Appendix A

Getting started with Octave and MATLAB

This book uses the Octave / MATLAB computing language and commands. While we expect students to have some familiarity with it, we provide a brief review of some basic features and syntax here. For more information, there is an enormous amount of free literature available that can be found with internet searches. Probably the best place to look is on the MATLAB website http://www.mathworks.com/help/matlab/index.html. The book Numerical Methods for Engineers and Scientists by Gilat and Subramaniam also has a nice MATLAB introduction in their appendix.

A.1 Basic operations

The most basic way to use MATLAB is for simple calculations. The symbols used for scalar operation can be found in table A.1.

Operation	**Symbol**	**Operation**	**Symbol**
Addition	+	Division	/
Subtraction	−	Exponentiation	^
Multiplication	*		

TABLE A.1. MATLAB Operations

Mathematical expressions can be typed directly into the command window. For example:

```
>> 7 + 6/2
ans =
     10
>> (7+6)/2 + 27^(1/3)
ans =
     9.5000
```

Numerical values can be assigned to variables, which can then be used in mathematical expressions and functions:

```
>> a = 11
a =
     11
```

```
>> B = 3;
>> C = (a - B) + 40 - a/B*10
C =
     11.3333
```

MATLAB also includes many built-in functions. These functions can take numbers, variables, or expressions of numbers and variables as arguments. Examples of common MATLAB functions:

- Square Root:

```
>> sqrt(64)
ans =
     8
```

- Exponential (e^x):

```
>> exp(2)
ans =
     7.3891
```

- Absolute value:

```
>> abs(-23)
ans =
     23
```

- Natural logarithm (Base e logarithm):

```
>> log(1000)
ans =
     6.9078
```

- Base 10 logarithm:

```
>> log10(1000)
ans =
     3.0000
```

- Sine of angle x (x in radians):

```
>> sin(pi/3)
ans =
     0.8660
```

Operator	Description	Operator	Description
<	Less than	>=	Greater than or equal to
>	Greater than	==	Equal to
<=	Less than or equal to	~=	Not equal to

TABLE A.2. Relational Operators

Operator	Name	Description
&	AND	Operates on two operands. If both are true, yields true (1). Otherwise, yields false (0).
|	OR	Operates on two operands. If at least one is true, yields true (1). If both are false, yields false (0).
~	NOT	Operates on one operand. Yields the opposite of the operand.

TABLE A.3. Logical Operators

Relational and logical operators can be used to compare numbers, and are given in tables A.2 and A.3. When using logical operators, a nonzero number is considered true (1), and a zero is false (0).

```
>> 6 > 9
ans =
     0
>> 4 == 8
ans =
     0
>> 3&10
ans =
     1
>>  a = 6|0
a =
     1
>> ~1
ans =
     0
```

The default output format in MATLAB is 'short' with four decimal digits. However, this may be changed by the user using the "format" command. Several available formats are listed in table A.4. Examples are given below.

```
>> format short
>> pi

ans =

     3.1416

>> format long
>> pi

ans =

     3.141592653589793

>> format shorte
>> pi

ans =

     3.1416e+00
```

Command	**Description**
format short	Fixed point with four decimal digits for $0.001 \leq number \leq 1000$. Otherwise, display format for shorte.
format long	Fixed point with 16 decimal digits for $0.001 \leq number \leq 1000$. Otherwise, display format for long e.
format shorte	Scientific notation with four decimal digits

TABLE A.4. Display Format

A.2 Arrays

An array is the form utilized in MATLAB to store data. The simplest array (a vector) is a row or column of numbers. A matrix is a more complicated array, which puts numbers into rows and columns.

A vector is created in MATLAB by assigning the values of the elements of the vector. The values may be entered directly inside square brackets:

```
>>var_name = [number number . . . number]
```

In row vectors, entries are separated by a comma or a space. In a column vector, they are separated by semicolons. Vectors whose entries have equal spacing can be specified using colon notation via

```
>>variable_name = m:q:n
```

where m is the first entry, q is the spacing between entries, and n is the last entry. If q is omitted, it is assumed to be equal to one. Several examples of constructing vectors are given below.

```
>> y = [1 19 88 2000 ]
y =
     1     19    88   2000
>> p = [1; 4; 8]
p =
     1
     4
     8
>> x = [3:2:11]
x =
     3     5     7     9     11
>> x=[-2:1]
x =
    -2    -1     0     1
```

A matrix can be created row by row, where the elements in each row are separated by commas or spaces. Note that all rows in a matrix must contain the same number of elements. Examples of creating matrices are:

```
>> A = [1 2 3; 4 5 6; 7 8 9]

A =

     1     2     3
     4     5     6
     7     8     9

>> a = 1; b = 2; c = 3;
>> B = [a b c; 2*b b+c c-a]

B =

     1     2     3
     4     5     2
```

Arrays and their elements may be addressed in several ways. For a vector named ve, $ve(k)$ is the element in the kth position (counting from 1). For a matrix named ma, $ma(k,p)$ is the element in the kth row and pth column. Single elements in arrays can be reassigned and changed, or used in mathematical expressions.

```
>> V = [1 8 4 2 5]
```

```
V =

     1     8     4     2     5

>> V(3)

ans =

     4

>> V(3)+V(4)*V(1)

ans =

     6

>> A = [1 2 3; 4 5 6; 7 8 9]

A =

     1     2     3
     4     5     6
     7     8     9

>> A(1,3)

ans =

     3

>> A(3,1)

ans =

     7
```

MATLAB allows you to not only address individual entries in matrices, but also sub-matrices. To do this, the row and column argument can contain a vector of indices (1 being the first element) or a colon ":" for the whole row or column. Using the construction "m:q:n" is very useful here. Note that the indices do not need to be sorted and will return the corresponding entries. Some examples are below.

```
>> MAT = [3 11 6 5; 4 7 10 2; 13 9 0 8]
MAT =
     3    11     6     5
     4     7    10     2
    13     9     0     8
```

- The whole column of column 2 and 4:

  ```
  >> MAT(:,[2,4])
  ```

```
ans =
    11     5
     7     2
     9     8
```

- Row 2, columns 1 to 3:

```
 >> MAT(2,1:3)
ans =
     4     7    10
```

- Row 3, 1, and 3 again (in that order) and the whole column:

```
>> MAT([3,1,3],:)
ans =
    13     9     0     8
     3    11     6     5
    13     9     0     8
```

This advanced indexing can also be used for vectors (of course only having one argument):

```
>> v = [4 1 8 2 4 2 5]
v =
    4     1     8     2     4     2     5
>> u = v(3:7)
u =
     8     2     4     2     5
```

Several built-in functions for arrays are listed in table A.5.

A.3 Mathematical Operations with Arrays

Addition, subtraction, and multiplication of arrays follow the rules of linear algebra. When a scalar is added to or subtracted from an array, it is applied to all elements of the array.

```
>> u = [1 2 3]; v=[4 5 6]

v =

     4     5     6

>> v = [1 2 3]; v=[4 5 6];
>> w = u+v

w =

     5     7     9
```

Command	Description
`length(x)`	Returns the number of elements in vector x
`size(A)`	Returns a row vector of two elements, $[m, n]$, m and n are the size $m \times n$ of A.
`zeros(m,n)`	Creates a zero matrix of size $m \times n$.
`ones(m,n)`	Creates a matrix of ones of size $m \times n$.
`eye(n)`	Creates an identity matrix of size $n \times n$.
`mean(x)`	Returns the mean value of all entries in a vector x
`sum(x)`	Returns the sum of the entries in a vector x
`sort(x)`	Arranges the elements of a vector x in ascending order
`det(A)`	Returns the determinant of a square matrix A
`A'`	Returns the transpose of the matrix or vector A

TABLE A.5. Built-in functions for arrays

```
>> u*v
??? Error using ==> mtimes
Inner matrix dimensions must agree.

>> u*v'

ans =

    32

>> u'*v

ans =

     4     5     6
     8    10    12
    12    15    18

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

A =

     1     2
     3     4
     5     6
```

```
>> B = [7 8;9 10]

B =

     7     8
     9    10

>> A*B

ans =

    25    28
    57    64
    89   100
```

Element-by-element multiplication, division, and exponentiation of two arrays of the same size can be carried out by adding "." before the operator. Examples of element-by-element operations:

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

A =

     1     2     3
     4     5     6

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

B =

     7     8     9
    10    11    12

>> A.*B

ans =

     7    16    27
    40    55    72

>> A./B

ans =

   1.4286e-01   2.5000e-01   3.3333e-01
   4.0000e-01   4.5455e-01   5.0000e-01

>> A.^3

ans =

     1     8    27
    64   125   216
```

A.4 Script Files

A script file (or program) is a file that contains a series of commands, executed in order when the program is run. If output is generated by these commands, it appears in the command window. A script file can be thought of as nothing more than a series of commands that are entered in sequence on the command line.

Script files are edited in the Editor Window. To make a new script file, select "New", followed by "M-file" from the "File" menu. A script file must be saved before it can be executed. Once saved, it can be run by entering the file name into the Command Window, or directly from the Editor Window.

Variables utilized in a script file can be set and changed in the file itself, or in the Command Window, either before the program is run, or as input during execution.

A.5 Function Files

A function file is a program in MATLAB that is used a function. It is edited and created in the Editor Window like a script file, but its first line must be of the following form:

```
function [outputArguments] = function_name(inputArguments)
```

- `function` must be the first word in the function file.
- If there are more than one input or output argument, they are separated by commas.
- All variables in a function file are local; they are defined and recognized only inside the file.
- Closing the function with `end` as the last instruction in the file is optional.

The following is a simple example of a function:

```
function [y] = myfun(x)
  y = exp(x) - x^2;
end
```

This function returns $e^x - x^2$ for any given x. It can be called by

```
>> myfun(1)

ans =

    1.7183
```

You can turn the function defined as above into a handle that you can pass to a different function (this will be important in some of the chapters) with the expression `@myfun`.

A.5.1 Inline functions

It is sometimes useful to define a function "inline" (not in a separate file as explained above), if the function is a mathematical formula (basically if you can write it in a single expression). The following code creates a function $f(x, y) = x + 2y + 1$ with two arguments and assigns it to the handle `myfunction`:

```
>> myfunction = @(x,y) x+2*y+1;
```

and you can evaluate the function like this:

```
>> myfunction(2,0)
    3
```

A third option is to create an inline function like this:

```
>> myfunction2 = inline('sin(x)-exp(x)');
```

Here MATLAB automatically detects that `x` is the argument of the function. This does not work reliably with more than one argument. This is why we prefer the syntax above.

A.5.2 Passing functions to other functions

We will need to pass functions to other functions many times in this semester. An example would be a function `integrate(f,a,b)` that computes (approximates) $\int_a^b f(x)$. You have two options for this:

1. Define the function inline:

   ```
   >> myfunction = @(x) exp(x)-x*cos(x);
   >> integrate(myfunction, a, b)
   ```

2. Or by creating a file `myfun1.m` with the contents

   ```
   function y = myfun1(x)
   y = exp(x)-x*cos(x);
   end
   ```

   and then call

   ```
   >> integrate(@myfun1, a, b)
   ```

Note that the `@` symbol is required to turn the function defined in the .m file into a handle.

A.6 Outputting information

MATLAB has several commands used to display output, including `disp` and `fprintf`. The `disp` command accept variable names or text as string as input:

```
disp(x)
disp('text as string')
```

The `fprintf` command has the form:

```
fprintf('text as string \%5.2f additional text\n', x)
```

The "%5.2f" specifies where the number should go, and allows the user to specify the format of the number.

A.7 Programming in MATLAB

In programming in MATLAB, some problems will require more advanced sequences of commands. Conditional statements allow for portions of code to be skipped in some situations, and loops allow for portions of code to be repeated.

A conditional statement is a command which MATLAB decides whether execute or skip, based on an expressed condition. The "if-end" is the simplest conditional statement. When the program reaches the "if" statement, it executes the commands that follow, if the condition specified is true. It then follows these commands down to the "end" statement, and continues executing the program. If the conditional statement is false, it skips the commands between "if" and "end" and continues with commands after the "end" statement.

Another conditional structure is "if-else-end". This provides MATLAB the means to choose one of two groups of commands to execute. If the condition associated with the "if" statement is satisfied, that code is executed, and the program skips to the code after the conditional structure (skipping the "else" section). If the it is not satisfied, the program skips this section of code and instead executes the commands in the "else" section.

If it is necessary for MATLAB to choose one of three or more groups of commands to execute, a similar structure is followed, with "elseif" statements added between the first "if" section and the last "else" section of commands.

Loops are another way to alter the flow of commands in a MATLAB program. Loops allow a group of commands to be repeated. Each repetition of code inside a loop is often referred to as a pass.

The most common type of loop in MATLAB is the "for-end" loop. These loops dictate the the group of commands inside the loop are run for a predetermined number of passes. The form for such a loop is:

```
for k = f:s:t
```

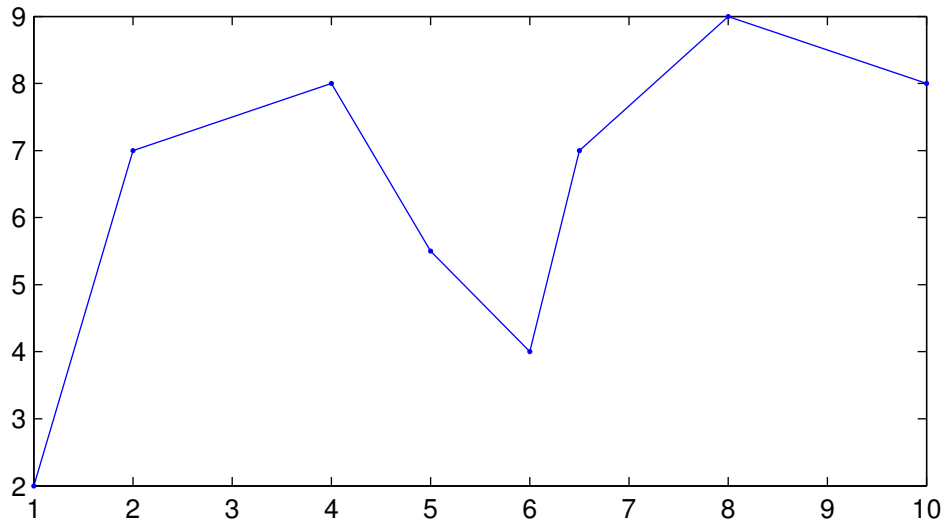

FIGURE A.1. A plot of data points

```
    ...
    ...
end
```

In the first pass, k=f. After the commands up until "end" are executed, the program goes back to the "for" and the value for k is changed to k+s. More passes are done until k=t is reached. Then, the program goes to the code past the "end" statement. For example, if k = 2:3:14, the value of k in the passes is 2, 5, 8, 11, and 14. If the increment value s is omitted, the default value is 1. Note that any vector can be given to the right of the equal sign and the loop will iterate over every element of the vector.

A.8 Plotting

MATLAB has a variety of options for making different types of plots. The simplest way to make a two-dimensional plot is using the plot command: "plot(x,y)". The arguments x and y are each vectors of the same length. The x values will appear on the horizontal axis and the y values on the vertical axis. A curve is created automatically connecting the points. The following code is an example of a basic plot, which is shown in figure A.1.

```
>> x = [1 2 4 5 6 6.5 8 10];
>> y = [2 7 8 5.5 4 7 9 8];
>> plot(x,y,'.-')
```

With the "plot" command, there are additional arguments which can be used to specify line color and marker type, if markers are desired. The command takes the form "plot(x,y,'line specifiers')". A list of there arguments is given in A.8.

Line Style	Specifier	Line Style	Specifier
solid (default)	-	dotted	:
dashed	–	dash-dot	-.

Line Color	Specifier	Line Color	Specifier
red	r	blue	b
magenta	m	black	k
green	g	cyan	c
yellow	y	white	w

Marker	Specifier	Marker	Specifier	Marker	Specifier
plus sign	+	asterisk	*	square	s
circle	o	point	.	diamond	d

TABLE A.6. Line and marker specifers

A.9 Exercises

1. Create a vector of the even whole numbers between 29 and 73.
2. Create a column vector with 11 equally spaced elements, whose first element is 2 and whose last is 32.
3. Let `x = [2 5 1 6]`.
   a. Add 16 to each element
   b. Add 3 to just the odd-index elements
   c. Compute the square root of each element
   d. Compute the square of each element
4. Given `x = [3 1 5 7 9 2 6]`, explain what the following commands "mean" by summarizing the net result of the command.
   (a) `x(3)`