \(ANEW\), and the old \(A\) should only be overwritten with \(ANEW\) when all of \(ANEW\) has been found.

Many successive generations can be computed by using a suitable loop.

5.3 A programming exercise

Suppose we are asked to write a program to implement and visualise 1000 generations of the population model in the previous subsection, starting from a random initial state of ones and zeros. A suitable pseudocode would be:

1. Read in \(n\), the number of cells in each direction.

2. Create an initial \(n \times n\) random distribution \(A\) containing only 0’s and 1’s.

3. Visualise \(A\).

4. Initialise \(ANEW\) and \(L\) to be matrices of zeros of the same size as \(A\) (it turns out to be convenient, but not essential, to define \(ANEW\) and \(L\) to be \(n + 2 \times n + 2\) matrices - see below).

5. Then enter a loop to compute successive generations. (The variable \(gen\_no\) will count the number of generations which have been computed.)

• for \(gen\_no = 1 : 1000\)

  (a) Implement the boundary condition by adding a border of zeros to \(A\):

  \[
  A = \begin{bmatrix}
  0 & \ldots & 0 \\
  \cdot & \cdot & \cdot \\
  \cdot & A & \cdot \\
  \cdot & \cdot & \cdot \\
  0 & \ldots & 0
  \end{bmatrix}
  \]

  \(A\) is then an \((n + 2) \times (n + 2)\) matrix.

(b) For \(i, j = 2, \ldots, n + 1\), compute the number of live cells that are nearest neighbours of cell \((i, j)\). Put it in the matrix entry \(L(i, j)\).

(c) To find \(ANEW\), initialise by setting \(ANEW = A\). Then change some entries of \(ANEW\) as appropriate:

(d) For \(i = 2, \ldots, n + 1\) and \(j = 2, \ldots, n + 1\),

- If \(A(i, j) = 1\) and \(L(i, j) < 4\) set \(ANEW(i, j) = 0\);
- If \(A(i, j) = 0\) and \(L(i, j) > 4\) set \(ANEW(i, j) = 1\).

(e) Create a new \(n \times n\) matrix \(A\) that consists of the interior values of \(ANEW(i, j)\), for \(i, j = 2, \ldots, n + 1\).
(f) Visualise $A$.

- end of for loop

The MATLAB code for this pseudocode can be found at `~masigg/ma20014/matlab/manual/majority.m`. In order to visualise each generation, a small MATLAB function `lookat.m` is also provided. It draws a blue dot for every live cell and leaves the dead ones blank. It incorporates the `drawnow` command and so supports animated plots. It labels the plot with the generation number (`gen_no`), which is one of the inputs to `lookat.m`. Note that `lookat` visualises the matrix $A$ by plotting the $j$ index on the vertical axis and $i$ on the horizontal, with the origin at the bottom left-hand corner.

Note also that the program `majority.m` has used the dynamic storage facility in MATLAB extensively. In the program the matrix $A$ switches from being $n \times n$ to $(n+2) \times (n+2)$ and back again during each cycle of the `for` loop. This dynamic storage facility is not present in all programming languages and is a feature which can be used here to simplify the coding substantially.

### 5.4 Exercises for Chapter 5

1. Play with the programs `majority.m` and `lookat.m` until you understand them fully. Remember that by removing a semicolon at the end of a line and adding a `pause` statement you can see the result of each computation line by line. This will help you understand the program, but note that printing the matrix $A$ on the screen will be helpful only for small values of $n$. Here are some further hints to help you gain some understanding.

2. Look up the help page for the command `round`. Explain to yourself how the initial random distribution of 1’s and 0’s is computed in `majority.m`.

3. Look up the help page for the command `size`. By putting the line `size(A);pause;` at appropriate places in the program `majority.m`, illustrate to yourself the dynamic storage mentioned above.

4. For small $n$ (e.g. $n = 4$) print out the matrix $A$ and the corresponding matrix $L$. Check manually that the program is doing the correct computation.

5. Perform similar investigative work on `lookat.m`. What happens if the `drawnow` command is removed? Experiment with different plot colours and point markers. More detailed information on `plot` can be got by typing `doc plot`.

### 6 Sixth Tutorial: Further Experiments on Population Dynamics

This tutorial extends tutorial 5 from investigation to modification of the CA program.

1. Run the program `majority.m` a few times. Try different values of $n$ and different random initial states. You should observe that a very irregular random initial state is smoothed out by the birth and death rules, and eventually arrives at a pattern of blue and white regions with quite smooth interfaces between them. (Some discussion of this can be found in *What’s happening in the Mathematical Sciences*, by B. Cipra, American Mathematical Society, Providence, 1996, pages 70-81.)

2. Rewrite the program `majority.m` so that it reads in an integer $p$ as a variable and then replaces the majority rule above with a variable rule:

- A live cell dies in the next generation only if more than $p$ of its immediate neighbours at the present generation are dead; otherwise it stays alive.
• A dead cell comes to life in the next generation only if more than p of its immediate neighbours at the present generation are alive; otherwise it stays dead.

Perform experiments with your program to see how the choice of p changes the outcome.

3. Modify the program majority.m to model the case where all cells outside the boundary of the universe are assumed to be live instead of dead.

Think of further generalisations of the rules and add them to your program. The next item is one suggestion.

4. Consider a modified population model where the fate of a cell is determined not just by its 8 immediate neighbours but also by their immediate neighbours, (excluding the initial cell, of course), a total of 24 cells in all. Call these cells the near-immediate neighbours and replace the majority rule by the following.

• A live cell dies in the next generation only if more than 12 of its near-immediate neighbours at the present generation are dead; otherwise it stays alive.
• A dead cell comes to life in the next generation only if more than 12 of its near-immediate neighbours at the present generation are alive; otherwise it stays dead.

Rewrite the program to implement and visualise this model.

5. Can you devise an alternative version of the program majority.m where the matrices A, L and ANEW are always $n \times n$? Write such a program and test it by comparing the results with those produced by majority.m.

7 The Matlab Symbolic Math Toolbox

This tutorial investigates the symbolic algebra capabilities of MATLAB. You will learn

• how to use MATLAB to solve algebraic equations and systems of equations,
• an easy way to plot graphs and surfaces given by formulae,
• how to use MATLAB to differentiate and integrate, and
• how to use MATLAB to solve differential equations.

Up till now all the computation we have done has been numerical or floating point computation. Numbers which do not have a finite decimal expansion are approximated by finite decimal expansions, creating a (small) error which remains throughout the computation. This error is sometimes referred to as rounding error. An example of rounding error was given in Example 2 of Chapter 1 of the MATLAB manual.

Although much computation done in science and technology is of floating point type, there also exist some computational systems that do not suffer from rounding error. Mathematical formulae, equations etc are manipulated symbolically without approximation. MATLAB provides a toolbox (the Symbolic Math Toolbox) that enables this type of computation. This is not available automatically with MATLAB - usually it has to be purchased separately. The UNIX MATLAB system on BUCS has a large number of Symbolic Math toolboxes available for use. These toolboxes are based on the Maple symbolic manipulator (see http://www.maple.com). Maple is also available as a stand-alone package on BUCS UNIX machines (type xmaple to the $prompt).

We choose to introduce this type of computation via the MATLAB Symbolic Math Toolbox because students will already be familiar with the MATLAB computational environment and hence should be able to progress more quickly than in a completely new environment. The student who feels confident with the MATLAB Symbolic Math Toolbox is invited to progress on to working on Maple proper in order to gain further experience with this type of computation.
7.1 Exercises on Chapter 7

1. Copy the file ~masigg/ma20014/matlab/manual/tutorial7.m to your workspace and run it.

2. Type `demo` to the `>>` prompt, double-click in the Toolboxes item and run the demo on the Symbolic Math Toolbox. It might take some time.

3. Read the help pages for the commands `solve` and `ezplot`. Let \( f \) be the cubic \( f(x) = x^3 + 6x^2 + 9x + 2 \). Solve the equation \( f(x) = 0 \), and plot a graph of \( f \). It should be clear from the graph that the cubic has three real roots.

4. Find out how `solve` can be used to solve simultaneous equations. Hence solve the three simultaneous equations 
   \[ 
   x + 2y + 3z = 1, 
   4x + 5y + 6z = 2, 
   7x + 8y + 9z = 3. 
   \]  
   What happens if the last equation is changed to \( 7x + 8y + 9z = 3 \)? What happens if it is changed again to \( 7x + 8y + 9z = 4 \)? Can you explain the results?

5. Find out how to use the symbolic differentiation command `diff` and the substitution command `subs`. (NB: there is also a `diff` command in vanilla MATLAB; you must therefore type `help sym/diff`.) Consider the function \( f(x) = \sin(\tan(x)) - \tan(\sin(x)) \). Find the derivatives of \( f \), \( df(x)/dx^i \), for \( i = 1, \ldots, 6 \). Evaluate \( f \) and each of its derivatives at \( x = 0 \). What do you find? Use the command `ezplot` to draw a graph of this function. Does the graph support your computations?

6. Look up the help page for the integrator `int`. Find the indefinite integral of the function \( \sqrt{\tan(x)} \). Use the command `pretty` to make a more readable output.

7. Read the help pages for the differential equation solver `dsolve`. Use it to solve the differential equation 
   \[ 
   \frac{d^2y}{dt^2} + y = 0 \]  
   for \( y = y(t) \). The solution will contain two arbitrary constants \( C_1 \) and \( C_2 \). Find the particular solution of this differential equation which satisfies \( y = 1 \) and \( dy/dt = 1 \) at \( t = 0 \). Plot a graph of the solution you have found. Repeat the exercise with the differential equation replaced by 
   \[ 
   \frac{d^2y}{dt^2} + 100y = 0. 
   \]

8. Read the manual page for `ezsurf`. Plot the function \( 1/\sqrt{x^2 + y^2} \).

A Appendix: Solutions to selected exercises

All the programs mentioned below can be found in the directory ~masigg/ma20014/matlab/manual and copied using the UNIX command `cp`

Chapter 3

7. Program in `guess.m`

9. Program in `quadratic.m`

Chapter 4

1. Solution in `twice.m`
   To compute twice 2, for example, type `twice(2)`.
2. Solution in swop.m.
   To swop the the order of the numbers 1 and \( \pi \), type \([a_1,a_2] = \text{swop}(1,\pi)\). The swopped numbers will be loaded into the variables \( a_1 \) and \( a_2 \).

3. Solution in myexp.m.
   To compute an approximation to \( e^4 \), type \( \text{myexp}(4) \). This function will only agree with the MATLAB built-in function \( \text{exp} \) up to about 7 decimal places. What happens if we compare \( \text{exp}(-20) \) with \( \text{myexp}(-20) \)?

4. Solution in mysqrt.m.
   To compute an approximation to \( \sqrt{\pi} \), type \( \text{mysqrt}(\pi) \). Note that the program also takes care of the cases \( a < 0 \) (when there is no real answer) and \( a = 0 \) (when the iteration fails but the answer is 0). The MATLAB commands \text{error} \ and \text{break} \ are used in the handling of these cases.

B Appendix: Glossary of UNIX commands

Note that the \textit{spaces} \ (marked with a \( \_ \)) in the statements of these commands are crucial. You will get errors if you leave them out.

Creating a file: Type \texttt{touch}_{\_}filename \ to create the file \texttt{filename}. If this file already exists then the effect of the command will be to change the time when it was last updated to the present time. \( \text{(You can also create a file using an editor such as pico; type pico}_{\_}filename.}\)

Copying a file: Type \texttt{cp}_{\_}~masigg/ma20014/matlab/manual/tutorial1.m \texttt{~ma****} \ to copy the file named \texttt{tutorial1.m} in masigg’s directory ma20014 to the home directory of ma****. If you are in \texttt{~masigg/ma20014/matlab/manual}, just type \texttt{cp}_{\_}tutorial1.\texttt{m} \texttt{~ma****}.

Listing a file: Type \texttt{more}_{\_}tutorial1.m \ to list the contents of the file \texttt{tutorial1.m} on the screen. You must be sitting in the directory in which \texttt{tutorial1.m} is stored. If not you must type the full name of the file, \( \text{e.g.} \texttt{more}_{\_}~masigg/ma20014/matlab/manual/tutorial1.m \).

Printing a file: Type \texttt{lwf}_{\_}tutorial1.m \texttt{| lpr}_{\_}P11 \ to print out \texttt{tutorial1.m} on the library laser printer.

Removing a file: Type \texttt{rm}_{\_}rubbish.m \ to remove the file \texttt{rubbish.m}. \( \text{(Make sure you get it right, as you will not get a chance to change your mind after removing it!)} \)

Finding what directory you are in: Type \texttt{pwd} \ to find your current working directory.

Listing the contents of a directory: Type \texttt{ls} \ to get the names of all the files in the current directory. Type \texttt{ls}_{\_}-l \ to get a listing of the files together with their read, write and execute permissions.

Creating a directory: Type \texttt{mkdir}_{\_}directoryname \ to create a subdirectory \texttt{directoryname} of the current directory. \( \text{(A directory in UNIX is analogous to a folder in Windows/DOS.)}\)

Changing directory: Type \texttt{cd}_{\_}~masigg \ to change to the home directory of user masigg. To change to directory ma20014, a subdirectory of masigg’s home directory, type \texttt{cd}_{\_}~masigg/ma20014. Type \texttt{cd} \ to change to your own home directory. \( \text{If you’re in your home directory, type cd}_{\_}subdir1 \text{ to change to subdir1, a subdirectory of your home directory.} \)

Changing the permissions on a file or directory: Type \texttt{chmod}_{\_}u+x_{\_}filename \ to add execute \( (x) \) permission for the owner \( (u) \). Other versions of the same command can be constructed by replacing \( u \) by \( g \) (group) or \( o \) (others) or replacing \( x \) by \( r \) (read) or \( w \) (write). Change \( + \) to \( - \) to take away the permission rather than add it.
Appendix: Some useful MATLAB commands

**On-line help**
- `help` lists topics on which help is available
- `helpwin` opens the interactive help window
- `helpdesk` opens the web-browser-based help facility
- `help topic` provides help on topic
- `lookfor string` lists help topics containing string
- `demo` runs the demo program

**Workspace information and control**
- `who` lists variables currently in the workspace
- `whos` as above, giving their size
- `what` lists M-files on the disk
- `clear` clears the workspace, removing all variables
- `clear all` clears all variables and functions from the workspace

**Command window control**
- `clc` clears command window, command history is lost
- `home` same as `clc`
- ↑ recall previous command

**Graphics**
- `plot` plots a graph
- `xlabel('x')` labels x axis x
- `ylabel('y')` labels y axis y
- `title('title')` gives a figure a title title
- `axis` fixes figure axes
- `clf` clears figure from figure window
- `cla` clears figure from figure window, leaving axes

**Controlling program execution**
- `break` terminates execution of a for or while loop
- `error('message')` aborts execution, displays message on screen
- `return` exit from function, return to invoking program

**Input from and output to terminal**
- `x=input('type x:')` asks user to give a value to be assigned to x
- `disp('string')` outputs string to terminal

**String-number conversion**
- `num2str` converts a number to a string (so it can be output e.g. as part of a message)

**Logical functions**
- `isempty` true (=1) if a matrix is empty
- `find` finds indices of non-zero elements of a matrix

**Arithmetic functions**
- `sum(x)` calculates the sum of the elements of the vector x
- `prod(x)` calculates the product of the elements of the vector x

**Termination**
- `^c` (Control-c) local abort, kills the current command execution
- `quit` quits MATLAB
- `exit` same as `quit`
This section contains more advanced information on MATLAB programming style. It is designed to illustrate performance improvements that may be obtained by replacing operations on scalars with operations on vectors, wherever possible.

When you run a MATLAB script or function program by typing its name, it is automatically interpreted into a program written in the language C. This program is compiled, linked, and run, all without you having explicitly to ask for this. This is tremendously helpful and means that you can develop programs in a fraction of the time it takes to develop the corresponding C code. But you may pay a price for this in the run time of certain types of program. In particular, because of the way the interpreter works, loops created with the MATLAB code (using the for, while or if commands) may run slowly, especially if they contain a lot of arithmetic calculations.

Thus it is a very good idea to avoid such loops in your program where at all possible. It is not always possible to avoid them, but there are a surprising number of ways around the use of loops which may not be obvious at first glance. The secret of good MATLAB programming is to try to identify situations when the same arithmetic operations are being applied to a whole vector of numbers (or maybe even a matrix of numbers). MATLAB contains many highly optimised built-in functions that allow you to operate on all the elements of vectors and matrices with one command. These can be used to achieve effects that you may first think can only be done using loops. Because you are using the highly optimised built in-functions your program can run very much faster than if you wrote everything elementwise inside loops.

This process of vectorising your code is not only relevant to the production of efficient MATLAB code, but is also useful for coding on certain special types of computer architecture that can exploit it.

The purpose of this section is to give you a sequence of examples illustrating these techniques.

Examples 1 and 2: Numerical integration using the composite trapezoidal rule.
To integrate a function \( f \) over the interval \([0, 1]\), choose an integer \( n > 0 \) and set \( h = 1/n \) and \( x_i = ih \), \( i = 0, \ldots, n \). Then the composite trapezoidal rule is constructed by applying the trapezoidal rule on each subinterval \([x_{i-1}, x_i]\) to get:

\[
\int_0^1 f(x) \, dx \approx \frac{1}{2} hf(x_0) + h \sum_{i=2}^{n-1} f(x_i) + \frac{1}{2} hf(x_n) .
\]

Computing the right-hand side involves doing the sum in the middle and then adding on the first and last terms. Apparently the simplest way of doing this is by using a for loop. This is illustrated in program ex1.m, in which we have also added a couple of lines to count and print out the number of flops (floating point operations) and cpu time (time taken by the processor) for the program. In this case we use the program to compute the integral of \( \exp(x) / \pi \). Here is a sample output from ex1.m:

<table>
<thead>
<tr>
<th>n</th>
<th>flops</th>
<th>cputime</th>
<th>approximation to integral</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>2503</td>
<td>0.03</td>
<td>0.856469345582</td>
</tr>
<tr>
<td>1000</td>
<td>5003</td>
<td>0.06</td>
<td>0.856469323888</td>
</tr>
<tr>
<td>2000</td>
<td>10003</td>
<td>0.10</td>
<td>0.856469318464</td>
</tr>
<tr>
<td>4000</td>
<td>20003</td>
<td>0.19</td>
<td>0.856469317108</td>
</tr>
<tr>
<td>8000</td>
<td>40003</td>
<td>0.40</td>
<td>0.856469316769</td>
</tr>
<tr>
<td>16000</td>
<td>80003</td>
<td>0.81</td>
<td>0.856469316685</td>
</tr>
</tbody>
</table>

Note that the cputime may vary a little each time the program is run, due to the timesharing property of the machine. The cputimes reported are the averages of three consecutive experiments on the BUCS machine amos. Here is the listing for the program ex1.m:
Program ex1.m: script file to integrate \( e^{x}/1\pi \) over \([0,1]\) using the composite trapezoidal rule on a uniform mesh with \( n \) subintervals and using a loop to do the adding up

```matlab
clear % remove any old variables
format long % prints out unformatted numbers to 12 digits
n = input('type n: ') % input n from the keyboard
h = 1/n; % uniform mesh
flops(0); % sets flop counter to 0
t = cputime; % records cpu time

% In the formula for the composite trapezoidal rule the first and last terms are weighted by \( h/2 \):
integral = (h/2)*(exp(0)^(-1/pi) + (exp(1))^(-1/pi));

% The remaining terms are weighted by \( h \) and can be done in a loop:
for j = 1:n-1 % repeats what's inside the loop for j = 1,..,n
    integral = integral + h*((exp(j*h)^(-1/pi));
end % end of for loop

fprintf('
 flops taken: %8.0f, CPUtime taken: %16.6g: 
',flops,cputime-t)
fprintf('
 Approximate integral : %16.12f
',integral)

% end of program ex1.m
```

Note that the 1D trapezoidal rule with 4000 points is extremely accurate. We use such a large number of points merely to illustrate the efficiency of the program.

The program ex1.m contains one loop inside which most of the computation is done. (Exercise: edit the program so that it prints out the number of flops done inside and outside the for loop.) The loop is used simply to add up the terms in the sum in the trapezoidal rule formula. A better way to do this is to use the MATLAB built-in function \texttt{sum}. (Exercise: look up the help page for the command \texttt{sum}.) This command is used in the program ex2.m, which is given as follows.

Program ex2.m: script file to integrate \( e^{x}/1\pi \) over \([0,1]\) using the composite trapezoidal rule on a uniform mesh with \( n \) subintervals and use the built-in function \texttt{sum} to do the adding up

```matlab
clear % remove old variables
format long % prints out unformatted numbers to 12 digits
n = input('type n: ') % input n from the keyboard

% In the formula for the composite trapezoidal rule the first and last terms are weighted by \( h/2 \):
integral = (h/2)*(exp(0)^(-1/pi) + (exp(1))^(-1/pi));

% The remaining terms are weighted by \( h \) and can be done in a loop:
for j = 1:n-1 % repeats what's inside the loop for j = 1,..,n
    integral = integral + h*((exp(j*h)^(-1/pi));
end % end of for loop

fprintf('
 Approximate integral : %16.12f
',integral)

% end of program ex2.m
```
\[ h = \frac{1}{n}; \quad \% \text{uniform mesh} \]

\[ \text{flops}(0); \quad \% \text{sets flop counter to 0} \]
\[ t = \text{cputime}; \quad \% \text{records cputime} \]

\% First and last terms are done manually

\[ \text{integral} = \frac{h}{2} \left( \frac{\exp(0)}{-1/\pi} + \frac{\exp(1)}{-1/\pi} \right); \]

\% the vector \((1:n-1)*h\) gives the \(n-1\) equally spaced points inside the interval \([0,1]\)
\% we need to evaluate the function \((\exp)^{-1/\pi}\)
\% at each of these points and sum up the answers

\[ \text{integral} = \text{integral} + h*\text{sum}((\exp((1:n-1)*h))^{(-1/\pi)}); \]

\% end program ex2.m

Here is a sample output from program ex2.m. Note that the use of the built-in function \texttt{sum} has made this program more than ten times as efficient as program ex1.m. Note that the flop count is also considerably lower for program ex2.m, although the difference to ex1.m on this indicator is not nearly as great as for the cputime.

<table>
<thead>
<tr>
<th>(n)</th>
<th>\texttt{flops}</th>
<th>\texttt{cputime}</th>
<th>approximation to integral</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>8007</td>
<td>0.01</td>
<td>0.856469318464</td>
</tr>
<tr>
<td>4000</td>
<td>16007</td>
<td>0.02</td>
<td>0.856469317108</td>
</tr>
<tr>
<td>8000</td>
<td>32007</td>
<td>0.03</td>
<td>0.856469316769</td>
</tr>
<tr>
<td>16000</td>
<td>64007</td>
<td>0.05</td>
<td>0.856469316685</td>
</tr>
</tbody>
</table>

\textbf{Examples 3 and 4: Assembly of the Hilbert matrix.}

The \(n \times n\) Hilbert matrix, given by

\[ H_{i,j} = \frac{1}{i+j-1}, \quad i, j = 1, \ldots n \]

is a famous example of a highly ill-conditioned matrix.

Suppose we wanted to write a program to assemble the Hilbert matrix for general \(n\). Because matrices have two indices (one each for the rows and columns) it is natural to do this using two nested loops as in the program ex3.m:

\% program ex3.m : script to assemble the \(n \times n\) Hilbert matrix

\[ n = \text{input}(''type n: ''); \]
\[ H = \text{zeros}(n,n); \quad \% \text{Initialise H} \]
\[ t = \text{cputime}; \]
for i = 1:n % Use a double do loop to assemble H
for j = 1:n
H(i,j) = 1/(i+j-1);
end % end of j loop
end % end of i loop

fprintf('\n time for assembling H: %16.8g\n',cputime-t)

% end of program ex3.m

Because of what we have learned from Examples 1 and 2 we may reasonably suspect that this program will not be the most efficient way of handling the assembly of $H$ because of the double loop inside which computation is done. The program ex4.m addresses this problem by using built-in functions to assemble $H$ and avoiding loops entirely.

% program ex4.m: script to assemble the n x n Hilbert matrix

n = input('type n: ')
H = zeros(n,n); % Initialise H
t = cputime;
H = ones(n,n)./(((1:n)'*ones(1,n) + ones(n,1)*(1:n) - ones(n,n));

fprintf('\n time for assembling H: %16.8g\n',cputime-t)
% end of program ex4.m

Exercise: check that you understand why program ex4.m has the same output as ex3.m. In doing so it is very important to note that MATLAB distinguishes between a $1 \times n$ row vector and an $n \times 1$ column vector. The product of the former times the latter (using the product function *) is a scalar, whereas the product of the latter times the former is an $n \times n$ matrix!

The relative performance of programs ex3.m and ex4.m is illustrated in the following table:

<table>
<thead>
<tr>
<th>n</th>
<th>cputime (ex3.m)</th>
<th>cputime (ex4.m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.36</td>
<td>0.01</td>
</tr>
<tr>
<td>200</td>
<td>1.41</td>
<td>0.05</td>
</tr>
<tr>
<td>400</td>
<td>5.56</td>
<td>0.31</td>
</tr>
<tr>
<td>800</td>
<td>21.59</td>
<td>1.38</td>
</tr>
</tbody>
</table>

Note that in both columns the cputime is growing with $O(n^2)$, but that the asymptotic constant in this estimate is about 16 times bigger for ex3.m than for ex4.m. Put more simply, ex4.m is about 16 times faster than ex3.m.

Of course a time of 22 seconds may be reasonable if your only aim was to compute the 800 x 800 Hilbert matrix. However, if processes with loops lie inside programs that are called many times over, it is clear that a quite severe efficiency loss may be experienced. This can be alleviated by using the built-in matrix and vector functions where possible.

Exercise: look up the MATLAB built-in function toeplitz. Can you find a method for assembling the $n \times n$ Hilbert matrix that is more efficient than ex4.m? Exercise: look up the MATLAB built-in function condest. Find an estimate for the condition number of $H$ when $n = 5, 10$ and $20$. 

30