1. MATLAB Programming

What is MATLAB?

- MATLAB provides an interactive environment.
- Building block of MATLAB: matrix. (MATLAB = MATrix LABoratory.)
- Fundamental data type: array. All data in MATLAB are stored as arrays.
- On-line help is available.
MATLAB Windows

In a default setup, there are four main windows: Command Window, Current Directory, Workspace, and Command History.

- **Command Window**: This is the main window where all commands including those for running user-written programs are typed in. The MATLAB command prompt is `>>`. E.g.
  ```
  >> help help (which gives a brief synopsis of the help system)
  >> help (for a list of help topics)
  >> help topic
  ```

- **Current Directory**: This is where all your files from the current directory are listed. To change directories, click on the menu bar or in the Command Window:
  ```
  >> cd new directory
  ```
  To use a file from another directory, use `addpath`.
• **Workspace**: This window lists all variables you have generated in this session, together with their type and size. Double click on a variable in this window to view its value. Right click for other options.

• **Command History**: This window records all commands typed in the Command Window. Double click on a command to execute it again. You can also use the up-arrow on the keyboard.

**Demos**

A comprehensive set of demonstrations is available by typing the command

```matlab
>> demo
```

and following the instructions.
System Information

The command `version` returns information about the MATLAB version. E.g.

```
>> version
ans =
7.0.4.352 (R14) Service Pack 2
```

The command `ver` returns information about MATLAB as well as installed toolboxes. Use this command to find out all toolboxes installed on your computer.
Data type

The fundamental data type in MATLAB is the array which includes integers, real numbers (double precision), matrices, character strings, structures, and cells.

No declarations of data type or data object are needed. Real numbers are set to be of double precision. Dimension statements are not required for vectors and arrays.

Variable names

Variable names are case-sensitive, can contain up to 31 characters, and must start with a letter. These are allowable:

- My_Variable, my_variable (two different variables), Math2301.

These are not allowable:

- my-variable, 2301math, @UNSW
There is a reserved word list. To check if a variable name (e.g. my_variable) has been used or is in the reserved list:

```matlab
>> exist('my_variable')
ans =
   0               % my_variable does not exist
>> a = 5;
>> exist('a')
ans =
   1               % a is a variable in the workspace
>> exist('pi')
ans =
   5               % pi is a built-in MATLAB function
>> exist('sin')
ans =
   5               % sin is a built-in MATLAB function
```

Use help ( help exist) to learn more.
Some special variables:

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ans</td>
<td>default variable name used for results</td>
</tr>
<tr>
<td>pi</td>
<td>$\pi$</td>
</tr>
<tr>
<td>eps</td>
<td>smallest number such that when added to 1 gives a number greater than 1 on the computer</td>
</tr>
<tr>
<td>inf, Inf</td>
<td>$\infty$</td>
</tr>
<tr>
<td>NaN, nan</td>
<td>not a number, e.g. $0/0$</td>
</tr>
<tr>
<td>i, j</td>
<td>imaginary number $\sqrt{-1}$</td>
</tr>
</tbody>
</table>

If you assign a new value to one of the special variables above, its prior value is overwritten and lost.

`clear variable_name`: clears that variable from the memory.

`clear all`: clears all variables.
>> i, j
ans =
    0 + 1.0000i
ans =
    0 + 1.0000i
>> i = 3, j = 2;
i =
    3
>> clear i
>> i, j
ans =
    0 + 1.0000i
j =
    2
>> clear all
>> i, j
ans =
    0 + 1.0000i
ans =
    0 + 1.0000i
Suppression of Outputs

A semicolon at the end of a command suppresses the screen output, except for graphics and on-line help commands. The command `more on` directs MATLAB to show one screen of output at a time. To turn it off, type `more off`.

Output Format

Though MATLAB performs computations in double precision, the default appearance is a scaled fixed point format with 5 digits (`format short`). The command

```
>> format compact
```

suppresses blank lines in the output, thus allowing more information to be displayed on the screen.
The following example shows different outputs of $\pi$ in different formats:

<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>format short, pi</td>
<td>3.1416</td>
</tr>
<tr>
<td>format short e, pi</td>
<td>3.1416e+00</td>
</tr>
<tr>
<td>format long, pi</td>
<td>3.14159265358979</td>
</tr>
<tr>
<td>format long e, pi</td>
<td>3.141592653589793e+00</td>
</tr>
<tr>
<td>format rat, pi</td>
<td>355/113</td>
</tr>
<tr>
<td>format bank, pi</td>
<td>3.14</td>
</tr>
</tbody>
</table>

For other formats use help.
MATLAB Editor

MATLAB provides its own built-in editor with an integrated debugger. You can use this editor to create and save your own programs in files. Alternatively, you can use any text editor, like vi or emacs.

File Types in MATLAB

MATLAB has 3 types of files:

- M-files: standard ASCII text files, with a .m extension to the file name. There are 2 types of M-files: script files and function files.
- Mat-files: binary data files with a .mat extension to the file name.
- Mex-files: MATLAB callable Fortran and C programs, with a .mex extension to the file name.
Script Files and Mat Files

A MATLAB session can be performed by typing commands one by one in the Command Window. This session can be saved by using `diary`. E.g.

```
>> diary this_session
>> a = pi
a =
    3.1416
>> a/2
ans =
    1.5708
>> b = a^2
b =
    9.8696
>> diary off
```

A file named `this_session` is then created in the current directory which contains all of the lines above, except the first line.
The same commands can be run by creating a script file, named e.g. `this_session_script.m`, which contains the following lines

```
% The values of the variables a and b can be saved for next sessions, using the commands `save filename` and `load filename`. A file named `filename.mat` is created in the current directory.
a = pi
a/2
b = a^2
```

To run this file, in the Command Window type

```
>> this_session_script  % Note: name without .m.
   % Things after % are comments
```

The values of the variables `a` and `b` can be saved for next sessions, using the commands `save filename` and `load filename`. A file named `filename.mat` is created in the current directory.
Matrices and Vectors

A row vector is defined by listing numbers which are separated by either commas or spaces. A column vector has entries separated by semicolons or new lines. E.g.

```
>> v1 = [ 1 2, 3 4 ]
v1 =
    1    2    3    4

>> v2 = [ 3 ; 4
          6 ]
v2 =
    3
    4
    6
```

```
>> v1(3), v2(2)
ans =
    3
ans =
    4
```
Row and column vectors are one-dimensional arrays. Matrices are two-dimensional arrays. It is not necessary to declare the dimension of a matrix. E.g.

```
>> A = [ 1 2 3
      4 5 6 ]
A =
  1  2  3
  4  5  6

>> B = [ 3 7 9 ; 2 6 4 ]
B =
  3  7  9
  2  6  4
```

If it is not possible to type the entire input on the same line, use an ellipsis (three dots) and continue the input on the next line. E.g.

```
>> A = [ 4 7 9 7 ... 
       8 9 ]
A =
  4  7  9  7
  8  9
```
Length and Size of Arrays

The `size` command gives the number of rows and columns of an array. The `length` command gives the maximum value of these two numbers. E.g.

```plaintext
>> A = [1 8 0 3 ; 2 4 9 6 ; 0 3 7 9];
>> size(A)
ans =
   3   4
>> length(A)
ans =
   4
>> [m n] = size(A)  % assign these values to variables
m =                % m and n for later access
   3
n =                
   4
```
>> size(A,1)
ans =
    3
>> size(A,2)
ans =
    4
>> a = [2 3 4 5 9];
>> size(a)
ans =
    1    5
>> length(a)
ans =
    5
>> B = []% empty array
B =
    []
>> size(B)
ans =
    0    0
Accessing and Correcting an Entry of an Array

E.g. If \( a \) and \( A \) are defined as in previous pages. Then

\[
\begin{align*}
\text{>> } a(3) \\
\text{ans} &= \\
&\quad \begin{array}{l}
4 \\
\end{array} \\
\text{>> } a(2) = 0 \\
\text{a} &= \\
&\quad \begin{array}{rrrrr}
2 & 0 & 4 & 5 & 9 \\
\end{array} \\
\text{>> } A(2,3) \\
\text{ans} &= \\
&\quad \begin{array}{l}
9 \\
\end{array} \\
\text{>> } A(3,1) = 8 \\
\text{A} &= \\
&\quad \begin{array}{rrrrr}
1 & 8 & 0 & 3 \\
2 & 4 & 9 & 6 \\
8 & 3 & 7 & 9 \\
\end{array}
\end{align*}
\]
Matrix Arithmetic Operations

Given a scalar \( a \) (real or complex) and two matrices \( A \) and \( B \). The following operations are available under some appropriate conditions.

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C = a+A )</td>
<td>( C_{ij} = a + A_{ij} )</td>
<td></td>
</tr>
<tr>
<td>( C = a-A )</td>
<td>( C_{ij} = a - A_{ij} )</td>
<td></td>
</tr>
<tr>
<td>( C = a*A )</td>
<td>( C_{ij} = a * A_{ij} )</td>
<td></td>
</tr>
<tr>
<td>( C = A/a )</td>
<td>( C_{ij} = A_{ij}/a )</td>
<td></td>
</tr>
<tr>
<td>( C = A+B )</td>
<td>( C_{ij} = A_{ij} + B_{ij} )</td>
<td>( \text{size}(A) = \text{size}(B) )</td>
</tr>
<tr>
<td>( C = A-B )</td>
<td>( C_{ij} = A_{ij} - B_{ij} )</td>
<td>( \text{size}(A) = \text{size}(B) )</td>
</tr>
<tr>
<td>( C = A*B )</td>
<td>( C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj} )</td>
<td>( n = \text{size}(A,2) = \text{size}(B,1) )</td>
</tr>
<tr>
<td>( C = A^n )</td>
<td>( C = A \ast \ldots \ast A )</td>
<td>( n ) ( \text{times} ) ( \text{size}(A,1) = \text{size}(A,2) )</td>
</tr>
<tr>
<td>( C = A' )</td>
<td>( C_{ij} = A_{ji} )</td>
<td></td>
</tr>
<tr>
<td>( C = \text{inv}(A) )</td>
<td>( C = A^{-1} )</td>
<td>( \text{size}(A,1) = \text{size}(A,2) )</td>
</tr>
<tr>
<td>( C = A/B )</td>
<td>( C = AB^{-1} )</td>
<td>( \text{size}(A,2) = \text{size}(B,1) = \text{size}(B,2) )</td>
</tr>
<tr>
<td>( C = A\backslash B )</td>
<td>( C = A^{-1}B )</td>
<td>( \text{size}(A,1) = \text{size}(A,2) = \text{size}(B,1) )</td>
</tr>
</tbody>
</table>
E.g. Solving a matrix equation $Ax = b$.

$$
\begin{align*}
>> & \quad A = \begin{bmatrix} 1 & 8 & 3 \\ 2 & 4 & 6 \\ 8 & 3 & 9 \end{bmatrix}; \quad b = \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix}; \\
>> & \quad x = \text{inv}(A)\times b \\
& \quad x = \\
& \quad \begin{bmatrix} 0.1500 \\ 0.0833 \\ 0.3944 \end{bmatrix} \\
>> & \quad x = A\backslash b \quad % \text{Almost the same as inv}(A)\times b \text{ but} \\
& \quad x = \\
& \quad \begin{bmatrix} 0.1500 \\ 0.0833 \\ 0.3944 \end{bmatrix}
\end{align*}
$$
\[
\begin{align*}
\text{>> } & \quad c = [1 \ 3 \ 4]; \\
\text{>> } & \quad x = A \backslash c \\
\text{??? Error using } & \quad \text{==> mldivide} \\
\text{Matrix dimensions must agree.} \\
\text{>> } & \quad x = A \backslash c' \\
x & = \\
& \begin{bmatrix}
-0.1500 \\
-0.0833 \\
0.6056
\end{bmatrix}
\end{align*}
\]

\textit{linspace}(a,b,n)\textit{ generates a vector of }n\textit{ linearly equally spaced points between (and including) }a\textit{ and }b. \textit{linspace}(a,b)\textit{ means }n = 100. \text{ E.g.}

\[
\begin{align*}
\text{>> } & \quad \text{linspace}(1,10,5) \\
\text{ans} & = \\
& \begin{bmatrix}
1.0000 \\
3.2500 \\
5.5000 \\
7.7500 \\
10.0000
\end{bmatrix} \\
\text{>> } & \quad 1:(10-1)/(5-1):10 \quad \% \ [a:(b-a)/(n-1):b] \\
\text{ans} & = \\
& \begin{bmatrix}
1.0000 \\
3.2500 \\
5.5000 \\
7.7500 \\
10.0000
\end{bmatrix}
\end{align*}
\]
Element-by-element Operations

If $a$ is a scalar, and $A$ and $B$ are two arrays of the same size, then the element-by-element operations give the following arrays:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C = a./A$</td>
<td>$C_{ij} = a/A_{ij}$</td>
</tr>
<tr>
<td>$C = A./a$</td>
<td>$C_{ij} = a/A_{ij}$</td>
</tr>
<tr>
<td>$C = A.^a$</td>
<td>$C_{ij} = A^{a}_{ij}$</td>
</tr>
<tr>
<td>$C = a.^A$</td>
<td>$C_{ij} = a^{A}_{ij}$</td>
</tr>
<tr>
<td>$C = A.*B$</td>
<td>$C_{ij} = A_{ij}B_{ij}$</td>
</tr>
<tr>
<td>$C = A./B$</td>
<td>$C_{ij} = A_{ij}/B_{ij}$</td>
</tr>
<tr>
<td>$C = A./B$</td>
<td>$C_{ij} = B_{ij}/A_{ij}$</td>
</tr>
<tr>
<td>$C = A.^B$</td>
<td>$C_{ij} = A^{B}_{ij}$</td>
</tr>
</tbody>
</table>
E.g.

```matlab
>> v = 1:5
v =
     1     2     3     4     5
>> 2.^v
ans =
     2     4     8    16    32
>> v.^2
ans =
     1     4     9    16    25
>> v^2
??? Error using ==> mpower
Matrix must be square.
```
The commands \([a:b]\) and \([a:s:b]\)

\[
\begin{align*}
&\text{>> } v = [3:9] \quad \% \text{ increment by } 1 \\
&v = \\
&\quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \\
&\text{>> } v = 3:2:9 \quad \% \text{ increment by } 2 \\
&v = \\
&\quad 3 \quad 5 \quad 7 \quad 9 \\
&\text{>> } v = [3:2:8] \quad \% \text{ final value does not exceed } 8 \\
&v = \\
&\quad 3 \quad 5 \quad 7 \\
&\text{>> } v = [8:-2:3] \quad \% \text{ negative increment} \\
&v = \\
&\quad 8 \quad 6 \quad 4 \quad 3 \\
&\text{>> } v = [5:2] \quad \% \text{ final value does not go below } 3 \\
&v = \\
&\quad \text{Empty matrix: 1-by-0} \\
&\text{>> } v = 2:-1:3 \quad \% \text{ final value does not go below } 3 \\
&v = \\
&\quad \text{Empty matrix: 1-by-0}
\end{align*}
\]
Parts of Matrices

```matlab
>> A = [2 5 -1 7 ; 8 0 6 3 ; 9 8 -2 1 ; 3 7 9 2]
A =
    2     5    -1     7
    8     0     6     3
    9     8    -2     1
    3     7     9     2
>> B = A(2:4,3:4)
B =
   6     3
  -2     1
   9     2
>> C = A(1:2:4,:)
C =
    2     5    -1     7
    9     8    -2     1
```
>> A(2:4,3:4) = 0
A =
   2    5   -1    7
   8    0    0    0
   9    8    0    0
   3    7    0    0
>> A(1:2:end,1) = -1
A =
   -1    5   -1    7
   8    0    0    0
   -1    8    0    0
   3    7    0    0

Using an Array to Refer to Part of a Matrix

>> j = [1 3 4];
>> A(3,j)
an =
   -1    0    0
Stacking Small Matrices

>> A = [1 -1 ; 2 2]; B = [2 1 ; 0 0]; C = [1:4 ; 5:8];
>> E = [A B ; C]
E =
      1   -1    2    1
      2    2    0    0
      1    2    3    4
      5    6    7    8
>> D = [1:5, 5:8];
>> F = [A B ; D]
??? Error using ==> vertcat
All rows in the bracketed expression must have the same number of columns.
Zeros and Ones Arrays

>> zeros(3)
ans =
     0   0   0
     0   0   0
     0   0   0

>> zeros(2,3)
ans =
     0   0   0
     0   0   0
     0   0   0

>> A = [1 2 5 7; 2 4 8 1];
>> zeros(size(A))
ans =
     0   0   0   0
     0   0   0   0
     0   0   0   0

Replacing zeros by ones gives similar matrices with entries 1.
Identity Matrices

>> eye(2)
an =
    1  0
    0  1

>> eye(3,2)
an =
    1  0
    0  1
    0  0

>> eye(2,3)
an =
    1  0  0
    0  1  0

>> A = [1 2 5 7 ; 2 4 8 1];
>> B = eye(size(A))
B =
    1  0  0  0
    0  1  0  0
Diagonals of a Matrix and Diagonal Matrices

- If A is a matrix then \( \text{diag}(A,k) \), where \( k \) is an integer, produces a column vector whose components are entries on the \( k^{\text{th}} \) diagonal of A. E.g.

\[
\begin{align*}
\text{A} &= \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}; \quad v_1 = \text{diag}(A), \quad v_2 = \text{diag}(A,1), \ldots \\
v_3 &= \text{diag}(A,2), \quad v_4 = \text{diag}(A,3), \quad v_5 = \text{diag}(A,-1) \\
v_1 &= \\
&\begin{bmatrix} 1 \\ 4 \end{bmatrix} \\
v_2 &= \\
&\begin{bmatrix} 3 \\ 6 \end{bmatrix} \\
v_3 &= \\
&\begin{bmatrix} 5 \end{bmatrix} \\
v_4 &= \text{Empty matrix: 0-by-1} \\
v_5 &= \\
&\begin{bmatrix} 2 \end{bmatrix}
\end{align*}
\]
If \( v \) is a vector, then \( \text{diag}(v,k) \) creates a square matrix of appropriate size whose \( k^{\text{th}} \) diagonal is \( v \). E.g.

\[
\begin{align*}
\mathbf{v} &= [1:3]; \quad \mathbf{v}_1 = \text{diag}(v), \ \mathbf{v}_2 = \text{diag}(v,1), \ \mathbf{v}_3 = \text{diag}(v,-2) \\
\mathbf{v}_1 &= \\
&= \begin{bmatrix} 1 & 0 & 0 \\
0 & 2 & 0 \\
0 & 0 & 3 \\
\end{bmatrix} \\
\mathbf{v}_2 &= \\
&= \begin{bmatrix} 0 & 1 & 0 & 0 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 3 \\
0 & 0 & 0 & 0 \\
\end{bmatrix} \\
\mathbf{v}_3 &= \\
&= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 & 0 \\
0 & 0 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\end{align*}
\]
Sparse Matrices

```matlab
>> A = [0 0 1 0 0 ; -2 0 3 0 0 ; 0 0 -5 0 0]
A =
    0     0     1     0     0
   -2     0     3     0     0
    0     0    -5     0     0

>> nnz(A)  % number of non-zero entries of A
ans =
    4

>> B = sparse(A)
B =
    (2,1)  -2
    (1,3)   1
    (2,3)   3
    (3,3)  -5
```

- The command `full(B)` reproduces a matrix identical to `A`.

- Arithmetic and indexing operations can be applied to sparse matrices or to a mixture of sparse and full matrices.
Listing current variables: the commands ‘who’ and ‘whos’

>> clear all % clear all existing variables from memory
>> clc % clear the Command Window
>> A = [2 0 5 0 0 ; 0 0 0 4 0 ; 0 0 0 1 0]; B = sparse(A);
>> nnz(A); v = [9 11 8];
>> who
Your variables are:
A   B   ans   v
>> whos A
    Name  Size          Bytes  Class
    A     3x5           120     double array
Grand total is 15 elements using 120 bytes
>> whos
    Name  Size          Bytes  Class
    A     3x5           120     double array
    B     3x5           72      double array (sparse)
    ans   1x1           8       double array
    v     1x3           24      double array
Grand total is 23 elements using 224 bytes
>> A = [0 1 0 8 0 0 ; 0 0 0 0 -3 0 ; 0 -2 4 0 0 2 ; 0 5 0 0 1 1];
>> B = sparse(A);
>> whos
    Name      Size             Bytes  Class          Attributes
    A         4x6              192    double array
    B         4x6              136    double array  (sparse)
Grand total is 33 elements using 328 bytes

>> A = eye(200); B = sparse(A);
>> whos
    Name      Size             Bytes  Class          Attributes
    A         200x200          320000 double array
    B         200x200          3204    double array  (sparse)
Grand total is 40200 elements using 323204 bytes
Speye – Identity matrix in sparse mode

Large identity matrices should be created by using the `speye` command instead of `eye`.

```matlab
>> clear all
>> A = eye(300);
>> sA = speye(300);
>> whos
Name      Size      Bytes  Class          Value
A         300x300  720000  double array
sA        300x300  4804    double array (sparse)

Grand total is 90300 elements using 724804 bytes
```
More on the command ‘sparse’

If \( A \) is a given matrix, the command \( sA = \text{sparse}(A) \) produces a matrix \( sA \) which is a sparse version of \( A \).

A sparse matrix \( B \) of size \( 4 \times 4 \) with nonzeros entries \( B(2,1) = -2, B(1,2) = 1, B(4,2) = 1, B(2,3) = 3, B(3,3) = -5, B(4,4) = 1 \) can be created as follows:

\[
\begin{align*}
\text{>> } a &= [-2 1 1 3 -5 1]; &\text{ % non-zero entries of } B \\
\text{>> } r &= [2 1 4 2 3 4]; &\text{ % row indices} \\
\text{>> } c &= [1 2 2 3 3 4]; &\text{ % column indices} \\
\text{>> } B &= \text{sparse}(r,c,a,4,4) \\
B &= \\
&\begin{pmatrix}
(2,1) & -2 \\
(1,2) & 1 \\
(4,2) & 1 \\
(2,3) & 3 \\
(3,3) & -5 \\
(4,4) & 1 \\
\end{pmatrix}
\end{align*}
\]
Spdiags

Sparse matrices are naturally defined in terms of their diagonals using `spdiags` command. E.g

```matlab
>> e = ones(5,1); % some none-zero entries
>> A = spdiags([-e 2*e -e],... % 0 -> diagonal, -1 -> subdiagonal
               [-1 0 1],5,5); % 1 -> superdiagonal
>> full(A)
ans =
   2  -1     0     0     0
  -1     2  -1     0     0
   0  -1     2  -1     0
   0     0  -1     2  -1
   0     0     0  -1     2
```

The diagonals can be obtained from columns of another matrix. E.g.
\[
\begin{bmatrix}
2 & 5 & 6 & 0 & 0 & 0 \\
-3 & 1 & 7 & 0 & 0 & 0 \\
4 & 1 & 8 & 0 & 0 & 0 \\
0 & 2 & 8 & 0 & 0 & 0 \\
1 & 1 & -1 & 0 & 0 & 0 \\
\end{bmatrix}
\]


\[
\begin{bmatrix}
0 & 1 & 8 & 0 & 0 & 0 \\
0 & 0 & 1 & 8 & 0 & 0 \\
2 & 0 & 0 & 2 & -1 & 0 \\
0 & -3 & 0 & 0 & 1 & 0 \\
0 & 0 & 4 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

If a column of \( B \) is longer than the diagonal it is representing, elements of super-diagonals of \( A \) correspond to the lower part of the column of \( B \), while elements of sub-diagonals of \( A \) correspond to the upper part of the column of \( B \).
**Built-in Functions**

MATLAB has numerous built-in functions. Use the command `help elfun` to see available elementary mathematical functions.

**Scalar functions** naturally take scalar arguments; but will also act on matrices elementwise. E.g.

```
>> x = 1.2; y = [1:0.2:2]; X = [1 2 ; pi 5];
>> sin(x), sin(y), sin(X)
ans =
    0.9320
ans =
    0.8415  0.9320  0.9854  0.9996  0.9738  0.9093
ans =
    0.8415   0.9093
    0.0000  -0.9589
```
Vector functions act on vectors and matrices. If \( v \) is a row vector or column vector, \( \text{sum}(v) \) gives the sum of the elements of \( v \). The command \( \text{sum}(v,1) \) treats \( v \) as a matrix and adds its components columnwise, whereas \( \text{sum}(v,2) \) finds the sum of elements in each row. E.g.

\[
\begin{align*}
\text{v} &= [1:5]; \\
\text{s1} &= \text{sum}(v), \text{s2} = \text{sum}(v,1), \text{s3} = \text{sum}(v,2) \\
\text{s1} &= \\
&\quad 15 \\
\text{s2} &= \\
&\quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \\
\text{s3} &= \\
&\quad 15
\end{align*}
\]

\( \text{prod}(v) \), \( \text{prod}(v,1) \), and \( \text{prod}(v,2) \) have the same meaning for products of elements of \( v \).
If \( A \) is a matrix, \( \text{sum}(A) = \text{sum}(A,1) \) adds the elements in each column of \( A \), whereas \( \text{sum}(A,2) \) adds the elements in each row. E.g.

\[
A = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}
\]

\[
>> s1 = \text{sum}(A), s2 = \text{sum}(A,1), s3 = \text{sum}(A,2), s4 = \text{sum}(%s1)\text{sum}(A))
\]

\[
s1 = \begin{bmatrix} 3 & 7 & 11 \\ 3 & 7 & 11 \end{bmatrix}
\]

\[
s2 = \begin{bmatrix} 3 & 7 & 11 \\ 9 & 12 & 21 \end{bmatrix}
\]

\( \text{prod}(A), \text{prod}(A,1), \text{prod}(A,2), \) and \( \text{prod}(\text{prod}(A)) \) have the same meaning for products of elements of \( A \).
Matrix functions take matrix arguments. E.g.

```matlab
>> A = [1 3 5 ; 2 4 6 ; 9 2 1];
>> det(A)
ans =
   -12
>> inv(A)
ans =
   0.6667   -0.5833    0.1667
  -4.3333    3.6667   -0.3333
  -2.6667   -2.0833    0.1667
>> A = [1 3 5 ; 2 4 6 ; 9 2 1 ; 4 8 2];
>> reshape(A,2,6)
ans =
     1     9     3     2     5     1
     2     4     4     8     6     2
```

43
Logical Arrays

Logical arrays are arrays whose components are true or false. The true value is represented by 1 and false by 0. Note that they are logical 1 and logical 0, not numbers.

Creating a Logical Array

A logical array can be created by using

- The true, false functions
- The logical function: converts numeric values to logical.
- Relational operators.
- Logical operators.
- All is functions. E.g. isinteger, isempty, isnan, isinf, etc.
The true/false functions

>> x = [true false]
x =
    1  0

>> A = true(2), B = false(2,3)
A =
    1  1
    1  1
B =
    0  0  0
    0  0  0

>> whos x A B
Name      Size          Bytes   Class
A         2x2                4   logical array
B         2x3                6   logical array
x         1x2                2   logical array

Grand total is 12 elements using 12 bytes
The logical function

```matlab
>> a = 1; b = logical(a); % convert a numeric value to logical
>> whos a b
    Name      Size         Bytes  Class      Attributes
    a        1x1           8        double array
    b        1x1           1        logical array

Grand total is 2 elements using 9 bytes

>> A = [1 2 0; -4 0 3];
>> B = logical(A)
Warning: Values other than 0 or 1 converted to logical 1.
B =
      1   1   0
      1   0   1
>> whos A B
    Name      Size         Bytes  Class      Attributes
    A        2x3           48        double array
    B        2x3           6        logical array

Grand total is 12 elements using 54 bytes
Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equal</td>
<td>A (operator) B</td>
</tr>
<tr>
<td>~=</td>
<td>not equal</td>
<td>returns (componentwise)</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>logical 1</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
<td>if relation is true, and logical 0</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>if relation is false</td>
</tr>
</tbody>
</table>

```plaintext
>> 3 > 4
ans =
   0 % 0 when relation is false

>> s = 2 < 3 % assign the answer to variable s.
s =
   1 % 1 when relation is true
```
\[ u = [1 \ 3 \ 9 \ 0]; \ v = [2 \ 1 \ 9 \ 3]; \text{ note } u \text{ and } v \text{ of same size} \]

\[ s1 = u \neq v, \ s2 = u \geq v \]

\[
\begin{array}{cccc}
s1 &=& 1 & 1 & 0 & 1 \\
s2 &=& 0 & 1 & 1 & 0 \\
\end{array}
\]

\[ A = [2 \ 3; \ 4 \ 5]; \ B = [2 \ 4; \ 5 \ 5]; \text{ note } A \text{ and } B \text{ of same size} \]

\[ C = A \neq B \]

\[
\begin{array}{cc}
C &=& 0 & 1 \\
& &=& 1 & 0 \\
\end{array}
\]

\[ \text{whos } s1 \ s2 \ C \]

\begin{tabular}{llll}
\text{Name} & \text{Size} & \text{Bytes} & \text{Class} \\
C & 2x2 & 4 & \text{logical array} \\
s1 & 1x4 & 4 & \text{logical array} \\
s2 & 1x4 & 4 & \text{logical array} \\
\end{tabular}

\text{Grand total is 12 elements using 12 bytes}
## Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;</code></td>
<td>( C = A \land B, C(i,j) = \begin{cases} 1, &amp; A(i,j) \neq 0 \AND B(i,j) \neq 0 \ 0, &amp; \text{otherwise} \end{cases} )</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>~</code></td>
<td>( C = \neg A, C(i,j) = \begin{cases} 1, &amp; A(i,j) = 0 \ 0, &amp; \text{otherwise} \end{cases} )</td>
</tr>
<tr>
<td>any</td>
<td>( v = \text{any}(u), v = \begin{cases} 1, &amp; \text{at least one } u(i) \neq 0 \ 0, &amp; \text{otherwise} \end{cases} )</td>
</tr>
<tr>
<td>all</td>
<td>( v = \text{all}(u), v = \begin{cases} 1, &amp; \text{all } u(i) \neq 0 \ 0, &amp; \text{otherwise} \end{cases} )</td>
</tr>
</tbody>
</table>

*Note: \( A, B, C \) are arrays of same size.*
A = [2 0 3; 5 0 1]; B = [0 0 -3; 1 3 0];
C = A & B, D = A | B, E = ~A,
C =
0 0 1
1 0 0
D =
1 0 1
1 1 1
E =
0 1 0
0 1 0

u = [2 0 1];
v1 = any(u)
v1 =
1
v2 = all(B) % v2 = [all(B(:,1)) all(B(:,2)) all(B(:,3))]
v2 =
0 0 0
The is Functions

isequal, isempty, isfinite, isnan, isinteger, isprime, ...

>> x1 = []; x2 = [1 2];
>> v1 = isempty(x1), v2 = isempty(x2)
  v1 =
      1
  v2 =
      0

>> A = [2 0 3; 5 0 1]; B = [0 -3 ; 1 3]; C = [2 0 3; 5 1 1];
>> v1 = isequal(A,A), v2 = isequal(A,B), v3 = isequal(A,C)
  v1 =
      1
  v2 =
      0
  v3 =
      0
Use of Logical Arrays

Logical arrays are used

- in array indexing;
- in conditional statements: IF, ELSE IF.

Use of Logicals in Array Indexing

```matlab
>> x = [5 2 8 4 9 0 -1];
>> v = x < 5 & x >= 0
v =
     0  1  0  1  0  1  0
>> x(v)
ans =
    2  4  0
>> x(v) = 5 % same as x(x<5&x>=0) = 5
x =
     5  5  8  5  9  5  -1
```
>> v = [0 1 0 1 0 1 0];
>> x(v)
??? Subscript indices must either be real positive integers 
   or logicals.
>> A = [2 0 3 ; 5 0 1]; B = [0 0 -3 ; 1 3 0];
>> A(A&B) = inf
A =
    2   0   Inf
   Inf   0   1
>> A = magic(3), A(~isprime(A)) = 0
A =
    8   1   6
    3   5   7
    4   9   2
A =
    0   0   0
    3   5   7
    0   0   2

53
Using Logicals in Conditional Statements

Example 1:

if I == J
    A(I,J) = 2;
elseif abs(I-J) == 1
    A(I,J) = -1;
else
    A(I,J) = 0;
end

Example 2:

if x <= 0
    y = -x;
else
    y = x.^2;
end
Character Strings

In MATLAB a string is an array of ASCII character codes. (ASCII = American Standard Code for Information Interchange).

```matlab
>> s = 'Hello world!’
s =
Hello world!
>> whos s
    Name      Size     Bytes  Class     Attributes
    s          1x12     24      char array

Grand total is 12 elements using 24 bytes
>> isstr(s), isstr(5), isstr(‘5’) % check if the variable
ans =
    1 % is a string
ans =
    0
ans =
    1
```
Since strings are arrays, they can be manipulated with all array manipulation tools.

```
>> a = 'Hello'; b = 'world'; c = '!' ;
>> [a b c]
ans =
Hello world!
>> [a ' ' b c]
ans =
Hello world!
>> a(1:4)
ans =
Hell
>> b(end:-1:1)
ans =
dlrow
>> u = 'I don’t know how' % single quotes within character are symbolised by two consecutive quotes
u =
I don’t know how
```
Strings can have multiple rows, but each row must have equal number of columns.

\[
\begin{align*}
\text{>> } v & = [\text{\textquotesingle}Joshua\text{\textquotesingle}; \text{\textquotesingle}Jack\text{\textquotesingle}; \text{\textquotesingle}Ben\text{\textquotesingle}] \\
v & = \\
Joshua \\
Jack \\
Ben
\end{align*}
\]

 Artificial creation of blank spaces can be avoided by using \texttt{char} command.

\[
\begin{align*}
\text{>> } \text{student} & = \text{char(\textquotesingle}Joshua\textquotesingle, \textquotesingle}Jack\textquotesingle, \textquotesingle}Ben\textquotesingle) \\
\text{student} & = \\
Joshua \\
Jack \\
Ben
\end{align*}
\]
Horizontal concatenation of string arrays having the same number of rows is accomplished by the function `strcat`.

```matlab
>> strcat('Adam', ' Smith')
ans =
Adam Smith
>> students = char('Joshua', 'Jack', 'Ben');
>> surnames = char(' Waugh', ' Warne', ' Bevan');
>> v = strcat(students, surnames)
v =
Joshua Waugh
Jack Warne
Ben Bevan
>> whos v
    Name      Size            Bytes  Class      Attributes
    v     3x12             72     char array
Grand total is 36 elements using 72 bytes
```
Cell Arrays

A cell array is an array that can contain any MATLAB object, e.g., an array of numbers, logicals, a string, etc.

>> A = {[1 2; 3 4], 'MATH2301', 1 + 2i, 1 > 0}
A =
    [2x2 double]    'MATH2301'    [1.0000 + 2.0000i]    [1]
>> whos A
    Name      Size         Bytes  Class     Attributes
    A         1x4           305  cell array

Grand total is 18 elements using 305 bytes

The curly braces indicate that the expression represents a cell rather than, e.g., a numerical value.

We can preallocate a variable as a cell array, using the command cell.
>> B = cell(2,2)
B =
    []    []
    []    []
>> B{1,1} = ones(1,3); B{1,2} = rand(1,2);
>> B{2,1} = 'MATLAB'; B{2,2} = A
B =
    [1x3 double]    [1x2 double]
    'MATLAB'        {1x4 cell}

To display the content of a cell array, use `celldisp`.

>> celldisp(A)
A{1} =
    1     2
    3     4
A{2} =
MATH2301
A{3} =
    1.0000 + 2.0000i
A{4} =
    1
>> celldisp(B)
B{1,1} =
    1 1 1
B{2,1} =
MATLAB
B{1,2} =
    0.8913 0.7621
B{2,2}{1} =
    1 2
    3 4
B{2,2}{2} =
MATLAB
B{2,2}{3} =
    1.0000 + 2.0000i
B{2,2}{4} =
    1
To display the content of a component of a cell array:

```matlab
>> A{1,2}
ans =
MATH2301
>> B{2,2}
ans =
    [2x2 double]    'MATH2301'    [1.0000 + 2.0000i]    [1]
>> B{2,2}{2}
ans =
MATH2301
>> B{2,2}{1}
ans =
    1     2
    3     4
>> B{2,2}{1}(2,2)
ans =
    4
```
A cell can be displayed in a graphics window, using the command \texttt{cellplot}. E.g. \texttt{cellplot(A)}, \texttt{cellplot(B)}

Cell arrays are often used to handle strings. The command \texttt{deal} extracts the contents of a cell array and assigns them to a variable. E.g.:

\begin{verbatim}
>> names = {'Joshua', 'Jack', 'Ben', 'Peter'}
names =
    'Joshua'    'Jack'    'Ben'    'Peter'
>> [a b] = deal(names{1:2}), c = deal(names{3})
a =
Joshua
b =
Jack
c =
Ben
\end{verbatim}
Structures

A structure is a collection of objects labelled by field names. E.g.:
```matlab
>> s = struct('name', 'harry', 'age', 45)
s =
    name: 'harry'
    age: 45
>> whos s
    Name      Size       Bytes  Class      Attributes
    s         1x1         266      struct array
Grand total is 8 elements using 266 bytes
>> s.name, s.age
ans =
    harry
ans =
    45
>> fieldnames(s)
ans =
    'name'
    'age'
```
We can also form array of structures:

```matlab
>> t = struct('name', {'harry', 'sue', 'john'}, ...
    'age', {46, 37, 29})
```

```
t =
1x3 struct array with fields:
    name
    age
>> t(2).name
ans =
sue
>> t.age
ans =
    46
ans =
    37
ans =
    29
>> disp([t.age])
    46   37   29
```
Graphics

The `plot` command:

```matlab
>> % plot y = exp(x) for 0 <= x <= 3
>> x = 0:0.3:3; % create some points on the domain [0,3]
>> y = exp(x); % compute function values at x
>> plot(x,y)
>> % We may need more points to get a smooth plot
>> x = linspace(0,3); % create 100 equally spaced points on [0,3]
>> y = x.^2 .* exp(x); plot(x,y)
```

The `fplot` command selects points on the x-axis automatically.

```matlab
>> fplot('sin(10*x)', [0 3])
>> fplot('x^2 * exp(x)') % or fplot('x.^2 .* exp(x)')
>> axis([0 0.5 0 1]) % Change axes. Type help axis for more info
```

Different colours and linestyles can be chosen: `+ * -`, `y m c r g b w k`. E.g. `plot(x,y,'g+')`.
The command `close 2 (close all)` closes figure 2 (all figures).
Several curves can be drawn at once. Plot the Bessel functions of order zero, one, and two on \([0, 10]\). (Bessel functions of order \(\mu\) are solutions of the ODE \(x^2y'' + xy' + (x^2 - \mu^2)y = 0\). Type \texttt{help besselj} to get more info.)

\begin{verbatim}
>> figure(2)
>> x = [0:0.2:10];
>> y0 = besselj(0,x); y1 = besselj(1,x); y2 = besselj(2,x);
>> plot(x,y0,x,y1,x,y2)
>> legend('
\mu=0','
\mu=1','
\mu=2')
\end{verbatim}

Another way to plot multiple graphs in one figure window is to use \texttt{hold on}:

\begin{verbatim}
>> x = linspace(0,2); y1 = x.^2; y2 = sin(x);
>> plot(x,y1)
>> hold on
>> plot(x,y2)
>> hold off
\end{verbatim}
Parametric curves can be drawn as follows:

```matlab
>> t = linspace(-pi,pi);
>> plot(cos(t),sin(3*t))
>> xlabel('x(t)') % x-axis label
>> ylabel('y(t)') % y-axis label
>> title('Parametric curve (cos(t),sin(3t))') % title plot
```

Curves can be drawn using polar coordinates:

```matlab
>> theta = linspace(-pi,pi,300);
>> r = 1 ./ (1+0.6*cos(10*theta));
>> polar(theta,r)
```

**Subplots**

The figure window can be divided into rectangular panes that are numbered rowwise. Plots are then output to each pane.

```matlab
>> subplot(1,2,1), plot(x,y1)
>> subplot(1,2,2), plot(x,y2)
```
3D Plots

Plot the surface $z = 1 - x^2/3 - y^2/5$ for $-2 \leq x \leq 2$ and $-1 \leq y \leq 1$.

$$\begin{align*}
>> & x = -2:2; \quad y = -1:1; \\
>>& [X,Y] = \text{meshgrid}(x,y) \\
>>& X = \\
&& \begin{bmatrix}
-2 & -1 & 0 & 1 & 2 \\
-2 & -1 & 0 & 1 & 2 \\
-2 & -1 & 0 & 1 & 2 
\end{bmatrix} \\
>>& Y = \\
&& \begin{bmatrix}
-1 & -1 & -1 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 
\end{bmatrix} \\
>>& Z = 1 - X.^2/3 - Y.^2/5; \\
>>& \text{surf}(X,Y,Z) \quad \% \text{ or } \text{surf}(x,y,Z)
\end{align*}$$

In general, $X, Y, Z$ are matrices of size $n \times m$ where $m$ and $n$ are the lengths of $x$ and $y$, resp, and $X(i,j) = x(j)$, $Y(i,j) = y(i)$. The surface is determined by points $(x(j), y(i), Z(i,j))$. 
Spheres can be generated by the command `sphere`.

```matlab
>> help sphere
SPHERE Generate sphere.
   [X,Y,Z] = SPHERE(N) generates three (N+1)-by-(N+1) matrices so that SURF(X,Y,Z) produces a unit sphere.


SPHERE(N) and just SPHERE graph the sphere as a SURFACE and do not return anything.

See also ellipsoid, cylinder.
```

```matlab
>> [X,Y,Z] = sphere(15);
>> subplot(2,2,1), mesh(X,Y,Z), hidden on % hidden line removal on
>> subplot(2,2,2), mesh(X,Y,Z), hidden off % hidden line removal off
>> subplot(2,2,3), surf(X,Y,Z)
>> subplot(2,2,4), surf(X,Y,Z), shading interp
```
Programming in MATLAB

Recall that MATLAB has three types of files, M-files (ASCII text files), Mat-files (binary data files), and Mex-files; see page 12. A program in MATLAB is written in M-files, script files and function files.

Script files

A MATLAB script is a text file containing MATLAB commands, and having a .m extensions. To execute commands in a script, simply type the name of the file, without the .m extension, at the MATLAB prompt. The script file has to be in the current directory, or in a directory of the MATLAB search path. The command what can be used to check if the file is in the current directory. Recall that a path can be added by the command addpath.

Running a script file is equivalent to typing all the commands (in that file) one by one at the MATLAB prompt. The advantage is obvious: script files are easy to modify.
1.1 Example. Plot $\sin(x)$ and $\cos(x)$ in one figure for $a \leq x \leq b$, where $a$ and $b$ are input parameters.

A script file, named e.g. `eg_1.m`, containing these lines solves the problem.

```matlab
% Script file to plot sin(x) and cos(x), a <= x <= b.
a = input(‘a = ’); % prompt for user input
b = input(‘b = ’); % prompt for user input
x = [a:0.1:b];
y1 = sin(x);
y2 = cos(x);
plot(x,y1,x,y2)
legend(’sin x’, ’cos x’)
```

Remark: The above script file needs to be improved further to make sure that $b > a$ using, for example, an `if` statement to be discussed later.
1.2 Example. Plot the Riemann zeta function $\zeta_N(s) = \sum_{n=1}^{N} n^{-s}$, $s \in [a, b]$, $a > 1$

% Script file to plot the Riemann zeta function on [a,b]
disp(‘Plot zeta(s) for a < s < b where a > 1’.)
a = input (‘a = ’);
b = input (‘b = ’);
umterms = input(’Number of terms =’);
s = linspace(a,b);
bases = [1:numterms]’ * ones(size(s));
exponents = -ones(numterms, 1) * s;
y = sum(bases .^ exponents);
plot(s,y)
xlabel(’s’)
ylabel(’zeta(s)’);
Some explanations of $\text{eg12.m}$:

Let $s = [s_1 \ s_2 \ \cdots \ s_{100}]$ and $N = \text{numterms}$

$$\begin{align*}
\text{bases} &= \begin{bmatrix} 1 \\ 2 \\ \vdots \\ N \end{bmatrix} [1 \ 1 \ \cdots \ 1] = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 2 & 2 & \cdots & 2 \\ \vdots & \vdots & \ddots & \vdots \\ N & N & \cdots & N \end{bmatrix} \\
\text{exponents} &= \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} [s_1 \ s_2 \ \cdots \ s_{100}] = \begin{bmatrix} -s_1 & -s_2 & \cdots & -s_{100} \\ -s_1 & -s_2 & \cdots & -s_{100} \\ \vdots & \vdots & \ddots & \vdots \\ -s_1 & -s_2 & \cdots & -s_{100} \end{bmatrix} \\
\text{bases} \ ^\wedge \ \text{exponents} &= \begin{bmatrix} 1^{-s_1} & 1^{-s_2} & \cdots & 1^{-s_{100}} \\ 2^{-s_1} & 2^{-s_2} & \cdots & 2^{-s_{100}} \\ \vdots & \vdots & \ddots & \vdots \\ N^{-s_1} & N^{-s_2} & \cdots & N^{-s_{100}} \end{bmatrix}
\end{align*}$$

**Warning:** The name of a script file cannot be the same as the name of a variable it computes.
Function Files

Function files (which are M-files) allow one to create one’s own MATLAB commands. A function file usually has these lines:

```matlab
function [yout1, yout2, ...] = funcname(xin1, xin2, ...)
% comment line 1
% comment line 2
% ...
% ...
yout1 = ...;
yout2 = ...;
...
```

Notes:

- The function M-file name and the function name that appears in the first line of the file should be identical. In reality, MATLAB ignores the function name in the first line and executes functions based on the file name.
• All comment lines immediately following the function definition line are displayed by `help funcname`.

• A single output does not require `[]`. E.g.
  
  ```
  function y = funcname(xin1,xin2).
  ```

• Input variable names given in the function definition line are local to the function. A variable name can be the name of another function. On execution, this name must be passed on as a character string, i.e., enclosed within two single right quotes.

• All variables inside a function are local and are erased after execution of the function, whereas those in a script file are left in the MATLAB workspace after execution of the script.

• Functions can have arguments, script files do not.

• Inside the body of a user-defined function, `nargin` and `nargout` return the numbers of declared input and output arguments. `nargin('funcname')` and `nargout('funcname')` return those values for the M-file function `funcname.m`. 

76
1.3 Example. Write a MATLAB function defining the following mathematical function:

\[ f(x) = \begin{cases} 
0 & \text{for } x < -2 \\
\frac{(x + 2)^2}{4} & \text{for } -1 \leq x < 0 \\
\frac{3 - x}{3} & \text{for } 0 \leq x < 3 \\
0 & \text{for } x \geq 3 
\end{cases} \]

function y = eg1_3(x)
% EXAMPLE of a piecewise-defined function
%
% (x+2)^2/4 for -2 <= x < 0,
% y = (3-x)/3 for 0 <= x < 3,
% 0 otherwise

y = zeros(size(x)); % preallocate y
i1 = find(-2 <= x & x < 0)
i2 = find(0 <= x & x < 3)
y(i1) = (x(i1)+2).^2/4;
y(i2) = (3-x(i2))/3;
1.4 Example. Write a function file to define the probability density function

\[ f(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \]

function density = normal_dist(x,mu,sigma)
% function density = normal_dist(x,mu,sigma)
% computes the probability density function for a normally
% distributed random variable with mean mu and standard deviation
% sigma. If the argument mu and sigma are omitted, then a standard
% normal distribution is assumed (mu = 0 and sigma = 1).
% if nargin == 1
  mu = 0;
  sigma = 1;
end
density = exp(-0.5* ((x-mu)/sigma).^2)/(sqrt(2*pi)*sigma);
A script file to execute this function can be:

```matlab
% Script file exec_normal_dist.m
% using the function normal_dist, with 1 and 3 input arguments
x = linspace(-4,4);
y1 = normal_dist(x);
y2 = normal_dist(x,1,0.5);
plot(x,y1,x,y2)
legend('standard normal','mu=1, sigma=0.5')
trapz(x,y1) % trapezoidal rule to evaluate integral
trapz(x,y2) % type "help trapz" to learn more
```
**Inline Functions**

A mathematical function can be defined by using the built-in function `inline`. E.g.

```matlab
>> fun = inline('abs(x).*sin(x)', 'x')
fun =
    Inline function:
    fun(x) = abs(x).*sin(x)
```

This can be done by editing a script file (named e.g. `funinline` containing the line

```matlab
fun = inline('abs(x).*sin(x)', 'x')
```

To activate this function in a MATLAB session:

```matlab
>> funinline
```

Functions Having Another Function as Input

Some MATLAB built-in functions (e.g. \texttt{fzero} and \texttt{feval}) or user-defined functions have as input another function. The input function can be defined as an inline function or by a function file and passed onto the called function as follows.

\begin{verbatim}
>> fun = inline('abs(x).*sin(x)', 'x');
>> feval(fun,pi)  % This is the same as fun(pi)
   ans =
         3.8473e-16
>> feval('eg1_3',1) % This is the same as eg1_3(1)
   ans =
         0.6667
>> feval(@eg1_3,1) % Use function handle @
   ans =
         0.6667
\end{verbatim}
1.5 Example. Consider the problem
\[
\frac{dy}{dt} = f(t, y), \quad t \in [a, b],
\]
\[
y(a) = y_0.
\]
Let
\[
a = t_0 < t_1 < t_2 < \cdots < t_N = b.
\]
The values of the solution \( y \) at \( t_n \) can be approximated by \( y(t_n) \approx y_n \) where
\[
y_{n+1} = y_n + hf(t_n, y_n), \quad n = 0, 1, \ldots, N - 1.
\]
The following code finds \( y_n \) for any given function \( f \).

------------ Run file eg1_5.m ------------

N = 20;
a = 1; b = 2; y0 = -1;
[t,Y] = Euler(’eg1_5f’,a,b,y0,N);
y = eg1_5exact(t);
function [t,y] = Euler(fstring, a, b, y0, N)

% Euler(fstring, y0, N) solves the ode
% \[ y'(t) = f(t,y) \] for \( a < t < b \),
% with initial condition
% \[ y(a) = y0. \]
% The argument \texttt{fstring} is the name of the function that computes
% \[ f(t,y). \]
% The argument \texttt{N} is the number of intervals in the grid.
% The output \texttt{t} is a row vector of length \( N+1 \) containing the grid
% points
% \[ a = t(1) < t(2) < \ldots < t(N+1) = b, \]
% where \( t(n) = a + (n-1)h \), \( h = (b-a)/N. \)
%
% The output \texttt{y} is a row vector of length \( N+1 \) containing the
% approximate values \( y(n) \) of the exact solution \( y(t(n)). \).
h = (b-a) / N;
t = a:h:b;
d = length(y0);
if size(y0,1) ~= d
    y0 = y0';
end
y = zeros(d,N+1);
y(:,1) = y0;
for n = 2:N+1
    y(:,n) = y(:,n-1) + h * feval(fstring, t(n-1), y(:,n-1));
end;

-------- File eg1_5f.m ---------------
function ydot = eg1_5f(t,y)
ydot = 1 ./ (t.^2) - y ./ t - y .^2;

-------- File eg1_5exact.m ---------------
function y = eg1_5exact(t)
y = -1 ./ t;
Subfunctions

An M-file can contain code for more than one functions. The additional functions within that file are called *subfunctions* and are only visible to the primary function and other subfunctions in the same file.

### 1.6 Example

This example is taken from MATLAB on-line help:

```
function [avg, med] = newstats(u) % Primary function
% NEWSTATS Find mean and median with internal functions.
n = length(u);
avg = mymean(u, n);
med = mymedian(u, n);

function a = mymean(v, n) % Subfunction
% Calculate average.
a = sum(v)/n;
```

function m = mymedian(v, n)    % Subfunction
% Calculate median.
w = sort(v);
if rem(n, 2) == 1
    m = w((n+1) / 2);
else
    m = (w(n/2) + w(n/+1)) / 2;
end

------------- end of file eg1_6.m -------------
Profiler

MATLAB Profiler helps you improve the performance of your M-files. The Profiler can be accessed from the Desktop menu. Run your M-file there to get a report on where the time is being spent. The Profiler can also be accessed by using the `profile` function.

1.7 Example. In Example 1.5, to get a report on the time spent on `Euler.m` the run file can be as follows:

```matlab
N = 20;
a = 1; b = 2; y0 = -1;
profile on
[t,Y] = Euler('eg1_5f',a,b,y0,N);
profile report
profile plot
profile done
y = eg1_5exact(t);
```
If Statements

If statements provide a mechanism for branching in a computer program, i.e. they make it possible to execute selectively on the basis of information generated while the program is running. In MATLAB, an if statement takes the following form:

```matlab
if <logical expression 1>
   <statement 1>
elseif <logical expression 2>
   <statement 2>
else
   <statement 3>
end
```

Zero or more than one elseif branch is permitted. The else branch can also be omitted.
1.8 Example. Here is a modified version of `eg1_1.m`:

```matlab
% Script file to plot sin(x) and cos(x), a <= x <= b.
a = input('a = '); % prompt for user input
b = input('b = '); % prompt for user input
if a >= b
    error('a must be strictly less than b')
end
x = [a:0.1:b];
y1 = sin(x);
y2 = cos(x);
plot(x,y1,x,y2)
legend('sin x', 'cos x')
```

The `error` built-in function displays the message, exits the M-file, and returns control to the keyboard.
1.9 Example. Here is a modified version of `eg1_2.m`:

```matlab
% Script file to plot the Riemann zeta function on [a,b]
disp('Plot zeta(s) for a < s < b where a > 1')
a = input ('a = ');
b = input ('b = ');
if a <= 1
    error('a must be strictly greater than 1')
elseif b <= a
    error('b must be strictly greater than a')
end
numterms = input('Number of terms = ');
s = linspace(a,b);
bases = [1:numterms]' * ones(size(s));
exponents = -ones(numterms, 1) * s;
y = sum(bases .^ exponents);
plot(s,y)
xlabel('s')
ylabel('zeta(s)');
```
Loops

A loop is a programming construct in which a group of statements is executed repeatedly. In MATLAB, loops are of two kinds: for loops and while loops.

A for loop has the form

```
for n = 1:N  % a value for N has to be pre-defined
    <statements>
end
```

A while loop has the form

```
while <logical expression>
    <statement>
end
```
1.10 Example. The output from

```matlab
for n = 1:5
    disp([num2str(n) '^2 = ' num2str(n^2)])
end
```

is

\[
\begin{align*}
1^2 &= 1 \\
2^2 &= 4 \\
3^2 &= 9 \\
4^2 &= 16 \\
5^2 &= 25
\end{align*}
\]

`num2str` converts numbers to a string.
1.11 Example. To find the unique whole number $K$ such that

$$(K - 1)^2 \leq 1073 < K^2,$$

we could use the following code:

```matlab
for n = 1:40
    if n^2 > 1073
        break
    end
end
disp(['K = ' num2str(n)])
disp([num2str(n-1) '^2 = ' num2str((n-1)^2)])
disp([num2str(n) '^2 = ' num2str(n^2)])
```

The output is

$K = 33$

$32^2 = 1024$

$33^2 = 1089$
1.12 Example. The for loop in the last example could be replaced with the following while loop:

\[
\begin{align*}
n &= 1; \\
\text{while } n^2 < 1073 & \quad n = n + 1; \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{disp(['K = ' num2str(n)])} \\
\text{disp(['num2str(n-1) \times2 = ' num2str((n-1)^2)])} \\
\text{disp(['num2str(n) \times2 = ' num2str(n^2)])}
\end{align*}
\]

Obviously, the <statements> in a while loop must modify the value of the <logical expression>, so that the latter eventually becomes false and the loop terminates. There is always the danger of an infinite loop, in which the <logical expression> never becomes false, but is always true, perhaps due to programmer not considering all possibilities. The for ... break is safer.
Recursion in MATLAB

1.13 Example. Legendre polynomials are recursively defined as

\[
P_0(x) = 1, \quad P_1(x) = x, \\
P_{j+1}(x) = \frac{2j+1}{j+1} x P_j(x) - \frac{j}{j+1} P_{j-1}(x), \quad j \geq 1
\]

function y = rlegendre(n,x)
% y = rlegendre(n,x) -- Legendre polynomial of degree n
if n > 1
  y = (2*n-1)*x.*rlegendre(n-1,x)/n ...
    - (n-1)*rlegendre(n-2,x)/n;
elseif n == 1
  y = x;
elseif n==0
  y = ones(size(x));
else
  error('rlegendre(n,x): n must be non-negative')
end
Efficiency

Performance of your MATLAB code can be improved by

- Vectorising loops;
- Preallocating arrays.

Vectorisation

MATLAB is a matrix language designed for vector and matrix operations. To speed up your M-file code you should use vectorising algorithms that take advantage of this design. Whenever possible, convert for and while loops to vector or matrix operations.

Test the following code on your system. In the code, \texttt{cputime} is used to compute the difference in time. \texttt{cputime} is a built-in function that returns the CPU time (in seconds) that has been used by the MATLAB process since MATLAB started.
1.14 Example. We sum the first $N$ terms ($N = 100,000$ or $N = 1,000,000$) of $\sum_{n=1}^{\infty} \frac{1}{n^2}$ in two ways:

N = 1000000;
% Use vector
t0 = cputime;
s = sum([1:N] .^ (-2));
t1 = cputime;
vec_time = t1-t0
% Use loop
% Use loop
% Use loop
% Use loop
% Use loop
% Use loop
t0 = cputime;
s = 0;
for n = 1:N
    s = s + n^(-2);
end
\begin{verbatim}
t1=cputime;
loop_time = t1-t0
ratio = loop_time/vec_time
\end{verbatim}
Run the script above on your machine and observe the difference in vec_time and loop_time. The difference is much bigger for MATLAB Version 6 and before. In general, it is safer to avoid loops whenever possible. For more information, go to http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/ then click on Improving Performance and Memory Usage.

Another example of a code that can be vectorised:

**1.15 Example.** The Hilbert matrix $A = [a_{ij}]$ has entries

$$a_{ij} = \frac{1}{i + j - 1}, \quad 1 \leq i \leq n, \; 1 \leq j \leq n.$$
An easy coding for setting up $A$ is

function $A = \text{Hilmat}(n)$
$A = \text{zeros}(n,n)$
for $i = 1:n$
    for $j = 1:n$
        $a(i,j) = 1/(i+j-1)$;
    end
end

The vectorised calculation (used in MATLAB built-in file $\text{hilb.m}$) is

function $A = \text{hilb}(n)$
$J = 1:n$;
$J = J(\text{ones}(n,1),:)$; \hspace{1cm} % J(i,j)=j
$I = J’$; \hspace{1cm} % I(i,j)=i
$E = \text{ones}(n,n)$; \hspace{1cm} % E(i,j)=1
$H = E ./ (I+J-1)$;
Preallocating Arrays

A loop that incrementally increases the size of a data structure each time through the loop can adversely affect performance and memory use. Repeatedly resizing arrays often requires that MATLAB spend extra time looking for larger contiguous blocks of memory and then moving the array into those blocks.

If you cannot vectorise a piece of code, you can make your loops go faster by preallocating any vector in which output results are stored.
1.16 Example. Compare the following two pieces of code:

% Without preallocation
clear x
for i=1:n
    for j=1:n
        x(i,j)=(i+j)/n;
    end
end

% With preallocation
clear x
x=zeros(n,n); % preallocate x
for i=1:n
    for j=1:n
        x(i,j)=(i+j)/n;
    end
end