A GUIDE TO MATLAB

Fifth Edition

Roger Herz-Fischler

Ce livre est dédié
à la mémoire de mon ami

Albert Badrikian

mathématicien, homme d'esprit, sportif,
et amateur de bons mots

Mzinhigan Publishing
ᒪᓯᓇᐃᑲᐣ

Canadian Cataloguing in Publication Data

Herz-Fischler, Roger

A guide to MATLAB, fifth edition

ISBN 0-9693002-4-7

1. MATLAB (Computer file). 2. Numerical analysis–Data Processing. I. Title

Published by

MZINHIGAN PUBLISHING

340 Second Avenue, Ottawa, Ontario

Canada, K1S 2J2

ᒪᓯᓇᐃᑲᓐ

MZINHIGAN is the word for "book" in the Odawa (Ottawa) dialect of Ojibwa. The logo is "Mzinhigan" written in the related Algonquin language of Cree—as spoken on the west coast of James Bay—using Cree syllabics.

HOW TO USE THIS GUIDE

Before turning the page to the table of contents you should consider the following options:

1. You want to see immediately how MATLAB will solve equations or do eigenvalue problems:

   Type matlab at the system prompt.
   Turn to sections III.1 and/or III.2 and type in the examples.

2. You are a first time user and you want to have a good idea of how MATLAB can do straightforward matrix type calculations:

   Do the sample MATLAB session in section I.1 (10 - 15 minutes).
   Refer to the table in section II.3 for the MATLAB operations of interest to you.
   Turn to sections III.1, III.2, III.3 for sample sessions dealing with equations, eigenvalues and matrix models.

3. You want to learn more about MATLAB

   At the beginning of each section there is a summary together with a quick reference for the commands. The summary is followed by a more detailed explanation of the commands, together with illustrative examples and programs.

   Read sections I.2, I.3, II.1, II.2.
   Read sections I.4, I.5, II.4, II.5.
   Read sections II.7, II.8.

4. You are mainly interested in graphing.

   The section on the graphing of vectors and matrices (II.6) has been made as independent as possible of the rest of the guide.
   Graphing of user defined functions (section II.7) requires a knowledge of sections I.3 and II.1.

INTRODUCTION

"Qu'on ne dise pas que je n'ai rien dit de nouveau: La disposition des matières est nouvelle." - Pascal, *Pensées*, number 22 (Edition L. Brunschvieg, Hachette, 1950, p. 30)

MATLAB is a program which is especially designed for matrix type operations. The following examples illustrate, without any explanations for the present, some of the range of computations that MATLAB was designed to do.

Pocket type calculator for linear algebra

We enter the matrix A:

```
>> A = [.2 .6;.4 .3]
```

and find the inverse using the MATLAB function inv( ):

```
>> A_inv = inv( A )

   -1.6667    3.3333
    2.2222   -1.1111
```

Then we check using multiplication:

```
>> A_inv * A

     1     0
     0     1
```

Advanced calculations

For A as above we find, using the MATLAB function eig( ), the eigenvalues and store them in a diagonal matrix EIG_VAL. At the same time we find the corresponding eigenvectors and store them as the columns of a matrix EIG_VEC:

```
>> [EIG_VEC , EIG_VAL] = eig( A )

EIG_VEC =

   -0.8048   -0.7418
    0.5935   -0.6706

EIG_VAL =

   -0.2424         0
         0    0.7424
```

Next, we pull out the first column of EIG_VEC so as to obtain the eigenvector ready for further computations:

```
>> eigvec1 = EIG_VEC( : , 1 )

   -0.8048
    0.5935
```

Programs

At the next level we can program MATLAB to do a sequence of calculations and/or determine when certain conditions are met. If for example we want to determine the first power of P for which all the elements are less than .1, we need only write the following:

```
P = [.2 .7; .4 .3]
n = 1;

while       max(max(P^n)) > .1
              n = n + 1;
end

disp('first index when power <= .1 : ') , disp(n)
disp(P^n)
```

The output of this program is:

```
first index when power <= .1 :

     8

    0.0630    0.0916
    0.0523    0.0761
```

Plotting involving vectors and matrices.

MATLAB makes the plotting of vectors and matrices of values against vectors of domain or index values particularly simple. This combined with the various built-in functions and the possibility of programming makes MATLAB very useful in connection with the visual display of matrix calculations.

An example of this type will be discussed in section II.6 and involves the calculation and plotting of the eigenvalues of a matrix. The back cover shows a typical output.

Another type of application involves the evolution in time of a system described by means of a matrix model. Examples of matrix models are given in section III.3.

The cover shows a typical outcome of calculations involving an "initial vector" multiplied by powers of a 3#x 3 "stochastic matrix". Each of the lines represents the values associated with each of 3 states as time evolves. The values were stored in a matrix M and the graph was produced by the simple command:

```
>> plot( index , M )
```

The purpose of this guide is to show the reader how to use MATLAB for the above and other purposes. It is called a guide rather than a manual because of the way it has been organized. The organization is by activity rather than by commands. For quick reference these activities, as well as the associated commands, are listed at the top of each section in which they are discussed. Then the sections continue with a discussion of the commands and examples which illustrate the use of the command with real numbers. References are made to later examples where the function is used again. I have deliberately avoided the use of arcane symbols which, according to my experience with classes, serve mainly to confuse and frustrate users.

My philosophy while writing this guide has been the following:

a. Only those functions and variations which have been found to be most useful should be included; too many functions and too many variations make for confusion and abandon, especially for first time users.

b. Informal suggestions should be made to make life simpler and to point out common errors.

The guide starts off with a sample MATLAB session in Section I.1 so that the first time user can immediately start to do calculations. All the common MATLAB operations and functions are discussed and illustrated in Chapter II, which has been split into eight subsections for quick reference.

Once the user is ready for more sophisticated material, they can consult sections I.2 (the five basic MATLAB symbols) and I.3 (simple loops and functions). With these two sections, the user is ready for advanced manipulations. Sections I.4 (advanced importing and exporting) and I.5 (formatting the output) can be read when needed.

Three special sections at the end give more detailed and advanced examples dealing with the solution of equations, eigenvalue problems, and matrix power models.

Overall, MATLAB produces quick solutions to straightforward matrix problems, via simple transparent commands. For more complicated problems, MATLAB programs tend to be very concise and fairly easy to write (see e.g. the program for the spectral decomposition of a matrix in section III.8). For plotting graphs based on vectors MATLAB is, in my opinion, a sheer delight (see sections II.6 and

III.3) with the combination of flexibility, simplicity and ease of use being unmatched by any other software package. As far as accuracy is concerned, the reader is invited to look at example 5 of section III.1 where the solution of 1000 non-homogeneous equations in 1000 unknowns is discussed.

This book contains shortened versions of various programs that I have written for my classes. Complete programs can be obtained free of charge by sending an electronic-mail message to me at rhfischl@mathstat.carleton.ca .

EDITORIAL CONVENTIONS

MATLAB functions are given in boldface type, at the principal points of discussion e.g. diary calcul1.dia or inv(A).

System commands e.g. RETURN are given in uppercase letters.

MATLAB programs that I have written are printed in courier type. The numbering starts again in each section, and is referred to by section, e.g program 1 of section III.1. As mentioned above, the programs used to produce the actual output have been printed here in shortened form so as not to obscure the major features.

MATLAB output is also printed in courier type. Sometimes, especially in the earlier sections, my comments have been interspersed with the output so as to provide a running commentary on what is happening; these are printed in Helvetica as is the main text. Sometimes, italics have been used in the output for clarity. At other times, the comments have been put in the programs or added to the MATLAB commands using the MATLAB command % .

The output is that produced by MATLAB version 4.0a (dated December 11, 1992) as installed on the Carleton University UNIX machine called "superior". Various redundant or blank lines have been edited out and spacing has sometimes been changed. When the output involves complex numbers, I have often put braces {###} around them for emphasis.

Example numbering starts again in each section, and is referred to by section, e.g example 1 of section I.2.

I find that it is good practice to indicate vectors and scalars by lower case letters, e.g. "a" or "vect_1" and matrices that are not vectors by upper case letters, e.g. "A" or "STORE" and I have employed this device here.

Another good practice is to use the extensions *.dia and *.gra, so as to easily locate diary and graph files. This practice has been followed here.

TECHNICAL NOTES

MATLAB has implementations on various systems, but any differences are related to how the system works rather than the MATLAB program itself. An "*.m" file written in ASCII format for one system will work on another system. The greatest difference between systems will involve the printing out of graphs; see item 3 below.

Most of the basic commands in MATLAB have stabilized, but since MATLAB is being constantly refined, some of the commands have been changed. If you created an "*.m" file on one system and then transferred it to another only to find that it no longer works, it is probably due to a change in syntax. For example older implementations used the command eye(A) to create an identity matrix of the same size as the square matrix A. Newer versions however require the command eye(size(A)); see Section II.2. Some of these changes have been noted in the text, but it is possible that MATLAB will respond to a command with an error message on the particular system that you are using. You should then use the command help axis etc. to find what the proper convention is. For some applications, in particular in connection with plotting and printing, the number of options is very large. I have only listed the most common and most useful options, but the others may be determined by typing help plot, help axis, help title, help print.

TABLE OF CONTENTS

There is a summary of the commands at the beginning of each section This summary is followed by more detailed explanations of the commands, illustrative examples and programs.

Many of files that appear in this book can be found at:
`people.math.carleton.ca /˜rhfischl/`

Table of Contents

CHAPTER I: RUNNING OCTAVE/MATLAB

SECTION I.1. BASICS

SUMMARY

If this is your first session with OCTAVE/MATLAB, go through the following OCTAVE/MATLAB session and then do another session with a few of the functions in section II.3.

i. To start OCTAVE/MATLAB.

Type OCTAVE/MATLAB. at the comand line.

ii. To quit OCTAVE/MATLAB.

Type quit.

iii. To save your work in a diary file calcul1.dia.

Open a "diary" using the diary command: diary calcul1a.dia. (the extension .dia helps you identify your diary files)

To toggle the diary: diary off, diary on.

To end the diary session: diary off.

A VERY STRONG SUGGESTION

In principle, do everything without a diary. In practice however, especially when you are learning or when you run a program more than once, or if you run several programs, you are going to spend a lot of frustrating time figuring out what went wrong, what you want to change etc. Diaries permit you to direct the output of different trials or programs to different files. Those files that you want to retain will be small and easily edited.

Suppose that you have a functions tele1.m and tele2.m that you wrote to analyse a telephone system. You want to run them, or debug them, for different sets of values. An efficient way to keep track of what you doing is to create different diaries for different trials and lable them: tele1a.dia, tele1b.dia ..., This way you when you look at your files, you will immediately see that you are dealing with the first, second ... diaries for functions tele1.m and tele2.m. You can check these diaries in the middle of a session by using the ! command as in part v. below.

iv. Editing in the middle of a session.

Use up arrow, down arrow, delete/backspace, or ENTER commands on the keyboard.

[Note: These commands are very useful as they help avoid the necessity of retyping an entire line. These commands are also valid in window versions.]

v. Returning to the system.

Use ! followed by commands such as DIR, TYPE, LS or by giving the name of a text editor. At the end of the system operation control returns to OCTAVE/MATLAB.

vi. Comments.

Use % to place comments at the OCTAVE/MATLAB prompt or after OCTAVE/MATLAB commands.

vii. List of variables.

Type whos.

viii. To clear the work space.

Type clear.
To clear just some variables: clear A1, ans

ix. To clear the screen (not the work space).

Type clc (= clear command).

x. To list the files in the current directory.

.ls

Notes AND WARNINGS

a. OCTAVE/MATLAB is case sensitive; thus variables A and a are not the same. [If you really do not like case sensitive programs then use casesen off and casesen on to toggle, but first read the next suggestion].

b. To help you distinguish matrices from vectors and scalars it is suggested that you use uppercase for the former and lowercase for the latter, e.g. A or STORE for matrices and a or vect_1 for scalars and vectors.

c. Variable names can be any length and may contain an underbar e.g. MATRIX_1. Do not write MATRIX-1, because OCTAVE/MATLAB will think that you are subtracting. Do not write 1_MATRIX (i.e. don't start the name with a number) as OCTAVE/MATLAB thinks that you are3 dealing with a number.
d. Do not use the name of a OCTAVE/MATLAB function for your variables 3because OCTAVE/MATLAB will think that you are trying to evaluate the function and forgot the ( ). Thus since det( ) is a built in OCTAVE/MATLAB function, if you want to designate the determinant of your ou.ls3tput matrix, call it deter or d, but not det! A very common error is to write something similar to:

```
>>  sum = 2 + 5
```

If you next use the OCTAVE/MATLAB function sum( ) you will find that it does not work!

```
>>  b = [2 5]
>>  sum(b)
```

OCTAVE/MATLAB responds with the error message:

```
" ???? index exceeds matrix dimensions. "
```

Instead of calling your variable sum, call it sum1 or total.
Similarly do not use prod for product, or mean for the mean (average).
e. OCTAVE/MATLAB functions are all lowercase and use parentheses. The name usually gives a good idea of what the function does e.g inv(A) finds the inverse of A and rref(A) give the reduced row ech.lselon form of A. Some functions however return a pair of matrices.
f. OCTAVE/MATLAB is often able to interpret statements that are incorrect in the sense that they do not stand for what you wanted them to stand for. In particular you can often get away without using parentheses, but not only may the answer be different from what you had intended (see II.1), but also later you won't be able to understand what you had written. So USE PARENTHESES. Always test programs and defined functions on examples for which you can do a hand check.
g. OCTAVE/MATLAB is now fairly stable as far as the names of commands are concerned. If you use a command that has been replaced, you will usually see a message that gives you the new form of the command.
h. If you quit OCTAVE/MATLAB, then return to OCTAVE/MATLAB and use a diary name that you have previously used, then OCTAVE/MATLAB adds to the previous diary file.

A SAMPLE OCTAVE/MATLAB SESSION

Here is a sample OCTAVE/MATLAB session.

First we call up OCTAVE/MATLAB at the system co.lsmmand:

```
{superior:1} OCTAVE/MATLAB
.ls
```

SECTION I.1. BASICS

We now open a "diary" file called calcul1.dia in wh.lsich to save the commands and output of the session. If you add the suffix ".dia" it will be easier to locate the diary when you go to edit it.

```
>> diary calcul1.dia
```

To enter a matrix to which we assign the variable name A, type in the matrix separating the rows by a semicolon. Push ENTER and OCTAVE/MATLAB automatically types out the variable name and the answer, unless commanded not to do so by the use of a " ; ":

```
>> A = [.2 .6; .4 .3]

A =

   0.2000    0.6000
   0.4000    0.3000
```

I now want to find the inverse matrix which I call A_inv.
% inv( ) is the OCTAVE/MATLAB function for finding inverses:

```
>> A_inv = inv(A)

A_inv =

  -1.6667    3.3333
   2.2222   -1.1111
```

Check using multiplication, there is no need to give a variable name. Whenever no variable name has been assigned to an operation OCTAVE/MATLAB assigns the value to the variable ans.:

```
>> A*A_inv

ans =

   1.0000    0.0000
  -0.0000    1.0000
```

I now decide to see what happens if I replace the .4 by .1. Instead of retyping the entire matrix, I use the up arrow until I arrive at the data statement. By using the arrows, DEL or BACKSPACE, I change the .4 to .1. I also decide to change the name of the matrix to A2. When ENTER is pushed the new A2 appears:

```
>> A2 = [.2 .6; .1 .3]
```

```
A2 =
.ls
    0.2000    0.6000
    0.1000    0.3000
```

If I now use the up arrow to go to A_inv and change A to A2 and then push RETURN, the command is repeated. Or I could just retype

```
>> A2_inv = inv(A2)
```

```
Warning: Matrix is singular to working precision.
```

OCTAVE/MATLAB has come up with an error message.

I could make more changes, but I have done enough on this problem, so I close the diary

```
>> diary off
```

All of a sudden, I remember that I want to find the determinant of A. So I turn the diary on and make sure that I turn it off again at the end. The new material is added at the end of the diary.

```
>>  diary on
```

In order to continue with the computations, the names of the variables and their dimensions must be known. This is done via the OCTAVE/MATLAB command whos:

```
>> whos

Name          Size           Total     Complex

A            2 by 2          4         No
A2           2 by 2          4         No
A2_inv       2 by 2          4         No
A_inv        2 by 2          4         No
ans          2 by 2          4         No
```

The variable ans came about from the check on A_inv to which no variable name had been assigned. If we had done this several times only the last ans is kept.

I now proceed

```
>> determinant = det(A)
```

```
determinant =

   -0.1800

>> diary off                    % close the diary
```

The additional material has been added to the diary

We could now start a whole new set of computations; first clear all the variables and then open a new diary:

```
>> clear

>> diary calcul2.dia
```

or we could just clear A2 and its inverse

```
>> clear A2, A2_inv
```

We now go through a series of calculations using some of the operations of section II.1 or the functions of section I.3.
When we are done we close the diary:

```
>> diary off
```

At this point I want to see what the results were. There is only one problem, I can't remember the name of the first diary!! (We could also have used the UP ARROW to find the original diary command.)

To produce a list of *.dia files I type ! at the OCTAVE/MATLAB command line to invoke the system directory commands. For UNIX systems, the directory command is "ls". For DOS systems, use DIR *.dia and for window systems click on the file listings.

```
>>  ! ls *.dia
```

From the file listing, I remember that the diary name was calcul1.dia.

To actually see what is in the file calcul1.dia, I use either a word processor or a viewing command. At this point, I can either leave OCTAVE/MATLAB or use the system command " ! " :

```
>> ! {name of text editor} calcul1.dia   % invoke the editor
```

The viewing command in UNIX is "more". For DOS systems use TYPE calcul1.dia:

```
>> ! more calcul1.dia
```

I can now continue my calculations, with the same or another diary. If I want a clear screen I use clc

```
>> clc
```

To leave OCTAVE/MATLAB:

```
>> quit
```

The file calcul1.dia can now be edited and printed using a text editor.

SECTION I.2. FIVE SPECIAL SYMBOLS: [ ] ( ) ; , :

SUMMARY

These five symbols are the most important in MATLAB. They have several uses and are used in conjunction with one another. The summary is grouped by symbol, whereas the example section is grouped by function.

NOTE: In order to make the symbols stand out, space has been left between symbols throughout this guide. This is not necessary when actually using MATLAB, but neither does it affect the result.

i The square bracket [ ]

To build matrices: A = [1 2]

ii The parentheses ( )

To indicate functions: B = eig(C)

To extract elements:
 b = A(1 , 2) for matrices
 b = a(3) for vectors

To redefine elements: A(1 , 2) = -5

iii. The semicolon ;

To separate the rows of a matrix: A = [1 2 ; 3 4]
To place submatrices under one another: C = [A ; B]
To suppress printing: A = [1 2 3 4];

iv. The comma ,

To place matrix blocks side by side: B = [vec1 , vec2]
To extract elements of a matrix: a = C(2 , 3)
To separate functions on a line: a = 1 , B = eig(C)

v. The colon :

 1 to 4 by steps of 1: 1 : 4
 -1 to 2 by steps of .5: -1 : .5 : 2
 5 to 1 in decreasing steps of 1: 5 : -1 : 1

--

DETAILS AND EXAMPLES

i. The use of [ ] to indicate the building of a matrix.

SECTION I.2. FIVE SPECIAL SYMBOLS: [ ] ( ) ; , :

The use of ; and , as matrix separators.

When we enter the elements of a matrix, we use square brackets, and separate the rows (which can be thought of as row vectors) by a semicolon.

```
>> A = [1 2 3 ; 4 5 6]        % two vectors on separate lines

A =
     1     2     3
     4     5     6

>> B = [7 8 9 ; 10 11 12]

B =
     7     8     9
    10    11    12
```

We can now join the two matrices A and B in two ways to form a new matrix:

We can put B underneath A by using a semicolon.

```
>> C = [A ; B]              % place submatrix B under A

C =
     1     2     3
     4     5     6
     7     8     9
    10    11    12
```

We can put A and B next to one another by using a comma.

```
>> D = [A , B]

D =
     1     2     3     7     8     9
     4     5     6    10    11    12
```

ii. The use of ( ) as a function indicator.

If we wanted to find the rank of C, we would use the rank function rank( ). The dependent variable is placed inside the parentheses.

```
>> rank_C = rank(C)

rank_C =
     2
```

iii. The use of ( ) and , to extract or redefine an element.

To find the element in row 2 and column 3 of A above we use parentheses which is the extraction function indicator and then enter the row and column values separated by the comma:

```
>> b = A(2 , 3)                % row 2 , column 3

     6
```

If a is a vector then all we have to do is give the element number:

```
>>  a =  [1 3 5];

>>  b = a(2)

     3
```

On the other hand suppose that we wanted to change the element in row 2 and column 3 so that it is -9, while leaving all the other elements alone. There is no need to retype all of A. We just specify the new value.

```
>> A(2,3) = -9                 % redefine row 2, column 3

A =
     1     2     3
     4     5    -9
```

The use of the colon to extract submatrices is discussed in subsection v, below and in section II.4

iv. The use of : to indicate the "range" of indices.

The colon could be loosely translated by the term "range". Thus if we wanted a row vector with the integer range -1 to 4 we write:

```
>> x = [-1 : 4]

    -1     0     1     2     3     4
```

Note that since we are building a vector we use brackets, [ ].

If we wanted to go from -1 to 2 by steps of .5, we would include the .5 in the statement. Note that there are now two colons.

```
>> x_step = [-1 : .5 : 2]

 -1.0000   -0.5000  0    0.5000    1.0000    1.5000  2.0000
```

 To generate a vector in reverse order use negative steps.

```
>> X_dec = [5 : -1 : 1]

X_dec =

     5     4     3     2     1
```

[Note: Writing [5 : 1] would give the empty matrix [ ]; see section II.2.vi].

For examples of the use of the colon, see Section II.1, example 1 and Section II.4

v. The use of : when extracting submatrices.

Subsection iv illustrated the used of the colon as a range indicator. By combining the colon with the extraction function of subsection iv, we can extract rows, columns and submatrices.

To extract the second row of A we fix the row value at 2 and let the column values range over all values as indicated by the colon.

```
>> row_2 = A(2 , : )            % the entire column range in row 2

     4     5     6
```

To extract the third column of A we fix the column value at 3 and let the row values range over all values as indicated by the colon.

```
>> col_3 = A( : , 3)            % the entire row range in column 3

     3
     6
```

If we only wanted to extract columns 2 through 3 of row 2 of A we would limit the column range by use of [2 : 3]

```
>> row_2_shortened = A(2 , [2 : 3])  % row 2, columns 2 to 3

     5     6
```

We could also write:

```
>> indices  = [2 : 3]

     2     3
```

and then

```
>>  row_2_shortened = A(2 , indices)

     5     6
```

More examples of the use of the colon are given in section II.4 which deals with extracting and building matrices.

vi. The use of ; to suppress printing.

We reenter B, but this time we put a semicolon after the bracket to suppress printing. Note that the semicolon is also used inside the brackets to separate the rows:

```
>> B = [7 8 9 ; 10 11 12] ;         % outside to suppress printing
```

To print B we simply type:

```
>> B

B =

     7     8     9
    10    11    12
```

To print just the matrix without "B = " we would type

```
>> disp(B)             % see section I.5.i ].
```

To suppress printing several variables in a row we put a semicolon after each:

```
>> B; C;
```

vii. Using , to place commands on the same line.

If we wanted to find the ranks of C and D, we could write the two commands on separate lines or use a comma.

```
>> rank_C = rank(C) , rank_D = rank(D)
```

SECTION I.2. FIVE SPECIAL SYMBOLS: [ ] ( ) ; , :

```
rank_C =

    2

rank_D =

     2
```

SECTION I.3. PROGRAMS, SIMPLE LOOPS AND FUNCTIONS

SUMMARY

i. Creating and importing your own program or data file.

Use a text editor to create an "m-file", e.g. "program1.m" in which you have typed the instructions, and data, exactly as you would do at the MATLAB command line. Invoke the program by typing the name without the extension:

```
>> program1
```

[Note: Do not start file names with numbers, e.g. "1program.m" is not a valid name.]

ii. Pauses

To have a program pause until return is pushed: pause
To have a program pause for 10 seconds: pause(10)

iii. "For-loops"

```
for  n = 1 : 5
          P^n
end
```

[Note: MATLAB is most efficient when working with vectors (or matrices). Thus whenever possible work with vectors instead of using loops. See Section II.1, example 1 for the conciseness and power of the vector approach.]

iv. Creating and using your own functions

Create the file trans1.m which starts off with:
```
   function NEW = trans1(OLD)
```
to evaluate a function at the variable A:

```
>> B = trans1(A).
```

[Note: For function names, as well as for the variables, do not use the names of MATLAB functions e.g. "det" .

Note: In certain versions you can use capital letters for your functions; e.g. in Section II.3 R( ) is defined to be equal to the MATLAB function sqrt( ).]

--

SECTION I.3. PROGRAMS, SIMPLE LOOPS AND FUNCTIONS

DETAILS AND EXAMPLES

i. Creating and importing your own program or data file.

Instead of typing in commands at the MATLAB prompt it is often preferable to create a program file which contains the instructions and then call up the program. This is particularly handy when we have several steps to perform in our computation or when we want to perform other calculations which are modifications of a previous one. It makes corrections much simpler and avoids a great deal of frustration caused by typing and other simple errors. When the amount of data to be used in a program is small, as in program.1 of example 1, then the program file will contain both data and instructions. Some files contain just data and are then called up in another program file or via a function (see iv). As far as MATLAB is concerned there is no distinction between a program file and a data file. In both cases we create an "m-file" or "*.m file". If we have two large sets of data that we want to analyse we might put them into two files data1.m and data2.m.
In our program file we would then change the call-up portion of the file from data1.m to data2.m. The use of a file that just contains data and is called up by a program is shown in example 2. If we are going to analyse many sets of data, we may prefer to write a function (see iv).

Program vs. function. The main difference between a program and a function, is that with a program all the variables become MATLAB variables (they are listed when we type whos - Section I.1, viii) that can be used later on. For a function only the designated dependent variables become MATLAB variables. This distinction often helps us to decide if we want to write a program or a function.

To write a program or data file we simply use a text editor (to create a pure ASCII file) and type in the instructions and the data. When naming the file, the name portion is arbitrary (except that it must start with a letter and not a number), but the extension must be "m", e.g. program1.m. To call the program one types the name, e.g. program1 of the file, without the extension. Large data sets, saving data and loading data are discussed in more detail in section I.4.

WARNING: Do not use the name of a MATLAB function for your variables because MATLAB will think that you are trying to evaluate the function and forgot the ( ). Thus since det( ) is a built in MATLAB function, if you want to designate the determinant of your output matrix, call it "deter" or "d", but not "det"!

ii. Including a pause in a program.

Example 1 also illustrates the use of pause. When there are many variables the earlier ones will be "gone" before we have had the time to read and check them. So we put either pause in which case MATLAB waits until you push ENTER (or any key), or pause(10) - or pause(15) etc. - in which case MATLAB automatically resumes after 10 seconds.

Example 1. I used a text editor to type the following program "prog1.m" which contains some data and a sequence of commands to manipulate this data. Note that we can leave a space between lines and put in comments using %.

Program 1

```
% [prog1.m]

C = [1 2 3 ; 4 5 6]          % data

pause(10)

ra_C = rank(C)                % find the rank

pause

ech_C = rref(C)               % reduced row echelon form
```

I invoke the program with the command prog1; this loads the variable "C" and then invokes functions which find the rank and reduce the matrix to row echelon form are invoked. The output is assigned to the variables "ra_C" and "ech_C".

```
>> prog1                      % no extension

C =                        % MATLAB prints C

     1     2     3
     4     5     6

MATLAB pauses for 10 seconds

ra_C =   2


MATLAB pauses until enter is pushed

ech_C =

     1     0    -1
     0     1     2
```

[Note: In this program the matrix C is automatically printed. For large matrices we may want to suppress printing. To do so we would place a semicolon after the matrix; see Section I.2,vi.

SECTION I.3. PROGRAMS, SIMPLE LOOPS AND FUNCTIONS

```
>> C = [1 2 3 ; 4 5 6];           % data
```

If we needed to see C we would use disp( ); see Section I.5,i:

```
>> disp(C)
```

]

Instead of placing the data inside the program we can also place it in a data file C.m

Program 2

```
% [data1.m]  contains only data

C = [1 2 3 ; 4 5 6]            % data
```

We now modify prog.1 so that it calls up the data file data1.m.

Program 3

```
% [prog1b.m]

data1                          % calls up data file [data1.m]

pause(10)

ra_C = rank(C)                 % find the rank

pause

ech_C = rref(C)                % reduced row echelon form
```

We now call up prog1b:

```
>> prog1b

C =
     1     2     3
     4     5     6

ra_C =   2

ech_C =
    1     0    -1
    0     1     2
```

When we type whos (Section I.1, viii) to find out what our variables are we see that all variables that appear in prog1b are there

```
>> whos
              Name        Size          Total     Complex

                 C       2 by 3             6        No
             ech_C       2 by 3             6        No
              ra_C       1 by 1             1        No
```

iii. "for - loops".

Often we want to repeat an operation many times and instead of constantly recomputing individual quantities we can use a "for-loop". This consists of a statement of the form:

```
for n = 1 : 5

     instructions to do something

end
```

Note that for and end are in lower case letters.

Loops are normally created in "m-files" and then called up in MATLAB, although simple ones can be typed in at the prompt. It is suggested that neither i nor j be used for indices since these are also used for imaginary numbers (section II.3.iii) and confusion in reading the program might result.

Example 2. We have a recursive relationship of the form:

nk = Pk * n0 ,

where P is a square matrix and nk and n0 are column vectors. To compute nk, for a sequece of values of k, we create the m-file called power1.m which contains P, n0 and a "for-loop":

Program 4

```
% [power1.m]  compute powers.

% enter initial vector and take the transpose; since the % elements are real we
```

can use '.

```
n_0 = [2  3]'

P = [.2 .6 ; .4 .3]  % enter the matrix;

%loop starts
for n = 1 : 3
     u = P^n * n_0
end
```

We call up "power1.m" by typing power1 without the extension *.m . Notice how for each loop the variable "u" changes its value and is printed out.

```
>> power1

n_0 =

    2
    3

P =

   0.2000    0.6000
   0.4000    0.3000

u =

   2.2000
   1.7000

u =

   1.4600
   1.3900

u =

   1.1260
   1.0010
```

In the above loop we could have indicated the subscript by adding the index n in the loop:

Program 5

SECTION I.3. PROGRAMS, SIMPLE LOOPS AND FUNCTIONS

```
% [program3.m]    have the program print the value of n

for n = 1:3
     n
     u = P^n * n_0
end
```

A nicer output can be obtained by suppressing printing using ; and then using disp(n) , disp(u); see section I.5.

Program 6

```
% [program3.m]    suppress "n = " and use disp(n)

for n = 1:3
     n;                      % suppress printing n
   u = P^n * n_0;              % suppress printing u
   disp(n) , disp(u)
end
```

iv. Creating and using your own functions.

Programs perform a sequence of operations on one set of data. When the same operations are going to be performed over and over it is preferable to define a function. To create a new function we create an "m-file" whose name (without the ".m") is exactly the same as the function. Thus if we want to define a function "trans1( )" we have to create a file "trans1.m"

The first statement - not including comments which start with % - in trans1.m must start off with the word function so that MATLAB knows that it is not dealing with a simple sequence of commands and data as in subsection i. The word function is followed by the statement that defines the dependent variable(s) as trans1( ) of the independent variables. The function name should be written in lower case letters (some older versions will accept upper case). Thus in the file "trans1.m" we start off with:

```
function  NEW = trans1(OLD)
```

In this case we have one dependent variable, called NEW and one independent variable called OLD, but MATLAB functions can have several dependent and several independent variables as will be seen in the examples below. These variables are all internal variables, i.e. they will not appear directly in the list of MATLAB variables that we obtain when we type whos; see Example 3 for a further discussion of this.

WARNING (second time in this section): Do not use the name of a MATLAB function for your variables because MATLAB will think that you are trying to evaluate the function and forgot the ( ). Thus since det( ) is a built in MATLAB function, if you want to designate the determinant of your output matrix, call it "deter" or "d", but not "det"!

Once the name of the function and symbols for the dependent and independent variables have been decided the next step is to give an algorithm which relates the variables:

Example 3. We have a fixed matrix F = [-2 2 ; 3 1] and we want to take various 2 x 2 matrices, square them and then add F. The function program would be:

```
Program 7
% [trans1.m]

function  NEW = trans1(OLD)

F = [-2 2; 3 1];                % suppress printing

% give the formula. add ; to avoid printing twice
NEW = (OLD)^2 + F;
```

To use this function we have to assign a variable name to the independent variable that whose functional value we are going evaluate. We do not have to use OLD as in the program and can use any variable name, e.g. A:

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

Next, we a assign a variable name, e.g. B, to the value of the function (we do not have to use the name NEW) as follows

```
>> B = trans1(A)

B =

     5    12
    18    23
```

It is preferable, but not necessary, to assign a variable to the value of the function. If we do not then the value is assigned to the variable ans as follows:

```
>> trans1(A)

ans =

     5    12
```

```
   18    23
```

Internal variables. Note that in the function trans1.m the variables OLD, NEW, F are variables which are internal to the function; they will not appear when we type whos (Section I.1, viii). Sometimes we do not want to keep a quantity as a MATLAB variable, but we want to print out the value, perhaps as a check when we run the program. The quantities can also be introduced as internal variables as shown in the next example:

Example 4. Suppose that as we go along in the above calculation we want to check the determinant of A to make sure that it is non-zero:

Program 8

```
% [trans2.m]
% prints out determinant as part of function call

function  NEW = trans2(OLD)

F = [-2 2; 3 1];                % suppress printing

det_check = det(OLD)            % do not suppress printing!!

% alternative method of printing out the value
disp(det(OLD))

NEW = (OLD)^2 + F;
```

We run the program and then type whos:

```
>> B = trans2(A)

det_check =    -101

B  =
     5    12
    18    23

>> whos
              Name        Size          Total     Complex
                 A       2 by 2            4        No
                 B       2 by 2            4        No
```

Note that the quantity det_check has been printed out, but does not appear as a variable. The value of det_check was printed out because we did not put a semicolon. An alternative method, as shown in the program, is to use the disp function. The value assigned to NEW by the function is listed under the variable B.

Functions of several independent variables. MATLAB functions can also be functions of several independent variables. Suppose that instead of a fixed adder in example 3, we want to have a variable adder. We simply include the adder as a variable in the defined function as in the following example.

Example 5.
Program 9

```
% [trans3.m]  - illustrates a function of two variables

function NEW = trans3(OLD, ADDER)

NEW = (OLD)^2 + ADDER;    % give the formula do not print out.
```

To use this function we need to define the adder, using any symbol:

```
>> C = [1 -2; 0 1]
```

We designate the dependent variable as D:

```
>> D = trans3(A , C)

     8     8
    15    23
```

Several dependent variables. In calculus a true function can only have one dependent variable, but a MATLAB function can compute and store several variables. For the function trans2( ) in example 4, we computed the determinant of the input matrix as we went along and even had the value printed out, but we did not assign a variable name to it. As we saw in example 4 the quantity does not appear in the "whos" list and thus cannot be used again. If we want to save the quantity as a variable then we proceed as in the following example:

Example 6. Suppose that in example 5 we had wanted to compute the rank of the output matrix, and also to assign a variable name to it. In this case, there are two dependent variables, which we designate as NEW and rank_new. We place both of these variable names inside square brackets on the first line of the m-program. Note that the two variables do not need to have the same dimensions, i.e. we are not forming a true matrix:
Program 10

```
% [trans4.m]  - illustrates a multi-variable output
% modification of program 5
```

```
function [NEW, rank_new] = trans4(OLD)

F = [-2 2; 3 1];    %suppress printing

NEW =(OLD)^2 + F;  % give the formula do not print out.

rank_new = rank(NEW);
```

At the MATLAB prompt we define our dependent variables. In this case we will assign the function value NEW to a variable whose name is also NEW. We assign the function value rank_new to a variable whose name is different namely lin_ind.

```
>> [NEW, lin_ind] = trans4(A)  % A was entered above.

NEW =

     5    12
    18    23

lin_ind =

     2
```

If we had just written:

```
>> Z = trans4(A)
```

then the value of Z would be that of the first of the variables (NEW) and the second variable lin_ind would not have appeared.

No independent variables. It is possible to define a function with no independent variables. In the following example we use the input function, which is discussed in I.4, to obtain that value we use in the computation.

Program 11

```
% [squa_it.m] - a function with zero variables

function c = squa_it;

N = input(' what is N? ');
c = N^2;
```

We could have obtained exactly the same value for c if we had not put in the function statement, in which case we would have had an ordinary MATLAB program. In the function format only c becomes a MATLAB variable, whereas with the program format both N and C become variables that can be used later on.

Functions with one or several dependent variables. For some built-in MATLAB functions there is the possibility of assigning either one or two dependent variables. An example of this is the function diag( ) which is discussed in detail in sections I.3.1 and III.2. If we type:

```
>> EIG_VAL = eig(A)
```

then we will obtain a diagonal matrix whose diagonal elements are the eigenvalues of A. But if we type:

```
>> [EIG_VEC , EIG_VAL] = eig(A)
```

we obtain two matrices, the first one having the eigenvectors of A stored as columns and the second again being a diagonal matrix whose diagonal elements are the eigenvalues of A.

SECTION I.4. ADVANCED IMPORTING AND EXPORTING TECHNIQUES

SUMMARY

[Typical names of files are indicated in the summary by italics; note that in the save and load commands discussed in iv and v, the extension .mat is added automatically by MATLAB. The expression "m-file" refers to a file with the extension *.m]

i. Importing data.

Use a word editor to create an "m-file" e.g. import1.m in which you have entered the variables and data exactly as you would at the MATLAB command line.

ii. Long lines of data.

The data is enclosed in brackets [ ] as is the case with all matrices of data. At the end of each line we type ellipses ... , push ENTER, then continue the data on the next line. This applies to "m-files" as well as at the MATLAB command.

[Warning! Leave a space after the last number, e.g. 5 ...
Otherwise MATLAB thinks that you have put decimal points.]

iii. Importing data from the keyboard.

```
vect_1 = input('type in your first vector:  ');
```

for string variables use the form:

```
     comment = input('type your comment: ' ,  's' );
```

A string input is made without ' ', e.g yes (and not 'yes').

iv. Saving variables to a "mat-file" called results.mat.

To save all variables: save results (no extension).
To save several specific variable: save results A1 a
[Note: there is no comma anyplace in the statement].

v. Reloading variables from a "mat-file" results.mat.

```
>> load results.
```

vi. Saving one variable A in ascii format to a file data1

```
>> save data1 A  /ascii  /double.
```

[Note and warning: As discussed in section I.1 output can be saved in a "diary". Although the output from such a diary can be placed (by use of a text editor) in an m-file and then loaded into MATLAB, this is not advisable when decimals are present and high accuracy is desired. A variable saved in a mat-file is saved to a high degree of accuracy, whereas a variable saved in a diary has been permanently truncated!

ASCII output (see vi) can only be done one variable at a time and the accuracy may not be as high as with mat-files.]

DETAILS AND EXAMPLES

i. Importing data.

For small data sets one can enter the numbers at the MATLAB control line. For larger sets of data and for situations where one wants to make repeated changes, it is better to use a text editor to create an ASCII file which contains the data. When naming the file, the name is arbitrary, but the extension must be ".m", e.g. data1.m. To call the program one types the name, e.g. data of the file, without the extension.

The concept of m-files was discussed in section I.3.i in connection with programs. From the viewpoint of MATLAB a program file is no different from a data file. The concept of putting data in an m-file is repeated here so that a comparison can be made with other methods discussed later in this section. Recall also that in section I.3.iii, we saw how user defined functions were also created by putting the instructions in a file with the extension .m . The difference between function files and program/data m-files is that a function file starts off with the word "function". Functions, via function files, operate on variables, whereas m-files contain a once only sequence of instructions.

Example 1. A simple data file.

The file contains the names of two variables: a matrix and a scalar, and the corresponding data. Note that the variable name MAT1 is not the same as the file name. However it could be the same if we wished.

Program 1

```
% [import.m]

MAT1  = [1 2 3; 4 5 6];          % the semicolon suppresses printing

a = sin( pi/2 );                 % sin(  ) and pi are MATLAB functions
```

To run this program in MATLAB we simply type the name of the file, without the extension, at the command line.

```
>> import1
```

To check that the variables are there:

```
>> whos

Name         Size         Total        Complex
MAT1         2 by 3       6            No
a            1 by 1       1            No
```

ii. Long lines of data.

For long strings of data we want to be able to continue data entry on the next line. To do so, type three or more ellipses "... ", push ENTER and continue the data on the next line. Make sure that you put a semi-colon if you do not want a huge mass of data to be printed out!

Ellipses can be used in "m-files" as well as at the MATLAB command.

Program 2

```
% [dots.m]

% place the data in brackets; add ellipses at the end
% of each line of data. But make sure that you leave a space
% between the last number and the ellipses.
% place a semicolon after the data to suppress printing.
% there is no need to have the same number of elements in each row.

a =  [1  3  ...
            4 ];

s = sum(a)
```

```
>> dots        % call up the program "dots.m"; no extension

a =

    1     3     4

s  =

    8
```

SECTION I.4. ADVANCED IMPORTING AND EXPORTING TECHNIQUES

If we have large sets of data and want to have the same number of elements in each row we can place ... in the first row.

```
a = [ ...                              % first row is empty
      *  *                      *  ...   % !! space after the last number
      *  *                      *  ...
                                   ]; % ! closing bracket
```

[Warning: Leave a space between the last number and the ellipses; otherwise MATLAB thinks that you have put a decimal point!]

[Note: Nothing prevents you from using a semicolon to form a matrix. For example:

```
>> F = [4 5 6 ...       % ellipses to continue row 1
       7 8;           % end of row 1
       9 10 11 12 13]  % row 2 all on one line

   F =

      4   5   6   7  8
      9  10  11  12  13
```

iii. Inputing data from the keyboard

One can also import data directly from the keyboard. This is especially useful in connection with a program that we want to check or run several times and which involves small sized matrices. By using the keyboard, we can avoid switching to a text editor in order to change the data.

To invoke the keyboard, we assign a variable, e.g. vect_1 to the MATLAB function input('   '). Inside the apostrophes, we place a prompt message:

```
>> vect_1 = input(' type in your first vector:  ')
```

Note that for clarity a colon has been included in the prompt message and a little space has been left after the colon.

The input can also be a string, i.e. text, and to indicate this we add , 's' :

```
>> comment = input('type your comment: ' , 's');
```

When we input the text we do not put the apostrophes around it; i.e if the input asks yes or no? we type yes, not 'yes'.

Example 2. Program 3 illustrates the use of input('   ') to input two variables and to input a comment.

Note that I have not placed a semicolon after vect_1 and vect_2 and so MATLAB prints out the values. I could have placed a semicolon and then used disp(vect_1). This however has been done with the variable "comment"; a semicolon has been placed at the end of the line which defines "comment" and then disp(comment) has been used.

The example shows that when we multiply a column vector by a row vector we obtain a matrix of rank 1. We input vect_1 as a row vector and then take its transpose by using the true transpose operator .' ; see section II.3.i for a discussion of the transpose operators.

Program 3

```
%[keyb1.m]

vect_1 = input(' type in your first vector:  ')
vect_2 = input(' type in your second vector: ')

% multiply the transpose of vect_1 and vect_2; find the rank
% .' is the true transpose; see section II.3.i

col_x_row = (vect_1).' * (vect_2);
ran_1 = rank(col_x_row) ;

comment = input(' type your comment: ', 's');
disp(comment)
```

Invoke this program by typing the file name keyb1 without the extension. Boldface in the following indicates what the prompt text is. Italics in the following indicates what is typed in at the keyboard.

```
>> keyb1

type in your first vector: [1 2]

vect_1 =

     1     2

 type in your second vector: [3 4]

vect_2 =

     3     4
```

```
col_x_row =

     3     4
     6     8

ran_1 =

     1

type your comment:  each row is a multiple of first

each row is a multiple of first
```

iv. and v. Saving variables to "mat-files" and reloading them.

In example 2 we imported two variables "vect_1" and "vect_2", and then calculated the values of two variables "col_x_row" and "ran_1". If we wanted to save all four variables for later use, the simplest and best manner is to save them to a "mat-file". This is a file written in machine language in which the variables are saved to a very high degree of accuracy. In principle, one could take the contents of a "diary" that has been created (section I.1), put them into an "m-file" (as in subsection i above) and then call up the "m-file". Not only would that way require a lot of work, but it also has the added inconvenience that accuracy is lost because the output appears in truncated form in the "diary". If you need to save the variables in ascii format, e.g. for export to another program, see vi below and the save ... ascii command; the precision may be less however.

To place all the current variables into a "mat-file" called results we give the command save and follow this by the name (without the extension ".mat")

```
>> save  results
```

MATLAB will automatically assign the extension .mat so that in the directory there will be a file results.mat.

If during another MATLAB session we want to have access to the variables we use the load command:

```
>> load  results
```

Sometimes we only want to save a few of the variables. Supose that we only wanted to save "vect_1" and "col_x_row" to a mat-file results2.mat. We could either first clear "vec_2" and "ran_1" by using:

```
>> clear vec_2, ran_1
```

and then typing:

```
>> save results2
```

or we could use save with the name of the file results2 followed by the list of the variables that we want to save. Notice that there is no comma anyplace in the statement.

```
>> save results2 vec_1  ran_1
```

To re-obtain the variables from results2, we use the load command:

```
>> load results2
```

To find what variables were stored in results2, and what their dimensions are, we use whos:

```
>> whos

Name        Size        Total       Complex
vec_1       1 by 2       2            No
ran_1a       1 by 1       1            No
```

We can now display the variables that we want by typing their names or by using disp( ):

```
>> vec_1                        % "vec_1 =" is printed
>> disp(ran_1a)                    % only the value is printed
```

Note: It is not possible to load just one of the variables from results. Just load and clear the ones that you do not want.

vi.   Saving one variable A in ascii format to a file data1

To save the variable "A" in ascii format to a file data1, we use save with the option /ascii:

```
>> save data1 A /ascii /double.
```

The option /double is usually added to ensure a high degree of accuracy.

SECTION I.5. FORMATTING THE OUTPUT

SUMMARY

i. Using disp() to avoid "A = ".
disp(A) gives A without the statement, " A = "
disp(' A is a matrix ') gives the statement typed between the apostrophes.
Use disp(' ') to skip a line.

ii. The number of decimal places in the output.
For 14-15 places:

```
   format long      57.27564927611034
```

to return to the standard fixed point format (4 places):

```
   format short     57.2756
```

floating point (scientific) notation:

```
   format short e    5.7276e+001
   format long e     5.727564927611034e+001
```

converting matrices to the + { } - format

```
   format +
```

[Note: Recent versions of MATLAB have more commands for specifying the format: e.g.

```
   format bank       57.27 (2 decimal places, for money!)
```

sprintf This command is based on C syntax and can be used to control the number of decimal places more precisely. String statements (such as in disp('hello ')) can also be included:

```
      a = 57.27564927611034
      sprintf( ' %10.5f ' , a ) = 57.27564
```

[5 decimal places; the 10 indicates the printing is being done in a minimum of 10 spaces with the "4" being placed on the extreme right.]

using e instead of f will give the answer in floating point notation.

format hex the output will be in hexadecimal format.

type help format, help fprintf for details of what is available on the system that you are using]

DETAILS AND EXAMPLES

i. Using disp() to avoid "A = "; construction of simple tables.

MATLAB output is usually in the form: "A = " followed by the value of A. To print out just the value of A we use disp( ) :

```
>> disp(A)
```

The display command is also used to produce alphanumeric text. The text to be displayed is enclosed within apostrophes. Thus if we enter:

```
>> disp(' A is a matrix ')
```

we obtain:

```
 A is a matrix
```

The use of disp( ) is illustrated in the following two methods of producing a table of sines and cosines. There are various ways of producing the individual results and these are also illustrated. In the first method we produce individual row vectors of values one at a time and print them out; this method is shown first in step by step format and then in a program. In the second method, we compute the vectors, take their transposes and create a matrix whose columns are these transposes. We then ask MATLAB to display the matrix.

Example 1. Producing the output in rows.

We first form a vector "values" that gives the range of values that interests us.

```
>> values =[0 pi/4 pi/2];          % ; to suppress printing

>> disp(values)

         0    0.7854    1.5708
```

Next, we calculate the sines of the values in "values" by using the MATLAB vector function sin( ). MATLAB trigonometric and other functions are discussed in detail in section II.3. We use both sin( ) and disp( ) in the same statement

```
>> disp( sin( values ) )

         0    0.7071    1.0000
```

For the sine, we did not assign a symbol, but we do so for the cosine values. A common error is to use a function name for a variable, e.g to let cos = cos(values). Note that we have used the underbar in the variable name "cos_variables", because the use of a minus sign, e.g. "cos-values" would make MATLAB think that we want to subtract!

```
>> cos_values = cos(values);
>> disp(cos_values)
```

```
    1.0000    0.7071    0.0000
```

Finally we produce a title for our table. To indicate that we are dealing with an alphanumeric quantity we enclose it in apostrophes. Some space is left after the first comma to shift the title over to the right:

```
>> label = '     table of sines and cosines';
>> disp(label)

    table of sines and cosines             % MATLAB response
```

Above, the commands were given on different lines, but they could also have been put on one line using the comma. Since the line is too long we use ellipses to indicate that we want to continue on the next line, although in this case it is just as simple to push ENTER and go to the next command line. Note how we use disp('   '), i.e. the message is empty, in order to have MATLAB skip a line

```
>> disp(label) , disp(values) , disp(sin( values )) , ...    disp('   ') , disp(cos_values)

      table of sines and cosines

         0    0.7854    1.5708
         0    0.7071    1.0000
    1.0000    0.7071    0.0000
```

Instead of doing the above steps individually we can write a short program "disp2.m".

Program 1

```
% [disp2.m]

x = [0:.2:1];               % ; to suppress printing
disp('  first row  values; second is sine; third is cosine')
disp('   ')                 % empty message to leave a space
disp(x)
disp( sin(x) )
disp( cos(X) )
```

We now call up "disp2.m". Note that we just type "disp2" and not "disp2.m"

```
>> disp2

  first row is values; second is sine; third is cosine

0        0.2000   0.4000   0.6000   0.8000   1.0000
```

```
0        0.1987   0.3894   0.5646   0.7174   0.8415
1.0000   0.9801   0.9211   0.8253   0.6967   0.5403
```

Example 2. Producing the output in columns.

In the first method, the output is in rows, but if the range of values is above nine or ten, then the rows will not fit on one line of print. So, instead of working exclusively with row vectors we take the transpose of the vectors, create a matrix whose columns are these vectors and then print the matrix. This method is followed in the program "disp3.m". Note how we compute the variables, but do not print them immediately. It is sometimes difficult to place the title exactly and it is often simpler to make a final adjustment with a text editor. Since the entries are all real, we can use the operator ' for transpose instead of .' (see section II.1). In the program we first use the semicolon (section I.2.v) to generate a domain vector with values going form 0 to 1 by steps of .2.

Program 2

```
% [disp3.m]

y = ( [0 : .2 : 1] )';          % ' change to a column vector
si = sin(y);                    % ; to suppress printing
co = cos(y);
A = [y , si , co];      % put in a matrix by means of `,'

disp('   x  sin(x)  cos(x)  ')   % display the title
disp('   ')                     % empty message, leaves a space
disp(A)                             % display the matrix
```

We now call the program:

```
>> dispro3           % type filename - without the extension .m

        x  sin(x)  cos(x)

    0         0         1.0000
    0.2000    0.1987    0.9801
    0.4000    0.3894    0.9211
    0.6000    0.5646    0.8253
    0.8000    0.7174    0.6967
    1.0000    0.8415    0.5403
```

One should not spend too much time lining up a title inside the disp('   '); it is usually simpler to do it later with a text editor.

ii. The number of decimal places in the output.

SECTION I.5. FORMATTING THE OUTPUT

MATLAB usually shows the results to five digits; to display more places use format long:

```
>> s1 = 81*sin(pi/4)

   57.2756

>> format long
>> s1                    % type s1 to display it

  57.27564927611034
```

To display the answer in "floating point" notation use `format short e' or `format long e'

```
>> format short e
>> s1

  5.7276e+001

>> format long e
>> s1

    5.727564927611034e+001
```

To return to the standard format use format short

```
>> format short
>> s1

   57.2756
```

For some matrices we may want to just see if elements are negative, zero or positive. Use the "format +" command. MATLAB leaves a very small space where the terms are 0.

```
>> format +
>> C = [-2 0 5]

     -  +                      % There is a space between the - and +
```

SECTION II.1. BASIC MATLAB OPERATIONS

SUMMARY

ia. Transpose of a matrix with real elements: '

ib. Transpose of a matrix with complex elements: .'

ii. Addition: A + B ; A + 5 adds 5 to each element.

iii. Multiplication: A*B; 2*A

iv. Powers: A^3

v. Element by element multiplication: (A).* B

[Note the dot; use parentheses to avoid problems and confusion. A and B must have the same dimensions.]

vi. Element by element division: (A)./B

[Note the dot; use parentheses to avoid problems and confusion. A and B must have the same dimensions.]

vii. Element by element powers: scalar a, vector x:
(a).^x ; (x).^a ; (x).^x

[Note the dots; use parentheses to avoid problems and confusion. Vectors a and x must have the same dimensions.]

viii. For very large matrices:

Instead of A*inverse(B) use A/B.
Instead of inverse(B)*A use B\A.

See the examples in section III.1.vii

ix. Use parentheses!

MATLAB interprets 5/2*4 as (5/2)*4. Is this what you wanted or was it 5/(2*4)?

DETAILS AND EXAMPLES

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

     1     2
     3     4
```

i. Transpose of a real matrix: ' or .'

```
>> TRANS = A'

     1     3
     2     4
```

[Note: See section II.3 for examples involving complex numbers and the difference between ' (conjugate transpose) and .' (true transpose).

ii. Addition: +

```
>> S = A + A'


     2     5
     5     8
```

To add 5 to each element of A there is no need to define a matrix of the same dimensions as A, and having all entries equal to 5. MATLAB interprets A + 5 in this sense:

```
>> B = A + 5

     6     7
     8     9
```

iii. Multiplication: *

```
>> A * B

    22    25
    50    57

>> 5*A

     5    10
    15    20

>> L = (1/5)*A

    0.2000    0.4000
    0.6000    0.8000
```

iv. Powers: ^

SECTION II.1. BASIC MATLAB OPERATIONS

```
>> A^3

   37    54
   81   118
```

Element by element operations

v. Element by element multiplication.

To multiply each element of A by the corresponding element of B use .* . Note the dot which distinguishes pointwise multiplication from regular matrix multiplication. It is advisable to get in the habit of using parentheses, even though they are not always required, to avoid problems and confusion that arise with numbers. A and B must have the same dimensions.

```
>> D = (A).*B

    6    14
   24    36
```

vi. Element by element division.

To divide each element of A by the corresponding element of B use (A)./B. A and B must have the same dimensions. To remember the direction of the slash in ./ think of 6/2 = 3. Division tends to be confusing so use many parentheses.

```
>> (A)./B

   0.1667    0.2857
   0.3750    0.4444
```

To divide each element of A by the number 5, use (A)./5. Note that this is equivalent to doing element by element division of A by a matrix all of whose elements equal 5.

```
>> F3 = (A)./5

   0.2000    0.4000
   0.6000    0.8000
```

To divide each element of A into the number 5, use (5)./A. Note that this is equivalent to doing element by element division of a matrix all of whose elements equal 5, by the matrix A

```
>> F =(5)./A
```

```
    5.0000    2.5000
    1.6667    1.2500
```

See the function "normalize.m" in section III.3, example 1 where pointwise division is used to normalize a positive matrix so that the row sums are all 1.

Example 1. The sum and products of the first eight odd numbers.

```
% [div.m]
% Illustrate term by term division of a vector
% Sum and product of the reciprocals of the first 8 odd
% numbers.
% note how the upper index of the vector a is a variable % that has been
calculated on the previous line.

odd_8 = 1 + (8-1)*2;          % 8th odd number

a = [ 1: 2 : odd_8];          % I.2,iv

% Note the dot and the ( ) in the following!!
recip_a = (1)./a;

sum_recip = sum(recip_a);      % II.3,iv

prod_recip = prod(recip_a);    % II.3,iv

disp('The eighth odd number: '), disp(odd_8)
disp('The first eight odd numbers:'), disp (a)

disp('The reciprocals:'), disp(recip_a)
disp('Their sum and product are: ')
disp(sum_recip), disp(prod_recip)
```

The output of this program is:

```
The eighth odd number: 15

The first eight odd numbers:

     1     3     5     7     9    11    13    15

The reciprocals:

1.0000    0.3333    0.2000    0.1429    0.1111    0.0909    0.0769    0.0667
```

Their sum and product are:

```
   2.0218    4.9333e-007
```

vii. Element by element powers

We start with a vector:

```
>> x = [-1: .5 : 1]   % section I.2,v

   -1.0000   -0.5000     0    0.5000    1.0000
```

Suppose that we want to raise each element of x to the third power. We use the point-wise power operator .^ and write (x).^3. As with multiplication it is a good habit to use many parentheses.

```
>> (x).^3

   -1.0000   -0.1250   0    0.1250    1.0000
```

On the other hand, if we want to use the elements of the vector x as powers to which we raise the number 3 then we write (3).^x.

```
>> (3).^x

    0.3333    0.5774    1.0000    1.7321    3.0000
```

To show what can happen if we do not put parentheses consider:

```
>> 3.^x
```

This is ambiguous since we are perhaps referring to the number 3.0. MATLAB responds with:

Error using ^
Matrix must be square.

One can also raise each element of a vector to a power determined by the elements of another vector of the same size.

```
>> (x).^x

  -1.0000  {0.0000 - 1.4142i}   1.0000  0.7071 1.0000
```

The element is the complex number obtained by evaluating (-.5)^(-.5):

```
 >> (-.5)^(-.5)
```

SECTION II.2. FIVE SPECIAL MATRICES

SUMMARY

[Note: It is for the commands in this section that MATLAB has changed the most. Because of this the newer syntax is given first and then this is followed by the older syntax. If the newer command does not work, then try the older form or type help eye etc. One difference that is common to the first four forms is that the term size is now needed; e.g before one could write ones(A) to obtain an identity matrix with the same dimensions as A, but now one must write ones(size(A)).]
i. Identity matrix.

eye(3) or eye(size(A)), if A is already defined and square.

older versions: eye(A).
ii. Matrix of ones.

ones(1 , 5):   a vector of length 5 of ones.
ones(3 , 5):   3 rows, 5 columns.
ones(3):   3 x 3 square matrix of ones.
ones(size(A)):   If A is already defined.

older versions: ones(A)
iii. Matrix of zeros.

zeros(1 , 5):   a vector of length 5 of zeros.
zeros(3 , 5):   3 rows, 5 columns.
zeros(3):   3 x 3 square matrix of zeros.
zeros(size(A)):   If A is already defined.

older versions: zeros(A)
iv. Matrix of random elements.

random elements in [0,1) ("uniform generator")

rand or rand(1) - one random number
rand(1 , 5):   a vector of length 5 of random numbers.
rand(3 , 5) - 3 rows, 5 columns.
rand(3) - 3 x 3 square matrix of random numbers
rand(size(A)) - If A is already defined.

older versions: rand(A)

To have a different set of random numbers corresponding to the initial seed 9956295 : rand('seed' , 9956295)

Repeating this command will result in the same initial seed as the first time that this command was issued.

To print out the present value of the seed: rand('seed') [newer versions only].

See the discussion for details and problems associated with the seed.

[note: for testing purposes use the built in seed or a fixed seed; for serious work pick your seed at random, e.g. via randomly picked telephone numbers.]

normal random numbers with mean 0 and variance 1

In the newer versions the commands are the same as for random numbers in [0,1) except that rand is replaced by randn, i.e. "n" is added on. A seed change for the normal generator does not change the seed for the [0,1) generator. To return to the [0,10 generator one simply uses the rand( ) commands given above.

newer versions: randn(1,5), rand('seed' , 127269), randn('seed' , 0), randn('seed')

older versions:
to switch to the normal generator: rand('normal')
to return to the [0,1) generator: rand('uniform')
to change the seed: rand('seed', 7626)
v.   Row vector of indices.

[-1 : .5 : 2] gives -1 to 2 by steps of .5
vi.   The empty matrix [ ].
--

DETAILS AND EXAMPLES

i. Identity matrices.

MATLAB uses the function eye( ) to define the identity matrix.

```
>> IDENT = eye(2)

IDENT =
    1     0
    0     1
```

If however, we have already defined a square matrix A and we want an identity matrix of the same size, we can simply type eye(A):

```
>> A = [4 -1 ; 3 0]

    4    -1
    3     0
```

```
>> eye(size(A))

     1     0
     0     1
```

The use of eye(size (A)) is very useful in multiple operations:

```
 >> A - 3*eye(size(A))

     1    -1
     3    -3
```

ii. Matrix of ones.

A matrix all of whose elements equal 1 is defined using the function ones(2 , 3).

```
>> a = ones(3 , 1)   % column vector of ones

     1
     1
     1
```

If the matrix is square one can write ones(2):

```
>> E = ones(2)

     1     1
     1     1
```

If A has already been defined we can use ones(size(A)):

```
>> E = ones( size(A) )

     1     1
     1     1
```

A vector of 1's is useful in constructing a matrix all of whose rows are equal to a given vector:

```
>> b = [2 4 6];

>> a*b                     % a = ones(3,1); see above

     2     4     6
     2     4     6
     2     4     6
```

This idea is used in the function "normaliz.m" which is defined in section III.3 program 1.

iii. Matrix of zeros.

A matrix all of whose elements equal 0 is defined using the function zeros(2 ,3).

```
>> B = zeros(2 , 3)

     0     0     0
     0     0     0
```

If the matrix is square one can use zeros(2).

```
 >> D = zeros(2)

     0     0
     0     0
```

If A has already been defined we can use zeros(size(A)):

```
>> F = zeros( size(A) )

     0     0
     0     0
```

A vector of zeros is very useful in predefining the size of a matrix. We can then fill in various positions with non-zero elements. Suppose that B is a 2 x 3 matrix of zeros obtained, as above, by zeros(2 , 3) as above. If we redefine the element in row 2 and column 3 to be 5, then we obtain a matrix of the same dimensions as B, but with a 5 replacing the 0 in row 2, column 3.

```
>> B(2 , 3) = 5      % see section I.2.iii

B =

     0     0     0
     0     0     5
```

The use of a predefined shape helps to avoid confusion and error statements. Thus you may think that you are working exclusively with row vectors, but since MATLAB tends to work with column vector --e.g. when finding the eigenvalues of a matrix--you may end up with incompatible matrices. This idea is exploited in section II.4, viii and the program "rand1.m" which is discussed in the section II.6.

SECTION II.2. FIVE SPECIAL MATRICES

iv. Matrix of random elements.

A matrix whose elements are "random" (i.e. are numbers between 0 and 1 which act statistically as if they were picked by chance) is defined using the function rand( ). The examples here use the older commands, for the newer ones and also additional commands see the summary.

To obtain single elements we use rand( ) or simply rand:

```
 >> rand

    0.2190

>>  rand(1)

    0.0470
```

To obtain a random vector of length 3:

```
>>  rand(1 , 3)

    0.2190    0.0470    0.6789
```

If the matrix is square one can just write rand(2).

```
>> rand(2)

    0.6793    0.3835
    0.9347    0.5194
```

If A has already been defined we can use rand(size(A)).

```
>> rand( size(A) )

    0.5017    0.7013
    0.4919    0.9217
```

Seeds: The basic MATLAB random number generator is based on a multiplicative congruential generator, i.e., a relationship of the form:

Zn = a* Zn-1 mod m

The initial value Z0 is called the seed or the initial seed.

In the context of MATLAB the Zn are also "seeds" (see below) although it would be better to think of them as the present value of the seed.

In some versions of MATLAB the following values of a and m are used:

a = 75 = 16807,   m = 231 -1 = 2147483647

The actual implementation of the algorithm depends upon the machine and is based on various numerical "tricks". If you write and test your program on say a PC, and then switch to a big UNIX machine, the numbers that you obtain might be different. This in itself is not a problem; however difficulties might arise at the level of testing your program (see below)

Once the values of the Zn are obtained we obtain our random numbers, which fall in the interval [0,1) by division:

xn = Zn/m

Random numbers are used in simulations and in Monte Carlo evaluations, and they are also used to generate data for the purpose of testing statistical procedures. They are also quite useful for making up numerical examples involving matrices (see Sections III.1.vii and III.4).

For the purpose of testing programs we would like to be able to always obtain the same set of random numbers. On the other hand when actually running an experiment on a computer we want to make sure that we obtain a completely new set of random numbers. For otherwise our simulation would not be completely random. In order to replicate the sequence of random numbers we simply reset the seed (which we can consider as the new initial seed). Suppose that we decide to use 9956295 (the telephone number of NSERC, the Natural Sciences and Engineering Research Council of Canada), we use the command:

```
>> rand('seed', 9956295)  % set the seed

>> rand                   % generate a random number
   0.9216
```

Now we reset the seed to the same value 9956295

```
>> rand('seed', 9956295)
>> rand                   % generate a random number
   0.9216                  % the same number before
```

[Note: MATLAB has a built in inItial seed. On Unix version 5.3.1.29215a (R11.1) the value is 931316785. In order to avoid obliging the user to retype this number each time, MATLAB switches to this value when 0 is used as the seed.

Using the function rand( ) is a very quick way of generating matrices for testing.

In section III.1.vii we test how long it takes for MATLAB to solve equations by generating 1000 equations in 1000 unknowns where the coefficients and constants are picked at random. This only takes a few seconds whereas typing in 1,001,000 numbers would take slightly longer.

v. Vectors of Indices.

The concept of building a row vector of indices has already been discussed in section I.2.iv, in connection with the semicolon, but a brief summary is repeated here for reference and also to emphasize that we are creating a type of matrix.

To obtain the index set -1, 0,... , 4 we write:

```
>> x = [-1 : 4]

   -1     0     1     2     3     4
```

To obtain the index set -1, -.5, 0, .5, ... , 4 we write:

```
>> x_step = [-1 : .5 : 2]

   -1.0000   -0.5000  0   0.5000  1.0000   1.5000   2.0000
```

To obtain the index set 5, 4, 3, 2, 1 (i.e. in reverse order) we write:

```
>> X_dec = [5 : -1 : 1]

     5     4     3     2     1
```

(Note: Writing [5 : 1] would result in MATLAB responding with the empty matrix [ ]).

vi. The empty matrix [ ].

This matrix is often very useful because it can serve as the initial matrix when building up matrices by concatenation, see section II.4.ix. The empty matrix is treated by MATLAB in exactly the same way it treats any other matrix.

```
>> length( [ ] )

           0

>> A =  [ ]*[ ]
   A =
       [ ]
```

As the last note in v. above shows, an incorrect statement may lead to MATLAB returning the empty matrix.

SECTION II.3. COMMON MATLAB FUNCTIONS

SUMMARY

In the following A represents a matrix, square where required, and x, y vector (row or column). Some of the functions are discussed in detail in other sections and for these only a short example or the function are given together with a reference.

Note that in general MATLAB functions, e.g. sum( ), operate on the columns of a matrix; to work with rows we use the ordinary transpose of A which is written A' if all the entries are real or (A).' . The commands are in usually written in terms of a matrix A, but they apply equally well to a vector v.

i. Linear algebra.

Examples of the following functions are given in the details section.

The solution of equations using rref( ), inv( ), COEFF\const and lu( ) is discussed in detail in section III.1.

The computation of eigenvalues and eigenvectors is discussed in detail in section III.2.

ordinary transpose when all entries are real: A'
ordinary (true) transpose with complex entries: (A).'
conjugate transpose: A'
inverse: inv(A)
reduced row echelon form (Gauss-Jordan): rref(A)
determinant: det(A)
eigenvalues as a column vectors of a matrix A: eig(A)
eigenvectors (right) & eigenvalues in a diagonal matrix:
    [VEC , VAL] = eig(A)
eigenvectors (left) & eigenvalues in a diagonal matrix:
    first write: [VEC_L , VAL] = eig(A.')
    then: VEC_L = (VEC_L).'
dot product (program 1): dot(x,y);
cross product (program 2): cross(x,y)

ii. Matrix manipulations.

Short examples of the following functions are given in the details section. More involved examples are given in section II.4 (extracting and building matrices) and throughout chapter III.

transpose: see i above
dimensions of a matrix A as variables: [rows , cols] = size(A)
length of a vector v: length(v)

SECTION II.3. COMMON MATLAB FUNCTIONS

diagonal of a matrix as column vector: diag(A)
to form a matrix whose diagonal is a vector d: diag(d)
off diagonals of a matrix: diag(A , 1) , diag(A , -1)
trace (sum of diagonal): trace(A)
rotate 90o counterclockwise: rot90(A)
change the order of a vector: rot90(rot90(A))
reflect matrix in vertical axis (flip left to right): fliplr(A)
reflect vector in vertical axis (flip left to right): fliplr(v)
reflect matrix in horizontal axis (flip up to down): flipud(A)
reflect vector in horizontal axis (flip up to down): flipud(v)

iii. Pi, square and other roots, complex numbers, roots of polynomials

3.14159 : pi
square root of a number (real or complex)
 sqrt( ) or define the function R( )
principal nth root: ( )^(1/n)
square root of -1: i or j
 [avoid using i and j elsewhere; use n for loops]
complex number a + bi: (a + b*i)
 [use parentheses to be sure that the entry is correct.]
zeros of a polynomial:
p = [1 0 0 1] is the vector of coefficients
roots(p)

iv Summing, maximum etc.

Examples of the use of the following operations are given in section II.5, rather than in this section.

Notice the use of the transpose operator **'** *when working with rows. This is done because the functions are intrinsically column operators.*

sums of the columns of a matrix as a row vector: sum(A)
sums of the rows of a matrix A as a row vector: sum(A.')
sum of a vector (row or column): sum(x)
 [for cumulative sums use cumsum(A)]
products: prod(A)]
averages of the columns of a matrix as a row vector: mean(A)\mni
averages of the rows of a matrix A as a row vector: mean(A.')
average of the elements of a vector (row or column): mean(x)
 [for medians use median(A); for standard deviation: std(A)]
maximums of the columns of a matrix as a row vector: max(A)
maximums of the rows of a matrix A as a row vector: max(A.')

minimums of the columns of a matrix as a row vector: min(A)
minimums of the rows of a matrix A as a row vector: min(A.')
maximum and minimum of a vector x: max(x), min(x)

v. Real and imaginary parts etc.

matrix of real parts: real(A)
matrix of imaginary parts: imag(A)
matrix of conjugates parts: conj(A)
matrix of amplitudes (radians): angle(A)

vi. Rounding the answer etc.

matrix of absolute values: abs(A)
matrix of values rounded to the nearest integer: round(A)
matrix of values rounded downwards : floor(A)
matrix of values rounded upwards : ceil(A)
matrix of values rounded towards 0: fix(A)
fractional part: define the function fract( )
matrix of sign function (1, 0, -1) : sign(A)

vii. Exponential type (pointwise and matrix)

exponential (pointwise): exp(A)exponential (matrix): expm(A)
logarithmic pointwise: log(A)
logarithmic (base 10): log10(A)
square root: sqrt(A)

viii. Trigonometric.

trigonometric: sin(A), cos(A), tan(A)
inverse trigonometric: asin(A), acos(A), atan(A)
hyperbolic: sinh(A), cosh(A), tanh(A)
inverse hyperbolic: asinh(A), acosh(A), atanh(A)

ix. Timing.

time: clock
elapsed time: t1 = clock; t2 = clock; etime(t2,t1)
--

SECTION II.3. COMMON MATLAB FUNCTIONS

DETAILS

When several functions in a group work in exactly the same way, then only one example will be given, e.g only sin in the trigonometric group.

i. Linear algebra.

The solution of equations using rref(); inv( ) and COEFF\const is discussed in detail in section III.1. The computation of eigenvalues and eigenvectors is discussed in detail in section III.2.

The following matrices and vectors will be used in this subsection.

```
A =
   -1     2     0
    3     1    -2

B =
   -1     2
    3     1

x =
   -1     2     4

y =
    3     1    -3
```

Transposes require special attention in MATLAB. If the entries of a matrix are real then we can use '

```
>> A_TRANSPOSE = A'

   -1     3
    2     1
    0    -2
```

However if the entries are complex then in order to obtain the ordinary transpose, which will usually be referred to as the true transpose, we have to use .'

```
complex =

   -1     i        % i is the square root of -1; see iii.

>> (complex).'      % true transpose uses dot-apostrophe
```

```
   -1
    i
```

If however we were to use the symbol **'** without the period then what we obtain is the conjugate transpose:

```
>> (complex)'       % conjugate transpose

   -1
   -i
```

Special care must be taken when dealing with eigenvalues and eigenvectors, because even though all the entries in a matrix may be real, the eigenvectors and eigenvalues may be complex; see the examples in section III.2.

```
>> B_INVERSE = inv(B)

  -0.1429    0.2857
   0.4286    0.1429

>> det_B = det(B)

   -7
```

Eigenvectors are discussed in detail in section III.2.

```
>> eigenvalues = eig(B)

  -2.6458
   2.6458

>> [EIGENVECTORS , EIGENVALUES] = eig(B)

EIGENVECTORS =

  -0.7722   -0.4810
   0.6354   -0.8767

EIGENVALUES =

  -2.6458         0
        0    2.6458

>> ECHELON_a = rref(A)

   1.0000         0   -0.5714
```

```
         0    1.0000   -0.2857
```

For the dot and cross products the following user defined functions may be used.

Program 1

```
% [dot.m]
%  dot(x,y) computes the dot product of two vectors x and y,
%  if x and y are both row or both column vectors.

function v = dot(x,y)

% use point wise multiplication & add terms
v = sum(x .* y);
```

Program 2

```
% [cross.m]
% cross(x,y) computes the cross product of two
% 3-dimensional row vectors.

function v = cross(x,y)

B =[x ; y];                         %  form 2 x 3 matrix

% the ti are minors of 3 x 3 matrix with i j k in first row
t1 =  det( B(: , [2 , 3]) );        % for t1 use cols 2 & 3
t2 = -det( B(: , [1 , 3]) );        % for t2 use cols 1 & 3
t3 =  det( B(: , [1 , 2]) );        % for t3 use cols 1 & 2

v = [t1, t2, t3];                   % put terms in a vector
```

We test these functions out on x and y as above:

```
>> do = dot(x,y)

   -13

>> cr = cross(x , y)

   -10     9    -7
```

ii. Matrix manipulations.

The following matrices and vectors wiil be used in this subsection:

SECTION II.3. COMMON MATLAB FUNCTIONS

```
C =

   -3     6    12
   -1     2     4
    3    -6   -12

x =

   -1     2     4

>> [ rows , cols] = size(C)

     3  3               % rows = 3, cols = 3

>> length(x)

     3                  % length of x

>> diag(C)              % for the main diagonal

   -3
    2
  -12

>> diag(C , 1)          %  first "off diagonal" above the main

    6
    4

>> diag(C , -1)         %  first "off diagonal" below the main

   -1
   -6

>> trace(C)   % sum of diagonal elements

  -13

>> rot90(C)   % counter clockwise rotation

   12     4   -12
    6     2    -6
   -3    -1     3
```

SECTION II.3. COMMON MATLAB FUNCTIONS

To change the order of a matrix.

Sometimes we want to change the order of a vector row or column. For example the function sort( ) - section II.5 - sorts the elements of a vector in increasing order and we may want the elements in a decreasing order. The use of the function rot90( ), which rotates a matrix 90o in a counterclockwise direction, twice in succession will reverse the order for both column and row vectors. For this reason it should be used in programs rather than the next two functions fliplr( ) and flipud( ), which will work on row and column vectors respectively.

```
>> rot90(rot90( x ))

     4     2    -1
```

To take the *mirror image* of a matrix about a vertical axis:

fliplr(   )   (FLIP Left to Right)

```
>> C3 = fliplr(C)

    12     6    -3
     4     2    -1
   -12    -6     3
```

To take the mirror image of a matrix about a horizontal axis:

flipud( )   (= flip Up to Down)

```
>> C4 = flipud(C)

     3    -6   -12
    -1     2     4
    -3     6    12
```

iii. Pi, square roots and other roots, complex numbers,
              zeros of polynomials.

The number pi:

```
>> pi

    3.1416

>> format long               % section I.5.ii

>> pi
```

```
   3.14159265358979

>> format             % return to short format (or format short)
```

Roots:

MATLAB use the symbol sqrt( ) for the square root of a real number and the principal square root of a negative or complex number. This can be cumbersome so we use the original square root symbol R( ) which goes back to the Italian Renaissance (R stood for radix = radicii = racine].

```
% [r.m]
% instead of sqrt(x)
function y = R(x)
  y = sqrt(x);
```

Consider the following calculation:

```
>> c =  R(1 + R(1 + R(1 + R(1 + R(1 + R(1 + R(1 + R(1))))))));

1.61785129060968
```

Of course if we wanted to continue the process to see if there is convergence we would want to use a "while-loop" (II.8.5)

```
% [it_gn.m]
format long

initial = input(' What is the initial value? ');
a = initial;
b = -100;
counter = 0;
while abs(a-b) > 10^(-15)
     b = a;
     a = R(1 + a);
     counter = counter + 1;
end
```

Run the program:

```
>> it_gn

initial value is: 1
last value is     1.61803398874990
the process took  30 iterations.
```

In fact the number to which the process converges is:

```
>> G = (1 + R(5))/2;

                 1.61803398874990
```

For other roots use ( )^(1/n)

```
 (2)^(1/5)

    1.1487
```

Complex numbers.

The imaginary number "square root of -1" is obtained by typing i or j. Note that "i" appears twice in the following. On the right, and in all complex numbers, it is used to indicate the imaginary part. On the left i is used to obtain the complex number with real part = 0 and imaginary part 1.

```
>> i

        0 + 1.0000i

>> j

        0 + 1.0000i
```

Because of this use of i and j one should avoid using i and j elsewhere; use m or n for indices in loops.

If we were to define i = 5 then whenever we type i we obtain the value 5:

```
>> i = 5

i =

     5
```

We can also obtain 0 + 0i ourselves by computation:

```
>> im = (-1)^(1/2)

   0.0000 + 1.0000i

>> sqrt(-1)            % sqrt(  ) is the square root function
```

```
   0 + 1.0000i

>> b = (-1)^(1/3)

   0.5000 + 0.8660i
```

We check this last calculation by taking powers:

```
>> b^2

  -0.5000 + 0.8660i

>> b^3

  -1.0000 + 0.0000i      % complex number with imaginary part 0
```

zeros of polynomials.

Note that only only one cube root of -1 is returned in the calculation for "b". To obtain all the cube roots of -1 we use the function roots( ); see also section II.7.ii.

First we define the polynomial vector which corresponds to the defining equation for the cube roots of -1, namely x^3 - 1 = 0. We simply write down the coefficients, putting a 0 whenever a term is missing, and then use roots( ):

```
>> p = [1 0 0 1];

>> roots(p)

  -1.0000
   0.5000 + 0.8660i
   0.5000 - 0.8660i
```

iv. Summing, maximum etc.

Examples of the use of the following operations are given in section II.5, rather than in this section.

v.   Real parts etc.

We start with the following two matrices, which we will then combine and manipulate:

```
REA =

   -1     2     0
    3     1    -2

 IMAG =

    0.5000   -0.6000    0.7000
   -0.1000    0.2000    0.3000


>> i * IMAG               % multiply IMAG by i

        0 + 0.5000i        0 - 0.6000i        0 + 0.7000i
        0 - 0.1000i        0 + 0.2000i        0 + 0.3000i

>> C = REA + i*IMAG       % form a complex matrix

  -1.0000 + 0.5000i   2.0000 - 0.6000i        0 + 0.7000i
   3.0000 - 0.1000i   1.0000 + 0.2000i  -2.0000 + 0.3000i

>> real(C)                          % real part of C = REA

   -1     2     0
    3     1    -2

>> imag(C)                         % imaginary part of C = IMAG

    0.5000   -0.6000    0.7000
   -0.1000    0.2000    0.3000

>> conj(C)                         % conjugate matrix

  -1.0000 - 0.5000i   2.0000 + 0.6000i        0 - 0.7000i
   3.0000 + 0.1000i   1.0000 - 0.2000i  -2.0000 - 0.3000i

>> abs(REA)                      % absolute values of REA

REA =

     1     2     0
     3     1     2

>> abs(IMAG)                     % absolute values of IMAG

IMAG =
    0.5000    0.6000    0.7000
    0.1000    0.2000    0.3000

>> abs(C)                          % absolute values of C

    1.1180    2.0881    0.7000
    3.0017    1.0198    2.0224

>> ang = angle(C)                  % amplitudes in radians

    2.6779   -0.2915    1.5708
   -0.0333    0.1974    2.9927

>> ang_deg = (180/pi)*ang          % amplitudes in degrees

  153.4349  -16.6992   90.0000
   -1.9092   11.3099  171.4692
```

vi. Rounding the answer etc.

Often we want an integer valued answer. The following MATLAB functions are used for this type of operation:

matrix of values rounded to the nearest integer:	round(A)
matrix of values rounded towards - :	floor(A)
matrix of values rounded towards + :	ceil(A)
matrix of values rounded towards 0:	fix(A)
fractional part: define the function	fract( )
matrix of sign function (1, 0, -1) :	sign(A)

```
>>  z =

   -5.6000    5.6000

>> round(z)

    -6     6

>> floor(z)

    -6     5

>> ceil(z)
```

```
    -5     6


>> fix(z)

    -5     5
```

For the decimal part of a number, written as a positive quantity we can use the following function.

```
% [decimal.m]
% finds absolute value of fractional part

function y = decimal(x)

x1 = abs(x);
y =  (x1 - floor(x1));

>>  z =

   -5.6000    5.6000

 decimal(z)

    0.6000    0.6000
```

Another function which is useful is sign( ) which returns -1, 0 or 1 depending on whether the number is negative, zero or positive:

```
>> w =

   -5.4000         0    5.4000

>> sign(w)

    -1     0     1
```

Instead of -1, 0, 1 we can also return -, {space} or + by using the format + command:

```
>> format +
>> w

    -  +                  % there is a space in the second position
```

SECTION II.3. COMMON MATLAB FUNCTIONS

vii. Exponential type (pointwise and matrix)

MATLAB functions operate pointwise on vectors and matrices

Suppose that we have a vector:

```
x =

     1     2     3     4
```

When we write exp(x) MATLAB calculates "e" to each element of x

```
>> exp(x)

    2.7183    7.3891   20.0855   54.5982
```

In this case we can obtain the same answer by taking “e” *pointwise* to the “power” x (see section II.1)

```
>> base_e = exp(1)

         2.7183

>> ( base_e) .^(x)      % .^ is pointwise power

   2.7183    7.3891   20.0855   54.5982
```

We can also work with matrices

```
M5=

    0.5000   -0.6000    0.7000
   -0.1000    0.2000    0.3000

>> exp(M5)

    1.6487    0.5488    2.0138
    0.9048    1.2214    1.3499
```

There is a matrix version expm( ) of the exponential function which uses the power series for e^x, but with powers of the matrix. The difference can be seen by working with the identity matrix.

SECTION II.3. COMMON MATLAB FUNCTIONS

```
>> R = eye(2)

     1     0
     0     1

>> exp(R)                  % pointwise powers of e

    2.7183    1.0000        % e1 = e;  e0 = 1 ;
    1.0000    2.7183

>> expm(R)                  % expm(R) = R + R2/2! + R3/3! + ....

    2.7183         0
         0    2.7183

B1 =

   -0.1250    0.2500
    0.3750    0.1250

>> log(B1)

  {-2.0794 + 3.1416i}    -1.3863
  -0.9808                -2.0794

>> sqrt(2)
    1.4142

>> format long

>> sqrt(2)

   1.41421356237310

>> sqrt(B)

   {0 + 1.0000i}      1.4142
   1.7321             1.0000
```

vii. Trigonometric functions.

The function names for the MATLAB trigonometric and hyperbolic functions are the same as the usual names; the inverse functions have the prefix "a" e.g. "atan". These functions operate element wise on matrices. See section II.5.i for an example of the construction of a trigonometric table.

```
B =
    -1     2
     3     1

>> tan(B)

   -1.5574   -2.1850
   -0.1425    1.5574

>> 4*atan(1)                    % arctan(1) = pi/4

   3.14159265358979  % = pi

B1 =

   -0.1250    0.2500
    0.3750    0.1250

>> acos(B1)

    1.6961    1.3181
    1.1864    1.4455


>> sinh(B)                    % hyperbolic sine

   -1.1752    3.6269
   10.0179    1.1752

>> atanh(B1)                  % inverse hyperbolic tangent

   -0.1257    0.2554
    0.3942    0.1257
```

viii. Timing.

MATLAB has a clock function clock, which gives the time on the computers clock:

```
>> t1 = clock

t1 = {1.0e+03} *
1.9940   0.0100   0.0170   0.0090  0.0520  0.0384
```

which is to be interpreted as:

1994, month 10, day 17, 09 hours, 52 minutes, 38.4 seconds)

SECTION II.3. COMMON MATLAB FUNCTIONS

To find out how long it takes for MATLAB to do a computation (or how long it took to debug your program!) we use clock in conjunction with the function etime(,) in the following way:

```
>> t1 = clock
>> ....  do some calculations
>> t2 = clock
>> elapsed_time = etime(t2 , t1)
```

Note how the last time comes first in etime.

Example 1. The program "elapsed.m" calculates how long it takes MATLAB to generate a vector of random numbers. The number of terms is an input quantity from the keyboard (section I.4.iii).

Program 1

```
% [elapsed.m]
% input size of vector from the keyboard
terms = input(' how many random numbers, eh?  ')

t1 = clock;

vector = rand(1, terms);                    % do not print!

t2 = clock;

elapsed_time = etime( t2 , t1)              % last time first
```

We now run this proram for progressively larger vectors of random numbers:

```
>> elapsed

how many random numbers, eh?  1000

    0.0284   (seconds)

how many random numbers, eh?  10000

   .1629      (seconds)

how many random numbers, eh?  100000

   2.9340     (seconds)
```

```
how many random numbers, eh?  1000000

   35.5314 (seconds)
```

Note that the time goes up by much more than a factor of 10 each time which shows that the time to generate vectors of random numbers is not linear. The time also varies as we repeat the program with the same numbers; this is because MATLAB is generating a different set of random numbers.

In section III.1.vii we will use the above technique to see how long it takes to solve 1000 linear equations in 1000 unknowns when the coefficients and constants are chosen at random. Guess now how long it will take.

II.4. Extracting and Building Matrices

SUMMARY

i. Building matrices from vectors.

To form the matrix with row 1 = vec1 and row 2 = vec2:

```
A = [vec1; vec2]
```

To form the matrix with column 1 = (vec1)', column 2 =(vec2)'

```
B = [vec1', vec2']
```

ii. Extracting elements and vectors from matrices.

To extract elements:

```
b = A(1 , 2) for matrices
b = a(3) for vectors
```

To extract vectors:

```
row_2 = M(2 , : )      [extract row 2 as a vector]
column_3 = M( : , 3)   [extract column 3 as a vector]
```

iii. Extracting several rows or columns from a vector or matrix.

vectors:
For sequential terms in vector v:

```
v1 = v( [2 : 5] )   [elements 2 through 5]
```

for a non-sequential, not necessarily increasing, set of indices of vector v:

```
index_1 = [5, 2 , 4]
v2 = v(index_1)     [elements 5, 2, 4 of v]
```

matrices:
To form the matrix N1 which consists of rows 3 through 6 of matrix M. Note the colon " : " which says that we should keep the elements in all the columns of M.

```
N1 = M( [3 : 6] , : )
```

To form the matrix N2 which consists of those columns of matrix M that are given by the vector index_2 and in the same order as in index_2. Note the colon " : " which says that we should keep the elements in all the rows of M.

```
index_2 = [5, 1:3]   [col. 5 and then cols. 1 through 3]

N2 = M( : , index_2)
```

To form the matrix N3 which consists of those elements of M which are in the rows given by an index vector index_3 and columns of matrix M that are given by the vector index_4.

II.4. Extracting and Building Matrices

```
N3 = M( index_3 , index_4  )
```

iv. Replacing, or defining, a column or row of a matrix by a vector. Replacing certain elements of a vector by another vector

```
M4 = M;         [we wish to save M so copy it]
M4( : , 4) = vec_4   [column 4 of M4is now equal to vec_4]
M4( 3 , :) = vec_5   [row 3 of M4 is now equal to vec_3]
```

index = [2 : 5]; w = [9 -7 3]
v(index) = w   [v(j) = w(j) if j is not in *index*]

v. Building matrices from submatrices.

M = [A1 , A2 ; A3 , A4] where the submatrices are compatable.

vi. Replacing a block of a matrix by a given matrix.

To replace the upper right 2 x 3 block of a matrix M by a 2 x 3 matrix ADD:

```
M( [1 : 2] , [3 : 5]) = ADD
```

vii. Building matrices recursively by assignment and by concatenation; the empty matrix: [ ].

viii. Storing and "fetching" sequences of matrices.

ix. Building a diagonal matrix whose diagonal is a given vector. Extracting a diagonal from a matrix. Off-diagonal matrices.

```
D = diag( dia )
```

x. Building matrices with many zeros; predefining the dimensions.

Construct a matrix of zeros and place the vectors of values via assignment and simple "for-loops"
--
xi. Forming a vector from the elements of a matrix and inversely.

To form a row vector from the elements of M: vec = M(:)
To form a 3 x 4 matrix from a 1 x 12 vector a:

```
M = zeros(3 , 4)
M( : ) = a
```

[The command reshape(M, 3 , 4) can be used to reshape a matrix]

II.4. Extracting and Building Matrices

DETAILS AND EXAMPLES

i. Building matrices from vectors.

To illustrate the building of matrices from row vectors we create three row vectors of indices in the manner discussed in section II.2.v.

```
>> row_1 = [3 : 6]

     3     4     5     6

>> row_2 = [7 : -1 : 4]

     7     6     5     4

>> row_3 = [-3 : 0]

    -3    -2    -1     0
```

To form a matrix M whose rows are equal to these three row vectors, we use square brackets together with the semicolon to indicate that we want the vectors in diffferent rows:

```
>> M = [row_1 ; row_2 ; row_3]

     3     4     5     6
     7     6     5     4
    -3    -2    -1     0
```

If we had used commas then MATLAB would have put the three vectors in the same row:

```
>> M1 = [row_1 , row_2 , row_3]

3   4   5   6    7    6    5   4  -3    -2    -1   0
```

If instead of row vectors, we have column vectors then we form a matrix using commas to indicate to MATLAB that the vectors should be placed side by side. To illustrate this, we take the transpose of each row vector using the true transpose operator ' .

```
M2 = [row_1' , row_2', row_3']

3    7    -3
4    6    -2
5    5    -1
```

```
6    4     0
```

ii. Extracting elements and vectors from matrices.

If a is a vector then all we have to do is give the element number:

```
>>  a =  [1 3 5];

>>  b = a(2)

      3
```

If A is a vector then we specify the row and column:

```
>>  c = M(2 , 3)

      5
```

To extract row 2 from the matrix M we use parentheses and separate the row and columns by a comma. Since the column values range over all the columns of M we indicate this by a colon.

```
>> extract_row_2 = M(2 , : )

     7     6     5     4
```

To extract column 3, we let the rows range over all values, again using :

```
>> extract_col_3 = M( : , 3)

     5
     5
    -1
```

iii. Extracting several rows or columns from a matrix.

If we want to pull out the range of columns 2, 3, 4 we list these in the column position using
[2 : 4]:

```
>> extract_1 = M(: , [2 : 4])

     4     5     6
     6     5     4
    -2    -1     0
```

II.4. Extracting and Building Matrices

Instead of listing the columns inside the parentheses, we can use an index vector:

```
>> index_1 = [2 : 4]

     2     3     4
```

We then place this index vector in the columns position:

```
>> extract_1a =M( : , index_1)

     4     5     6
     6     5     4
    -2    -1     0
```

In this last example, we listed the columns that we wanted to extract in increasing order. If we specify a different order, then MATLAB will automatically do the required switching of the columns:

```
>> index_2 = [3  4  2]

     3     4     2

>> extract_2 = M( : , index_2)

  % [col3  col4  col2  of M]

     5     6     4      % interchanged elements of row 1
     5     4     6
    -1     0    -2
```

We can work with both row and column index vectors at the same time:

```
>> index_3 = [3 : -1 : 1]

     3     2     1

>> extract_3 = M(index_3, index_2)

    -1     0    -2
     5     4     6
     5     6     4
```

iv. Replacing, or defining, a column or row of a matrix by a vector.

Suppose that we want to replace the fourth column of the matrix M by the column vector:

```
vec_4 =

   11
   12
   13
```

Since we may want to use the matrix M again later on, we first make a copy of it:

```
>> M4 = M;
```

To assign the vector col_4 to the fourth column of M4, we use a colon to indicate that we are ranging over all the rows of M4 and then we indicate that we want column 4 to be changed.

```
>> M4(:, 4) = vec_4

 %                  the new column 4 = vec_4

    3     4     5    11
    7     6     5    12
   -3    -2    -1    13
```

If had wanted to replace the third row of M by a 1 by 4 row vector vec_5 we would write:

```
>> M4(3, :) = vec_5
```

Suppose that we want a 1 x 6 vector such that the even elements are all equal to 3 and the odd ones are 7. We first define a 1 x 6 vector z of 0s.

```
>> z = zeros(1,6)
```

We then define two index sets and the vectors of elements that we want in those indices

```
>> index1 = [2, 4, 6]  % or index1 = [2 : 2 : 6]
>> w1 = [1 2 3]

>> index2 = [1, 3, 5]  % or index2 = [1 : 2 : 5]
>>  w2 = [4 5 6]
```

Next we replace the elements of z:

```
% first replace those elements of z which correspond to elements of index1
% by the element of w1 which occupies the same position.
```

```
>> z( index1 ) = w1

%  the elements of z in positions 2, 4 and 6, will be replaced by
%  1, 2 and 3 respectively. The other positions will remain unchanged.

>> disp(z)

    0 1 0 2 0 3

%  now do the same with index2

>> z( index2 ) = w2

>> disp(z)

    4 1  5  2 6 3
```

The same technique can be used to replace elements of the rows (or columns) of a matrix which correspond to a given index set. See example 1 of section x for another example.

v. Building matrices from submatrices.

In i we built up matrices from vectors. A more general situation is to build up matrices from submatrices and to do this the semicolon and comma are used in the same way as with vectors. The key to combining matrices (even more so than in marriages) is compatibility. A general rule of thumb is that MATLAB will do more or less anything if the dimensions are compatible.

We first form four submatrices of dimensions 1 x 2, 1 x 3, 3 x 3 and 3 x 2 using the function ones( ) discussed in II.2.ii.

```
>> SUB_1 = 1*ones(1 , 2)

    1     1

>> SUBS_2 = 2*ones(1,3)

    2     2     2

>> SUB_3 = 3*ones(3,3)

    3     3     3
    3     3     3
    3     3     3
```

II.4. Extracting and Building Matrices

```
>> SUB_4 = 4*ones(3,2)

    4    4
    4    4
    4    4
```

We can combine the first two and then the third and fourth so as to form a 1 x 5 row vector and a 3 x 5 matrix. Note how the comma is used to put the submatrices on the same "row".

```
>> COMBINED_1 = [SUB_1 , SUB_2]

    1    1    2    2    2

>> COMBINED_2 = [SUB_3 , SUB_4]

    3    3    3    4    4
    3    3    3    4    4
    3    3    3    4    4
```

We join COMBINED_1 and COMBINED_2 together using the semicolon to put them on separate "rows":

```
>> COMBINED_3 = [COMBINED_1 ; COMBINED_2]

    1    1    2    2    2
    3    3    3    4    4
    3    3    3    4    4
    3    3    3    4    4
```

We could have done all the combining at once:

```
>> COMBINED_4 = [SUB_1 , SUB_2 ; SUB_3 , SUB_4]

    1    1    2    2    2
    3    3    3    4    4
    3    3    3    4    4
    3    3    3    4    4
```

To again illustrate extraction we pull out rows 1 and 2 of columns 1 , 3, 5 of COMBINED_4. In order to save COMBINED_4 for later use we give a name to the extracted matrix:

```
>> EXTRACT_4 = COMBINED_4( [1 2] , [1 3 5] )

     1     2     2
     3     3     4
```

vi. Replacing a block of a matrix by a given matrix.

Suppose that we wanted to replace the upper right 2 x 3 block of COMBINED_4 by submatrix of 5's. If we do not care about saving "COMBINED_4" we could change it right away.

```
>> ADD = 5*ones(2,3);

>> COMBINED_4( [1 : 2] , [3 : 5] ) = ADD         % see iii above

COMBINED_4 =

     1     1     5     5     5
     3     3     5     5     5
     3     3     3     4     4
     3     3     3     4     4
```

However if we wanted to save COMBINED_4, we would first make a copy of it and then work with this copy:

```
>> COPY_1 = COMBINED_4;

>> COPY_1( [1 : 2] , [3 : 5]) =  ADD            % see iii above

     1     1     5     5     5
     3     3     5     5     5
     3     3     3     4     4
     3     3     3     4     4

>> NEW_1 = COPY_1;
```

vii. Building matrices recursively by assignment and by conatenation; the empty matrix: [ ].

Subsection iv shows how we can assign a column vector to a specific column of a matrix and subsection i shows how matrices can be built up from vectors by concatenation; i.e. by linking the vectors together. Both of these ideas can be used to build up matrices recursively.

Which of the two methods one uses is often a question of taste, but sometimes one or the other is more suited to the situation. What is particularly nice is that in neither of the two methods is there a need to predefine the dimensions of the final matrix.

To illustrate the techniques suppose that we want to construct a matrix A whose first row consists of the integers 1,...,5 and whose second row consists of the squares of these integers.

In the first method we simply define the first row and then use pointwise powers to find the squares. We then concatenate the two rows, placing the second underneath the first by means of a semi-colon ; as in subsection i. This technique is given as method 1 in the program concat.m .

In the second method we loop five times. At each stage n we construct the column vector c whose first element is n and whose second is n2. We then assign vector c to column n as in subsection iv.

In the third method we start with the empty matrix, which is denoted by [ ]. We then go into our loop, and, as in the second method, we construct the column vector c. This column vector is then concatenated to the matrix that we had at the previous stage. At the first stage we concatenate c with the empty matrix and so obtain c.

```
% [concat.m]

% method 1: define the first row, use pointwise powers.

a = [1: 5];     % the row vector [1,...,5]; section II.2.v
b = (a).^2;     % pointwise powers;  section II.1.vii
A = [a; b];     % concatenate;  subsection i

disp('final matrix ='), disp(A)

% method 2: assign a column vector to each column of A.

for n = 1:5            % loop; section 1.3.iii
    c = [n; n^2];      % ; puts n^2 underneath n
    A(:, n) = c;       %  column n; subsection  iv
end

disp('final matrix ='), disp(A)

% method 3. start with the empty matrix and concatenate.

A = [ ];               % start with the empty matrix
for n = 1:5
    c = [n; n^2];      % ; puts n^2 underneath n
    % concatenate c, using , to the previous matrix A
    A = [A, c];        % , puts c on the "side' of A
end

disp('final matrix ='), disp(A)
```

In all three cases we obtain:

```
final matrix =

     1     2     3     4     5
     1     4     9    16    25
```

A more involved example of the second method is given in Section III.3, example 2. It could of course have been done by the third method.

viii. Storing and "fetching" sequences of matrices.

One of the weaknesses of MATLAB is that it does not allow for automatic sequencing of matrices. If we have only a small fixed number of matrices, we can call them A_1, A_2, etc., but sometimes we want to generate matrices using a "for-loop" and it would be useful to have the matrices indexed. One way to get around this deficiency is to store the matrices generated as submatrices of a larger matrix. We then "fetch" the submatrices from the larger matrix as needed.

Example 2. The following program creates a 2 x 2 random matrix A and then stores the first three powers of A in a matrix called STORE. This is printed out here for the purposes of illustration, but for large systems this would not be done; submatrices would be pulled out as needed.

```
Program 1
% [store1.m]

A = rand(2 , 2)                 % section II.2.iv

for n = 1:3                     % for-loop; section I.3.iii
    B = A^n;
    index = [(2*n -1) : (2*n)]
    STORE(: , index) = B;        % extract several columns; subsect. iii
end
```

We now call up the program "store1.m"

```
>> store1                      % use only the name not store1.m
```

```
STORE =

    0.2190    0.6789    0.0799    0.6098    0.0462    0.4685
    0.0470    0.6793    0.0423    0.4934    0.0325    0.3638
```

To "fetch" the submatrices, we use the function "fetch(k, STORE)" which is defined below. This function of two variables (see section I.3.iv) retrieves the kth submatrix of a matrix STORE. It was written for square matrices stored horizontally in the matrix STORE, but it can easily be modified to fetch non-square matrices or matrices stored vertically.

The program includes a check to make sure that the number of rows of the storage matrix "STORE" is an integer multiple of the number of columns. This check uses:

1. a conditional if-end loop (section II.8.iii).
2. the inequality relation symbol ~= (section II.8.i).
3. the error function error(' message ') which returns control to the MATLAB command line and displays the enclosed message (section II.8.iv).
4. the double valued function size( ) to find the dimensions (section II.3.ii)
5. the function round( ) which rounds to the nearest integer.

Program 2

```
% [fetch.m]

function B = fetch(k, STORE)

[rows, cols] = size(STORE);     % sect. II.3.ii.

% Check that the matrix is o.k.        % sect. II.8,i,iii.
if round(cols/rows) ~= (cols/rows)     % round: sect. II.3.vi
    error('columns not multiple of rows or matrix is vertical' )
end

% The dimension of the submatrices is rows x rows; there are
% (cols/rows) of them as checked just above.
dim = rows;
num_mat = cols/dim;

% The submatrix that we want is in columns (k-1)*dim + 1 to
% k*dim of "STORE". For complicated ranges do in steps.
lo = (k-1)*dim + 1;
up = k*dim;
range = [lo : up];
```

II.4. Extracting and Building Matrices

```
B = STORE(: , range);   %  use `;' to avoid a repeat printout
```

Example 3. We "fetch" out the third of the submatrices of the storage matrix "STORE" which was calculated as A^3.

```
>> fetch(3,STORE)

    0.0462    0.4685
    0.0325    0.3638
```

A more involved example of the storage and fetching of matrices - so that linear combinations can be calculated - is given in section III.2 in connection with the spectral decomposition of a matrix.

ix. Building a diagonal matrix whose diagonal is a given vector.   Extracting a diagonal from a matrix. Off-diagonal matrices.

We construct a 1 x 4 vector "dia" all of whose elements equal 2.

```
>> dia = 2*ones(1,4)

     2     2     2     2
```

To form a diagonal matrix using "dia" we use the diag( ) command:

```
>> D = diag(dia)

     2     0     0     0
     0     2     0     0
     0     0     2     0
     0     0     0     2
```

The function diag( ) also works in the other way; we can obtain the vector "dia" back again:

```
>> diag(D)

     2     2     2     2
```

If we want to construct an off-diagonal matrix we can use a two-valued form of diag

```
>> off_dia = 2*ones(1,3)

     2     2     2
```

To place off_dia below the main diagonal we write:

```
>> D = diag(off_dia, -1)

    0     0     0     0
    2     0     0     0
    0     2     0     0
    0     0     2     0
```

To place the vector off_dia above the main diagonal we would use the command:

```
>> D = diag(off_dia, 1)
```

x. Building matrices with many zeros; predefining the dimensions.

If a matrix has many zeros one can often avoid, particularly if the non-zero entries are off-diagonals or rows, typing in all the values.

Example 1. We construct a Leslie matrix (see the population example in section III.2) which has 2's in the first row and 7's just below the main diagonal. First we define the corresponding vectors and construct a matrix of zeros which has the correct dimensions. The first vector is placed in the first row by assignment and the second is placed via a "for-loop" (section I.3.iii) which assigns one element of the vector at a time to the off-diagonal.

```
>> A = zeros(4,4);        % predimension: a 4 x 4 matrix of 0's
>> r1 = 2*ones(1,4);      % a 1 x 4 vector of 2's
>> d2 =7*ones(1,3);       % a 1 x 3 vector of 7's

>> A(1 , :) = r1;         % assign r1 to the first row of A

>> for n = 1:3            % for-loop; section I.3.iii
    A(n+1, n) = d2(n);   % element (n+1,n) of A = element n of d2
  end

>> disp(A)                % display A; see I.5.i

    2     2     2     2
    7     0     0     0
    0     7     0     0
    0     0     7     0
```

xi. Forming a vector from the elements of a matrix and inversely.

Sometimes we want to examine the elements of a matrix in vector form (see example 1 of section II.5). If we use the command M(:), then MATLAB pulls out the elements of M one at a time starting in the upper left corner and working down the first column and then does the same for the next columns. Since the vector formed would be a column vector, we use the transpose operator to obtain a row vector.

```
>> vec = M( : )'        % use '  for real matrices

3   7  -3   4    6   -2    5    5   -1    6    4   0
```

Now we want to take the twelve elements of "vec" and form them into a 4 x 3 matrix whose elements are obtained by filling the first column from "vec", then the second, etc.. First we give it the desired shape. Then we use M(:), but this time the vector is assigned to M(:) whereas above M(:) was assigned to the vector. Notice how the order, starting from the upper left and working down, of the elements is exactly the same as in the original matrix M, but that now the matrix is 4 x 3 instead of 3 x 4.

```
>> M3 = zeros(4,3);

>> M3( : ) = vec       % assign the vector to M

     3     6    -1
     7    -2     6
    -3     5     4
     4     5     0
```

[Note: If had simply wanted to reshape M we could have used the command:

```
>> M3 = reshape(M, 4, 3)  ]
```

II.5. Matrix column operations (summing etc.) AND SORTING

SUMMARY

i. Column functions.

MATLAB has various column functions including sum( ), mean( ), max( ), min( ), cumsum( ), prod( ), cumprod( ), median, std( ).

[Warning: Do not use "sum" etc. as variable names]

ii. Sorting

To sort all the columns of a matrix G in increasing order:

```
sort(G)
```

To sort a vector x and obtain the set of indices indicating the position of the entries before sorting:

```
[x_sort , indices] = sort(x).
```

The vector "indices" can be used in turn to change the position of elements of another vector or matrix.

iii. Determining the dimensions of a matrix or vector.

```
[rows , cols] = size(A)
```

If a is vector:

```
terms = length(a)
```

DETAILS AND EXAMPLES

i. Column functions.

We use the following matrix in the examples that follow:

```
G =

   -1    2    3
    1   -1   -1
    1    2    2
    0    0    0
```

MATLAB matrix functions are column functions rather than row functions. Thus sum(G) calculates the sums of the columns of G and places the sums in a row vector which lies underneath the columns:

II.5. Matrix column operations (summing etc.) AND SORTING

```
>> col_sum = sum(G)

     1     3     4
```

To obtain row sums we need to use transposes, and if we want the row sums in a column vector so as to correspond to the "side" of G, we have to take another transpose. Since G is real we can use ' the transpose for real matrices.

```
 >> row_sum = sum( G' ) '

     4
    -1
     5
     0
```

The average of the columns is obtained using mean( ).

```
>> h1 = mean(G)

    0.2500    0.7500    1.0000
```

h1 is a vector so we can find the mean of its elements:

```
>> a = mean( h1 )

   a = .667
```

We could have found a in one line:

```
>> a = mean( mean(G) )
```

or we could have first formed a vector from the elements of G (see II.4, xi):

```
a =  mean( G( :  ) )
```

We can also obtain the "median" or "middle" value of the columns of a matrix or of a vector by using median( )

```
x =
     1     2     3     4     5

>> median(x)

     3
```

II.5. Matrix column operations (summing etc.) AND SORTING

Since the columns of G have an even number of elements, MATLAB takes the average of the two middle elements. For example the elements of column 1 are -1 0 1 1 and so MATLAB takes the average of 0 and 1.

```
>> h2 = median(G)

   0.5000    1.0000    1.0000
```

For statistics, one often needs the standard deviation of a vector or matrix

```
>> std(x)

   1.5811

>> h3 = std(G)

   0.9574    1.5000    1.8257

>> max_col = max(G)

    1     2     3

>> min_col = min(G)

   -1    -1    -1
```

We can also obtain cumulative sums as we go down the column

```
>> cumulative_sum = cumsum(G)

   -1     2     3
    0     1     2
    1     3     4
    1     3     4
```

ii. Sorting.

To sort the columns of a matrix in increasing order, we use the function sort( ). The function can be used as a single valued function to simply sort or as a double valued function to find the original position of the elements

II.5. Matrix column operations (summing etc.) AND SORTING

```
>> sort(G)

   -1    -1    -1
    0     0     0
    1     2     2
    1     2     3
```

Example 1. We form a vector from a matrix using D(:) (see section II.4) and then order these elements for further analysis.

```
D =

    0     0     7     4
    2    -1    -3     0
    0     5     0    -1
```

To put all the elements of D, in the order (1,1), (1,2), (1,3), (2,1), ... in a row vector we use a single colon.

```
>> x = D( : )'

0    2    0    0   -1    5    7   -3    0    4    0  -1
```

We can now sort x in increasing order:

```
>> y = sort(x)

 -3   -1   -1   0    0    0    0    0    2   4   5  7
```

Not only can we sort the columns, but we can also find the position of the elements in the original columns. Thus from the first row of INDICES below we can tell that the three terms -1, -1, -1 were respectively in the rows 1, 2 and 2.

To find INDICES we use sort( ) as a double valued function (see section I.4; and the other common double valued function eig( )).

```
>> [NEW_MAT , INDICES] = sort(G)

NEW_MAT =

   -1    -1    -1
    0     0     0
    1     2     2
    1     2     3
```

```
 INDICES =

     1     2     2
     4     4     4
     2     1     3
     3     3     1
```

Sorting indices are also used to change the order of a vector (or matrix) according to the order of the sorted matrix.

Example 2. Suppose that we want to rearrange the rows of G so that the corresponding row sums are in increasing order. Above we found the row sums as a column vector; this is repeated here for reference:

```
>> row_sum = sum(G')'  % original row sums

     4
    -1
     5
     0
```

We now order the row sums and find the corresponding index set.

```
>> [row_sum_2 , indices] = sort(row_sum)

row_sum_2 =               % ordered row sums

    -1
     0
     4
     5

indices =
     2
     4
     1
     3
```

Now we rearrange the rows of G using the index set "indices" (see Section II.4)

```
>> G_2 = G(indices , :)      % rows contained in indices

     1    -1    -1
     0     0     0
    -1     2     3
     1     2     2
```

II.5. Matrix column operations (summing etc.) AND SORTING

We check that G_2 does indeed have its row sums in increasing order:

```
>> row_sum_3 = sum(G_2')'

    -1
     0
     4
     5
```

iii. Determining the dimensions of a matrix and the length of a vector.

```
>> [rows , cols] = size(G)

    rows = 4
    cols = 3

>>  terms = length(indices)

    terms = 4
```

SECTI0N II.6. GRAPHING VECTORS AND MATRICES

SUMMARY

First time users of the plot( ) function should read the introduction to graphing at the beginning of the EXAMPLES section.

Simple plots can be done from the MATLAB command line, but it is generally advisable to write a program (see section I.3.i). This allows for changes and corrections.

Because plotting involves various options the summaries here are more detailed than in other sections. The Details and Examples section illustrates commands via examples and are not presented in the same order or at the same number as in this summary.

i. The basic plotting command.

plot(domain_vector , value_vector)

Both domain_vector and value_vector are row vectors of the same length. How the values in the domain and value vectors are obtained is of no importance e.g.:

```
     domain_vector              value_vector
     x1 = [-1  6]              y1 = [.1   pi]
     x2 = [2 : 01 : 4]          y2 = sin(x2)
     x3 = [0 : 4]              y3 = rand(1 , 5)
     x4 = [-2]                y4 = [7]
```

The ordered pairs (x , y) - where x and y are the corresponding entries of the domain and value vectors - will be connected by a solid line.

To determine the length of your vectors use length( ).

[Note: there is no requirement that the values in the domain vector be increasing in value, although only examples where this is the case will be considered here. If the two vectors are such that the first and last values are the same, then the plot will be a closed circuit.]

[Warning: Special care is need if your index set starts with 0; see Section II.4, viii.]

SECTI0N II.6. GRAPHING VECTORS AND MATRICES

ii. To plot several pairs of domain and value vectors.

Several pairs, possibly of different lengths, can be plotted using a single plot( ) command e.g.:

```
plot( x1 , y1  ,  x2 , y2  ,  x3 , y3 )
```

[See hold in ix.]

iii. To plot the x and y axes.

Suppose that we want to plot the x-axis between a and b. This is just a special case of plotting, with the domain vector being [a b] and the range vector being [0 0]:

```
plot( [a  b] , [0  0] )
```

To plot the y-axis between c and d, we take [0 0] as the domain vector and [c d] as the range vector:

```
plot( [0 0] , [c  d] )
```

These axes plots can be added into other plots as in ii; see hold, below in ix.

The suitable values of a, b, c, d are often obtained from a few preliminary plots.

iv. To plot without connecting.

To plot the corresponding points in domain_vector and value_vector and to mark then with an asterisk:

```
plot( domain_vector , value_vector,  '*' )
```

Note how the ***** is enclosed in commas

Instead of using *****, one can mark the points with **+** **o** **.** (dot) **x** **s** (square) **d** (diamond) **p** (pentagon) **h** (hexagram)

Several pairs of domain and value vectors - with each pair having different lengths - can be plotted using different markers as in ii. Some of the pairs can be connected and others non-connected.

SECTI0N II.6. GRAPHING VECTORS AND MATRICES

v. To plot the rows of a matrix M.

If M has dimensions rows x cols then to plot the rows of M as lines on a graph, with the x-values being the column indices:

a. indices = [1 : cols]   % the domain vector
b. plot(indices , M)
c. The line types, or point formats, are defined as in iii and iv; the same format applies to all the rows.

To use the first column of the matrix for "initial conditions" and to have the indices start at 0; see the function shift_1(indices , M) in example 5.

[Note: Matrices used in plotting are often built-up matrices; see the examples below and in section III.6. In such cases, it is very good practice to predefine the size of matrices, e.g. M = zeros(rows , cols); and then fill in the rows or columns; see section II.4. Do not forget the semicolon or MATLAB will print out huge matrices.]

vi. To connect the points by other types of lines.

```
dashed line:   plot( x1 , y1 , '--' )
dash-dot line:   plot( x2 , y2 , '-.' )
dotted line:  plot( x1 , y1 , '..' )
solid line:  plot( x3 , y4 , '-' )
```

As in ii, different line type commands can be combined inside one plot( ) command.

vii. To fix the ranges of the axes of the graph.

MATLAB has automatic range selection. To fix the ranges yourself, with x going from x1 to x2 and y going from y1 to y2:

axis(**[** x1 x2 y1 y2 **]**)

Note there is only one set of brackets **[ ]** and that there are no commas.

If this command is placed first, it should be followed by hold (ix). The correct selection often requires a few preliminary plots to see what would be a suitable range of values.

Automatic range selection is re-obtained by: axis.

viii. To have the same scale on both axes.

```
axis('equal')
```

Automatic range selection is re-obtained by: axis

[Note: for some versions of MATLAB, the correct command is axis('square'); type help axis in case of problems.

ix. To hold a graph for the later superimposing of other graphs.

```
hold
```

The command hold is placed only once in a program, after the first plot command, rather than after the axis command.

Note that since the first plot has been held, parts of later plots will be lost if their domain and value ranges are not contained in the original ranges. Thus, care must be taken when using this.

To turn off the hold type: hold off or hold. The next plot( ) command will plot the graph on a clean screen.

x. To clear the graphics screen before a new plot.

```
clg
```

If you are writing a graphics program, place the command clg at the beginning. This will prevent previous plots from being held over.

If you have quit the graphics screen, without clearing it via clg, you should be able to show the graph again with the command shg.

xi. To put a title at the top of a graph.

```
title(' plot of averages ')
```

xii. To label the x and y axes.

```
xlabel(' real parts ')
ylabel(' imaginary parts ')
```

xiii. To print out text on the graph.

To print out a message at coordinates (.3, r3), where r3 is a scalar:

text(.3, r3, 'dominant eigenvalue')

Since the range of the plot and/or the free spaces on a graph are usually not known ahead of time, the placing of text usually requires a few trials.

xiv. To print out a variable on the graph.

To print out the value of element 1 of the vector eigenvalues at coordinates (.3 , r2), where the quantity r2 is a scalar:

```
text(.3, r2,  num2str( eigenvalues(1) ) )
```

xv. Printing a hard copy of a graph.

The programs and examples under details show how to produce a graph on the screen or in memory. The following algorithm is typical of how one obtains a hard copy from UNIX systems. Bold face indicates MATLAB commands and italics indicate system commands. The extension *.gra is not needed, but it makes finding your graph files easier! To find the correct commands for personal computers and other systems type: help plot

a. plot and add titles as above.

b. >> print -dps hart8.gra
[creates a postscript file "hart8.gra"]

c. Leave MATLAB (quit or !).

d. lpr -Php5230-ps hart8.gra
[sends the postscript file to a Postscript printer called "hp5230"; here lpr is the "line printer command on the UNIX system].

DETAILS AND EXAMPLES

An Introduction to plotting

Once the idea behind the basic MATLAB command:

```
plot( domain_vector , value_vector )
```

is understood, the plotting of simple connected graphs becomes very easy . What is sometimes confusing to users of MATLAB is the plotting of non-connected points and also the plotting of matrices. Because of this, I present the commands for lines, points and matrices in the same format. This will avoid problems such as incompatibility and the plotting of columns instead of vectors.

Adding titles and labelling axes (xi, xii) is also straightforward. But experience has shown that new users become confused and frustrated when they try to put too much in their first graphs.

It is thus suggested that the reader do a few simple graphs involving one pair of vectors, then some involving a few pairs. After this, they should do a few non-connected plots and then plot the rows of some matrices.

Then do some graphs with titles, and only when you are comfortable should you attempt to put in axes (iii), use fancy line formats (iv), fix the axes (vii), make the scaling equal (viii), hold plots (ix) or print text in the middle of a graph (xiii, xiv). As always it is much less frustrating when your instructions have been put in an *.m file (section I.3) where they can be easily corrected. If your system has text and graphics windows then you can see the changes made to the graph as you add commands. And to make yourself feel good and impress others, print out some graphs as early as possible (xv).

i. Plotting a value_vector against a domain_vector

The most important step is to make sure that both your domain_vector and value_vector are row vectors of the same length. How the values in the domain and value vectors are obtained is of no importance.

Example 1. The basic plotting command. It is suggested that the reader do the individual plots indicated. Figure 1 shows the result of doing the plot command for three functions at once as will be discussed at the end of the example.

We start off with the domain vector

```
>> x1 = [0 : .25 : 1]
```

and then we define the value vector in terms of x1

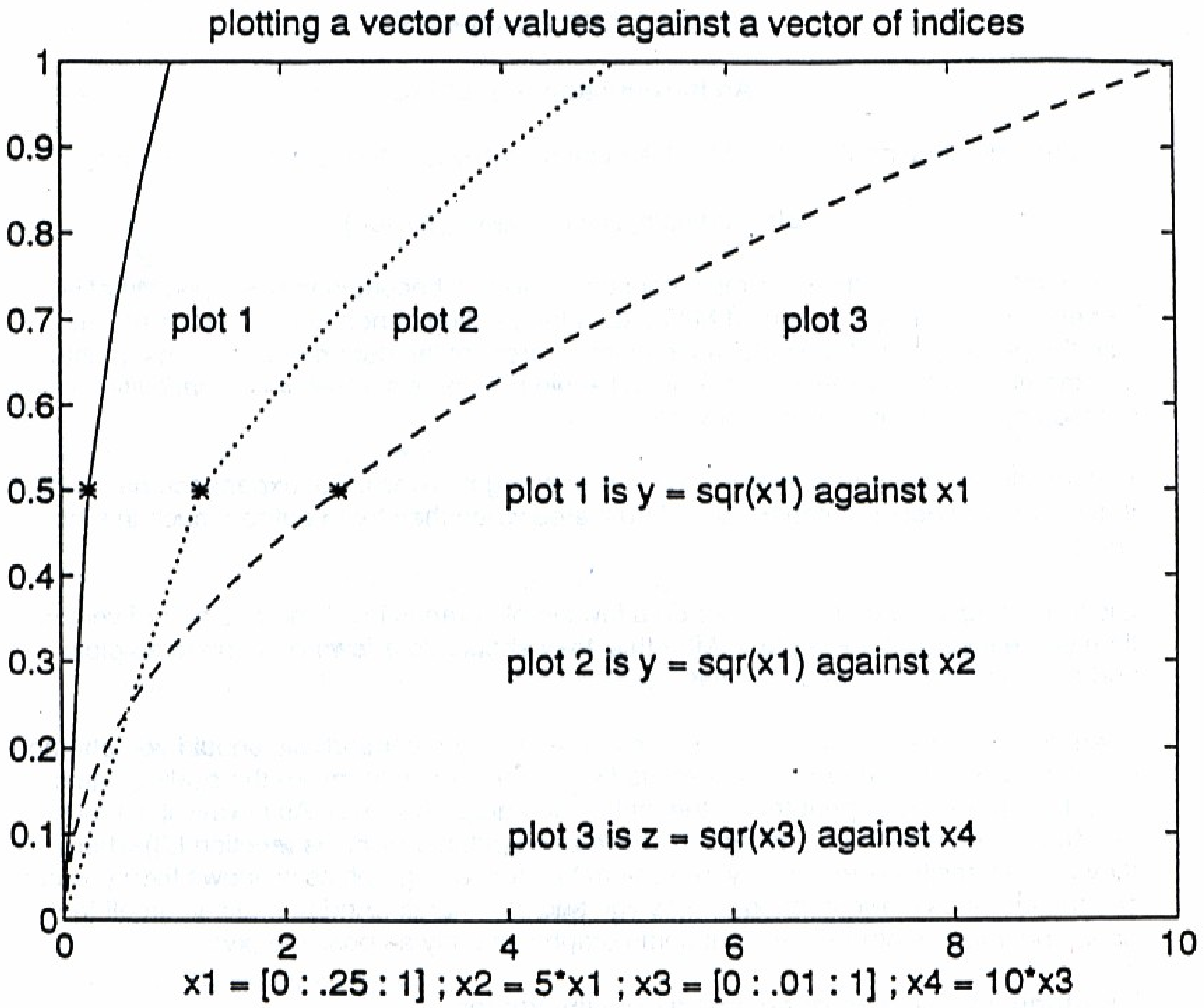
```
>> y1 = sqrt(x1)
```

Both of these vectors are row vectors of length 5 and so we can plot y1 against x1. This is done via the **plot(,)** command:

```
>> plot( x1 , y1 )
```

Note that the domain vector is stated first and then the value vector. MATLAB will take corresponding order pairs, plot them and then connect them with a solid line.

Thus since x1(2) = .25 and y1(2) = 0.50, the point (.25 , .50) will be on the graph. This point is indicated by a star on the first curve in the figure on the next page.



If we had just plotted y1 against x1, then the MATLAB plot would have occupied the entire width of the graph with the x values going from 0 to 1. However, since figure 1 shows three curves, the portion allotted by MATLAB to the first curve is limited to the segment [0 , 1].

Now we are going to do another plot which involves the same value vector y1 as in the previous plot, but we choose a different domain vector.

```
>> x2 = 5*x1                  % x2 also has 5 elements
```

Since x2 also has 5 elements, we can plot the values of y1 against x2. In other words, how we obtained y1 is of no consequence, we can plot it against any domain vector of 5 elements:

```
>> plot( x2 , y1 )
```

In this case since x2(2) = 1.25 and y1(2) = .50. Thus the point (1.25 , .50), again marked by a star, appears on the second graph. What we have done in effect is stretch out the first graph from the interval [0 , 1] to the interval [0 , 5]. The starred points correspond to one another.

There is one other thing that should be noticed, namely that the points are connected by straight line segments. In the plot of y1 against x1 this is barely noticeable, but it is very evident in the spread-out graph of y1 against x2.

If we want to plot a curve in which the straight line segments are not noticeable, we have to make sure that the elements in domain vector are close together. Thus, conceptually plotting a curve is just a special case of plotting one vector against another. To illustrate this we start off with:

```
>> x3 = [0: .01 : 1];          % suppress printing with ;
```

Note that x3 has 101 elements instead of 5 as was the case with x1 and x2. We again use the sqrt( ) function to obtain a vector z of values:

```
>> z = sqrt(x3);              % suppress printing with ;
```

Now x3(26) = 0.2500 and z(26) = 0.5000 so that if we plotted z against x3 the point (.25 , .50) would appear on this curve just as it did on the plot of y against x1.

However, instead of plotting z against x3, we are going to once again shift the curve over, this time to [0 , 10]

```
>> x4 = 10*x3;                % also 101 elements
>> plot( x4 , z )
```

Since x4(26) = 2.5 and z(26) = 0.50, the starred point on this plot corresponds to the starred point on the other two curves.

Up to now we have used the plot( ) command for one plot at a time. We can however plot all three curves at once:

```
>> plot( x1 , y  ,  x2 , y  ,  x4 , z )
```

The order in which we enter the plots is of no importance, MATLAB uses the largest spread of values in doing the plot. Notice that in each individual pairs the length is the same, but that the length changes from pair to pair.

Example 2. Plotting without connecting.
In the figure, there are three individual points (.25 , .50), (1.25 , .50) and (2.5 , .50) which correspond to one another. To plot these pairs without connecting them, we start off exactly as with a connected plot, i.e. we put the x-values in a domain row vector and the y-values in a value row vector. However to indicate that we want to have the isolated points plotted using a star, we add ' *' to the plot command.

Instead of using the numbers themselves, we assign them to variable names, both to save space and for later use.

```
>> a = x1(2),   m = y(2);              % (.25 , .50)
>> b = x2(2),   n = y(2);              % (1.25 , .50)
>> c = x4(26),  p = z(26);             % (2.5 , .50)
```

Now we have two choices. We can either display the variables explicitly in the plot command:

```
>> plot( [a b c] , [m n p ] , '*' )
```

or we can, as we did in example 1, place the values in domain and value vectors:

```
>> u = [a b c];                  % domain vector
>> v = [m n p ];                 % value vector

>> plot(u, v, '*')
```

The result of the plot command is the set of three points.

We could combine this plot with the plot of the three curves by putting all vectors in one command:

SECTI0N II.6. GRAPHING VECTORS AND MATRICES

```
>> plot( x1 , y  ,  x2 , y  ,  x4 , z , u , v, '*')
```

Instead of * one can mark the points with + , .'   o , x .

The following program was used to plot the curves. Roman numerals refer to the summary.

Program 1

```
% [plo_sq1.m]

axis( [ 0 10 0 1 ] )            %  (vii)
hold                            %  (ix)

x1 = [0 : .25 : 1];

% same curve shifted over to [0 , 5]
x2 = 5*x1;

% on [0 , 1] but with finer division points, do not plot
x3 = [0: .01 : 1];
x4 = 10*x3;

plot( x1 , y)                   %  (i)
plot( x2 , y, ':' )             %  (vi)
plot( x4 , z, '--' )            %  (vi)

pause                           % section I.3.ii

% a, b, c are corresponding points of x1, x2 , x3
a = x1(2),   m = y(2);
b = x2(2),   n = y(2);
c = x4(26),  p = z(26);         % m = n  = p

u = [a b c];                    % index vector
v = [m n p ];                    % value vector

% plot the three corresponding points, do not join, use a *
plot( u , v , '*')              %  (iv)

pause

% label the points
text( 1 , .7 ,  ' plot 1 ')     %  (xiii)
text( 3 , .7 ,  ' plot 2 ')
text( 6.5 , .7 , ' plot 3 ')
text(4, .5,  'plot 1 is y = sqr(x1) against x1')
text(4, .3,  'plot 2 is y = sqr(x1) against x2')
text(4, .1,  'plot 3 is z = sqr(x3) against x4')

% title  (xi)
title('plotting a vector of values against a vector of indices')

% xlabel, long lines; (xii) , sect. I.4.ii
xlabel(' x1 = [0 : .25 : 1] ; x2 = 5*x1 ; x3 = ...
[0 : .01 : 1]  ; x4 = 10*x3 ')
```

Example 3. Plotting the rows of a matrix M. When working with matrix models, we often wish to plot the vectors obtained from the rows of the matrix. For example, each row might represent the evolution of a "state" of the system with the column value representing the "time". Examples of this type are given in section III.3. Sometimes, as in example 5 below, we have vectors of the same length which can be put together to form a matrix M. Instead of writing the plot command for the individual vectors, we can simply plot the matrix M against the indices. Another advantage is that we can easily pull out selected columns from the matrix, as is done in example 5, and then plot the rows of this submatrix.

Suppose that the number of columns in our matrix M is cols. Then to plot the rows of M as lines on a graph, with the x-values being the column indices, we first define the domain vector which is precisely the vector of column indices:

```
>> indices = [1 : cols]
```

Then we type the plot command which has exactly the same format as for vector:

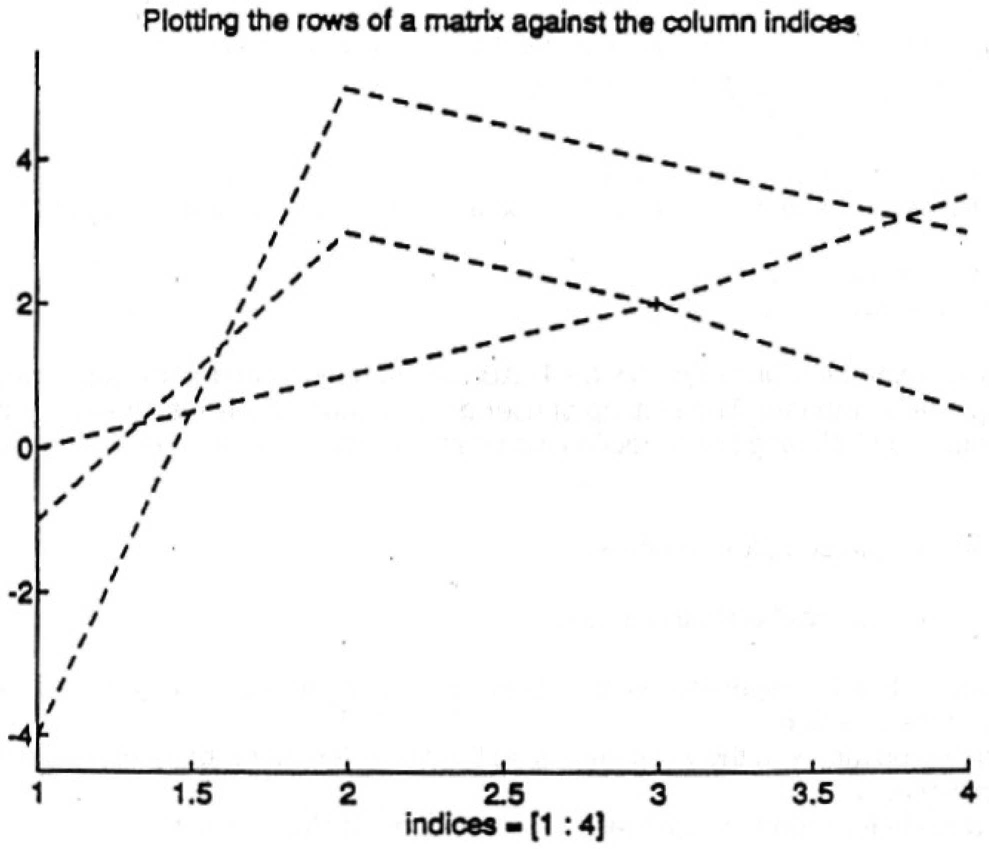
```
>> plot( indices , M)
```

The matrix plot and labelling was done via Program 2. Notice how the first row vector, [-1 3 2 .5], of the matrix defined in the program, is plotted as one line. The Roman numerals in parentheses refer to the numbers in the summary above. If you are reading about plotting for the first time, you need only consider the variable indices and the command plot.

Program 2

```
% [plot_mat.m]

% fix the range of the axes so as to leave some space at
% the top (vii)
axis([ 1 4 -4.55 5.5])

hold                    % hold the axes for later plotting (ix)
```

SECTI0N II.6. GRAPHING VECTORS AND MATRICES



```
% define the matrix
M = [-1 3 2 .5 ; 0 1 2  3.5 ; -4 5 4 3]

indices = [1 : 4]

% plot M against indices using a dashed line (vi).

plot( indices , M , '--')

% plot the vector [2] against the vector [3], i.e. plot the
% point (3,2), using a plus sign (iv).
plot( [3 ] , [2] , '+')

% put a title on the graph (xi).
title(' Plotting the rows of a matrix against the column indices ')
```

```
% label the x-axis (xii)
xlabel(' indices = [1 : 4] ')
```

The following examples illustrate how MATLAB can be used to do matrix calculations and then plot the outcome. The plotting of user defined functions is discussed in section II.7.vi. Examples of plotting in connection with matrix power models are given in section III.3.

Example 4. Plotting complex numbers.

The function "comp_drw" does the following:

1. Finds and orders the eigenvalues of a square matrix; it returns the eigenvalues as the value of the function.
2. Plots the eigenvalues in the x-y plane using the same length on both the real and imaginary axis.
3. Draws a circle through the eigenvalue of maximum absolute values.
4. Has an option for drawing lines from the origin to the eigenvalues.
5. Has an option for printing out the value of the maximum eigenvalue.

Numbers in parentheses refer to the summary. The various operations on complex numbers are discussed in section II.3.v.

Program 3

```
% [comp_drw.m]

function  eigenvalues = comp_drw(A)

clg                              % clear previous graphs (x)

eigenvalues = eig(A);

eigenvalues = (eigenvalues).'        % true transpose; sect. III.1

% sort eigenvalues by decreasing magnitude.
eigenvalues = sort(eigenvalues);          % sorts by increasing order
eigenvalues = rot90(rot90(eigenvalues));    % by decreasing order

% calculate and display absolute values
absolute = abs(eigenvalues);
disp('vector of absolute values'), disp(absolute)

% calculate and display arguments in radians and degrees
angl_rad = angle(eigenvalues);
angl_deg = (180/pi)*angl_rad;
```

SECTI0N II.6. GRAPHING VECTORS AND MATRICES

```
pause                              % push RETURN; section I.3.ii

% plot complex numbers
re = real(eigenvalues);
im =imag(eigenvalues);

axis('equal')                  % same scale (viii)
hold                           % hold plot (ix)

r = absolute(1);               % maximum absolute value
r1 = (1.15)*r;                 % leave some space outside of points

axis( [-r1 r1 -r1 r1 ])        % set range of MATLAB axes (vii)


plot( re, im, '*')  % put "*" at each point, do not connect (iv)

% plot x and y axes (iii)

plot( [-r1 r1], [0 0] ) , plot( [0 0], [-r1 r1])

% optional connect eigenvalues to origin (sections I.4.3, II.8.iii)
lines = input(' connect origin to origin? y or n:  ', 's');

if lines == 'y'
   for n = 1 : cols
      plot( [0, re(n)] , [0, im(n)] )     % connect (0,0) to (re,in)
   end
end

% plot circle by defining a domain vector corresponding to   x-values
% between - r and + r and against this plotting two value vectors
% "circle_up" and "circle_lo" which correspond to points on the circle.

x =  [-r : .005: r];
circle_up =  ( r.^2 - x.^2).^(.5);     % upper half of circle
circle_lo = -( r.^2 - x.^2).^(.5);     % lower half of circle

plot(x, circle_up , x, circle_lo)


% optional printing of dominant eigenvalue (sections I.4.3,
% II.8.iii, summary xiv)
dom = input(' print dominant eigenvalue? y or n:  ', 's');

if dom == 'y'
```

```
    r2 = 1.05*r;
    r3 = 1.1*r;
    text( .3, r3, 'dominant eigenvalue' )
    text( .3, r2,  num2str( eigenvalues(1) ) )  % maximum eigenvalue
end

% label graph (xii)
xlabel(' real parts ')
ylabel(' imaginary parts ')

pause                  % keeps on screen until RETURN is pressed.
disp( 'to put a title type: title(`  ` )  ')  % use ` for ' inside disp

hold off                        % (ix)
```

To illustrate the use of the function "comp_drw" with the first option that connects the eigenvalues to the origin, we find the fifth roots of i. This is a good test example because the eigenvalues are not symmetric about the x-axis.

The first step is to solve the equation z5 = i. We do this by rewriting the equation in the form z5 - i = 0 and then putting the coefficients, with 0 for missing terms, in a vector:

```
>> poly = [1 0 0 0 0 -i]
```

Next we find the roots of this polynomial using roots( ) (section II.7.ii).

```
>> vec = roots(poly)

vec =
  -0.9511 + 0.3090i
  -0.5878 - 0.8090i
   0.5878 - 0.8090i
   0.9511 + 0.3090i
        0 + 1.0000i
```

To use "comp_drw" we need a square matrix whose eigenvalues are the elements of "vec". We use diag( ) which produces a diagonal matrix whose diagonal elements are the elements of "vec".

```
>> B = diag(vec);              % section II.3.ii.
```

We are now ready to use "comp_drw" to find and plot the fifth roots of -i:

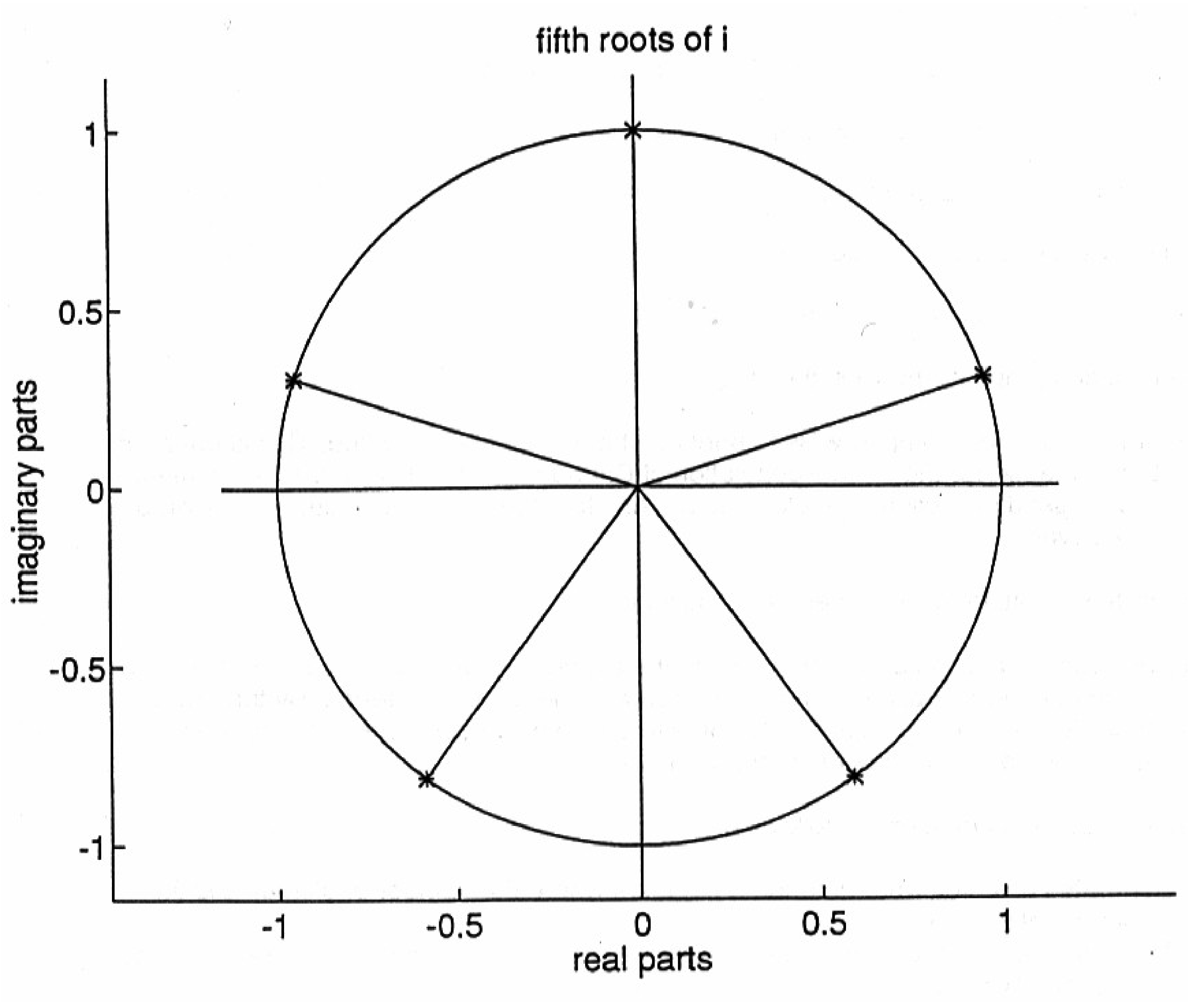
```
>> comp_drw(B)

eigenvalues in decreasing order
```

```
{ 0.5878 - 0.8090i}  {0.9511 + 0.3090i}  {0 + 1.0000i}

{-0.5878 - 0.8090i}  {-0.9511 + 0.3090i}

vector of absolute values

    1.0000    1.0000    1.0000    1.0000    1.0000

vector of arguments in radians

   -0.9425    0.3142    1.5708   -2.1991    2.8274

vector of arguments in degrees

  -54.0000   18.0000   90.0000 -126.0000  162.0000
```

The resulting graph is shown in the diagram on the next page

The other option for "comp_drw" is to print out the dominant eigenvalue. This is done for the "*Leslie matrix*" for the female population of Canada in 1981. This is a 10 x 10 matrix and is discussed in section III.3, example 3. The plot of the ten eigenvalues is shown on the back cover.

Whereas the eigenvalues in the fifth roots of -1 example are not symmetric about the x-axis, that is the case for the Leslie matrix. This is an example of the *Perron-Froebenius* theorem which says that for matrices which are not “cyclic” and where there is “communication” between different portions of a matrix of non-negative elements, the eigenvalues will indeed be symmetric about the x-axis. Furthermore the dominant eigenvalue will be real.



Example 5. Plotting moving averages; "closeups"

This program investigates what happens to the average value of the sum of a sequence of random numbers (section II.2.iv) as we increase the number of terms. By the "law of averages" (or "law of large numbers"), the averages should converge, if the random number generator is a good one, to the value .5.

In particular rand4.m does the following:

1. Generates a vector "vec" of random numbers where the number of elements is an input quantity.
2. Takes moving averages of the sum of the first n elements and stores these averages in a vector "vec_aver" .
3. Computes the fraction of time that the averages were above y = .5 (the so-called "arcsin law"). This part uses conditional vectors as discussed in section II.8.vii.
4. Plots the averages against the value of the indices starting with the index 0; using the function shift_1 of program 5.
5. Plots the lines y = .5, y = 0 and y = 1 by working with the functions ones( ) and zeros( ).
6. Does a blowup of the last segment of the graph; see the inside from cover. The number of elements is an input item. The range of the blowup is restricted to the interval [.4 , .6]. This part illustrates how we can extract part of an already plotted matrix and do a new plot. We could continue to blowup sections of the orginal, eventually zooming in on one element!

Program 4

```
% [rand4.m]
terms = input(' how many random numbers?  ');

vec = rand(1 , terms);              % generate random vector
vec_aver = zeros(size( vec ));      % dimensions the same as `vec'

total = 0;
for n = 1:terms                     % loop; section I.3.iii
       total = total + vec(n);      % keep adding a term
       vec_aver(n) = total/n;       % store averages in a  vector
end

% compute percentage of time that was spent above line y = 1/2
A = [vec_aver > .5];               % conditional vector, II.8.vii
fraction_above = (sum(A))/terms;

% plot
terms_2 = terms + 1;           % number ofcolumns of storage matrix
indices = [0 : terms];         % indices go 0 ... terms
expected_av = .5*ones(1 , terms_2);    % line y = .5
line_1 = ones(1, terms_2);             % line y = 1
line_0 = zeros(1, terms_2);            % line y = 0

% store  averages, y = .5 and y = 1 as rows of matrix M
M = zeros(4, terms_2);                 % dimensions of M
```

```
M(1, 1) = .5;   % start curve at .5
M(1, [2 : terms_2] ) = vec_aver;       %store vec_aver
M(2, :) = line_1;
M(3, :) = expected_av;
M(4, :) = line_0;

plot(indices, shift_1(indices, M))      % program 5

title('average of first n terms vs. n ')

% insert texts and values
a = terms/2;
aver = vec_aver(terms)
text( a , .9 , 'last average is: ')
text( a , .82 , num2str(aver ) )

text( a , .7 , ' fraction of time averages above .5  is: ')
text( a , .62, num2str( fraction_above ) )

disp(' graph is printed to "rand_av.gra" ')
print -dps rand_av.gra            % (xv)

% closeup
closeup = input('where should the closeup start?  ');

% truncate "vec_aver' and "expected_av" (i.e. y = .5)
% store in a new matrix M1
indices2 = [closeup : terms];

vec_trunc = vec_aver(indices2 );          % last part of vec_aver
exp_trunc = expected_av( indices2 );       % last part of expected_aver
M1(1 , :)  = vec_trunc;
M1(2 , :)  = exp_trunc;

% fix vertical range between .4 and .6
axis([ closeup   terms_2   .4  .6 ])
hold

% plot the 2-row matrix M2 against indices 2

plot( indices2 , M1 )

title(' closeup of the plot of averages ')
print -dps close_up.gra                    % (xv)
```

To run "rand4.m" we need the following program which allows us to start the

SECTI0N II.6. GRAPHING VECTORS AND MATRICES

index vector at 0 instead of 1

Program 5

```
% [shift_1.m]

% for an integer k and a matrix M the function pulls out
% column k+1 of the matrix

function v = shift_1(k , M)

v = M(: , k+1);
```

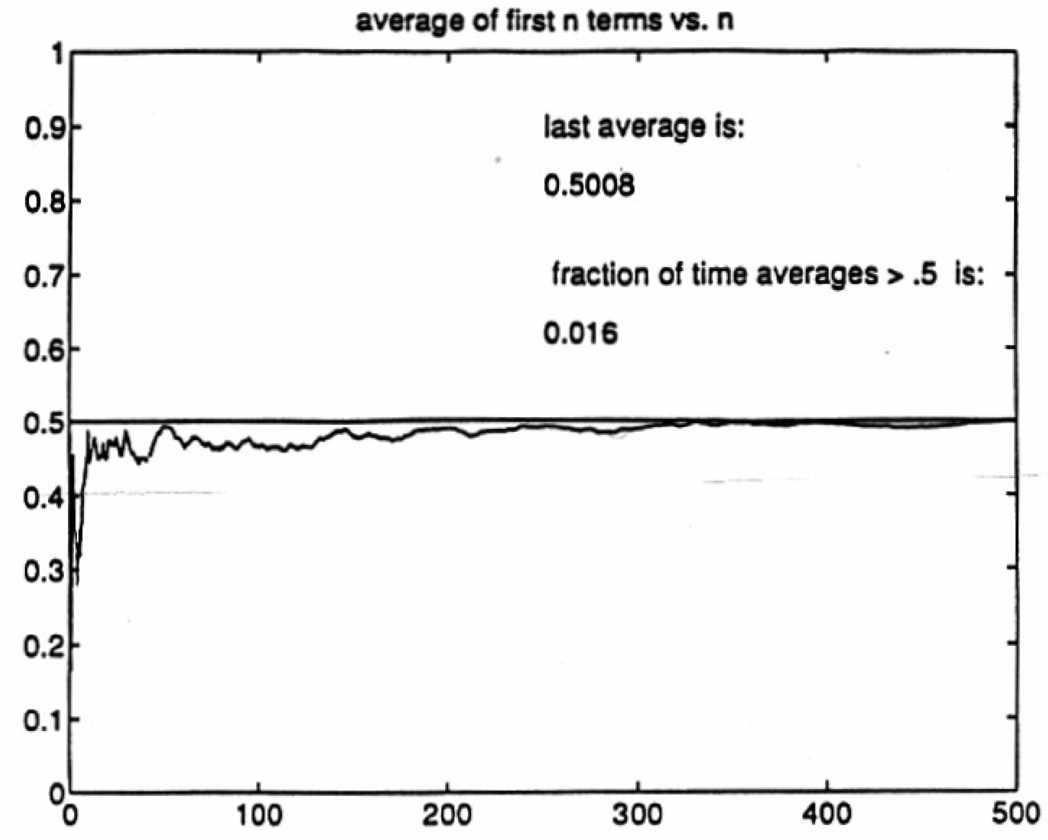
We now run the program is as follows:

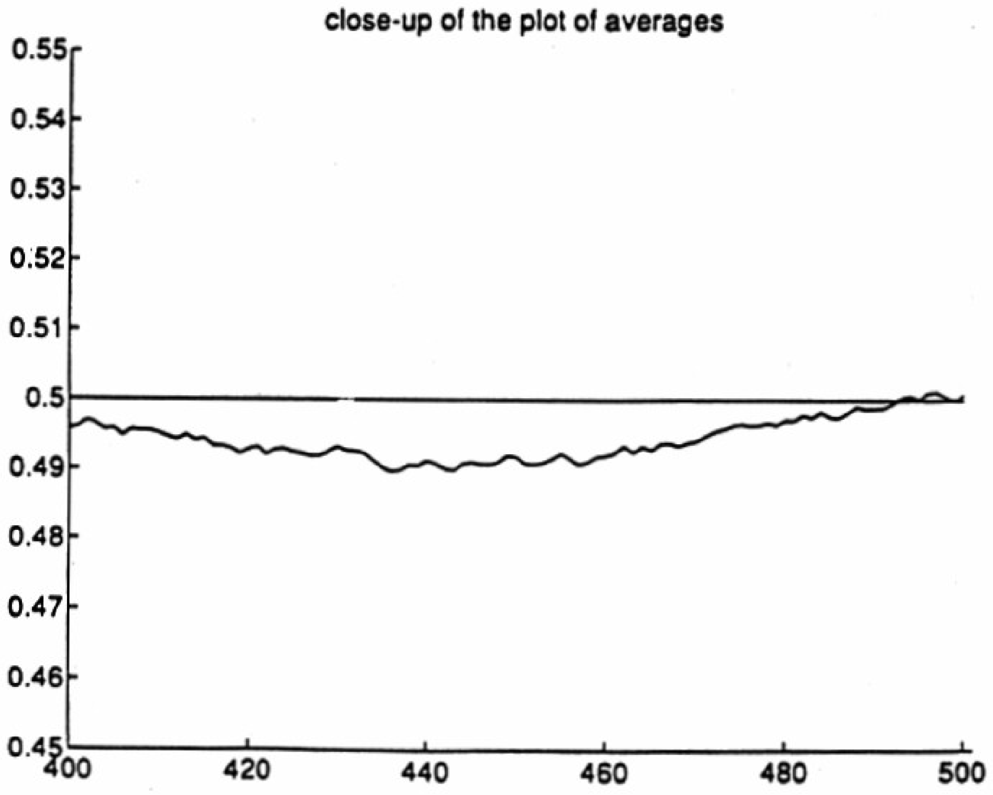
```
>> rand4

how many random numbers?  500

number of random numbers:
   500

the last average is:   0.5008

fraction of time averages above .5  :  0.0160
```

[Thus the averages are greater than .5 less than two percent of the time. Intuitively this may seem incorrect, but in fact it is predicted by the "arc-sine law".]

```
where should the closeup start?  400
```

The main graph and the blowup are shown on the next page and on the inside cover. The latter indicates that the averages stayed between .49 and .50 for the last 100 sums, except for a crossover just at the end.


average of first n terms vs. n
last average is:
0.5008
fraction of time averages > .5 is:
0.016
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
100
200
300
400
500


close-up of the plot of averages
0.45
0.46
0.47
0.48
0.49
0.5
0.51
0.52
0.53
0.54
0.55
400
420
440
460
480
500

SECTION II.7 NUMERICAL ANALYSIS

SUMMARY

i. To define a vector function "hart8(x)".

Create an m-file "hart8.m" which defines the function. Use pointwise MATLAB operators, (II.1).

To simply calculate the values of a function on a one-time basis just define the relationship in the main program.

For a polynomial defined by a vector of coefficients poly use the function coeff(poly, x).

ii. To find the roots of a polynomial

Enter the coefficients in decreasing order in a vector:

```
poly = [1 - 3 -9  5]
rac = roots( poly )
```

iii. To find the roots of a defined function "hart8(x)".

Search for roots near various points, e.g. near 2.

```
root1 = fzero('hart8', 2)
```

iv. To find the minimum (maximum) of "hart8" on [a,b].

```
x = [a : .001 : b]
y = hart8(x);
m1 = min(y)
m2 = max(y)
```

v. To find the integral of "hart8" between a and b.

```
hart8_int = quad8('hart8' , a , b )
```

vi. To plot the function "hart8" on an interval [a b].

```
x = [a : .001 : b];
plot( x , hart8(x) )
```

For axes, titles, printing etc. use the commands of section II.6.

vii. To find a least squares polynomial fit of of order 3.

```
x_values = [2.1  4.3   6.1    9.4]
y_values = [6.7  1.2  -0.6   -3.1]
```

SECTION II.7 NUMERICAL ANALYSIS

```
poly3 = polyfit(x_values , y_values , 3)
```

To plot the polynomial over x = [a : .001 : b];

```
y = coeff(poly3,x)
plot(x,y)
```

viii.   To estimate values / Interpolation

```
x_values = [2.1  4.3   6.1    9.4]
y_values = [6.7  1.2  -0.6   -3.1]

x = 5.1  or x = [5.1 5.4 .5.7]
y = interp(x_values , y_values, x, 'spline')
y = interp(x_values , y_values, x, 'linear')
--
```

DETAILS AND EXAMPLES

i. Defining the function.

Functions (section I.3.iv) are defined via a program which starts with the word function. The function is then defined by using the MATLAB pointwise functions (section II.1) .* , .^ , and ./ . A very good habit is to use lots of parentheses. One should always test functions with simple values; for complicated functions check steps by step. The reason for this is that even if you have made a mistake, what you have written may be valid MATLAB statements, but not the correct ones! So when you use your function you will have answers, but not the ones you wanted! Put a semicolon at the end of each line, including the last line which defines the value of the function; otherwise the final answer will be printed out twice.

The following examples were chosen from my father's calculus book; W. Hart, *Analytic Geometry and Calculus,* 1957.

Example 1. This involves a simple polynomial. Note that to write 9x, we need only write 9*x, because 9 is a scalar. *For powers we must use the pointwise power operator*, so that for x3 we write (x).^3. Remember that MATLAB functions are vector functions and that we are operating pointwise on them.

Program 1a

```
% [hart5.m]      Hart, p. 122

% f(x) = x^3 - 3x^2 - 9x + 5

function  y = hart5(x)          % indicate function

y = (x).^3 -3*(x).^2 - 9*x + 5;   % pointwise powers: .^
```

% suppress printing with ;

Since "hart5(x)" is a vector function, it can be used to calculate values of the polynomial over a range of values:

```
>> x = [0: .2 : 1]

   0    0.2000    0.4000    0.6000    0.8000    1.0000

>> hart5(x)

  5.0000    3.0880    0.9840   -1.2640   -3.6080   -6.0000
```

To display more decimal places, we would have first entered:

```
>> format long
```

If you want to find the roots, integral etc. of "hart5.x" then, as we shall see in the next subsections, you need to create a a function m-file as above. However if you just want to calculate values on a one-time basis, you can do the calculations directly.

```
>> x = [0: .2 : 1]
>> y = (x).^3 -3*(x).^2 - 9*x + 5;   % pointwise powers: .^
```

A useful tool when working with polynomials

In section 7 we will see how to find a polynomial that “fits” a data set. The problem is that the estimating polynomial will be returned as a *vector* of coefficients. But to *evaluate* the polynomlal at a given point, we need to have a *function* whose coefficients are precisely the elements of the vector. Instead of recopying the coefficients, it is simpler---in addition to avoiding copying errors---to write a short program that automatically creates the desired *function.*

The idea is as follows:

Suppose that we have the cubic x^3 -3x^2 -9x + 5. The vector of coefficients is [1 -3 -9 5] and the length of the vector is 4. If we reverse the proceedure and start with the vector [1 -3 -9 5], then we see that the length of the vector = 4. This means that the highest power of the polynomial will be (4-1) = 3 and the coefficent of the x^3 term will be the first term of the vector, namely 1. The second term will involve x^2 and the coefficient will be the second element of the vector, namely -3, etc.

Program 1b

```
% [coeff.m]
% define a function of x using the coefficents as stored  vector.
% the function is thus defined in terms of the two variables vector and  x.

function y = coeff(vector,x)

l = length(vector);

y = 0;
for k = 1:l
y = y + vector(k)*(x).^(l-k);
end
```

This how we would use coeff.m in the above example with "hart5(x)"

```
>> x = [0: .2 : 1]
>> vector_hart5 = = [1 -3 -9  5];
>> y = coeff(vector_hart5 , x)
```

Example 2. The second function involves a combination of polynomials (-5x and 2x) and the product of two transcendental functions. Both exp( ) and sin( ) are predefined MATLAB pointwise functions so we do not have to worry about them.

Program 2

```
% [hart8.m]       Hart, p. 283

% f(x) = [e^(-.5x) * sin(2x)]

function y = hart8(x)

A = exp( -.5*x ) ;            % do in parts; first term
B = sin( 2*x ) ;              % second term
y = (A).*B ;                  % pointwise multiplication
```

Example 3. This involves a ratio with the numerator having both the sine function and a polynomial and the denominator having a polynomial. Since sin( ) is already a pointwise function we do not have to worry about it. We do the numerator and denominator separately and then divide using the pointwise division operator ./ . Since this function is rather complicated, both the numerator and denominator should be checked individually.

Program 3

```
% [hart2.m]   Hart, p. 251, ex. 32

% f(x) = [{sin(3x)}^2 ] / [4x^2 - x]

function y = hart2(x)

num = ( sin(3*x) ).^2 ;             % do square pointwise;
den = 4(x.^2) - x ;
y = (num)./den ;                    % do division pointwise
```

ii. The roots of a polynomial.

To find the roots of a polynomial, we create a vector whose entries are the coefficients of the powers in decreasing order; do not forget 0 for missing powers. We then use the function roots(polynomial)

In example 1 we had:

```
hart5(x) = [x^3 - 3*x^2 - 9*x + 5]
```

and so we write:

```
>> poly = [1 - 3 -9 5]
```

then:

```
>> rac = roots(poly)

    4.6913
   -2.1801
    0.4889
```

iii. Roots of a defined function.

Finding the roots of a defined function requires that the function be defined and that a starting point for the search be given. Thus for the function "hart8" we might start at 2 (doing a plot first may be helpful; see the graph below). The command is:

```
>> root1 = fzero( 'hart8' ,  2 )

    1.5708              %  root at pi/2
```

If we had started at 1 we would have obtained the same root:

```
>> r2 = fzero( 'hart8' , 1 )

    1.5708
```

If we start at .5:

```
>> r3 = fzero( 'hart8', .5 )

 -5.4807e-018           %  root at 0
```

[Note: For complicated functions such as "hart2" in example 3, the algorithm does not work very well.]

iv. Maximum and minimum of a function.

For nice functions with continuous derivatives the function values at the local maxima and minima can be found using the functions max( ) and min( ). The smaller that we make the steps in the domain vector, the more precise will be the answer.

```
>> x = [-1 : .001 : 5];     % do not forget the ;

>> length(x)     % vector; use length( ) instead of size( )

        6001

>> y = hart8(x);

>> max8 = max( y )

    0.6964

>> min8 = min( y )

   -1.5275
```

To see more decimal places we use format long

```
>> format long

>> max8

   0.69644479848592

>> min8
```

```
  -1.52749846885105
```

We can also find the values of x at which the maximum and minimum occur. This is an interesting example of the uses of conditional vectors and is discussed in section II.8.ix. There the maximum is found to occur at element 1664 of x and the minimum at element 93. We find x and y at these elements:

```
>> x(1664)

   0.66300000000000

>> y(1664)

   0.69644479848592       % this is max8 as given above

>> x(93)

  -0.90800000000000

>> y(93)

  -1.52749846885105       % this is min8 as given above
```

[Note: MATLAB has a built-in function fmin( ) which gives the value of x at which the minimum occurs. Unfortunately, in the UNIX version that I used this function did not work properly!

```
>> fmin8 = fmin( 'hart8' , -1 , 5 )

   2.23371853388113
```

This is where the minimum is supposed to be, whereas above we saw that it is at -0.908. We find the value of "hart8" at this point:

```
>> hart8(fmin8)

  -0.31753574300154
```

So MATLAB has found a value greater than 0 (see the graph below)! To see what has happened, we again use the function min(   ), but this time we limit ourselves to positive values of x:

```
>> x1 = [0 : .001 :5];    % interval [0 , 1]

>> y1 = hart8(x1);
>> min( y1 )

  -0.31753568446987
```

Thus MATLAB has only found the minimum on [0 , 5]. See also the discussion in Section II.8, Example 12]

See also example 7c of Section II.8.vii where the conditional conditional vectors of the form:

```
b =[ abs(v - a) < .0001]
```

are used. Here v is a vector and a is a number and we want to find out how many elements of v are within .0001 of a.

v. Numerical integration of a function.

MATLAB has two functions for numerical integration: quad( ) and quad8( ).

```
>> quad('hart8', -1, 5 )

  -0.46158331057559

>> quad8('hart8', -1, 5 )

  -0.46158379374149
```

Instead of the integral, we may wish to find the area under the curve. To do this we need to work with the absolute value of "hart8.m" and so we define a function "hart8abs"

Program 4

```
% [hart8abs.m]    % absolute value of hart8

function y = hart8abs(x)

y = abs( hart8(x) );     % work with absolute value function
```

We check this function out by finding the maximum and minimum:

```
>> max( hart8abs(x) )

   1.52749846885105
```

This last value is -min8, as given above.

```
>> min( hart8abs(x) )

    0
```

We now find the integral using both quad( ) and quad8( ):

```
>> quad( 'hart8abs', -1 , 5 )
```

```
Recursion level limit reached in quad.  Singularity likely.
```

MATLAB has indicated that there may be some problems. This is probably due to the non-differentiability of "hart8abs" wherever "hart8" crosses the x-axis.

The answer was found to be:

```
   2.11673639164145
```

If we use quad8( ) there are no problems.

```
>> quad8( 'hart8abs', -1 , 5 )

   2.11673472997670
```

vi. Plotting.

Plotting user defined functions is just a special case of plotting a value vector against a domain vector as discussed in section II.6.i. In that section we used the domain vector

```
>> x1 = [0 : .25 : 1]
```

and then defined the value vector:

```
>> y1 = sqrt(x1)
```

To plot we simply wrote:

```
>> plot( x1 , y1 )
```

We do exactly the same thing for user defined functions. To make sure that the graph is smooth we use small steps.

Example 4. Suppose that we want to plot "hart8" over the interval [-1 , 5]. We define:

```
>>   x = [-1 : .01 : 5];
```

and then, making sure that we suppress printing:

```
>>  y = hart8(x);
```

To plot we enter:

```
>>  plot( x , y )
```

We can even avoid the step of defining y and simply write:

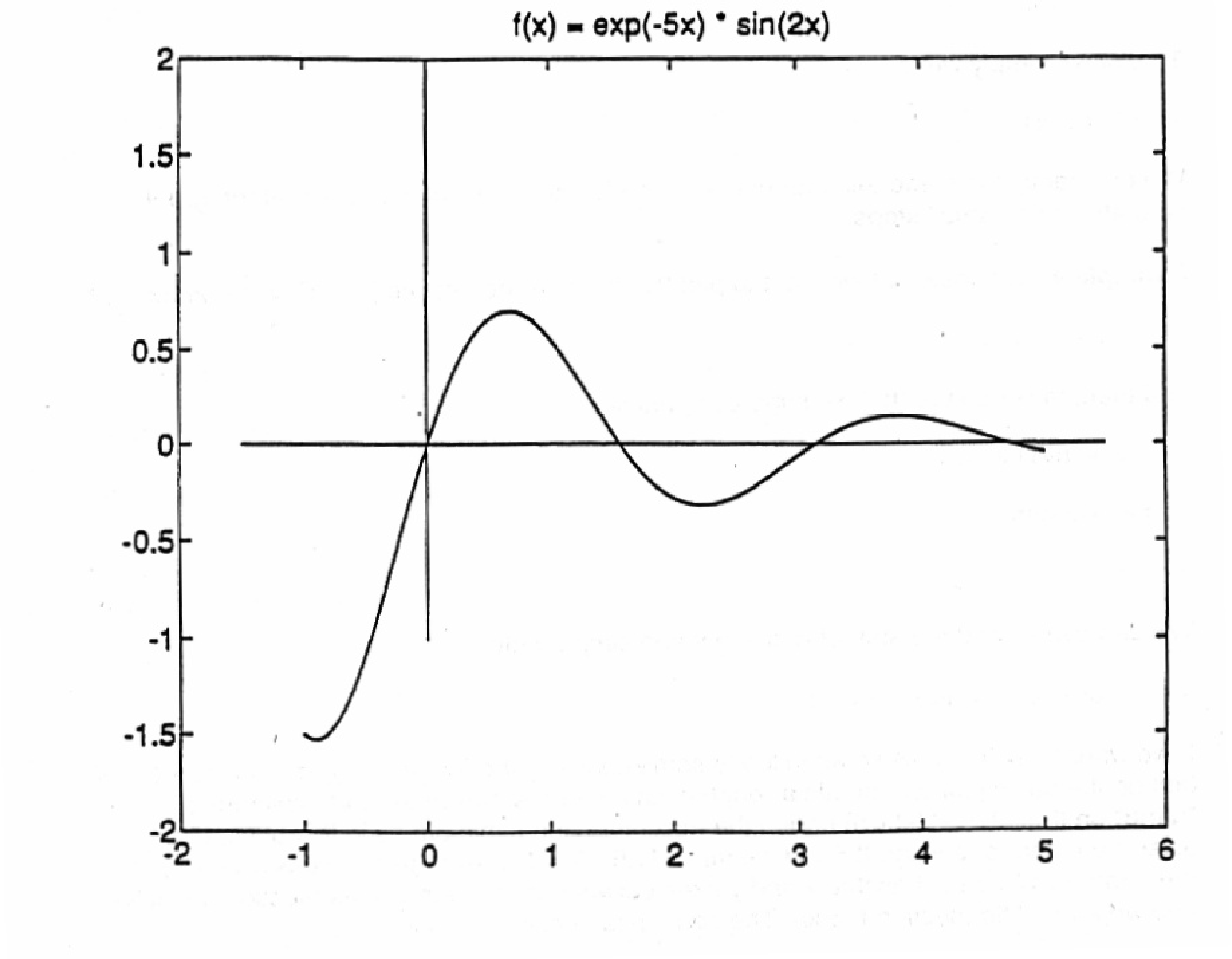
```
>>  plot( x  ,  hart8(x) )
```

If we want to fix the axes so as to leave some space at the top of the graph, we can either first do the basic plot and obtain a rough estimate of the maximum and minimum of "hart8" on the interval [-1 , 5] or find the maximum and minimum as in iv. above. Using the axis( ) command (section II.6.viii) we make [1.5 , 5.5] the x-range of values and [-2 , 1] the y-range. We also draw the x and y axes between these same limits (section II.6.iii,iv) and we add a title (section II.6.xi). The sequence of commands is:

```
>>  axis( [ -1.5  5.5   -2  1 ] )     % section II.6.viii

>>  hold            % section II.6.ix

>>  plot( x  ,  hart8(x) )

>>  plot( [-1.5  5.5]  [0  0] )          % x-axis; sect. II.6.iii

>>  plot( [0  0]  [-1  2] )            % y-axis; sect. II.6.iv

>>  title(' f(x) = exp(-5x) * sin(2x) ' )   % sect. II.6.xi
```

The graph of "hart8", produced by the above commands, is shown in the diagram on the next page. For a more complicated plot, involving several functions, marked points and text, see program 5.

[Note: For functions with vertical asymptotes or removable singularities, such as "hart2", MATLAB runs into difficulties. I was not able to obtain a satisfactory graph of "hart2" over the entire range. In some cases, a change in the step size, coarser or finer, will help as may plotting in segments]



vii. Curve fitting.

We know that 2 points determine a line, 3 points a conic (ellipse, parabola, hyperbola) and in general (weird cases excepted) n+1 points in the plane determine a polynomial of degree n. If however we have a medium or large size set of data points and we want a line or low-order polynomial to "fit" the data then we use a least squares polynomial fit. In general we will have two vectors x_values and y_values which correspond to the ordered pairs (x,y) of data points. If we want the polynomial of order n that provides the best least squares fit then the command is:

>> poly***n*** = polyfit(x_values, y_values, ***n***)

Here poly***n*** is the *vector* of coefficients of the polynomial of order ***n***

If the number of data points is n+1 then the polynomial will pass through all the points. (If the degree of the polynomial is equal to or larger than the number of data points then some of the coefficients will be 0 and in fact the approximations will generally not be as good).

Example 5. Six points were chosen on the curve of hart8(x). For simplicity the x values were taken to be the integers 0, 1, 3, 4, 5, 6, but with a jump of 2 in the middle. From the complexity of the function we can suspect ahead of time that 6 points will not give a very good fit. Note how the computed polynomials are used directly (via the function coeff) to obtain the y values over the plotting domain. Because the leading coefficients are very small, simply copying the coefficients from a 4 place output would result in errors. Program 5 is reproduced here will all the details; it involves superimposing the picked points; multiple curves, interpolation points, titles and text. The second part of the program, involving interpolations, is discussed in Example 6.

Program 5

```
% [fit_plot.m]
% Shows curve fitting and interpolation,
% uses hart8.m

clf   % clear the previous graphics, clg in older versions

% first plot  hart8(x) for an extended range.

x  = [-.5: .01 : 6.5];
y = hart8(x);
plot(x, y)

hold       % holds the plot

% now determine values of hart8 at  0, 1, 3, 4, 5, 6
% note that 2 is missing
x_values = [0 1 3 4 5 6];
y_values  = hart8(x_values);

l= length(x_values);
for j = 1 : l
    u = x_values(j); % read from vector
    v = y_values(j);% read from vector
    plot(u,v, '*')
end

% now calculate and plot 3th degree polynomial approximation

vector_coeff3 = polyfit(x, y, 3);     % vector_coeff3 is the vector of coefficients of
                                      %  the approximating polynomial of degree 3
```

```
disp('3rd degree approximating polynonial:'), disp(vector_coeff3)

%  The function coeff(vector, x) was created as program 1b above.
%  With vector = vector_coeff3  and  the domain vector for the plot chosen
%  above as x  = [-.5: .01 : 6.5] we can  obtain y3, the corresponding set of
%  values determined by the approximating polynomial of degree 3.
% We now use it to obtain the  polynomial function whose coefficients are given:

y3 = coeff(vector_coeff3, x);

plot(x , y3, '-.')

% Now we do exactly the same thing for the approximating polynomials of degree
% 4 and 5.

% Calculate and plot 4th degree polynomial approximation.
vector_coeff4 = polyfit(x, y, 4);
disp('4th degree polynonial:'), disp(vector_coeff4)

y4 = coeff(vector_coeff4, x);

plot(x , y4, '--')

% Calculate and plot 5th degree polynomial approximation
vector_coeff5 = polyfit(x, y, 5);
disp('5th degree polynomial: '), disp(vector_coeff5)

y5 = coeff(vector_coeff5, x);

plot(x , y5, ':')

title(' Curve fitting using values from f(x) = hart8(x)   -  Interpolation ')

xlabel(' solid line = hart8(x) * = chosen values    -. = 3rd degree  -- = 4th
degree  .. 5th degree ')

% For the purpose of comparison, we are going to calculate, display and plot  the
% values of hart8 and  the approximating polynomials at the point x =4.5

v0 = hart8(4.5);
v3 = coeff(vector_coeff3, 4.5);
v4 = coeff(vector_coeff4, 4.5);
v5 =coeff(vector_coeff5, 4.5);

disp('hart8(4.5)') , disp(v0)
disp('poly3(4.5)') , disp(v3)
disp('poly4(4.5)') , disp(v4)
disp('poly5(4.5)') , disp(v5)

plot(4.5, v0, '+')
plot(4.5, v3, 'x')
plot(4.5, v4, 'x')
plot(4.5, v5, 'x')
```

The polynomials obtained are:

3rd degree polynonial:

```
    0.0148   -0.1402    0.3156   -0.0337
```

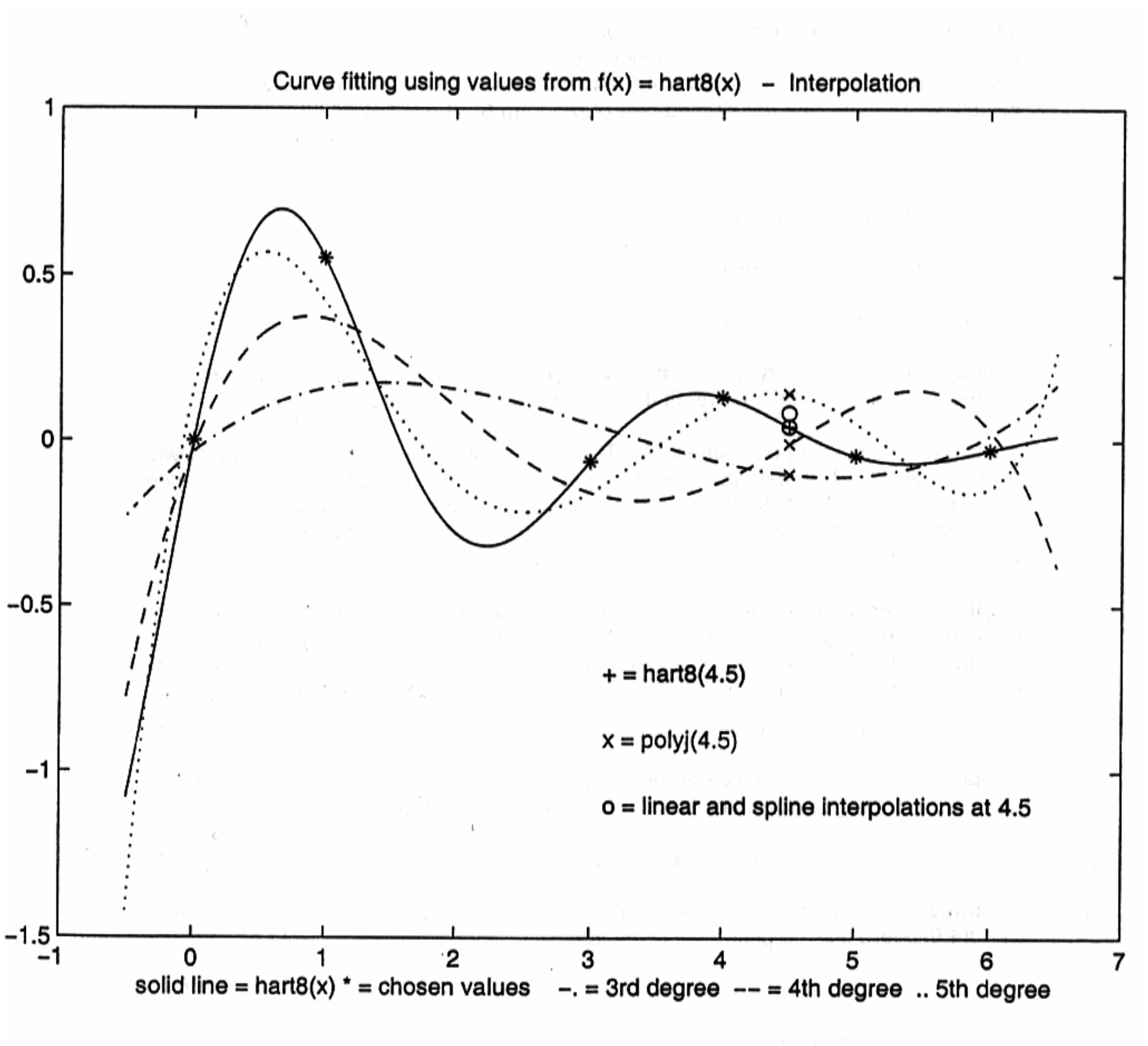
4th degree polynonial:

```
   -0.0161    0.2081   -0.8404    1.0376   -0.0203
```

5th degree polynomial:

```
    0.0098   -0.1631    0.9565   -2.2827    1.7471    0.1536
```

The plot is given on the next page:



viii. Estimating values / Interpolation

Example 6. One reason that we want approximating polynomials is so that we can estimate values at points other than those for which the values are given. If we evaluate our 3 approximating polynomials at 4.5 and compare the values obtained with the true values, we obtain---see the second part of program 5---the following:

hart8(4.5) = 0.0434

poly3(4.5) = -0.0998

poly4(4.5) = -0.0081

```
poly5(4.5)  =    0.1431
```

Because the fit is not very good in this example, we have---by accident---that the 3rd degree polynomial gives a better fit than the higher order polynomials.

Instead of first calculating polynomials MATLAB has an interpolation function interp1 which uses the given data to estimate values. There are two options that available with interp1. In both cases we start with two vectors x_values, y_values (as in polynomial approximation) and approximate values are given at x where x is either a single point or a vector of points.

a. Linear interpolation. Here pairs of points are connected with straight lines and values are computed using similar triangles. MATLAB will not extraplate using this method. This was the classical method before the advent of computers and any engineer of a certain age can verify that much time in school was spent interpolating log tables.

```
>> approx_linear = interp1(x_values, y_values, x, 'linear')
```

b. Spline interpolation. Here different cubic polynomials are used to connect pairs of adjoining points. The polynomials are picked so that they match up "smoothly".

```
>> approx_spline = interp1(x_values, y_values, x, 'spline')
```

Example 7. In example 6 we used the polynomials of example 5 to estimate the value of hart8(4.5). Here are the estimates using interp1.

```
hart8(4.5)             =    0.0434

linear estimate at 4.5 =    0.0446

spline estimate at 4.5 =    0.0852
```

Again by accident, the best estimate corresponds to drawing a line between the values at 4 and 5.

For illustrative purposes the number of points was kept small in these examples. For a large number of points we can expect the higher order polynomials to give the best fit and the spline estimate to be the best estimator. However strange things can happen and a set of points that vary rather wildly is a sign that one should proceed with caution.

SECTION II.8 LOGICAL OPERATORS

SUMMARY

A word of caution about a possible confusion.
The relational symbols (I) and logical operators (ii) appear in two contexts.

Context 1. In parts iii through vii, we are interested in applying a criteria and if the criteria is met we do something. Typically this occurs in a program with conditional statements (iii). Typically we might have something such as:

```
   if ( condition )  %  note the (  )
```

Context 2. In parts viii through x we want to determine which elements of a vector (matrix) satisfy certain conditions. This is done by placing a 1 or a 0 in a vector (matrix) of the same dimensions depending on whether or not the condition is satisfied. Typically we might have something such as:

```
        [a condition b]   %  note the [   ]
```

----

i. Relational symbols.

equality: ==;

[Do not confuse with the *assignment* operator =; e.g. a = 2]

not equal: ~=
    ex. if (integer ~= 2)

inequalities: < <= > >=

The relational symbols apply to vectors and matrices of the same dimension and can be used in conditional loops and conditional vectors.

ii. Logical operators.

notice the ( ) enclosing the logical statement

```
"and"   &          if ( (x < 6)  &   (x ~= 4) )
"or"    |          if ( (x < 6)  |   (x == 10) )
"not"   ~          if ( (x < 6)  &  ~(x == 4) )
```

[note: the outer parentheses are optional, but highly recommended.]

[note: statements such as (6 < x < 8) will give a wrong answer!]

SECTION II.8 LOGICAL OPERATORS

iii. Conditional " if - elseif - else - end " loops.

```
if          x > 0
                  x = x/sum(x)

elseif   +++
                  .....
elseif   ++++
                  .....
else
                  .....
end      [Octave will accept this, but prefers endwhile]
```

The short forms if - else - end and if - end are also valid.

iv. Breaking completely out of a program / error messages

Place **return** at the point that you want to quit the program

To break and print a message: **error('** matrix not square **')**

v. "while-loops".

```
while          +++++
                  .....
end
```

vi. Breaking out of for-loops and while-loops.

break

vii. Conditional vectors and matrices.

To replace the elements of a vector or matrix by 1 where a condition is satisfied and 0 where the condition is not satisfied. If u, v, w are vectors and A, B, C are matrices then we can determine conditional vectors and matrices in ways such as the following: Note the use of **[ ]** and not ( ) as in ii above

```
conditional_vector1 = [ v < 3 ]
conditional_vector2 = [ v == 3 ]
conditional_vector3 = [ abs(v-3) < .1 ]
conditional_vector4 = [ abs(v-3) < eps]    % eps = "smallest" number
conditional_vector5 = [ (u < v) & (v < w) ]
conditional_matrix1 = [ (A >= 2) | (B == 5)]
conditional_matrix2 = [ (A == B) & (B > 4) | (B < C)]
```

SECTION II.8 LOGICAL OPERATORS

viii. Testing conditional vectors and matrices with any( ), all( ).

```
any( x1 )
all( x1 )
```

ix.   Finding indices of a vector that satisfy certain conditions and obtaining the values as a new vector.

```
e = find( x1 )    [x1 is the conditional vector]
x2 = x1( e  )     [x2 = elements of x1 that satisfy the condition]
```

x.   Four programs to illustrate the use of four programming techniques to do the same problem.

DETAILS AND EXAMPLES

i. & ii.   Relational symbols and logical operators.

Equality of two elements is indicated by ==.

If a and b are two vectors (or matrices) of the same dimension then a == b means that all corresponding elements of a and b are equal. See the example in section iii for an illustration of this.

A common mistake is to confuse the relational symbol == with the assignment operator =. For example if we write:

if (integer == 2)   % note the **(   )**

we are checking to see if the variable "integer" is *equal to* the value 2. But when we write

integer = 2

we are *assigning* the value 2 to the variable "integer"

Non-equality is indicated by ~=
ex. if (integer ~= 2)

Inequalities are indicated by < , <= , > , >=

If a and b are two vectors (or matrices) of the same dimension then a < b means that every element of a is less than the corresponding element of b.

SECTION II.8 LOGICAL OPERATORS

Statements involving the relational symbols can be combined via logical operators as illustrated in the following statements:

```
"and"   &          if ( (x < 6)  &   (x ~= 4) )
"or"    |          if ( (x < 6)  |   (x == 10) )
"not"   ~          if ( (x < 6)  &  ~(x == 4) )
```

Note that the first and third conditions are logically equivalent. In the first we use the "not-equal" relational symbol ~=, whereas in the third we take the negation ~ of the equality relationship.

Statements such as:

(6 < x < 8)   [i.e. 6 < x and x < 8]

x == (4 | 7)   [i.e. x = 4 or x = 7]

where we try to combine two inequalities at once will give the wrong answer; try these statements with a = [0:10].
We must write

```
if (  (x < 6) & (x < 8)  );   if ( (x == 4) | (x == 7) )
```

The outer parentheses are optional, but highly recommended. The use of lots of parentheses helps to avoid logical errors. For simple statements such as:

if (integer == 2) , one could leave out the parentheses, but even in these simple cases they often show the statement more clearly.

iii.   Conditional if - elseif - else - end loops.

In simple cases a if - end loop or a if - else - end loop is sufficient and there will not be a elseif statement.

To illustrate we test the relationship between three numbers. First we use a simple if - end loop to print a statement only if we have equality. Then we use a if - else - end to have the program also print a statement even if we do not have equality.

Example 1a

```
% [equal1.m]

a = 1
b = 3
c = 4
```

```
disp(' part 1')
if  (a == b)
   disp('a equal b')      % print out only if equal
end

disp('part 2')
if (a < c)
   disp('a is less than c')       % print out if smaller
else
   disp('a is not less than c')   % print out if not smaller
end
```

We now run the program:

```
>> equal1

   part 1
   empty       % no equality so nothing was printed

   part 2
   a is less than c
```

In Example 1a we tested the relationship between numbers. We can do the same thing for vectors and matrices.

Example 1b.

```
% [equal2.m]
% testing == and < with vectors

a = [1 2]
b = [3 4]
c = [1 4]

% equality of vectors means that all elments are equal
if (a == b)
   disp('vector a equal vector b')
else
   disp('vector a is not equal to vector b')
end

% less than for vectors means that all elments are less than
if (a < b)
   disp('vector a is less than vector b')
else
   disp('vector a is not less than vector b')
```

```
end

if (a < c)
   disp('vector a is less than vector c')
else
   disp('vector a is not less than vector c')
end

% we run the program and obtain the following:

vector a is not equal to vector b
vector a is less than vector b
vector a is not less than vector c
```

Example 2. To illustrate the use of elseif we consider the vector [1 2 5 8 9] and let y = x(n). The following program examines the values of y and does the following:

1. We use the condition (y/2 == floor(y/2)) to test if y is even. Since this is the first condition it is preceeded by if. Whenever y is even this fact is printed out via disp(state1). Note the use of == to check equality.

2. Next we use the condition (y <= 3) to check if y is less than or equal to 3. Since this is neither the first nor the last condition we preceed it by elseif

3. We use the combination of two conditions, (4 < y) & (y < 7), to check if y is strictly greater than 4 and strictly less than 7. We again preceed the test statement with elseif.

4. Finally we take care of all other possibilities with else.

To end the if - elseif - else loop we write end.

Note the double loop. The outer one is the "for-loop" and the inner one is for the conditional statements.

Program 2

```
% [logic1.m]

x = [1 2 5 8 9]

state1 = 'even';
state2 = 'less than or equal to 3';
state3 = 'greater than 4 and less than 7';
state4 = 'none of above';
```

```
for n = 1:5
              y = x(n);
              disp('integer is:  '), disp(y)
          if       ( y/2 == floor(y/2) )
                        disp(state1)
          elseif   (  y <= 3 )
                        disp(state2)
          elseif  ( (4 < y) & (y < 7) )
                        disp(state3)
          else
                         disp(state4)
          end                          % end of conditional statements
end                                    % end of for-loop
```

We run the program:

```
>> logic1

x =
     1     2     5     8     9

integer is:     1
  less than or equal to 3

integer is:     2
  even

integer is:     5
  greater than 4 and less than 7

integer is:     8
  even

integer is:     9
  none of the above
```

iv.   Breaking completely out of a program.

Example 3. Sometimes, we want to terminate a program when a certain condition is met. To do this, we simply type return. This is illustrated in the following program where we keep taking powers of a matrix until all the elements are less than or equal to .1 . We know ahead of time that this will eventually happen since the row sums are less than 1 and so we pick an arbitrarily large number (in this case 100) and loop this number of times.

The same problem is solved by different techniques in examples 5, 12 and 13.

SECTION II.8 LOGICAL OPERATORS

Program 3

```
% [return1.m]

P = [.2 .6; .4 .3]

% loop 100 times. 100 is large enough to ensure that we break
for n = 1:100
        if ( max(max(P^n)) <= .1 )
             disp('first index when power < .1 is: '), disp(n)
             disp('P^n = '), disp(P^n)
             return
        end
end
```

To find the index we type return1

```
>> return1

P =

    0.2000    0.6000
    0.4000    0.3000

first index when power < .1 is:   7

P^n =

    0.0558    0.0758
    0.0505    0.0685
```

Example 4. Sometimes we want to check that conditions are met before the main part of a program proceeds. In case the condition is not met we issue an error statement in the form:

error(**'** *condition is not met* **'**).

If the condition is not met, then MATLAB will print the error message and break out of the program This is illustrated in the program "checksiz.m" which forms part of the complete version of program 1 of section III.1.

Program 4

```
% [checksiz.m]
function y =checksiz(A)

[rows, cols] = size(A);
```

```
if ( cols ~= (rows + 1)  )
   error( ' columns ~= rows + 1 ')     % returns to with message
end

disp('  ')
disp('number of rows is: '),  disp(rows)
```

We test this program first on a matrix which does not satisfy the condition and then on one that does. Notice the use of rand( ) to quickly generate matrices for testing.

```
>> A = rand(3,3)
>> checksiz(A)

Error using  checksiz
columns ~= rows + 1

>> B = rand(3,4);
>> checksiz(B)
  number of rows is:  3
```

v. "while-loops".

Sometimes, we want to continue a sequence of calculations until a certain condition is met. For such situations, "while-loops" are very useful. This involves the opening statement while and the closing statement end.

Example 5. The problem is the same as in example 3; we want to find the first power of a matrix such that all the elements are less than .1 . In example, we used a "for-loop" together with a conditional "if-loop". When the condition was satisfied, we broke out of the program with return.

With "while-loops" we do away with both loops; as soon as the condition is no-longer satisfied the rest of the program is executed. When using "while" loops, it is important to make sure that we start off correctly. For example, in the present example we have to give the initial value n = 1. If we had had P^1 > .1 then the "while-loop" would never have been executed. In addition to example 3 this problem is solved by different techniques in examples 12 and 13.

Program 5

```
% [while2.m]

P = [.2 .7; .4 .3]

n = 1;
```

```
while     (max(max(P^n)) > .1)
              n = n + 1;
end

disp('last index when power > .1 : '), disp(n-1)
disp(P^(n-1))

disp('first index when power <= .1 : '), disp(n)
disp(P^n)
```

The program is run:

```
>> while2

P =

    0.2000    0.7000
    0.4000    0.3000

last index when power > .1 :    7

    0.0806    0.1173
    0.0670    0.0973

first index when power <= .1 :   8

    0.0630    0.0916
    0.0523    0.0761
```

In program 5 there was only one condition in the while statement. The program [while6.m] in section xii illustrates a "while-loop" with two conditions.

vi. Breaking out of for-loops and while-loops.

To break completely out of a program, we use return (see iv above), but sometimes we only want to break out of a "for-loop" or a "while-loop". To do this we use break, usually in connection with a conditional statement. The command break is associated with an if (or elseif, or else) statement:

```
if       +++++
              break
elseif   +++++
```

Note that the break command only applies to the innermost loop containing the expression break. This is illustrated in the following program:

SECTION II.8 LOGICAL OPERATORS

Example 6. We start with the matrix P which has all the elements in row j equal to j. We then do the following:

1. We leave even rows alone by means of break, which causes the program to leave the inner column loop

2. We replace the elements in the third row by the sum of the 3 + 2*column.

3. We replace the elements of rows 1 and 5 by zeros.

Program 6

```
% [break1.m]

% generate intial matrix with elements of row j equal j
P = zeros(5,3);  % predefine matrix; II.4.vii,x
for j = 1:5
   P(j, :) = j*ones(1,3);
end

disp('original matrix'), disp(P)

for m = 1 : 5            % rows
    for n = 1:3          % columns
        if  (f loor(m/2) == m/2)  % row index is even
          % break out of columns loop and goes to next row
            break
        elseif  (m == 3 )   % row 3
             P(m,n) = 3 + 2*n;  % assign the value 3 + 2*n                          else
             P(m,n) = 0;  % assign the value 0 in rows 1 & 5
        end    % end for conditional statement
    end        % end of "for-loop" for columns
end            % end of "for-loop" for rows


disp('transformed matrix'), disp(P)
```

The result of the program is:

```
>> break1

original matrix

     1     1     1
     2     2     2
     3     3     3
     4     4     4
     5     5     5
```

transformed matrix

```
     0     0     0
     2     2     2
     5     7     9
     4     4     4
     0     0     0
```

vii. Conditional vectors and matrices.

In its simplest form a conditional vector (matrix) is defined in terms of another vector a (matrix B) and a condition involving one of the numerical relations. The conditional vector will be of the same size and its elements will be 1 or 0 depending upon whether or not a specified condition is satisfied.

Example 7a. Suppose that we have the vector:

```
>> a  = [1 : 5]

  1     2     3     4     5
```

We want to test the elements to see if they are less than 3. The logical statement is:

```
a < 3
```

Note that since MATLAB works with matrix operators we use the relational symbols with the *entire vector* a and not with the individual elements of a, i.e. *we do not write*: a(k) < 3

To define the conditional vector we simply put the conditional statement inside the vector brackets **[   ]:**

```
>> conditional_vector1 = [a < 3]
```

MATLAB replaces the elements of B by 1 where the elements are less than 3 (i.e. for the values 1 and 2) and replaces the elements by 0 when this condition is not satisfied:

```
conditional_vector1 =
  1     1     0     0     0
```

Example 7a involved a conditional vector, but the one works with a matrix in exactly the same way:

Example 7b. We generate a random 5 x 6 matrix (not shown) and then we form the conditional matrix cond_B which has a 1 where ever the random elements are less than .5 (in simulation this would correspond to tossing a coin and seeing if we have a head). We then add up the columns using sum( ) and this tells us how many 1s we had in each column:

```
>> B = rand(5,6);
>> cond_B = [B <= .5]

cond_B =

     1     1     0     1     0     0
     1     0     0     0     1     0
     0     0     1     0     0     1
     0     1     1     0     1     1
     0     1     1     0     0     0

>> sum(cond_B)

     2     3     3     1     2     2
```

Example 7c. Conditional vectors can also involve more complicated examples. Suppose we generate a random vector of length 4:

```
>> a = rand(1,4)

    0.2190    0.0470    0.6789    0.6793
```

If a were too large to examine visually we might want to know how many of the elements of a were within .0001 of the value .2190. To find out we use a conditional vector involving the function abs( ):

```
>> d = [abs(a-.2190) <= .0001]

     1     0     0     0
```

The number of elements satisfying the condition is given by the sum of the 1s:

```
total = sum(d)

    1
```

SECTION II.8 LOGICAL OPERATORS

Notice how we tested for numbers within .01 of .2190. Suppose we had used eps which is the smallest non-zero number that MATLAB can work with (eps depends upon the version and system):

```
>> e = [abs(a-.2190) < eps]

     0     0     0     0
```

The trouble is that when a was displayed only 4 decimals were given and the first element was not exactly .2190. Depending upon the situation one might want to approach a problem such as this in a way similar to that described in Section II.7.iv.

So far the examples have involved the comparison of a vector or matrix with a number. We can also compare vectors (matrices) with vectors (matrices) of the same size.

Example 8a

```
a = [1 2]
b = [3 4]
c = [1 4]

v1 = [a < b]
v2 = [ a < c]
v3  = [b == c]

v1 =      1     1
v3 =      0     1
v4 =      0     1
```

[Note: do not confuse the above which involves conditional vectors with statements such as " a < b " that one finds in linear algebra texts. In algebra a < b means that all the elements of a are less than the corresponding element of b].

Example 8b

```
B =
     1     2
     3     4

C =
     2     2
     3     4
```

SECTION II.8 LOGICAL OPERATORS

```
>> E = [B == C]

E =
    0     1
    1     1
```

When creating conditional vectors (matrices) we can use both the relational symbols and the logical operators. This is illustrated in the following example.

Example 9a.

```
>> x = [1:7]

    1     2     3     4     5     6     7

>> x1 = [x < 7]

    1     1     1     1     1     1     0

>> x2 = [ (x < 7) &  (x == 2) ]

    0     1     0     0     0     0     0

>> x3 = [ (x < 7) &  ~(x == 2) ]

    1     0     1     1     1     1     0

>> x4 = [ (x < 7) &  ~( (x == 2) | (x == 4) ) ]

    1     0     1     0     1     1     0

>> x5 = [ (x < 7) &  ( ~(x == 2) & ~ (x == 4) ) ]

    1     0     1     0     1     1     0
```

Note that by De Morgan's (1806 -71) laws the two statements:

```
~( (x == 2) | (x == 4) )
```

and

```
( ~(x == 2)  &   ~ (x == 4) )
```

are equivalent, which is why x5 is identical to x4
Example 9b. Instead of a number we can do the same thing with vectors:

SECTION II.8 LOGICAL OPERATORS

```
a =    [1     2]

b =    [3     4]

c =    [1     4]

>> e = [(a < b) & (b==c)]

e =      0     1
```

Conditional matrices are very useful when we want to count the number of elements in a matrix which satisfy a certain condition. This is because the sum of all the 1's that appear in the conditional matrix is precisely the same as the number of elements of the original matrix that satisfy the condition.

Example 10. In the following program we generate a random vector of length 1000. We then test each element to see if it lies in the interval [.2 , .3] . This condition determines the conditional vector d1 We then add up the elements of d1 using sum( ).

Program 7

```
% [logic2.m]

d = rand(1 , 1000);                   % section II.2.iv

% create a conditional matrix
d1 = [ (.2 < d) & (d < .3)];

counter = sum(d1);                    % add up the 1's

disp('number of elements of d between .2 and .3')
disp(counter)

disp('relative frequency of elements between .2 and .3')
disp('should be approximately .1')

disp(counter/1000)
```

The ouput of this program is:

```
>> logic2

number of elements of d between .2 and .3
     109
```

```
relative frequency of elements between .2 and .3
should be approximately .1
      0.1090
```

Example 11. We start with a 3 x 4 matrix:

```
D =

     0     0     7     4
     2    -1    -3     0
     0     5     0    -1
```

Suppose that we want to extract the negative entries of D and then count and order them. We first put all the elements of D, in the order (1,1), (1,2), (1,3), (2,1), ... into a row vector use a single : (section II.4.xi):

```
>> x = D( :)'   %  transpose via ' to obtain a row vector

0    2     0     0    -1     5     7    -3     0    4    0  -1
```

We now sort x in increasing order using sort( ); see section II.5.ii :

```
>> y = sort(x)

 -3    -1    -1     0     0     0     0     0     2     4   5  7
```

Next, we create a conditional vector

```
>> z = [y < 0]

1     1     1     0     0     0     0     0     0     0     0   0
```

The number of negative elements is just the sum of the 1's:

```
>> num = sum(z)

     3
```

Extract the first three, negative, elements from y

```
>> y  = y(1 : num)

   -3    -1    -1
```

viii. Testing conditional vectors and matrices with sum( ), any( ), all( ).

SECTION II.8 LOGICAL OPERATORS

We start with a vector and then obtain a conditional vector of 0's and 1's. To check if any element satisfies the condition we take the sum of the conditional vector. If the sum is greater than 0, then at least one of the elements satisfied the condition:

```
% [chk_con.m]

x =  [1 2 3 4 5]
x1 = [(x < 3)]
disp(x1)

s = sum(x1)   % warning:  DO NOT use 'sum' as a variable; it is a built-in
               %           function

if  ( s > 0)
    disp('at least one element is less than 3')
end
```

```
>> chk_con

   x1 =   1  1 0 0 0

   at least one element is less than 3
```

MATLAB has a function any( ) which assigns the value 1 to a conditional vector if at least one value is equal to 1.

```
>> any(x1)

       1                   % there is at least one 1
```

The function all( ) assigns the value 1 to a conditional vector only if all the values are equal to 1 (instead of all one could check that the sum of the elements of the conditional vector equals the length):

```
>> all(x1)

       0                   % not all elements are  1
```

[Note: For some reason all assigns 1 to the empty vector!!!]

For matrices, any( ) and all( ) operate on each column of the matrix and treat the column as a vector. The values 0 or 1 are assigned to each column and these are given in a vector:

SECTION II.8 LOGICAL OPERATORS

```
>> A =

      -1     4
       5     6

>>  A1 = [A <= 5]

         1     1
         1     0

>> any(A1)                 % operates on columns

         1     1
```

We now have a vector, namely any(A1) and so we can apply any() to this vector. If any of the entries in the conditional matrix were equal to 1 then any(any(A1)) will give a 1; if all were 0, we would obtain a 0.

```
>> any(any( A1 ))

          1
```

This same example could be done (Section II,5,i) using sum(sum A1)

The statement all(all(A1)) will assign the value 1 to A1 if and only all the entries are 1:

```
>> all(A1)                  % operates on columns

 1     0

>> all(all( A1 ))

    0
```

The following two examples and programs illustrate the use of any(any( )) and all(all( )) in connection with conditional matrices. In program 9 we keep raising the power of P while any of the elements are greater than .1 . In program 10 on the other hand, we break out of a for-loop using return as soon as all of the terms are less than or equal to .1 .

Programs 9 and 10 should be compared with programs 5 and 3 respectively. Both programs 9 and 5 use while-loops, but whereas 9 uses any(any( )), program 5 uses max(max( )). Similarly both programs 10 and 3 use a for-loop inside of which is nested a conditional statement, the difference being that program 10 uses all(all( )) whereas program 3 uses max(max( )).

SECTION II.8 LOGICAL OPERATORS

Example 12 .

Program 9

```
% [any_any.m]

P = [.2 .7; .4 .3]
n = 1;

% keep going if any element of P^n > .1
while      any( any ([P^n  > .1] ) ) == 1
           n = n + 1;
end

disp('last index when power > .1 : '), disp(n-1)
disp(P^(n-1))

disp('first index when power <= .1 : '), disp(n)
disp(P^n)
```

We run the program:

```
>> any_any

P =

    0.2000    0.7000
    0.4000    0.3000

last index when power > .1 :    7

    0.0806    0.1173
    0.0670    0.0973

first index when power <= .1 :    8

    0.0630    0.0916
    0.0523    0.0761
```

Example 13.

Program 10

```
% [all_all.m]
```

```
P = [.2 .7; .4 .3]

% pick a large upper limit; will exit before end (we hope!)

for n = 1:1000

   if  all ( all ([P^n <= .1] ) ) == 1  % max element <= .1
      disp('last index when power > .1 : '), disp(n-1)
      disp(P^(n-1))

      disp('first index when power <= .1 : '), disp(n)
      disp(P^n)

      return               % returns to MATLAB prompt
   end
end
```

When we run this program the answer is the same as the previous three times; see example 10 above and also program 3 and example 4.

ix. Finding indices of elements of a vector that satisfy certain conditions.

We consider the integers between -3 and 3

```
>> x = [-3 : 3]

   -3    -2    -1     0     1     2     3
```

Next we form the conditional vector x2 determined by the condition that the element of x is even (floor of half the number = half the number)

```
>> x2 = [floor(x/2) == x/2]

    0     1     0     1     0     1     0
```

The function find( ) gives the indices for which the conditional matrix has the entry 1.

```
>> even = find(x2)

   2     4     6
```

Now we "evaluate" the original vector x1 at the elements given by the entries even (see section II.4.iv). Note that x2 has the same dimensions as even:

```
>> x2 = x1( even  )
```

```
   x2 = [-2 0 2]
```

We can use the function find the x-values for which a function attains the maximum.

Example 12. In section II.7.iv we considered the maximum and minimum of the function "hart8". The first steps are repeated here for convenience:

```
>> x = [-1 : .001 : 5];          % use ; to avoid printing

>> length(x)               % vector; use length( ) instead of size( )

       6001

>> y = hart8(x);

>> length(y)

       6001                    % a check!

>> max8 = max( y )

    0.6964
>> min8 = min( y )

   -1.5275
```

We now form the conditional vector determined by the condition that y equals max8:

```
>> vec_max = [y  ==  max8];

>> length(vec_max)                 % a check!

       6001

>> find( vec_max )

       1664
```

This shows that the maximum is only reached at one point.

We find the x and y values for this index:

```
>> x(1664)
```

```
   0.66300000000000

>> y(1664)

   0.69644479848592                % this is max8
```

We do the same for the minimum:

```
>> vec_min = [y == min8];

>> find( vec_min )

    93

>> x(93)

  -0.90800000000000

>> y(93)

  -1.52749846885105

>> min8


  -1.52749846885105
```

x.   Four programs to illustrate the use of four programming techniques to do the same problem.

We saw in examples 3, 5 , 12 and 13 how different programming techniques can be used to solve the same problem.

Here we give another example and solve it by four different techniques:

Problem: for j = 2,3,4, find the smallest power such that j raised to that power is greater than 100. Do not however go above 5 as a power.

Technique 1: "while-loop" with a double condition (no break).
Technique 2: double "for-loop" with a break.
Technique 3 "while-loop" with one condition and a break.
Technique 4: "while-loop" with no conditions (!) and a double break.

Technique 1: "while-loop" with a double condition (no break)

```
% [while6.m]

% We wish to find the first power for which j^power > 100,
% but we do not wish to try values of power larger than 5.

% The inner while loop has two conditions
% This avoids the need for a "break".

% Note the "priming" values: value = 0; power = 1
% If we did not have values, the "while" statement could not
% be  evaluated.

store = [ ];

for  j = 2:4
   value = 0;     % need a starting value less than 100!
   power = 1;
while ( value <= 100 ) & (power <= 5)   % two conditions
        value = j^power;
        power = power + 1;
   end   % end of the while loop

   power = power - 1;  % reduce by 1 when we quit
   result = [j, power, j^power];
   store = [store; result];
end  % end of the for-loop

>> while6

%     j   power  (j)^power

     2     5    32    % stops at 5 as a power
     3     5   243
     4     4   256
```

Technique 2: double "for-loop" with a break.

```
% [for_br.m]

% Illustrates double "for-loop" with a "break" from the
% inner "for-loop".
% We wish to find the first power for which j^power > 100,
% but we do not wish to try above values of power larger than 5.
% Note that the variable "result" is defined after the inner
% "for-loop".
```

```
store = [ ];

for  j = 2:4                  % loop 1
    for power = 1:5           % loop 2
         value = j^power;
         if value > 100
             break            % break
      end
    end   % end of the inner for loop

    result = [j, power, j^power];
    store = [store; result];

end  % end of the outer for-loop

>> for_br

     2     5    32
     3     5   243
     4     4   256
```

Technique 3 "while-loop" with one condition and a break.

```
% [while_br.m]

% Illustrates an outer "for-loop" with a "break" from an
% inner "while-loop".
% We wish to find the first power for which j^power > 100,
% but we do not wish to try values of power larger than 5.

% Note that the variable "result" is defined after the inner
% "for-loop" when we have found the correct power.

% Note the "priming" values: value = 0; power = 1
% If we did not have values, the "while" statement could not be
% evaluated.

store = [ ];

for  j = 2:4
    value = 0;     % need a starting value less than 100!
    power = 1;

    while value <= 100        % 1 condition
```

```
         value = j^power;

        if power == 5   % don't do more than 5 times
          break              % break
      else
          power = power + 1;
      end
    end   % end of the while loop

    result = [j, power, j^power];
    store = [store; result];
end  % end of the for-loop

>> while_br

     2     5    32
     3     5   243
     4     4   256
```

Technique 4: "while-loop" with no conditions (!) and a double break.

```
% [whil2_br.m]

% We have an outer "for-loop" and an inner "while-loop".
% Illustrates the use of a condition that is always satisfied,
% namely  1 == 1. The true conditions are placed inside a
% conditional statement containg two "break" commands.

% We wish to find the first power for which j^power > 100,
% but we do not wish to try above values of power larger than 5.

% Note that whereas in while_br.m , we needed two "priming" values:
% value = 0; power = 1; the first is no longer needed here.

store = [ ];

for  j = 2:4
    power = 1;

    while 1 == 1        % no conditions, since always true

         value = j^power;

        if power == 5   % don't do more than 5 times
            break             % first break
        elseif value > 100
```

```
              break              % second break
         else
          power = power + 1;
      end
    end   % end of the while loop

    result = [j, power, j^power];
    store =  [store; result];

end  % end of the for-loop

>> whil2_br

     2     5    32
     3     5   243
     4     4   256
```

CHAPTER III

SPECIAL TOPICS

This chapter gives examples of MATLAB operations as applied to three topics: linear equations, eigenvalues and matrix power models. Section III.4 shows how to use MATLAB to perform advanced linear algebra computations.

SECTION III.1 LINEAR EQUATIONS

In this section we explore different ways of solving equations and finding parametric solutions. We test how accurate MATLAB is and how long it takes MATLAB to solve large systems of equations. Subsection viii deals with linear independence and equations.

i. General systems.

Example 1. Suppose that we are given three equations:

$$\begin{aligned} 2x - 1y + 1z &= 2 \\ 1x + 1y - 1z &= 7 \\ 1x + 1y + 2z &= 4 \end{aligned}$$

We write each of these equations in vector form:

```
>> eqn1 = [2  -1   1  2];    % suppress printing using " ; "
>> eqn2 = [1   1  -1  7];
>> eqn3 = [1   1   2  4];
```

Next, we create the augmented matrix whose rows are these vectors. To enter as separate rows use " ; " :

```
>> A = [eqn1 ; eqn2 ; eqn3]      % enter as rows using " ; "

    2    -1     1     2
    1     1    -1     7
    1     1     2     4
```

We solve by reducing A to reduced row echelon form using the command rref( ) :

```
>> A1 = rref(A)

    1     0     0     3
    0     1     0     3
    0     0     1    -1
```

We pull out the last column of A1. Here we use extraction using the command A1(: , 4). Here the " 4 " represents column 4:

```
>> solution = A1( : , 4)

     3
     3
    -1
```

We now check the answer. First form the coefficient matrix COEFF by extraction from A using the command A(: , [1 : 3]). Here [1 : 3] indicates the last 3 columns of A and the semicolon (:) means let the indices range over all rows.

```
>> COEFF = A( : , [1 : 3])

     2    -1     1
     1     1    -1
     1     1     2
```

Next we multiply the coefficient matrix by the solution vector:

```
>> COEFF * soln

     2
     7
     4
```

This column vector should equal the vector of constants. We use the extraction command A(: , 4) to obtain the vector of constants.

```
>> constants = A( : , 4)       % extract 4th column, all rows

     2
     7
     4
```

Therefore we have COEFF*solution = constants, as should be the case.

We can also think of the above solution set in terms of rank using rank( ):

```
>> rank(A)

     3

>> rank(COEFF)
```

```
     3
```

Since A is the augmented matrix of the system we can write:

rank(augmented matrix) = rank(coefficient matrix)

ii. An infinite number of parametric solutions.

Example 2. We work with the following system of equations in which eqn3 was simply taken as (eqn1 - eqn2) so as to make the rows of the augmented matrix linearly dependent:

```
>> eqn1 = [3  2  3  16];
>> eqn2 = [1 -1  2   4];
>> eqn3 = [2  3  1  12];

>> A = [eqn1 ; eqn2 ; eqn3]      % enter as rows using " ; "

     3     2     3    16
     1    -1     2     4
     2     3     1    12

>> A1 = rref(A)

    1.0000         0    1.4000    4.8000
         0    1.0000   -0.6000    0.8000
         0         0         0         0
```

To obtain an expression for the parametric solution, we first form a column vector from the last column of A1. We do this via extraction:

```
>> particular = A1(: , 4)     % extract 4th column, all rows

    4.8000
    0.8000
         0
```

Next form a vector called "parametric" from column 3:

```
>> parametric = A1(: , 3)

    1.4000
   -0.6000
         0
```

If we set z = t = parameter, then the most general solution set is of the form:

```
x = 4.8 - 1.4t
y = 0.8 + 0.6t
z =   0 +   1t
```

We can write this in vector form by first taking the negative of parametric and then replacing the third element by 1:

```
>> parametric = - parametric

>> parametric(3) = 1              % redefine parametric(3)

parametric =

   -1.4000
    0.6000
        1
```

Therefore in vector form the most general solution can be written:

solution = particular + t * (parametric)

We can use the MATLAB command , to place "particular" and "parametric" as columns of a matrix:

```
>> solution= [particular , parametric]

   4.8000    -1.4000
   0.8000     0.6000
        0         1
```

As in example 1, we can think of the solution set in terms of rank. Since the last row of A1 = rref(A) is 0, the rank of A is 2 as is the rank of the coefficient matrix. We can calculate these quantities directly using rank( ):

```
>> rank(A)

    2

>> COEFF = A(: , [1 : 3])        % pull out columns 1 to 3.

    3     2     3
    1    -1     2
    2     3     1
```

```
>> rank(COEFF)

     2
```

As in example 1, we have:

rank(augmented matrix) = rank(coefficient matrix)

Since the coefficient matrix is square in this example, we can speak of its determinant. Since the rank of the coefficient matrix is less than the number of rows, the determinant of the coefficient matrix must be 0. We check this via det( ):

```
>> det(COEFF)

     0
```

iii. No solutions.

Example 3. We modify the set of equations in example 2, by taking A(3 , 4) = 13. The relationship for the original set was eqn3 = eqn1 - eqn2 . This will still hold for the coefficient matrix, but not for the constants.

```
>> A(4 , 3) = 13              % replace element (4,3) by 13

A =

     3     2     3    16
     1    -1     2     4
     2     3     1    13

>> B3 = rref(A)

B3 =

    1.0000         0    1.4000         0
         0    1.0000   -0.6000         0
         0         0         0    1.0000
```

The transformed third equation now reads " 0 = 1 ". Since this is impossible, the original set had no solutions.

As in examples 1 and 2, we can look at this problem in terms of rank:

```
>> rank(A)
```

```
    3

>> rank(COEFF)

    2
```

rank(augmented matrix) > rank(coefficient matrix).

This is what always happens in the case of a system of equations with no solutions.

iv. Cramer's rule.

Example 4. We use the same equations as in example 1. We first copy COEFF - so as to be able to reuse it - and replace the first column by the vector "constants":

```
>> B1 = COEFF;                    % make a copy of COEFF!!
>> B1(: , 1) = constants

    2    -1     1
    7     1    -1
    4     1     2
```

We now find the determinants of B1 and COEFF and then divide to obtain the value of x:

```
>> det1 = det(B1)

   27

>> det_den = det(COEFF)

    9

>> x = (det1)/(det_den)

    3
```

For large systems, Cramer's rule is very slow and for very large systems the computation of the determinants becomes impossible; see example 6.

v. Inverses.

Example 5. We use the equations of example 1. If the equations are written in the form:

C*x = constant

then the solution vector is:

x = inv(C)*constant

```
>> soln_inv = inv(COEFF)*constants

     3
     3
    -1
```

     <esc>fpl<down>.5<tab>0<down>1<left>1<enter>
vi. Using the division operator; "LU decompositions"

Instead of using inv(COEFF) for the square matrix COEFF we can use the division operator (COEFF)\constants (section II.1), which obtains the solution via Gaussian elimination applied to the augmented matrix:

Example 6. We solve the system of example 1:

```
>> soln_div = (COEFF)\constants

     3
     3
    -1
```

Another way of solving equations involving square coefficient matrices is via "LU decompositions" (also called "LU factorizations"). By means of Gaussian elimination a square matrix is "decomposed" into lower and upper triangular matrices, such that the original matrix is the product of the two triangular matrices (see section II.1.i). The solution to the original equations is then found via reduction in two steps, first finding u = LOWER\constants and then finding UPPER\u:

```
>> [LOWER , UPPER] = lu(COEFF)

LOWER =

    1.0000         0         0
    0.5000    1.0000         0
    0.5000    1.0000    1.0000
```

```
UPPER =

    2.0000   -1.0000    1.0000
         0    1.5000   -1.5000
         0         0    3.0000


>> soln_intermediate = LOWER\constants

soln_intermediate =

     2
     6
    -3

>> soln_LU = UPPER\soln_intermediate

soln_LU =

     3
     3
    -1
```

vii. Large systems of equations.

For small matrices it does not make any difference if one uses inv(A)*b or A\b or Cramer's rule. The following example explores the accuracy and execution time of the three methods.

Example 7. Large systems.

The program "sol3.m" does the following:

1.   Finds the complete solution set by both the inv(A)*b and A\b methods.

They look the same, but how close are they really? Don't forget that MATLAB has calculated these products to a high degree of accuracy and perhaps the products do not agree at the higher decimal places. To check we first look at the P1*y1 and L1*y1 and then take the sum of these absolute differences:

2. Checks the accuracy by calculating the vector of the absolute values of the differences between between COEFF*soln and the constants and then finding the sum of these absolute differences. The solution obtained by division is checked in the same way. In addition the program checks the maximum difference and the sum of the absolute values of the difference of the two solutions.

3. Finds x1, the first variable by Cramer's rule.

4. It calculates the computation time for the first two methods and estimates the total time for Cramer's rule. This is done via the time function t1 = clock, t2 = clock and the elapsed time function etime(t2 , t1) discussed in section II.3:

```
    t1 = clock  % start
      ..... execution .....
    t2 = clock
    elapsed_time = etime(t2 , t1)
```

Program 1

```
% [sol3.m]
% for n non-homogeneous equations in n unknowns.

function soln = sol3(A)

disp(' number of equations = unknowns: '), disp(rows)

COEFF = A(:, [1: rows]);    % pull out coefficient matrix
const = A(:, cols);         % pull out last column

% solution by inverse, timing, check
t1 = clock;
soln_1 = inv(COEFF)*const;
t2 = clock;
e1 = etime(t2 , t1);
disp(e1)

% check soln_1 via sum of absolute differences
diff1 = sum(abs( COEFF * soln_1 - const ));
disp( diff1 )

% solution by "divison",
t3 = clock;
soln_2 = (COEFF)\const;
t4 = clock;
e2 = etime(t4 , t3);
disp(e2)

% check soln_2 via sum of absolute differences
diff2 = sum(abs( COEFF * soln_2 - const ));
disp( diff2 )
```

```
% check the difference in the two solutions, both maximum
% and sum of absolute differences.
diff = abs( soln_2 - soln_1 );
disp(max(diff))
disp( sum( diff ))

% Find the first variable x1 by Cramer's method
% timing is done for the determinant of the coefficient
% matrix and then for the numerator for x1 + division time
% and this is used to estimate the total time by Cramer's
% method.
M1 = COEFF;     % make a copy so as not to destroy

% determinant of denominator
t5 = clock;
det_den = det(M1);
t6 = clock;
e3 = etime(t6 , t5);
disp(det_den)
disp(e3)

% find x1 = first variable
t7 = clock;
M1(:,1) = const;
det_num = det(M1);
x1 = (det_num)/(det_den);
t8 = clock;
e4 = etime(t8 , t7);
disp(e4)

% estimate total time by Cramer's method
total = e3 + (rows * e4);
% number of variables = rows
disp(total)

% check solution x1 against solution for first variable from
% soln_2
disp('first variable x1 is: '), disp(x1)
z1 = soln_2(1);
d = abs( x1 - z1);
disp(d)
```

To generate examples for use with sol3 we use the command:

```
A = rand(n, n+1) ;
```

which produces an n x (n+1) augmented matrix with coefficients picked at random between 0 and 1 (see section II.2, iv).

We start by giving MATLAB a real workout; 1000 equations in 1000 unknowns:

```
>> A = rand(1000,1001);        % do not forget the semicolon

>> sol3(A) ;                   % do not forget the semicolon

number of equations = unknowns:

        1000

inverse method

total time in seconds to read matrices and solve by inv(COEFF)*const

  629.7649  {seconds = 10 minutes and thirty seconds}

check solution by sum of absolute differences

for inverse method sum of absolute differnces =

   5.3102e-10

augmented matrix method

total time in seconds to read matrices and solve by (COEFF)\const

  345.8351   {seconds = 5 minutes and forty-six seconds}

check solution by sum of absolute differences

for "division" method sum of absolute differnces =

   8.5478e-11

maximum difference between solutions:

   4.1278e-13

sum of differences:

   4.8734e-11

Cramer`s rule
```

```
determinant of coefficient matrix is:
   Inf  { = "infinity"}

time in seconds to calculate the determinant of denominator

  281.5857   {seconds = 4 minutes and 41 seconds}


time in seconds to substitute constants in first column, find determinant of
numerator and divide:

  304.4050    {seconds = 5 minutes and 4 seconds}

estimated total time in seconds by Cramer`s method

   3.0469e+05   { seconds = 84 hours !!!}

first variable x1 is:

   NaN     {"not a number"}

difference in solution for the first variable obtained by
Cramer`s rule versus "division" is:

   NaN
```

We see that both the reduction and inverse methods are incredibly accurate. We also note that reducing to row echelon form takes only half the time of the inverse methods. The determinant, involving 1000! products of 1000 terms each, is just too much for MATLAB.

We try smaller systems with the results indicated in the following table. In each case we have non-homogeneous equations with the same number of equations as variables. The last three columns give the time in seconds. In all cases the accuracy was extremely high.

variables	inverse	reduction	Cramer
100	0.3570	0.1030	8.0939
200	4.1337	0.7817	160.7606
500	58.6381	28.4732	1.8978e+04
1000	629.7649	345.8351	fails

The table illustrates how quickly computation time increases with the number of variables. Row reduction always takes about half the time of the inverse method, but there is no precise relationship. Cramer's method works up to 500 variables, but is very slow.

In case you were wondering:

for 100 variables:

determinant of coefficient matrix is: -4.8352e+24

time in seconds to calculate the determinant of denominator: 0.1238

viii. Linear independence.

Example 6. We take three vectors:

```
>> v1 = [ 2    1    0];
>> v2 = [-3    4    5];
>> v3 = [13  -10  -15];
```

Are these linearly independent or linearly dependent? To answer this question, we form the equation:

x*v1 + y*v2 + z*v3 = 0

and solve by reduction to row echelon form.

When we write out the above vector equation in non-vector form and then form the augmented matrix the vectors v1, v2 and v3 will appear as columns. We can thus form the augmented matrix by using the (real) transpose operator ' and entering v1', v2', v3' and z in a vector separated by commas. To obtain the 3 x 1 vector of zeros on the right we use the command zeros(3 , 1).:

```
>> z = zeros(3 , 1) ;

>> M = [v1', v2', v3', z] % separate by , to form columns

    2    -3    13     0
    1     4   -10     0
    0     5   -15     0

>> M1 = rref(M)

    1     0     2     0
    0     1    -3     0
    0     0     0     0
```

As in example 2 there will be an infinite number of solutions. In particular since there is a solution other than 0, 0, 0 the vectors are not linearly independent.

Since the vectors are not linearly indepedent, they are linearly dependent, i.e. some non-zero linear combination gives 0.

We can write the solution in vector form as in example 2. Because the constants are all 0, only the parametric vector is needed

```
>> parametric = M1(: , 3);
>> parametric = -parametric
>> parametric(3) = 1;


    -2
     3
     1
```

Thus one non-zero solution is x = -2, y = 3, z = 1. This in turn means that:

-2*v1 + 3*v2 + v3 = 0.

Or
v3 = 2*v1 - 3*v2

We check our answers:

```
>> 2*v1 - 3*v2

   13   -10   15

>> v3

   13   -10   -15
```

SECTION III.2. EIGENVALUES AND VECTORS

SUMMARY

i. Finding and checking right (column) eigenvectors
ii. Finding and checking left (row) eigenvectors
iii. Spectral decompositions.

DETAILS AND EXAMPLES

i. Finding and checking right (column) eigenvectors.

We start with a 3 x 3 matrix of positive elements generated randomly using the function rand( ) discussed in section II.2.iv.

```
>> P = rand(3)

    0.2623    0.8458    0.3424
    0.0146    0.2609    0.5263
    0.7956    0.4665    0.1368
```

To show more decimal places we could type

```
>> format long
```

First we find just the eigenvalues which include a conjugate pair.

```
>> eig_val = eig(P)

   1.1640
  -0.2520 - 0.3646i
  -0.2520 + 0.3646i
```

Notice that the largest eigenvalue is real; this always happens when the elements of P are positive.

To place the eigenvalues in a row vector, we use the true transpose .' (see section II.3.i).

```
>> eig_val = (eig_val).'

   1.1640    {-0.2520 - 0.3646i}   {-0.2520 + 0.3646i}
```

We use the save command to save the variables P and eig_val and in high precision form to a file "eig1.mat" for future use (section I.4.iv):

```
>> save eig1                   % no extension
```

To reload them at a later date and find out what variables were stored in eig1 we would use load and whos

We find the magnitudes of the eigenvalues (section III.3.v):

```
>> absol_val = abs( eig_val )

    1.1640    0.4432    0.4432
```

Now we sort the eigenvalues by decreasing order using sort( ); see section II.5.

```
>> eig_val = sort( eig_val )

{-0.2520 - 0.3646i} { -0.2520 + 0.3646i}   1.1640
```

MATLAB orders by increasing order, but we want the largest first, so we rotate the vector twice (we could have used flipud( ) or fliplr( ), but which one we use depends upon whether we have a row or column vector:

```
>> eig_val = rot90( rot90( eig_val ))

   1.1640  {-0.2520 + 0.3646i}  {-0.2520 - 0.3646i}
```

Used alone eig(P) only gives the eigenvalues. To obtain the right (column) eigenvectors, as well as the eigenvalues, we use the two-valued form of eig( ). The first matrix contains the right eigenvector as columns and the second matrix contains the eigenvalues as its diagonal elements:

```
>> [EIG_VEC, EIG_VAL1] = eig(P)

EIG_VEC =

   0.6281      -0.0521 + 0.4994i     -0.0521 - 0.4994i
   0.3993      -0.4524 - 0.3198i     -0.4524 + 0.3198i
   0.6678       0.6639 - 0.0156i      0.6639 + 0.0156i
```

Note that columns 2 and 3 are complex. These correspond to the complex eigenvalues.

```
EIG_VAL1 =

   1.1640                0                 0
        0         -0.2520 + 0.3646i        0
        0                0           -0.2520 - 0.3646i
```

SECTION III.2. EIGENVALUES AND VECTORS

We can once again obtain the eigenvalues as a column vector and then, as a row vector

```
>> eigen1 = diag(EIG_VAL1);     % section II.3.ii

>> eigen1= (eigen1).'

   1.1640    { -0.2520 + 0.3646i}  {-0.2520 - 0.3646i}
```

In most linear algebra texts, the term eigenvector refers to a column vector y such that

P*y = L*y,

where L is the corresponding eigenvalue. This is the type of eigenvector that MATLAB calculates, and it stores these eigenvectors as the columns of the matrix EIG_VEC. We pull out the first column:

```
>> y1 = EIG_VEC(: , 1)     % extraction; section II.4

    0.6281
    0.3993
    0.6678

>> L1 = eigen1(1)         % eigenvalue corrresponding to y1.

    1.1640
```

We now check:

```
>> P*y1

    0.7312
    0.4648
    0.7774

>> L1*y1

    0.7312
    0.4648
    0.7774
```

They look the same, but how close are they really? Do not forget that MATLAB has calculated these products to a high degree of accuracy and perhaps the products do not agree at the higher decimal places. To check we first look at the vector of the absolute values of the differences between P1*y1 and L1*y1 and then take the sum of these absolute differences:

```
>> sum( abs( P*y1 - L1*y1 ) )

  8.3267e-016
```

Thus the difference between the calculated values P1*y1 and L1*y1 is small indeed.

MATLAB gives us one set of eigenvectors, but if we multiply an eigenvector by a constant, we will still have an eigenvector. In certain cases we can normalize the eigenvectors so as to meet certain conditions. In this example y1 has strictly positive elements (this is because P is strictly positive) and so we can normalize y1 so that the sum of the elements is 1. We do this by finding sum(y1) and then dividing y1 by this sum:

```
>> y1 = y1/sum(y1)      % keep the same variable name: y1

    0.3705
    0.2355
    0.3939
```

We check the answer:

```
>> sum(y1)

     1
```

ii. Finding and checking left (row) eigenvectors.

As stated above in most linear algebra textbooks the term "eigenvector" refers to right (column) eigenvectors. But in certain applications (see part iii below and section III.3) we are interested in left (row) eigenvectors x which satisfy:

```
 x*P = L*x
```

It is easy to check that the left eigenvectors of P, are just the transposes of the right eigenvectors of the transpose of P (and not simply the transposes of the right eigenvectors) i.e. if y satisfies:

```
 P'*y = L*y
```

then if we put x =y', then x is a left eigenvalue as above.

Thus what we do is use MATLAB to find the matrix of right eigenvectors of the transpose of P and then take the transpose of this matrix. The eigenvalues will be the same for both the left and right eigenvectors.

When working with transposes of eigenvectors and eigenvalues we must use the true transpose operator .' and not the conjugate transpose operator ' . As shown in the examples of section II.3.i, if we start with a matrix M with complex elements then the elements of M' will be the transposes of the elements of M. However the elements of M.' will be the same as those of M.

```
>> P1 = (P).'          % take the true transpose of P

>> [LEFT_EIG_VEC, EIG_VAL2] =  eig(P1)    % gives as columns

LEFT_EIG_VEC =

  -0.4716    -0.3080 + 0.5506i    -0.3080 - 0.5506i
  -0.7110     0.5917 - 0.0358i     0.5917 + 0.0358i
  -0.5215    -0.0640 - 0.4965i    -0.0640 + 0.4965i

EIG_VAL2 =

   1.1640                  0                  0
        0           -0.2520 + 0.3646i         0
        0                  0           -0.2520 - 0.3646i
```

The eigenvalues are exactly the same as in EIG_VAL which we calculated when we computed the right eigenvectors in subsection i.

To obtain the left (row) eigenvectors of P, we now take the true transpose of "LEFT_EIG_VEC":

```
>> LEFT_EIG_VEC = (LEFT_EIG_VEC).'

LEFT_EIG_VEC =

  -0.4716             -0.7110              -0.5215
  -0.3080 + 0.5506i    0.5917 - 0.0358i    -0.0640 - 0.4965i
  -0.3080 - 0.5506i    0.5917 + 0.0358i    -0.0640 + 0.4965i
```

It is the rows of this matrix which are the left eigenvectors of P.

As in subsection i, we put the eigenvalues in a row vector. We have to redo the calculation, because we want to make sure that the eigenvalues and eigenvectors match up. Sometimes MATLAB places the eigenvalues in different positions when working with a matrix and its transpose:

```
>> eigen2 = diag(EIG_VAL2);         % section II.3.ii
>> eigen2= (eigen2).'

   1.1640    { -0.2520 + 0.3646i}  {-0.2520 - 0.3646i}
```

We now check the third eigenvalue and the corresponding eigenvector to see if they satisfy the equality

x3*P = L3*x3

To do this we pull out the third element of "eigen2" and the third row of LEFT_EIG_VEC

```
>> L3 = eigen(3)              % pull out the third eigenvalue

  -0.2520 - 0.3646i

>> x3 = LEFT_EIG_VEC(3 , :)    % 3rd row of matrix; sect. II.4

{-0.3080 - 0.5506i}  {0.5917 + 0.0358i}  {-0.0640 + 0.4965i}
```

As was the case for the right eigenvectors in subsection i, we look at the vector of absolute differences and then take the sum:

```
>> sum( abs( x3*P - L3*x3 ) )

  8.9227e-016
```

Once again the difference is negligible.

iii. Spectral decompositions.

"Nice" matrices, e.g. matrices with distinct eigenvalues (this includes all strictly positive matrices), have is what is called a "spectral decomposition":

P = L1*Z1 + L2*Z2 + L3*Z3

The Zi have the property that Zi*Zj = 0 if i and j are distinct and Zi*Zi = Zi. Further Z1 + Z2 + Z3 = Identity.

Because of the property of the Zi, we also have that for any power n:

Pn = (L1)n * Z1 + (L2)n * Z2 + (L3)n * Z3.

Computing the spectral decomposition is done via the program "spectral.m" which displays the ease with which many matrix calculations can be done automatically via MATLAB. The program first finds the matrix X of left eigenvectors. The eigenvalues are sorted in decreasing order and the rows of X are then rearranged accordingly. The matrix of right eigenvectors is computed as inv(X). The ith spectral component is simply yi*xi where yi and xi are the right and left eigenvectors corresponding to the eigenvalue Li.

The components Zi are stored in a matrix STORAGE which can either be displayed, or the submatrices can be pulled out using the function fetch(k, STORAGE) discussed in section II.4. The function spectral( ) is really a two valued function; aside from STORAGE the eigenvalues are also stored in eig_val.

Program 1

```
% [SPECTRAL.M]

function [eig_val, STORAGE] = spectral(A)

[rows, cols] = size(A);

% find X = matrix of left row eigenvectors;
A1 = (A).'                              % true transpose
[VEC , VAL] = eig(A1);
X = (VEC).';                            % want row vectors

% sort eigenvalues by decreasing magnitude and rearrange X.
eig_val = diag(VAL);                    % column vector
[eig_val, indices] = sort(eig_val);     % sort, sect. II.5
eig_val = rot90(rot90( eig_val ));      % decreasing order

indices = rot90(rot90( indices ));      % decreasing order
X = X(indices , :);                     % rearrange rows of X

eig_val = (eig_val).';                  % eigenvalues as row vector

% Obtain right eigenvectors; Y is obtained from X as inverse
% so no problem with order.
Y = inv(X);

% now find components

for k = 1:cols
    y = Y(: , k);           % column k of Y = right eigenvector
    x = X(k , :);           % row k of X = left eigenvector
    val = eig_val(k);         % kth eigenvalue
    Z = y*x;                % kth idempotent matrix
```

```
% store this component matrix in columns (k-1)*cols + 1 to
% k*cols of "STORAGE"
% for complicated ranges do in steps;thus avoiding charabia
     lo = (k-1)*cols + 1;
     up = k*cols;
     range = [lo:up];
     temp_STORAGE(: , range) = Z;

end                              % end of "for- loop"

STORAGE = temp_STORAGE;
```

As an example we apply the function "spectral" to the matrix P of subsection i:

```
>> [vec, STORE] = spectral(P)
```

The eigenvectors are in "vec" and the components are in "STORE".

Now we put the components back together again:

```
>> L1 = vec(1) , L2 = vec(2) , L3 = vec(3);

L1 = 1.1640

>> Z1 = fetch(1,STORE), Z2 = fetch(2,STORE), Z3 = fetch(3,STORE);

Z1 =
   0.3191 + 0.0000i   0.4810 + 0.0000i   0.3528 + 0.0000i
   0.2028 + 0.0000i   0.3058 + 0.0000i   0.2243 + 0.0000i
   0.3393 - 0.0000i   0.5115 - 0.0000i   0.3751 - 0.0000i

>> P1 = L1*Z1 + L2*Z2 +L3*Z3

   0.2623             0.8458 + 0.0000i   0.3424
   0.0146 + 0.0000i   0.2609 + 0.0000i   0.5263 + 0.0000i
   0.7956 - 0.0000i   0.4665 - 0.0000i   0.1368 - 0.0000i
```

According to the definition of the spectral composition given above P1 should be the same as P. We check this out using the sum of the absolute values of the differences between P1 and P.

```
>> P

    0.2623    0.8458    0.3424
    0.0146    0.2609    0.5263
    0.7956    0.4665    0.1368
```

SECTION III.2. EIGENVALUES AND VECTORS

```
>> sum(abs(P1-P))

  1.0e-015 * { 0.5295    0.9169    0.6884}
```

One of the uses of the spectral decomposition is to analyse the long term behaviour of powers of a matrix. If L1 is the dominant eigenvalue (which will be real and positive for positive matrices with distinct eigenvalues) and Z1 is the corresponding, spectral component then for "large" n we have approximately:

Pn = (L1)n * Z1

This is illustrated with P, L1 and Z1 as above:

```
>> P^10

    1.4571    2.1971    1.6114
    0.9264    1.3964    1.0244
    1.5496    2.3360    1.7130

>> (L1)^10 * Z1

   1.4573 + 0.0000i   2.1970 + 0.0000i   1.6113 + 0.0000i
   0.9264 + 0.0000i   1.3966 + 0.0000i   1.0243 + 0.0000i
   1.5494 - 0.0000i   2.3359 - 0.0000i   1.7132 - 0.0000i
```

So even for the 10th power the approximation is fairly good.

The relationship between powers of the matrix and powers of the dominant eigenvalue becomes even more striking when we do a plot using semi-log axes for then (L1)^n appears as a staight line. An example of this is given in section iii.3.example 2.

SECTION III.3. MATRIX POWER MODELS

Many matrix models can be written in one of two forms, depending upon whether we are working with column or row vectors:

i. nk-1 * A = nk (row vector form)

or equivalently

n0 * Ak = nk where n0 is an initial vector

ii. nk = A * nk-1 (column vector form)
or equivalently

nk = Ak * n0 where n0 is an initial vector

Calculating the values, and plotting the graphs, for such models is particularly straightforward.

Example 1. The function stochast is for models of type i. It was designed especially to illustrate what happens to "stochastic matrices", which are square matrices whose rows all add to 1. (These are sometimes called Markov chains (1906); algebra books tend to define them with the column sums being 1, but the row definition is more natural and is the one adopted in virtually all books on the topic). The function stochast stores the intial vector n0 in the first column of a matrix M, and then the other vectors nk as columns of M. To plot M we use the function shift_1 discussed in section II.6, example 5. At the end the matrix M is kept as a matrix STORE which can either be displayed or, if too big, the column vectors can be pulled out of STORE using the function fetch discussed in section II.4, program 2.

To create stochastic matrices easily, we use the following function normaliz:

Program 1

```
% [normaliz.m]

function y = normaliz(A)

b1 = sum(A.');                  % sum ( ) is for columns use transpose
b = (b1).'                      % use the true transpose .'

[rows, cols] = size(A);         % section II.3.ii

c = ones(1 , cols);

B = b*c;
% B has same dimension as A; elements of row i all = sum of row i
% of A

y = (A)./B;             % element by element division
```

The power calculations are done by the next function stochast.

Program 2

```
% [stochast.m]
% stochast.m works with left row vectors (do not have to add to 1)
% P is an mxm matrix, n_0 is a mx1 row vector; t = number of
% iterations
% The program finds the population vectors for t time periods
% and stores them in a matrix M which is returned as an
% m x (t + 1)  matrix STORE (which can be suppressed when calling
% via `;'. The graph of time vs the population at each time is
% then plotted. The value at time = 0 is the initial vector
% Total population is also computed at time = 0 and time = t.
% It is assumed that all numbers are real, so use ' for transpose.

function STORE = stochast(P, n_0, t)

% do computations and store
% Since the index 0 is not permitted, we put the transpose
% of the initial vector in column 1.
M(:,1) = (n_0)';

for n = 1:t
      u = n_0 *P^n;            % u = population row vector at time n
      M(:,n + 1) = u';         % store u' as column n + 1 of matrix M
end

STORE = M;

% plot rows versus time;
index = [0 : t];               % domain vector; sect. II.6

% matlab will not accept 0 as index in matrices, so use
% shift_1(k,M)  = column (k +1) of M. Section II.6, example 5
plot(index , shift_1(index, M) )

% label the graph
title(' stochastic matrix, "population" in different classes
 versus time ')
```

To illustrate these programs we use the same matrix P as in section III.1.i. This was saved in a file "eig1.mat".

```
>> load eig1                    % no extension; sect. I.4.v
```

Now obtain a stochastic matrix using the function normaliz of program 1 above. To check, we show more than the usual four decimal places:

```
>> format long                  % section I.5.ii

>> P2 = normaliz(P)

   0.18085444136721   0.58309924493029   0.23604631370250
   0.01816746059301   0.32539236171264   0.65644017769435
   0.56872601196630   0.33346231536217   0.09781167267153
```

We check the row sums

```
>> s1 = sum(P2')

1.00000000000000   1.00000000000000   1.00000000000000

>> format short                 % back to the short form
```

We now pick an initial row vector at random. Just to make everything even more random, we change the "seed" for the random number generator (section II.2.iv).

```
>> rand('seed', 78 )            % 78 is an arbitrary number

>> initial = rand( 1, 3);
```

Normalize "initial" so that the sum of the elements is one.

```
>> initial = initial/(sum(initial))

    0.0055   0.7310   0.2635

>> sum(initial)

   1.000

>> save eig2                    % save old and new variables to "eig2.mat"
```

The initial vector n0 is taken to be equal to "initial". We are now ready to find and plot the vectors n1, n2, ... n7, using the function stochast.

```
>> STORE = stochast(P2, initial, 7)
```

```
 the "initial" vector:

    0.0055    0.7310    0.2635

 the total initial population was:  1.0000

 number of iterations was:  7

last value of population vector:  { 0.2535    0.3934    0.3531 }

 the total population at the end is:   1.0000
```

The vectors nk, for k = 0, ..., 7 have been put as column vectors in STORE.
Because the intial vector is stored in column 1, nk has been stored in column k+1.
The first and last columns have already been printed out above by the program.

```
STORE =

Columns 1 through 7

0.0055    0.1641    0.3240    0.2384    0.2498    0.2573  0.2527
0.7310    0.3289    0.3718    0.4113    0.3897    0.3927  0.3945
0.2635    0.5069    0.3043    0.3503    0.3605    0.3500  0.3527

Column 8

0.2535
0.3934
0.3531
```

From the last columns we see that the vectors nk appear to converge to approximately: [0.2535 0.3934 0.3531].

To see what the limit is, we continue our calculations by taking a high power of P2:

```
>> POWER_40 = P2^40

   0.2536   0.3936   0.3528
   0.2536   0.3936   0.3528
   0.2536   0.3936   0.3528
```

We pull out the first row:

```
>> a = POWER_40(1 , :)
```

```
   0.2536  0.3936   0.3528
```

which is what nk would be for large values of k.

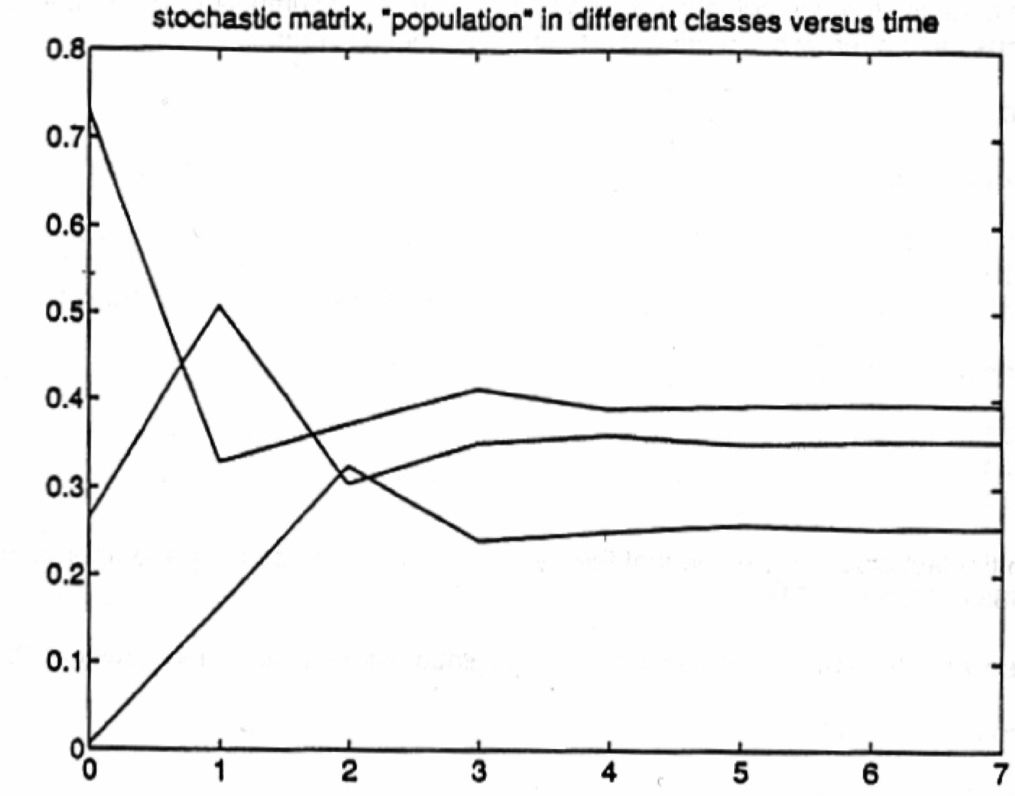
The vector "a" turns out to be (approximately, because we only took the 40th power) a left eigenvector of P2 corresponding to the eigenvalue 1 (which is always an eigenvalue for a stochastic matrix):

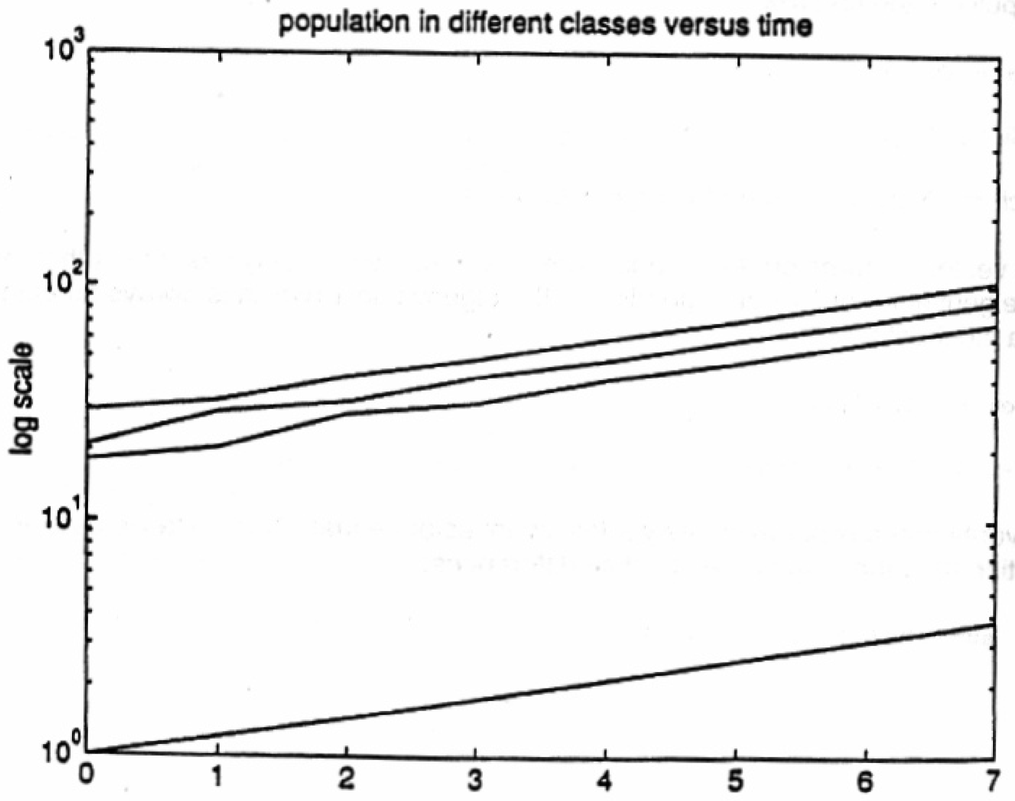
```
>> ei = eig(P2).'

  {-0.1980 + 0.3184i}  {-0.1980 - 0.3184i}   1.0000
```

To verify that a is approximately a left eigenvector i.e that a*P2 = 1*a we use, as in section III.2. the sum pf the absolute differences:

```
>>  sum(abs( a*P2 - 1*a ))

         1.1102e-016
```

The plot of the graph is shown in the figure on the next page


stochastic matrix, "population" in different classes versus time
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
1
2
3
4
5
6
7


population in different classes versus time
log scale
10^3
10^2
10^1
10^0
0
1
2
3
4
5
6
7

SECTION III.3. MATRIX POWER MODELS

Example 2. Human population population models; Leslie matrices

The following matrix is the "Leslie matrix" for the female population of the United States in 1965 (source: N. Keyfitz, Introduction to the Mathematicas of Population, 1968, pp. 42, 56; for other matrix models see also H. Caswell, Matrix Population Models, 1989). The rows and columns correspond to the female population in age groups 0-15, 15-30, 30-45 respectively. The off diagonal numbers represent the survival rate from 0-15 to 15-30 and 15-30 to 30-45. The numbers in row 1 represent the number of females born to females in each of the three groups.

```
    0.4271    0.8497    0.1273
    0.9924         0         0
         0    0.9826         0
```

The female population of the three groups in 1965, displayed as a row vector with the factor 10^6 omitted, is:

```
   29.4130   20.8860   18.0400
```

This model corresponds to the second type above, with an increase of the power by one being the same as fifteen real years.

The function popul_r was designed for models of type ii. The main difference from that of "stochast.m" in example 1 - aside from the change from rows to columns - is that the plot is done on semi-log axes (log10). Because of the importance of the dominant eigenvalue L1, as discussed in the part iii of section III,3, the population of each class will have log(L1) as an asymptote on these axes. The graph shows both the line log(L1) * k and the population of each class. The function also computes the population at each time and stores the vectors in a matrix STORE for printing out or later retrieval.

Program 3

```
% [popul_r.m]
% popul_r.m works with right column vectors (conver1.m is rows)
% P is an mxm matrix, n_0 is a mx1 column vector; t = number of
% iterations.
% The program finds the population vectors for t time periods
% and stores them in a matrix M which is returned as an
% m x (t + 1)  matrix STORE (which can be suppressed when calling
% via `;'. The graph of time vs the population at each time is
% then plotted. The value at time = 0 is the initial vector
% Total population is also computed at time = 0 and time = t.
% The program also computes the dominant eigenvalue L1 and plots
% the straight line log10(L1)*time
```

SECTION III.3. MATRIX POWER MODELS

```
function STORE = popul_r(P,n_0, t)

M(:,1) = n_0;     % initial vector in column 1;

for n = 1:t
       u = P^n * n_0 ;       % u = column vector of `population' at n
       M(:,n + 1) = u ;      % store u as column n + 1 of matrix M
end

STORE = M;

% now find the largest eigenvalue and set up plot of L1*time
EIG_VAL = eig(P)
L1 = max(max(EIG_VAL));   % first max on columns, then max of max!

% plot rows versus time;
index = [0 : t];
Z = (L1).^(index);   %  line with slope log10(L1) on log scale.

% matlab will not accept 0 as index in matrices, so use
% shift_1(k , M)  = column (k +1) of M

% make y axis log10 scale use 'semilogy'
% to plot the line log10(L1)*time repeat use 'index, Z)
semilogy(index, shift_1(index, M), index, Z )

% label the graph
title('population in different classes versus time')

% label the y axis
ylabel('log scale')
```

The data for the United States in 1965 was typed into a program called "usa_1965.m"; see section I.3.i. Note that the situation here is different from that of example 1. In that example we had created a matrix at random and then saved the matrix in a *.mat file which we later called up using load. But here "usa_1965.m" is a simple ASCII file that we created ourselves. So to call it up we simply type the name; see sections I.3.i and I.4.iv,v

```
>> usa_1965              % call up data from file usa_1965.m
```

We call up popul_r and store results in STORE

```
>> STORE = popul_r(LESLIE, initial, 7)

 the total initial population was:   68.3390
```

```
last value of population vector - displayed as a row vector is;

  105.1696   86.4479   70.0180

 the total population at the end is:  261.6356

the dominant eigenvalue is   1.2093

STORE =

  Columns 1 through 7  (column 1 = initial population)

29.4130   32.6056   41.3407   48.8022   59.7511   71.8036 87.1100
20.8860   29.1895   32.3578   41.0265   48.4313   59.2970 71.2579
18.0400   20.5226   28.6816   31.7948   40.3126   47.5886 58.2652

  Column 8  ( = final population vector)

  105.1696
   86.4479
   70.0180
```

Example 3. In example 2, we found the population vectors for each of three age classes at various time points. There the indices were the time points, and at each time point we had three values corresponding to the population in each of three classes.

In the present example, we do what might be called the "transpose" of problem 2. We start with the female population in Canada in 1981. whose age was less than 50 years. We divide this population into ten age classes, each class being five years in length. By proceeding as above, we then find the population in 1991, 2006 and the asymptotic population (these correspond to n = 2, n = 5 and the right eigenvector corresponding to the maximum eigenvalue.)

We normalize the population for each of the four time points, so as to find the relative frequency of females in each of the ten age classes. For each year our value vector will be the vector of relative frequencies for that year. We then plot each of the four value vectors against the domain vector [1 : 10] All the work is done with the program "canada81.m".

The graph appears at the end of this section.

Program 4

```
% [canada81.m]
```

SECTION III.3. MATRIX POWER MODELS

```
% analysis of fraction of female population in first 10 age classes

% based on the 1981 Canadian census (volume N92-901 table 1),
% values have been rounded off. Population = value *10^5

% The age and year intervals are 5 years.  There are 10 age
% classes (only one child was born to a woman over 50 in 1981):

% only real values, including the dominant eigenvalue, so can use '
% for long data sets use ellipses and continue on the next line

n0 = [8.69, 8.65, 9.36, 11.33, 11.69, 10.93, 10.17, 8.07, ...
6.63, 6.21];

n0 = (n0)';           % need as column vector for powers

% first row vector of 10 terms; use ellipses (section I.4.ii)

fertility = [0, .003, .031, .1448, .2639, .2292, .1024, ...
 .0267, .0039, .0002];

% off diagonal terms vector of 9 elements
survival = [.9982, .9999, .9983, .9977, .9975, .9971, ...
.9959, .9936, .9898];

% form Leslie matrix
CAN_81 = zeros(10 , 10);       % determine shape

CAN_81(1 , : ) = fertility;     % first row

% off diagonal = survival
for n = 1 : 9;                % start in row 2
   CAN_81(n+1 , n) = survival(n);
end

% population at t = 2 (10 years) and 5 (25 years)
n2 = (CAN_81)^2 * (n0);
n5 = (CAN_81)^5 * (n0);

% obtain as row vectors by taking transposes
n0 = (n0)';
n2 = (n2)';
n5 = (n5)';

% stable initial vector = right eigenvector corresponding to
% maximum eigenvalue
```

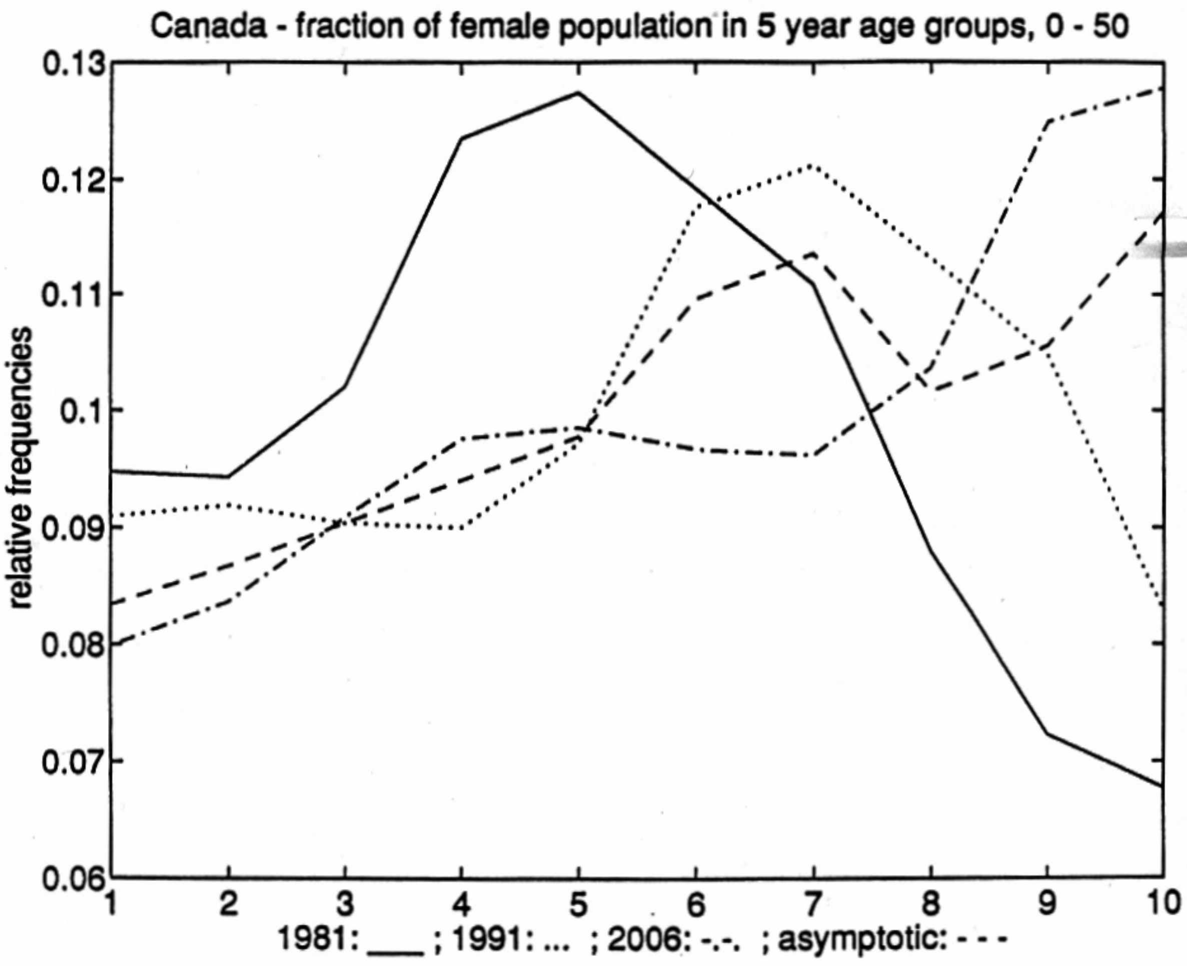
SECTION III.3. MATRIX POWER MODELS

```
[VEC, VAL] = eig(CAN_81);

% sort eigenvalues by decreasing magnitude and rearrange CAN_81
eig_val = diag(VAL);
[eig_val, indices] = sort(eig_val);   % sorts by increasing order
eig_val = rot90(rot90(eig_val));      % section III.2.ii
indices = rot90(rot90(indices));
% rearrange the rows of VEC to match eigenvalues
VEC = VEC(indices , :);
stable = VEC(: , 1);                  % pull out first column
stable = (stable)';

% relative frequencies obtained as row vectors
disp(' relative frequencies ')
y0 = (n0)/sum(n0)
y2 = (n2)/sum(n2)
y5 = (n5)/sum(n5)
y_stable = (stable)/sum(stable);

in = [1 : 10];
plot(in, y0, '-' ,  in,y2, ':', in, y5, '-.', in , y_stable, '--')
hold

title('Canada - fraction of female population in 5 year age groups, 0 -50')
xlabel('1981: ___ ; 1991: ...  ; 2006: -.-.  ; asymptotic: - - - ')
ylabel('relative frequencies')
```

The population (multiply by 100,000) and relative frequencies for 2005 are:

```
n2 =

  Columns 1 through 7

 8.7219    8.8149    8.6735    8.6344    9.3226   11.2757 11.6270

  Columns 8 through 10

   10.8536   10.0635    7.9366

y2 =

  Columns 1 through 7

0.0909    0.0919    0.0904    0.0900    0.0972    0.1175   0.1212
```

```
Columns 8 through 10

  0.1131    0.1049    0.0827
```



SECTION II.7 ADVANCED LINEAR ALGEBRA

"I can't tell you what the Matrix is, no one can." -Lawrence Fishburne (Morpheus) to Keanu Reeves (Neo) in *The Matrix* (contributed by Greg Doyle)

SUMMARY

i. Norms and orthogonalization

norm: norm(v)
normalizing the columns of a matrix: norz_col(A)
orthonormal vectors: dot(v1, v2), norm(v1) = norm(v2) = 1
orthogonalization (Gram-Schmidt): orth(A)
orthogonal (unitary) matrices

ii. Null spaces

null(A)

iii. Orthogonal transformations

Orthogonal transformations
QR-factorization: [Q,R] = qr(A)

iv. Eigenvalues and eigenvectors

Eigenvalues and eigenvectors: [EIG_VEC, EIG_VAL] = eig(A)
Characteristic polynomial: poly(A)
Cayley-Hamilton theorem
Linear independence: Section 3.1.viii
Spectral decomposition: Section 3.2

v. The similarity problem, diagonalization

"Nice" square matrices:

P = EIG_VEC; D = EIG_VAL

A = P*D*inv(P); D =inv(P)*A*P

"Non-nice" square matrices: Jordan normal (cannonical) form

[V, J] = jordan(A); A = V*J*inv(V) ; J= inv(V)*A*V

Non-square matrices: Singular value decomposition

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

SECTION II.7 ADVANCED LINEAR ALGEBRA

DETAILS AND EXAMPLES

[N.B. Nering = Nering, E. Linear Algebra and Matrix Theory, London: Wiley, 1963.

i. Norms and orthogonalization

The norm of a vector. The norm of a vector is given by norm(v) = R(sum((abs( ))^2). Here R( ) is the modern symbol for square root as defined in Section II.3.iii. The definition also holds for complex numbers. Matlab computes the norm as norm(v) = R(v * v') for row vectors; where ' is the conjugate transpose (Section II.3.i).

If divide each element of a vector by the norm then the new vector has norm 1. We say that the vector has been normalized.

Example 1

```
>> c = [ 1 2 3]

>> d =  norm(c)

      3.7417

>> e = c/d

      0.2673    0.5345    0.8018

>> norm(e)

     1
```

Often we need to normalize a set of vectors. Instead of working with each vector individually we can place them as the columns of a matrix and then use the following function:

```
% [norz_col.m]
% normalize the columns of a matrix (square or not)
% pull out the column vectors; find the norm, divide and replace!

 function   B = norz_col(A)

 [m,n] = size(A);
 B1 = zeros(m,n);      % storage matrix
 for k = 1:m
```

```
      v =A(:,k);    % kth column
      n = norm(v);
      u = v/n;         % normalize
      B(:,k) = u;      % replace
  end
```

Example 2. Let

```
A1 =  [1     4             % [Nering, 224]
       4    -5]


>> B1 = norz_col(A1)

    0.2425    0.6247
    0.9701   -0.7809

%  we check
>> v3 = B1(:,1)     % pull out column 1, Section II.4,ii
>> v4 = B1(:,2)     % pull out column 2

v3 =    0.2425
        0.9701

v4  =   0.6247
       -0.7809

>>  norm(v3),  norm(v4)

             1   1
```

[Note: Do not confuse norz_col( ) with normalize( )] of Section III.3 where the sum of the elements of each row was made equal to 1].

Orthonormal vectors. Vectors v1 and v2 are called orthonormal if:

a. dot(v1, v2) = 0   % dot is the dot (inner) product
b. norm(v1) = norm(v2) = 1

For example in R2 the vectors v1 = (1,0) and v2 = (0,1) are orthonormal. In the same way we can talk about orthonormal families where any two vectors are orthonormal.

[note: The function dot( ) is strangely missing from MATLAB; it was defined as a function in Section II.3.i]

SECTION II.7 ADVANCED LINEAR ALGEBRA

Gram-Schmidt orthogonalization: If we have a set of linearly independent vectors (Section III.1.viii) which spans some space, i.e., whose linear combinations generate the same vectors as the original vectors, then this spanning set is called the basis for the space. For example (see under null spaces below) we might want a basis for the set of solutions to a system of homogeneous linear equations. Knowing the basis means that we essentially know all the solutions.

Often we wish to replace a basis, whose members need only be linearly independent, by a linearly independent set which is also orthonormal. The usual method for finding an orthonormal basis for a set of linearly independent vectors is called the Gram-Schmidt orthogonalization process. This can be done in MATLAB by placing the linearly independent vectors as the columns of a matrix and using the command orth( ).

To apply the Gram-Schmidt orthogonalization process to A1 above we first check that the columns are linearly independent:

```
>> rank(A1)

      2
```

Since the rank equals the number of columns, the columns are linearly independent.

We find the dot (inner) product between v1 and v2

```
>>  dot(v1,v2)    % Section II.3.i

      -0.6060
```

So v1 and v2 are linearly independent, but not orthogonal.

Now we use the function orth( ) to find an orthonormal basis for the space spanned by v1, v2.

Example 3

```
>> GS1 = orth(A1)

    0.6247    0.7809
   -0.7809    0.6247
```

Suppose that we wanted to check that the columns of GS1 are indeed orthonormal. There are two ways of doing this. In the first we actually pull out the columns of GS1 and then check:

```
>> v3 = GS1(: , 1)    % pull out the first column
>> v4 = GS1(: , 2)    % pull out the secondcolumn

>> norm(v3), norm(v4)

        1     1

% check that orthogonal
>> dot(v3,v4)

   1.6653e-016     %  = 0
```

In the second method we think of GS1 as consisting of two column vectors v3, v4 and then consider the transpose:

```
              GS1     = [v1, v2]
              (GS1)'  = [v1'
                         v2']
```

If v1 and v2 are orthonormal, then we should have:

```
                  GS1 * (GS1)'  = I
```

```
>> GS1 * (GS1)'

    1   0
    0   1
```

A square real matrix M with the property that M*M' = I is called an orthogonal matrix. Since the product is the identity we have that the transpose of of M is the same as the inverse of M. Since the entries are real, the transpose ' operator coincides with the true transpose (see Section II.3.i).

For square matrices with complex entries, matrices having the property that M*M' = I are called unitary matrices. In this case the operator ' is the conjugate transpose (see Section II.3.i).

In the above example the matrix had rank 2 and so we had that the columns were linearly independent. Suppose we take A1 and add a column made up of numbers picked at random. Then we have a 2 x 3 matrix and the rank will still be 2.

Example 4

```
>> w = rand(1,2);   % pick a vector at random; Section II.2.iv
>> w = w';          % transpose
>> A2 = [A1,w]      % concatenate w with A1
```

```
A2 =

   1    4    .6789
   4   -5    .6793

>> rank(A2)

     2
```

Since the rank of A2 is 2, any orthonomal base for the space generated by the columns of A2 will consist of two vectors. As before we can find an orthonormal base for the space using orth( ):

```
>> orth(A2)

   0.6247    0.7809
  -0.7809    0.6247
```

Note that this is the same set of two vectors that we obtained from the command orth(A1). This is because the last column of A2 is linearly dependent on the first two columns (to find the relationship of the last column to the first two, just treat A2 as the augumented matrix corresponding to A1 and solve; see Section III.1.viii).

ii. Null spaces

Example 5

Consider the two homogeneous equations in 5 unknowns given by the matrix equation.

```
A3*s = 0;    s = [v w x y z]'
```

where:

```
>> A3 =     [ 2     0     5     0     1
             1     0     2     1     0
             1     1     0     0     0 ]
```

We reduce A3 using rref( ); see Section III.1.

```
>> R3 = rref(A3)

       1     0     0      5    -2
       0     1     0    -5      2
       0     0     1    -2     1
```

From R3 we see that the rank of A3 is 3. Since there are 5 unknowns, the first 3 variables v, w, x can be expressed in terms of the 5-3 = 2 other variables y and z (see Section III.1.ii, example 2). The set of all solutions of the equation A3*s = 0 is called the null space of A3.

What we want is a basis, and in particuar an orthonormal basis, for the null space. Because 3 of the variables can be expressed in terms of the other 2 variables, any basis for the solution space will consist of 2 linearly independent vectors. To construct a basis we simply take [y z] = [1 0] and [y z] = [1 0]. These latter two vectors are linearly independent in R2 and since the other variables depend on y and z we are then assured that we have linearly independent vectors in R5.

Because y and z are 0 and 1, the values for v, w and x are just the negatives of the elements of the last two rows of R3, i.e., (-5, 5, 2) for y = 1, z = 0 and (2, -2, -1) for y = 0, z = 1 (see Section III.1.ii, example 2). We use these values to create a basis {s1, s2} for the null space.

```
>> s1 = [-5  5  2  1  0]' ;  % transpose to make column vector
>> s2 = [ 2 -2 -1  0  1]' ;
```

To find an orthonormal basis we proceed as in the previous section and put s1 and s2 into a matrix and then use orth( ), as in part i above:

```
>> S= [s1, s2]

  -5     2
   5    -2
   2    -1
   1     0
   0     1

% we check that we did not make a mistake and that S is indeed
% a solution of the equation A3*s = 0
>> A3* S

      0 0
      0 0
      0 0

%  find an orthonormal base usin orth( )
>> BASE1 = orth(S)

   0.6742   -0.0000
  -0.6742   -0.0000
  -0.2697    0.1826
  -0.1348   -0.3651
        0   -0.9129
```

```
% check that orthonormal. Note the order with 2 x 5 (BASE1)' first
>> (BASE1)' * BASE1

   1.0000    0.0000
   0.0000    1.0000
```

The columns of BASE1 form an orthonormal basis for the null space of A3.

Instead of preceeding as we just did using orth( ) we can use the MATLAB command null( ) to obtain an orthonormal basis for the null space.

```
 >>BASE2 = null(A3)

  -0.6594    0.1403
   0.6594   -0.1403
   0.2258   -0.2347
   0.2079    0.3291
   0.1900    0.8929

 % check
>> (BASE2)'*BASE2
   1.0000    0.0000
   0.0000    1.0000
```

Note that the columns of BASE2 are not the same as the columns of BASE1. Thus we have obtained two distinct orthonormal bases for the null space.

iii. Orthogonal transformations

Example 6

We start with two arbitrary matrices and then find orthnormal bases

```
A4 =

     4     5    -1     2
     3    -7     4     6
     8     0    -3     5
     5     4     1     2

rank(A4)
```

```
        4

BASE_A4 = null(A4)

    0.3746   -0.4650   -0.1776   -0.7822
    0.2810    0.8049   -0.4653   -0.2383
    0.7493    0.1431    0.6334    0.1300
    0.4683   -0.3398   -0.5922    0.5608


B4 =

    11    -5     4     2
    13    -2     0     4
     3     3     1    -2
     5     6     1     4

rank(B4)

        4

BASE_B4 = null(B4)

    0.6111   -0.4317    0.1043    0.6552
    0.7222   -0.0380   -0.1384   -0.6766
    0.1667    0.4093    0.8966   -0.0285
    0.2778    0.8029   -0.4075    0.3347
```

Since both A4 and B4 have rank 4, both BASE_A4 and BASE_B4 are orthonormal bases for R4. We want to know if there is a relationship between these bases. In particular we want to know if there is there a matrix O such that:

```
BASE_A4 = O*BASE_B4
```

If so the matrix O will represent an orthogonal transformation from one (real) basis to another.

If O exists then:

```
>> O = (BASE_A4)*inv(BASE_B4)

   -0.1014    0.8421   -0.2649   -0.4588
   -0.3804    0.3979   -0.0341    0.8342
    0.5473    0.3601    0.7477    0.1084
    0.7386    0.0537   -0.6080    0.2863
```

To see what properties O has we find the transpose and then multiply:

```
O' =

   -0.1014   -0.3804    0.5473    0.7386
    0.8421    0.3979    0.3601    0.0537
   -0.2649   -0.0341    0.7477   -0.6080
   -0.4588    0.8342    0.1084    0.2863

O*O'

    1.0000    0.0000   -0.0000    0.0000
    0.0000    1.0000   -0.0000    0.0000
   -0.0000   -0.0000    1.0000   -0.0000
    0.0000    0.0000   -0.0000    1.0000
```

Since the matrix that we obtain is the identity matrix, O' is the inverse of O and furthermore the columns of O itself form an orthonormal basis.

The matrix O thus is an orthogonal matrix and represents an orthonomal transformation from the base obtained from B4 to the base obtained from A4. The linear transformation whose matrix is O' = inv(O) goes in the other direction.

QR-factorizations. To illustrate these factorizations we the MATLAB function qr( ) with A4:

Example 7

```
>> [Q, R] = qr(A4)

Q =

   -0.3746    0.4650    0.1776   -0.7822
   -0.2810   -0.8049    0.4653   -0.2383
   -0.7493   -0.1431   -0.6334    0.1300
   -0.4683    0.3398    0.5922    0.5608

R =

  -10.6771   -1.7795    1.0302   -7.1181
         0    9.3184   -2.9154   -3.9349
         0         0    4.1760    1.1642
         0         0         0   -1.2227
```

If we multiply Q and R we find that:

A4 = Q*R

The matrix R is an upper triangular matrix (see Section III.1.vi for the solution of equations by LU decompositions which involve both upper and lower triangular matrices).

What about the matrix Q?

```
>> Q*Q'

    1.0000    0.0000    0.0000   -0.0000
    0.0000    1.0000   -0.0000    0.0000
    0.0000   -0.0000    1.0000   -0.0000
   -0.0000    0.0000   -0.0000    1.0000
```

Since Q*Q' is the the identity matrix, Q is an orthogonal matrix and the discussion of the previous section holds, e.g., if we type:

```
>> E = Q*BASE_A4
```

then E will be an orthogonal matrix and its columns will forma a basis for R5.

iv. Eigenvalues, eigenvectors, spectral decomposition, linear independence.

Eigenvalues and eigenvectors are discussed in detail in Section III.2; spectral decompositions are discussed in detail in Section III.2.iii; linear independence is discussed in detail in Section III.1.viii. In the following we only consider right (column) eigenvectors.

Eigenvalues, eigenvectors

Example 8

```
A1 =  [1     4             % [Nering, 224]
       4    -5]

>> [EIG_VEC1, EIG_VAL1] = eig(A1)   % Section III.2. i

EIG_VEC1 =

   -0.8944    0.4472
   -0.4472   -0.8944

% the (right) eigenvectors are the columns of EIG_VEC1
```

```
EIG_VAL1 =

     3     0
     0    -7
```

The eigenvalues appear as the diagonal elements of EIG_VAL1; so we pull them out into a column vector and then take the transpose. Since the eigenvalues are often complex, even when the original matrix was real, we use the true transpose .' instead of ' which gives the conjugate transpose (Section II.3.i).

```
>> eig_val1 = diag(EIG_VAL1);
>> eig_val1 = (eig_val1).'

    3    -7
```

Characteristic polynomial

The characteristic polynomial can be obtained by the MATLAB command poly(A).

Example 9

```
>> p1 = poly(A1)

      1  4 - 21
```

The terms here represent the polynomial p1(x)= 1*x^2 + 4*x - 21 (note that p1 is used here for both the polynomial and the vector of its coefficients; see Section II.7.i). We can check that eigenvalues are given by eig_val1 by substitution into p1. This can be done using the function coeff( ) defined in Section II.7.i.

```
>> coeff(p1 , eig_val1)

     0     0
```

Instead of checking the eigenvalues that we obtained above using the function eig( ) we can solve the characteristic polynomial, using the function roots( ) discussed in Section II.7.ii

```
>> e1 = roots(p1)

    -7
     3

>> e1 = sort(e1')
```

```
    -7     3

% place in decreasing order
>> e1 = fliplr(e1)     % reverse order; Section II.3.ii

     3    -7
```

In some examples MATLAB does not give quite the answer when using eig(A) and roots(poly(A)). This is due to fact that different methods are used.

Cayley-Hamilton Theorem

This theorem says that if we substitute a matrix into its characteristic polynomial---using matrix powers---then we obtain the zero matrix.

Since the function coeff( ) of Section II.7.i works with pointwise powers it cannot be used. However all we have to do is modify [coeff.m] by using ^ instead of .^ (i.e. no dot). We do, however, put in a check for the squareness of the matrix.

```
% [coeff2.m]
% for pointwise powers use [coeff.m] (Section II.7.i)

 function y = coeff2(poly , A)

 % check if square
 [m,n] = size(A);
 if m ~= n
     error('not square')       % Section II.8.iv
  end

 l = length(poly);

 y = zeros(m);
 for k = 1:l
      y = y + poly(k)*(A)^(l-k);  % matrix power
 end
```

Example 10. We apply coeff2 to A1 and p1

```
>> coeff2(p1 , A1)

        0  0
        0  0
```

Spectral decompositions

SECTION II.7 ADVANCED LINEAR ALGEBRA

The spectral decomposition of "nice" matrices (i.e. linearly independent eigenvectors; this occurs e.g. when we have distinct eigenvalues) is discussed in Section III.2.3 where the code for the function spectral( ) is given.

Example 11

```
>> spectral(A1)

     term 1,  eigenvalue = 3
     component matrix

      0.8000    0.4000
      0.4000    0.2000

     term 2,  eigenvalue = -7
     component matrix

     0.2000   -0.4000
    -0.4000    0.8000
```

Since the eigenvalues of A1 are distinct we are automatically guaranteed that the eigenvalues are linearly independent (Section III.1.viii). We could verify this directly in this case using rank( ):

```
>> rank(A1)

     2
```

Since the rank equals the number of columns, the (column) eigenvectors are linearly independent.

v. The similarity problem, diagonalization

In general the similarity problem involves relating a matrix A to a matrix D which is simpler in form, hopefully a diagonal or nearly diagonal matrix. More specifically given A we would like to find an invertible matrix P such that:

A = P*D*inv(P)

Whether this is possible, and what the form of P and D are depend upon how nice A is. We discuss three cases.

"Nice" square matrices. In this case we let:

P = EIG_VEC; D = EIG_VAL

then the definition of eigenvectors and eigenvalues is:

A*P = P*D

If the eigenvectors, i.e. the columns of P, are linearly independent (a sufficient, but not necessary condition for this is that the eigenvalues be distinct) then inv(P) will exist and we will have:

A = P*D*inv(P); D =inv(P)*A*P

Example 12.

```
A5 =

     3     2     2    -3
    -1     6     0     5
     4    -3     9     8
     8     0     6     1

>> [E_VEC5, E_VAL5] = eig(A5)

E_VEC5 =

  -0.0887   -0.4642 - 0.3537i    -0.4642 + 0.3537i    0.2420
  -0.3309    0.2857 - 0.0695i     0.2857 + 0.0695i    0.9354
  -0.8128    0.5706 + 0.1568i     0.5706 - 0.1568i   -0.0727
  -0.4712   -0.4095 + 0.2348i    -0.4095 - 0.2348i    0.2474

E_VAL5 =

  12.8534           0                  0                  0
  0            -0.4585 + 3.7766i        0                  0
  0                  0            -0.4585 - 3.7766i        0
  0                  0                  0            7.0636
```

Notice that the eigenvectors given by the second and third columns are complex, even though A5 is real. These complex eigenvalues correspond to the conjugate eigenvalues given as the second and third elements of the diagonal of E_VAL5

For simplicity, we designate the matrix of eigenvectors by P and the matrix of eigenvalues by D.

```
>> P = E_VEC5;
>> D = E_VAL5;
```

We check the rank of P to see if it is invertible

```
>> rank(P)

     4
```

So we know that inv(P) exists.

We now check that A5 is similar to D

```
>> F5 = P*D*inv(P);
>> F5 - A5;

   1.0e-14 *
       {4 x 4 matrix} = 0

>> G5 = inv(P)*A5*P;
>> G5 - D

   1.0e-14 *
       {4 x 4 matrix} = 0
```

Because of similarity we should have the relationship:

(A5)n = P*(Dn)*inv(P);

```
>> H5 = P*(D^10)*inv(P);
>> H5 - (A5)^10

   1.0e-04 *
       {4 x 4 matrix} = 0
```

In fact since D is diagonal, any power of D can be obtained by taking the powers of the individual diagonal elements.

Because A5 and D are similar they should have the same determinants.

```
>> det(D)

  1.3140e+03 + 3.1368e-14i

>> abs(det(D))

        1314
```

```
>> det(A5)

       1314
```

"Non-nice" square matrices: Jordan normal (cannonical) form

```
            [V, J] = jordan(A);
```

A = V*J*inv(V) ; J= inv(V)*A*V

Example 13

```
>> A6                          % Nering, p. 107

     1     0    -1     1     0
    -4     1    -3     2     1
    -2    -1     0     1     1
    -3    -1    -3     4     1
    -8    -2    -7     5     4

>> [E_VEC6, E_VAL6] = eig(A6);
```

% check the rank of the matrix of eigenvectors

```
>> rank(E_VEC6)

     3
```

So the eigenvectors are not linearly independent and no inverse exists.

% display the eigenvalues in a row vector

```
>> eigenvalues = sort(diag(E_VAL6).')

2.0000 - 0.0000i   2.0000 + 0.0000i   2.0000 - 0.0000i

2.0000    2.0000 + 0.0000i
```

So the five eigenvalues of A6 are all equal to 2. We check this by finding the characteristic polynomial and then the roots.

```
>> poly(A6)

    1.0000  -10.0000   40.0000  -80.0000   80.0000  -32.0000
```

```
% f(x)  = x^5 - 10*x^4 + 40*x^3 - 80*x^2 + 80*x - 32
%       = (x-2)^5

>> roots(poly(A6))

   2.0024
   2.0007 + 0.0023i
   2.0007 - 0.0023i
   1.9981 + 0.0014i
   1.9981 - 0.0014i
```

Notice that accuracy is not as good by this second method; MATLAB is using a differenet proceedure

Now we find the Jordan canonical form using the MATLAB function jordan( ):

```
>> [V7, J7] = jordan(A6)

V7 =

     0    -1     2     0     1
     0    -6     0    -2     0
    -1    -3     0    -1     0
    -1    -4     1    -1     1
    -1   -11     0    -3     0


J7 =

     2     1     0     0     0
     0     2     1     0     0
     0     0     2     0     0
     0     0     0     2     1
     0     0     0     0     2
```

We see that J has the eigenvalues on the diagonal and three 1s on the off-diagonal.

We check that A6 is indeed similar to J6.

```
>> H6 = V6*J6*inv(V6)

>> H6 - A6


     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0

>> G6 = inv(V6)*A6*V6

>> G6 - J6

     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
```

Non-square matrices: Singular value decomposition

```
      [U, S, V] = svd(A);  A = U*S*V'
```

Example 14.

We use transformed random numbers (Section II.2.iv) to generate a 2 x 5 random matrix with integer entries belonging to {-9, .. -1, 0, ... 9}.

```
>> A8 = -9 + floor(19*rand(2,5))    % floor - Section II.3.vi

    -3     6     0    -6     2
     4     8     7    -3     7

>> r8 = rank(A8)

     2
```

First we will find the singular value decomposition and verify the similarity properties and then we shall see where the matrix S comes from. Note carefully the dimensions of the matrices that we obtain.

```
>> [U8, S8, V8] = svd(A8)   %  svd( ) = singular value decomposition

U8 =          %  U8  = 2 x 2
    0.4472   -0.8944
    0.8944    0.4472

% S8 = 2 x 5, the same dimensions as A8
% note how S8 is a "diagonal" matrix

S8 =

   14.8661         0         0         0         0
        0    7.1414         0         0         0


V8 =          % V8 = 5 x 5

    0.1504    0.6262   -0.7650         0         0
    0.6618   -0.2505   -0.0749    0.5687   -0.4125
    0.4212    0.4384    0.4416   -0.5269   -0.3972
   -0.3610    0.5636    0.3904    0.6157   -0.1434
    0.4813    0.1879    0.2484    0.1408    0.8072


>> % check similarity

>> E8 = U8*S8*V8';

 E8 - A8

   1.0e-14 * {2 x 5 matrix} = 0
```

Both U8 and V8 are orthogonal matrices. We can check this:

```
>> U8*U8'

      1  0
      0  1
```

% similarly for V8*V8' we obtain the 5 x 5 identity matrix

So both U8 and V8 are invertible and we can obtain S8 from A8:

```
>> H8 = inv(U8)*A8*V8;

>> H8 - S8

   1.0e-14 * * {2 x 5 matrix} = 0
```

We now investigate what the 2 x 5 matrix S8 represents:

```
>> B8 = (A8)'*A8
```

```
    25    14    28     6    22
    14   100    56   -60    68
    28    56    49   -21    49
     6   -60   -21    45   -33
    22    68    49   -33    53
```

So B8 is a symmetric 5 x 5 matrix. Let us find the eigenvalues of B8.

```
>> eigenvalues8 = eig(B8);

% sort the eigenvalues in decreasing order:
>> eigenvalue8 = fliplr(sort(eigenvalues8))

        221.0000   51.0000    0.0000    0.0000   -0.0000
```

Notice that we have two strictly positive values with the rest being 0. This is the same situation as with the diagonal elements of S8. In fact the values are related:

```
>> s8 = r(eigenvalue8)              % r = sqrt

 14.8661   7.1414   0.0000    0.0000   0 + 0.0000i
```

So we see that diagonal elements of S8 are just the square roots of the eigenvalues of B8 = A8'*A8.