MATLAB Primer

Brian Storey

with edits by Brian Hunt, Mark Shattuck, Shelby Wilson

version of 18 Jul 2016

MATLAB is a software tool and programming environment that has become commonplace among scientists and engineers. It is a useful programming language for scientific computing, data processing, and visualization of results. Many useful mathematical functions and graphical features are integrated with the language. MATLAB is a powerful language for many applications as it has high-level functionality for science and engineering applications coupled with the flexibility of a general-purpose programming environment.

This chapter does not provide a complete description of MATLAB programming, but rather an introduction to the basic capability and the basic syntax. As the title says, this chapter is a primer, not complete product documentation. In this chapter we provide some simple examples to get the student started and familiar with programming in the environment. There is extensive documentation on the Mathworks web site as well as the help feature built into the MATLAB environment. All the commands presented in this chapter have detailed explanations of the functionality via the built in help. This chapter is also not a general introduction to programming.

Most of the functionality will be presented through very simple (trivial) examples, just to present the basic syntax and capability. If you are an experienced programmer, you should pay particular attention to the sections on array operations and mathematics as this feature is one of the biggest differences with traditional programming languages.

To get the most out of this chapter you should read the notes with MATLAB open and type each command and write each program as you read this primer. Make some variations so that you understand each command and program. There are problems in the chapter meant to give you practice programming in MATLAB. Depending on your past programming experience these activities may vary in difficulty.

1 GETTING STARTED

The focus of the MATLAB graphical user interface (GUI) is the command window. In the command window you can type MATLAB instructions, the instructions will be executed instantaneously and the result will be displayed in the window. You may also store a sequence of commands in a file so that you may run a long sequence of instructions. We will get to that later, but we will start with running all our instructions in interactive mode. In the MATLAB command window the characters **>>** indicate the **prompt**. The prompt is waiting for you to type a command to be executed. The notation for commands in this primer is

```
>> command;
```

which simply means you should type `command;` at the MATLAB prompt. If a semi-colon is placed at the end of a command then all output from that command is suppressed and you will get the next prompt after execution. If you do not type the semi-colon, any output from the command will be displayed. Many times in this primer we use comments to help explain the purpose of each command. Comments are marked by the percent sign, MATLAB ignores anything to the right of percent sign. The comments allow the reader to follow written programs more easily. We will make use of comments to clarify the intent, the format will follow

```
>> command;  %% This is a comment to describe this command
```

Finally, any command presented in this primer has further information via the integrated help manual. To find details of commands that we present, simply type

```
>> help command
```

which will provide information on the inputs, outputs, usage, and functionality of the command. Most commands have a variety of options that may be invoked and these are explained in the help. A listing of commands sorted by functionality can be found by typing help.

2   CALCULATOR

The simplest function that MATLAB can perform is that of a very expensive calculator. You may go to the command window and try typing in basic mathematical operations such as the following three examples,

```
>> 4*9

ans =
    36

>> 12/8

ans =
    1.5000

>> 12+31 - (9*(2-9))/18

ans =
    46.5000
```

After hitting the return key following each command you will receive the answers shown. Note the basic operators are + for addition, - for subtraction, * for multiplication, and / for division. The order of operations follows the convention of basic algebra such that if you type 8*4 + 3 or 8*(4 + 3) you will get the appropriate answers. MATLAB understands most common mathematical operations such as trigonometric functions, exponentials, complex numbers, and common constants. Try typing some of the following operations at the command line.

```
>> pi

ans =
    3.1416

>> cos(pi/6)

ans =
    0.8660

>> 2^3

ans =
    8
```

Some common mathematical expressions are `sqrt(x)` for $\sqrt{x}$, `log(x)` for the base e logarithm of x, `log10(x)` for the base 10 logarithm of x, `cos(x)` for $\cos x$, and `exp(x)` for e^x. You can see what other elementary functions are known to MATLAB by typing `help elfun`. For the usage of any of these commands you can type, for example, `help acos` to get help on the usage of the inverse cosine function. The value you provide the function is called the **argument** and always must be included in parentheses after the function name.

3   VARIABLES

We can start to make the calculator a little more useful if we store data into variables so that we can collect the results of lengthy calculations in a straightforward manner. The following commands are examples of **assignment statements**.

```
>> x = 4.8;
>> y = 7;
>> z = x*y;
```

These commands define three variables, x, y and z, and assign their values 4.8, 7 and 33.6 respectively. If we type the commands as listed above, we will simply receive the prompt after hitting return (because we included the semicolon). To understand what MATLAB has done, type whos. The whos command lists all variables that are stored in the local **workspace**, or memory. After typing whos, we will see that MATLAB has created the variables x, y, and z and stored them in the workspace.

```
>> whos
  Name      Size            Bytes  Class     Attributes

  ans       1x1                 8  double
  x         1x1                 8  double
  y         1x1                 8  double
  z         1x1                 8  double
```

To remove all variables from the workspace use the command clear. If you now execute the whos command you will find that the workspace is empty. You may also selectively remove a variable, say x, from the workspace with clear x. There is also a graphical window that displays the workspace interactively in a side window.

There is no difference between the way integers or real numbers are used in MATLAB. All variables thus far are listed as an eight byte, 1x1 element, double array. In many programming languages, such as C, there is a difference between mathematics and storage of integer and real numbers, but not so in MATLAB. In the whos listing, the 1x1 refers to the fact that the variables are single entry of a table of numbers. We will demonstrate in short order that MATLAB is efficient in working with tables, or arrays, of numbers. MATLAB always stores numbers as arrays, even if there is only one entry. To view the values assigned to the variables, simply type the variable name with no semi-colon.

```
>> z

z =

   33.6000
```

You can use almost any name you like for variables, but there are restrictions. You must start each variable name with a letter, but numbers may follow (a2 is fine, 2a is not). You may not use spaces to separate words in a long variable name, but you may use the underscore (long variable name is fine). You may not use other symbols in the variables name. MATLAB also has many **keywords** that are part of the language and may not be used for variable names. These keywords include: for, if, else, elseif, while, function, return, continue, global, persistent, break, case, otherwise, try, and catch. If you use these names in variables you will receive an error immediately.

You may use variable names that match MATLAB function names, but this should always be avoided. Since MATLAB has a special place for names such as pi and cos you should not use these as a variable name, though you will be allowed to do so. If you use pi as a variable name and assign a value to it, it will no longer be available as 3.14159. If you issue the command cos = 10 you will no longer be able to use the cosine function until you clear the workspace.

4   ONE-DIMENSIONAL ARRAYS

In the last section we stored single numbers in variables. Often we have an entire series of data that we would like to manipulate rather than a single number. One of the advantages of MATLAB is that when we create lists of data, MATLAB can perform mathematical operations on those entire lists. Such lists are often called **arrays** or **vectors**. We demonstrate the manipulation of these arrays by a series of simple examples.

When dealing with arrays the orientation is important. To create a column oriented and a row oriented array we can type the following commands.

```
>> x = [1; 2; 3]  %% column

x =

     1
     2
     3

>> y = [1 2 3]  %% row

y =
     1     2     3

>> whos
  Name      Size            Bytes  Class      Attributes

  x         3x1                24  double
  y         1x3                24  double
```

The size of the data arrays are `3x1` and `1x3` for `x` and `y` respectively. The size `3x1` means that the data has three rows and one column. Note that spaces between elements create rows and semicolons create columns. To find what information is contained in the first element of the array you can use the **index** enclosed in parentheses to denote the position in the array,

```
>> x(1)

ans =
     1
```

which returns the first element of the array `x`. You can also access a range of elements in the array, for example,

```
>> x(1:2)

ans =
     1
     2
```

returns the elements of the array ranging from the first and second position.

One fundamental advantage of MATLAB is the ability to perform mathematical operations on entire arrays of data with one command. Using the basic mathematical operations we can multiply all the numbers in the array by a constant, add a constant to each element of the array, or add two arrays element by element. The following commands demonstrate these concepts,

```
>> x*4  %% multiply each number in the array by 4

ans =
     4
     8
    12

>> x+4  %% add 4 to each number in the array

ans =
     5
     6
     7

>> x+x/5  %% sum of x and 1/5 of x
```

```
ans =

    1.2000
    2.4000
    3.6000
```

We cannot add x and y since they are oriented in different directions, one is a row array and the other a column array, and MATLAB will return an error. In order to add arrays they need to be the same size and the same orientation.

```
>> x+y
Error using +
Matrix dimensions must agree.
```

To change the orientation of an array use the function transpose or the shorthand ' (the latter also takes the complex conjugate if the arrays contain complex numbers).

```
>> y'  %% flip the orientation of y

ans =

     1
     2
     3

>> x+y'  %% add x and y, flipping y so that x and y have same orientation

ans =

     2
     4
     6

>> x+transpose(y)  %% add x and y, flipping y so x and y have same orientation

ans =

     2
     4
     6
```

In order to perform an element by element multiplication (division) of two vectors, you must use the "dot" operations. The usual multiplication (division) sign will fail.

```
>> x*x
Error using *
Inner matrix dimensions must agree.
```

This error refers to a topic that is out of the scope of this course. MATLAB is a linear algebra package that performs matrix vector operations quite succinctly. In linear algebra, a multiplication means something somewhat different when dealing with one and two-dimensional arrays of data. We do not want to deal with linear algebra in this course. In order to force an element by element multiplication operator on a list of numbers you need the "dot" operator. The .* (dot multiply) operation performs x.*y = [x(1)*y(1); x(2)*y(2); x(3)*y(3)]. Observe the result of this series of examples.

```
>> x.*x  %% multiply x by itself, element by element

ans =

     1
     4
     9
```

```
>> x.^2  %% take x to the second power, same as last command

ans =
     1
     4
     9

>> x./x  %% divide x by itself, results in all ones

ans =
     1
     1
     1
```

The reader with programming experience might notice this feature to be a time saver. In many languages, such as C, when you wish to multiply each element of an array by a constant you must write a loop that explicitly multiplies each element of the array. Such whole array notation is extremely useful and will be used frequently in our MATLAB programs.

There are numerous methods to generate regular one-dimensional arrays. A simple way to create an array of integers between zero and five is, `t = [0:5]`. Arrays of equally spaced numbers can also be created using the command `linspace`, which requires three arguments: the start value, the last value, and the number of points. A variety of arrays of equally spaced points are generated as follows. Study the notation and experiment with your own values. These commands are quite useful and commonly used throughout this course.

```
>> t = [0:5]  %% array runs from 0 to 5 in increments of 1

t =
     0     1     2     3     4     5

>> t = [0:.1:.5]  %% array runs from 0 to .5 in increments of .1

t =
     0    0.1000    0.2000    0.3000    0.4000    0.5000

>> t = linspace(0,.5,6)  %% 6 element array from 0 to .5

t =
     0    0.1000    0.2000    0.3000    0.4000    0.5000

>> t = [0:5]/5  %% 6 element array from 0 to 1

t =
     0    0.2000    0.4000    0.6000    0.8000    1.0000
```

5   PLOTTING

One of the most powerful features of MATLAB is that you have the opportunity to easily integrate graphics into your calculations, analysis, and simulations. In this section we will demonstrate some of the basic plotting capabilities. A simple example is to create a list of known points and evaluate a function (sine in this example) at these sample points and plot the result. The basic usage of the plot command is `plot(t,y)`, which generates a plot of `t` versus `y`. The arrays `t` and `y` should be the same length and orientation. The commands `xlabel` and `ylabel` are used to add text to the axes. When plotting data, you should get into the habit of always labeling the axes. The following set of commands will create generate a list of sample points, evaluate the sine function, and then generate a plot of a simple sinusoidal wave putting axis labels on the plot.

```
>> t = linspace(0,1,200);   %% create 200 points from 0 to 1
>> f = sin(2*pi*t);         %% evaluate sin(2*pi*t)
>> plot(t,f);               %% create the plot
>> xlabel('t');             %% add axis labels
>> ylabel('sin(2 pi t)');   %% add axis labels
```

You can create other interesting plots such as the unit circle,

```
>> t = linspace(0,1,200);   %% create 200 points from 0 to 1
>> plot(cos(2*pi*t),sin(2*pi*t));
>> xlabel('cos')
>> ylabel('sin')
```

The appearance of plots can be manipulated either via simple commands embedded in your programs or you can edit plots directly through the graphical user interface. You may click on the axes or the lines and change numerous properties such as the line styles, the axis, the font sizes, colors, etc. Some of the more common controls are to add different line-styles to distinguish between curves and controlling the axis limits on graphs, and adding legends which are shown in the example below.

```
>> t = linspace(0,1,200);     %% create 200 points from 0 to 1
>> clf;                       %% clears the current figure
>> plot(t,exp(-t));           %% plots e^(-t)
>> hold on                    %% holds graphics for subsequent plots
>> plot(t,exp(-3*t),'r--');   %% plots e^(-3t) with red dashed line
>> plot(t,exp(-5*t),'k-.');   %% plots e^(-5t) with black dash-dot line
>> axis([ 0 1 0 1]);          %% set the axis limits between 0 and 1
>> xlabel('t');               %% x-axis label
>> legend('e^{-t}','e^{-3t}','e^{-5t}')   %% adds a legend
```

Embedding commands in your programs to change the appearance of plots is most important when the task is repetitive and you may want to generate many plots that all look the same.
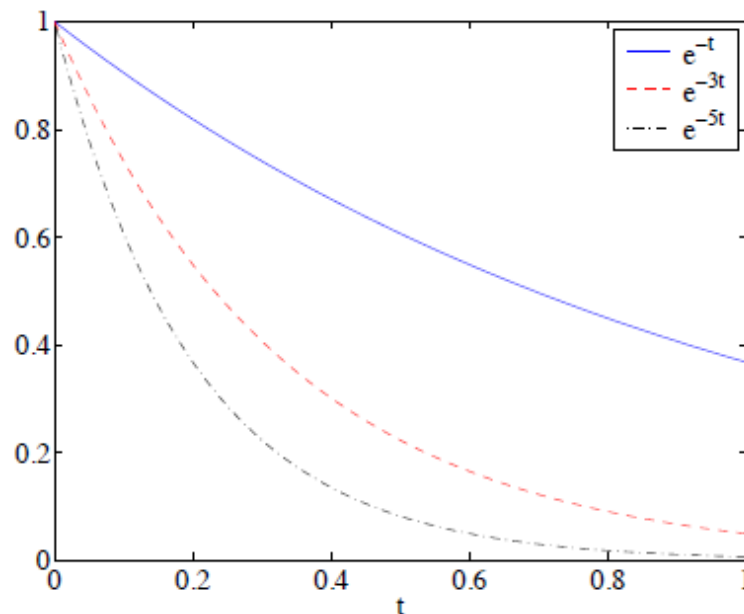

Figure 1: The resulting graph for the commands typed in the example above.

6 TWO-DIMENSIONAL ARRAYS

MATLAB also has syntax for dealing with two dimensional arrays, or tables of numbers. Many of the same rules from one-dimensional data are directly extended. To create a simple array, three rows by three columns, you can enter directly

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

a =
     1     2     3
     4     5     6
     7     8     9
```

Spaces indicate a change in column and semicolons a change in row. To access any particular element of this data simply use the notation `a(1,3)` to access the first row, third column. To extract the second column of a you can use the notation `a(:,2)` (meaning second column all rows)

```
>> a(:,2)

ans =
     2
     5
     8
```

or to extract the last row use the notation a(3,:) (meaning third row all columns)

```
>> a(3,:)

ans =
     7     8     9
```

It is also easy to extract a smaller two dimensional array. For example, the sub-array that consists of the first and third rows and the second and third column is generated via

```
>> a([1 3],2:3)

ans =
     2     3
     8     9
```

The notation for mathematical operations on the two-dimensional array of data is the same as with the one- dimensional data. As before you can perform operations such as `3*a`, `a-5`, or `a/10`, and the result will be that all elements of a are multiplied by 3, have 5 subtracted, and are divided by 10, respectively. For example,

```
>> 3*a

ans =
     3     6     9
    12    15    18
    21    24    27
```

To perform multiplication, division, or powers on an element by element basis requires the "dot" operators. For example to multiply the data element by element by itself try `a.*a` and `a.^2` which should provide the same result.

```
>> a.^2

ans =
     1     4     9
    16    25    36
    49    64    81
```

The operation a./a will return an array of all ones.

```
>> a./a

ans =
     1     1     1
     1     1     1
     1     1     1
```

Care must be taken when using the multiply operation as you can easily introduce errors. When the array `a` has the same number of columns as there are rows in the array `b`, the the operation `a*b` is defined and will return an answer. The result you obtain the matrix-matrix product of the two data arrays. This product is very different than the element by element squaring of the array. Matrix-matrix products are covered in a standard linear algebra course.

A two-dimensional extension to the `linspace` command is `meshgrid`. The command `meshgrid` transforms the one-dimensional vectors into two-dimensional arrays that can be used for evaluating two dimensional functions An example of the use of this function is provided below

```
>> clf;                        %% clears the current figure
>> x = linspace(0,2*pi,100);   %% x-axis ranges from 0 to 2 pi
>> y = linspace(0,4*pi,100);   %% y-axis ranges from 0 to 4 pi
>> [X,Y] = meshgrid(x,y);      %% create the 2-D array 100x100
>> contour(x,y,sin(X).*sin(Y)); %% create contour plot of 2-D function
>> xlabel('X')
>> ylabel('Y')
```
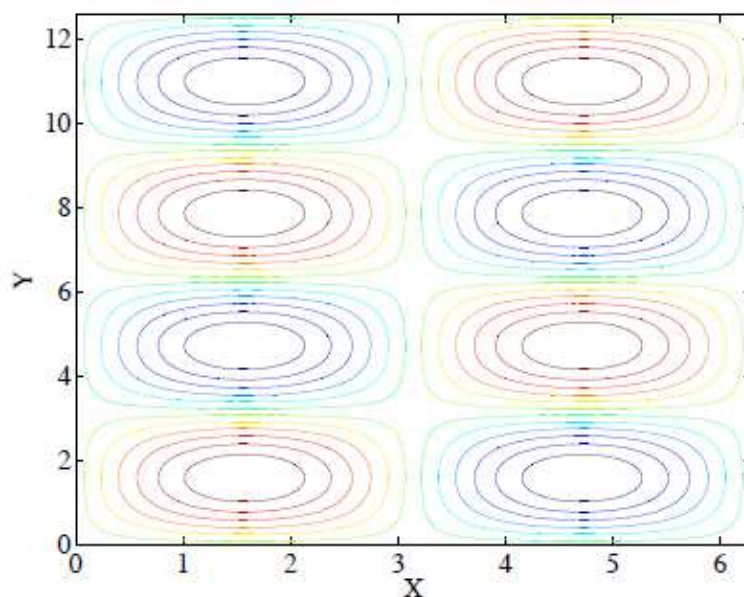

Figure 2: The resulting plot for the commands typed in the example above demonstrating the meshgrid and contour commands.

These commands create a contour plot of the function $\sin(x)\sin(y)$ on the domain $0 < x < 2\pi$ and $0 < y < 4\pi$. The first two lines create the one-dimensional vector representing the points along the x and y axes where the function is evaluated. The meshgrid command creates the two-dimensional arrays, X and Y. The next command creates the function and plots the contours in a single command. In a similar manner, a mesh plot of the function can be created using `mesh(x,y,sin(X).*sin(Y))`. In order to visualize the result of meshgrid, create a contour plot of the two arrays x and y. The array x should produce vertical contours and the array y should produce horizontal contours.

9

7   RELATIONAL OPERATORS

The relational operators can be used to compare numbers. The usual relational operators are greater than >, less than <, greater than or equal to >=, less than or equal to <=, equal to ==, and not equal to ~=. All of the operators return either a one or a zero if the condition is true or false, respectively. We demonstrate this in a simple example where we define two, one-dimensional arrays and apply the relational operators in turn. These same operators may be applied to two-dimensional data or single numbers. You should type each of these commands yourself and take note of the answers that are returned.

```
>> a = [1 2 3];
>> b = [1 1 4];
>> a > b

ans =
     0     1     0

>> a < b

ans =
     0     0     1

>> a <= b

ans =
     1     0     1

>> a >= b

ans =
     1     1     0

>> a == b

ans =
     1     0     0

>> a ~= b

ans =
     0     1     1
```

Note that the operators are applied element by element. In the first example above, `a(1)` is not greater than `b(1)` so `ans(1)` is false (zero), but `a(2)` is greater than `b(2)` so `ans(2)` is true (one), and `a(3)` is not greater than `b(3)` so `ans(3)` is false (zero). A useful command that uses logical operators is find. This command returns the index to the elements in the array that correspond to true from the relational operator. Some simple examples of the find command are provided below.

```
>> a = [-3:3]

a =
    -3    -2    -1     0     1     2     3

>> find(a >= 2)

ans =
     6     7

>> find(a == 0)
```

```
ans =
     4

>> find(a > 10)

ans =
   Empty matrix: 1-by-0

>> find(a > mean(a))

ans =
     5     6     7
```

In the first example above, we ask for the indices of all elements of a that are greater than or equal to 2, which are the sixth and seventh elements.

8   SCRIPTS

Currently we have been typing MATLAB commands at the prompt. This is clearly not efficient if you want to write programs of more than a few lines and programs that you might want to run repeatedly. As we will start writing programs that are multiple lines long it is useful to write a script file so if a mistake is made it is easy to fix the script, run it again, and reprocess the results. In MATLAB nomenclature this script file is called an **m-file**; an **m-file** is only a text file that processes each line of the file as though you typed them at the command prompt. These scripts can be written with any text editor, but it is best to use the editor that is included with MATLAB. In the graphical user interface you can click File→New→Script to launch the editor. One advantage of the MATLAB editor is that colors are used to highlight keywords and comments. This will help you make fewer syntax mistakes when typing your script for the first time.

The script may be executed by returning to the command window and typing the name of the script. In MATLAB, you must be in the directory that the script file is located to run the program. To see what directly you are currently in use the Unix command pwd. To list the contents of the current directory use ls. You may also access directory information through the graphical user interface.

9   CONTROL FLOW

There are a variety of commands that allow us to control the flow of commands inside a program. We will present some of the common ones through example in this section. A common construct is **if-elseif-else** structure. With these commands we can allow different blocks of code to be executed depending on some condition. This if structure allows the program to make decisions. Create the following program as a script file and execute it to study the behavior.

```
x = 5;
if (x > 10)
y = 10;
elseif (x < 0)
y = 0;
else
y = x;
end
```

This simple program will result in y = x when 0 < x < 10, y = 10 when x is greater than 10, and y = 0 when x is negative. The program has the following logic: the program starts by setting x = 5 (you should change this number and run the program many times). The next line asks the question, is x greater than 10? If the answer is true the program executes all the commands until it reaches an else, elseif, or end. In our example, the answer is false so the program proceeds to the next else, elseif, or end statement. In this example the program next asks the question, OK so x was not greater than 10, but is it less than zero? Again this statement is false in our example so the program proceeds the else statement. At the else statement

the program gives up and says; OK neither of those conditions were true, so now execute these lines of code and set `y = x`. If `x` were set equal to `11` then the first condition would be true and the program would set `x = 10`, and then jump to the end statement that closes the if-else block. Programming convention is to indent blocks of code that are surrounded by if-else-end statements. The indentation makes the code easier to read and debug, especially if the blocks of conditional code are lengthy. The MATLAB m-file editor will do the indentation automatically for you.

Another common control flow construct is the **for** loop. The for loop is simply an iteration loop that tells the computer to repeat some task a given number of times. The format of a for-loop is

```
>> for i=1:3
i
end

i =
     1

i =
     2

i =
     3
```

The above for-loop says that the variable `i` will be taken in increments of `1` (default) from `1` to `3`. All the statements until the next end is reached are executed three times. Other forms of the expression of the range of iteration may be used, for example

```
>> for i=3:-1:1
i
end

i =
     3

i =
     2

i =
     1
```

In this example, the statement `3:-1:1` takes the variable `i` from `3` to `1`, by increments of `-1`. The format for the for loop starts with the keyword for, followed by the variable name to be iterated, the start value, the increment value, and the end value. If the increment value is not provided, `1` is assumed. For loops are useful for many things and can be used to perform repetitive tasks; something computers are very good for. A simple example would be an algorithm that computed the factorial of an integer.

```
>> N = 6;
>> fac = 1;
>> for i = 1:N
fac = fac*i;
end
>> fac

fac =
   720
```

For loops are also useful for processing data inside of arrays. The iteration variable can be used as the index into the array. When possible the programmer should try to use MATLAB's whole array mathematics, as this results in shorter programs and more efficient code. Many times the use of for-loops is needed, do not be ashamed to use them. An example program is

```
N = 1000;
for i = 1:N+1
   t(i) = (i-1)/N;
   f(i) = sin(2*pi*t(i));
end

plot(t,f)
xlabel('t')
ylabel('sin(2 pi t)')
```

This program is perfectly acceptable, but the program will run much faster for large values of N if you allocate the arrays f and t before the for-loop. The reason is that the size of the array grows with each iteration and the computer must reallocate a block of memory to hold the entire array with each iteration in the loop. On the first time through the loop, t is allocated as one element long. On the second iteration it is two elements long, and so on. With each iteration a location in memory must be allocated to hold the variables t and f. The time required for memory allocation time will only be become noticeable at very large values of N, e.g. N = 100000000. Since each element is 8 bytes, when storing N = 100000000 we need 800 megabytes of free memory. While this amount of memory is easily available, the time required to continuously update the allocation can make the program run slow. It is faster to allocate the entire memory block at the beginning and fill the array with values in the for-loop. We initialize the arrays f and t to be zero, but of the proper size.

```
N = 1000;
f = zeros(N+1,1);
t = zeros(N+1,1);

for i = 1:N+1
   t(i) = (i-1)/N;
   f(i) = sin(2*pi*t(i));
end

plot(t,f)
xlabel('t')
ylabel('sin(2 pi t)')
```

Try the previous two programs and note how long it takes each to run. Try changing N and see at what value the time to allocate the data becomes prohibitive. Finally, the above programs can be constructed without for-loops; an equivalent program is

```
N = 1000;
t = [0:N]/N;
f = sin(2*pi*t);
plot(t,f)
xlabel('t')
ylabel('sin(2 pi t)')
```

It is always the case in programming that there is no single way to do a given task.

10   LOGICAL OPERATORS

The common logical operators are **and**, **or**, and **not**. The operators are fairly intuitive, the **and** operator checks two input states returns true only when both input conditions are true. The **or** operator checks two input states returns true if either is true. The **not** operator inverts the true/false state of the input. Logical operators are usually used with if statements and the symbols are & (and), | (or), and ~ (not). The statement

```
if (x > 10 & x < 20)
x = 0
end
```

sets `x` to zero if it is between `10` and `20`. As a further example,

```
if (x > 10 | x < -10)
x = 0
end
```

sets `x` to zero if it is outside the range `-10 < x < 10`.

11 FILES

One common use of MATLAB is to take data generated from another program or from an experiment and use MATLAB as a plotting and analysis tool. In this case you need to be able to read data from a file in order to manipulate it. If the data is stored as plain text, with whitespace separating the columns of the data and returns to denote new lines, you can use the command load to import the data. This command will simply create a two-dimensional array corresponding to the data. To use the load command the file must have the same number of columns in each row.

You can also write data to a file from MATLAB. There are several ways to do this, but the simplest is to use the `save` command. You can save data in MATLAB format or in ASCII. Saving variables in MATLAB format allows for easy loading and storing of data. The command save filename saves all the variables in the workspace to the file, filename. The command `save filename A B` saves only the variables `A` and `B`. Finally, `save -ascii filename A` saves the data `A` as an ascii text file. To test these commands, try the following program.

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> save -ascii test.dat A;
```

You should now look at the file `test.dat` and examine how the data was stored. Now returning to MATLAB, type

```
>> clear %% clear the workspace
>> whos %% confirm workspace empty
>> load test.dat %% load the file
>> whos %% see that a 3x3 double array is now in the worksapce
  Name      Size              Bytes  Class      Attributes

  test      3x3                  72  double

>> test  %% confirm that it is the array defined above

test =
     1     2     3
     4     5     6
     7     8     9
```

If we tried the same sequence of commands, only using `save test.dat A`, we get the same result when we open the file `test.dat`, but it is not readable to any application other than MATLAB.

Finally, we can also read and write data to files line by line using `fscanf` and `fprintf` commands. Help can be found on these commands in the MATLAB documentation, but we will not be using these functions herein. These input/output commands are very similar to their counterparts in C.

12 EXAMPLE: NUMERICAL INTEGRATION

In this course, we will often write programs for numerical approximations to common mathematical operations. A simple example to introduce the idea of numerical approximation is to take the numerical integral of a known function. The integral of a function is defined as the area contained underneath the curve. When we have a discrete representation of a function, we can easily compute the integral by using the trapezoidal rule. The trapezoidal rule simply breaks up a function into several trapezoids whos areas are easy to compute and sums the area, see Figure 3.
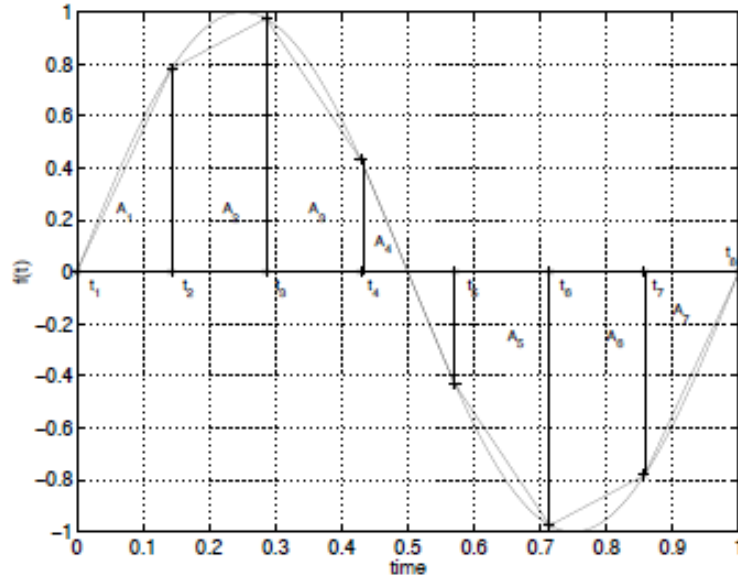
Figure 3: Schematic of trapezoidal rule applied to the function $f(t) = sin(2\pi t)$ sampled at 8 points. The trapezoidal rule simply computes the area of each block and sums them up.

In Figure 3 we see that there are seven blocks (A_j) and 8 data points (t_j). To compute the area, A_j, of the jth trapezoidal block we use

$$A_j = \frac{f(t_j) + f(t_{j+1})}{2}(t_{j+1} - t_j). \tag{1}$$

For now, let's assume that the sample points t_j are equally spaced such that $t_{j+1} - t_j = \Delta t$. Then the total area is

$$A = \Delta t \left(\frac{f(t_1) + f(t_2)}{2} + \frac{f(t_2) + f(t_3)}{2} + \cdots \frac{f(t_6) + f(t_7)}{2} + \frac{f(t_7) + f(t_8)}{2} \right), \tag{2}$$

which generalizes to

$$\int_{t_1}^{t_N} f(t)dt \approx \Delta t \left(f(t_1)/2 + f(t_N)/2 + \sum_{j=2}^{N-1} f(t_j) \right). \tag{3}$$

The trapezoidal rule provides a useful mechanism for approximating integrals to functions that are to difficult to evaluate analytically. There are more complex and more accurate methods of approximation that will be covered later in the course.

Now we will write a MATLAB program that takes a two arrays, `t` and `y`, assumes y is a function of t, and approximates the integral $\int_{t_1}^{t_N} y(t)dt$ using the trapezoidal rule (equation 1.3). Equation 1.3 assumes that the points in `t` are equally spaced, an assumption that the program does not make.

```
t = [0:100]/100;
y = sin(2*pi*t);
I = 0;

for i = 2:length(t)
   I = I + (y(i)+y(i-1))/2*(t(i)-t(i-1));
end
```

An equivalent algorithm that makes use of some of MATLAB's built in functionality could be,

```
I = (y(2:end) + y(1:end-1))/2;
I = I.*diff(t);
I = sum(I);
```

15

In the second example, we made use of the MATLAB functions `diff` and `sum`; you should read the MATLAB help on these functions to understand the usage and output. The second algorithm works whether or not the points are equally spaced.

13   EXAMPLE: RATE EQUATIONS

Many of the systems we will study in this course can be described by rate equations. Often we can write equations that represent the *rate of change* of some system and we will want to understand how the system evolves over time. Equations that describe the rate of change of a variable are called differential equations.

As a very simple example let us consider a car moving along at constant velocity. The position of the car at any instant in time, x, is given as the starting position plus the velocity (rate of change of position), v, multiplied by the time elapsed. We will numerically solve this problem (which we can also easily solve by hand) by using a for loop to take small increments in time of 1/10 minute. At each instant we will update of position of the care by the short time that has passed and accumulate a record of where the car was located at each instant.

```
t = 0; %% Initial time
x = 0; %% Initial position
v = 1; %% velocity in miles/minute (60 mph)
dt = .1; %% time increment in minutes

for i = 1:1000
   t = t + dt; %% new time = old time plus increment
   x = x + v*dt; %% new x = old x plus distance travelled in increment
plot(t,x,'.')
hold on
end
```

The result of program will be a straight line. Now we will add a second following car that starts ten miles behind the first car. This second car will travel at a top speed of 72 miles/hour (1.2 miles/minute) until it catches the first car. As it approaches the first car it will slow down and follow the lead car at 60 miles/hour. Rather than approach the lead car and slam on the brakes to slow down, the second car will approach the first car with a velocity proportional to the distance between the two cars.

```
close   %% Close previous figure
t = 0; %% Initial time
x1 = 0; %% Initial position car 1
v1 = 1; %% velocity in miles/minute of car 1 (60 mph)
x2 = -10; %% Initial position car 2
v2 = 0; %% velocity in miles/minute of car 2
dt = 0.1; %% time increment in minutes

for i = 1:1000
   t = t + dt; %% new time = old time plus increment
   x1 = x1 + v1*dt; %% new x = old x plus distance travelled in increment
   v2 = (x1-x2); %% velocity of car 2 depends on distance between cars.

   if (v2 > 1.2) %% max velocity is 72 mph
      v2 = 1.2;
   end

   x2 = x2 + v2*dt; %% new x = old x plus distance travelled in increment
   plot(t,x1,'b.');
   hold on;
   plot(t,x2,'r.');
end
```
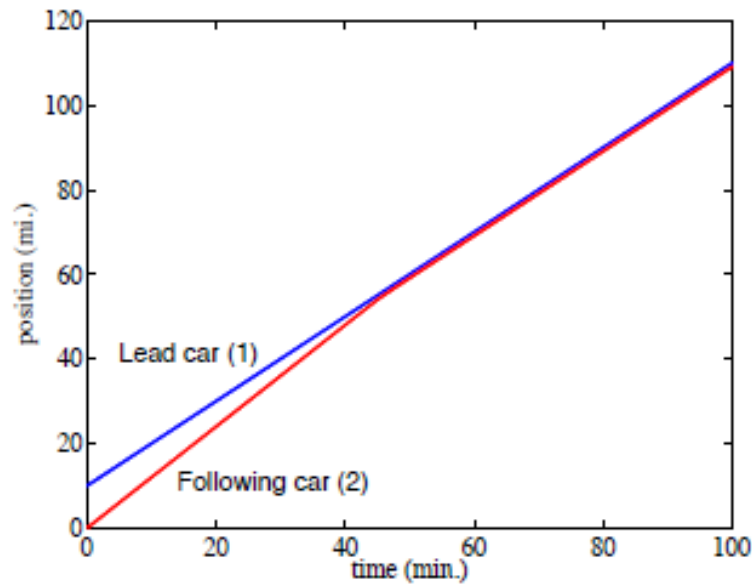
Figure 4: The resulting figure from the program created to predict the position of one car following a second. Note that the first car travels at constant velocity while the second car catches the first and then follows at a close distance.

Notice that this method of controlling the position of the second car means that the second car will always be one mile behind the first. When the difference between the two cars is one mile, `v2 = 1`, and therefore the distance between the two cars will never change. The system will behave very differently for different constants of proportionality, which we took as unity in the example. This is a somewhat contrived example and we will see more complex and authentic examples of similar control ideas throughout the course. Later, we also investigate the effects of changing the size of the incremental step in time and provide a more thorough mathematical background to such numerical methods.

14   FUNCTIONS

Functions are programs that accept inputs and provide outputs according to some rules. You have used many functions so far as MATLAB has many built in. For example the functions linspace, plot, and cos are just a few of the many examples we have covered. Now you will learn to write your own functions. Functions are extremely useful as you can write one piece of code to perform a task and then use that function repeatedly; requiring one time programming and debugging. How would you like to write out the lines of code that compute the cosine every time you needed it!

The first line of a function M-file starts with the keyword function. Next you provide the output variable, the function name, and the name of the input variables. This is best shown through example. Previously, we used the if statement to create a function that limited a variable between 0 and 10. We will create this code as a function. In a separate file named `limit0_10.m` try the following function:

```
function B = limit0_10(A)

if (A > 10)
   B = 10;
elseif (A < 0)
   B = 0;
else
   B = A;
end
```

You can test this function by saving the file and running the function in the command window,

17

```
>> limit0_10(11)

ans =
    10

>> limit0_10(-1)

ans =
     0

limit0_10(2)

ans =
     2
```

The function above has the name `limit0_10`, the input variable name is `A` and the output variable name is `B`. **It is very important that the function name and file name match.** It is also very important to realize that functions have their own workspace. They only know variables that have been passed as input arguments or calculated inside the function itself. They are not aware of the main MATLAB workspace. This is important because then when we write functions we know that they are self contained and cannot be contaminated by variables already in memory. To test this we make a simple edit to the limit function, by adding a whos statement inside to check the workspace.

```
function B = limit0_10(A)

if (A > 10)
   B = 10;
elseif (A < 0)
   B = 0;
else
   B = A;
end
whos
```

We now can move to the command window and monitor the workspace as we use the function. We start with a clear workspace and define a variable `X` . We then check to see that is the only variable in the workspace. We then run the function to define `Y`, and see that the whos statement inside the function shows that two variables in the *function's* workspace are `A` and `B`, where `A` is the input argument and `B` is created in the function. We then check the workspace *back in the command window* and see that `X` and `Y` are the only known variables. Functions have an independent workspace, containing variables that are defined as input arguments or defined within.

```
>> X = 5.6;
>> whos
  Name      Size            Bytes  Class     Attributes

  X         1x1                 8  double

>> Y = limit0_10(X);
  Name      Size            Bytes  Class     Attributes

  A         1x1                 8  double
  B         1x1                 8  double

>> whos
  Name      Size            Bytes  Class     Attributes

  X         1x1                 8  double
  Y         1x1                 8  double
```

18

We can extend our function so that it works on data arrays as well. While there are many ways to execute such a function, one method using the find command is shown as

```
function B = limit0_10(A)

B = A;              %% set A=B
i = find(A > 10);   %% find indices where A>10
B(i) = 10;          %% upper limit to 10
i = find(A < 0);    %% find indices where A<0
B(i) = 0;           %% lower limit to 0
```

which can perform the function element-wise on data arrays as follows.

```
>> X = [-2 0 5.5 13];
>> limit0_10(X)

ans =

         0         0    5.5000   10.0000
```

The final modification to this program, we allow the maximum and minimum outputs be determined by the user. These values are added as input arguments that should be supplied by the user.

```
function B = limit_min_max(A,maximum,minimum)

B = A;                   %% set A=B
i = find(A > maximum);   %% find indices where A > maximum
B(i) = maximum;          %% upper limit to maximum
i = find(A < minimum);   %% find indices where A < minimum
B(i) = minimum;          %% lower limit to minimum
```

which now has a more flexible functionality.

```
>> X = [-2 0 5.5 13];
>> limit_min_max(X,6,-1)

ans =
   -1.0000  0  5.5000  6.0000
```

To provide more function examples, we can create functions for a variety of mathematical operations. Below we provide an example of a factorial function, a function that sums all elements of an array, one that finds the greatest common divisor of two integers, and one that takes the integral of two vectors assuming that one is a function of the other. MATLAB already has `factorial`, `sum`, `gcd`, and `trapz` as built-in functions, but we will construct our own for examples. Write the following programs in separate files named `fac.m`, `summit.m`, `gdivisor.m` and `integrate.m`. In the `gdivisor` program we make use of a few new MATLAB commands. The command `error` will print the text that is passed to the function and then quit the execution and return to the command prompt. The function `mod` provides the remainder after division, zero if the number is evenly divisible.

```
function X = fac(N)
% Computes the factorial of N and returns the result in X

X = 1;
for i = 1:N
   X = X*i;
end

function X = summit(A)
% Sums all the elements in array A and returns the result in X.
% The sum is taken over all elements
```

```
X = 0; %% sum
A = A(:); %% if 2-D stretch into 1-D
for i = 1:length(A)
   X = X + A(i); %% add up
end


function N = gdivisor(A,B)
% Find the greatest common divisor of A and B and return answer.
% A and B must be positive integers or an error is returned.
% A and B are only single numbers, not arrays.

%% make sure A and B are integers
if (round(A) ~= A | size(A(:)) > 1 | A < 0)
   error('A is not a single, positive, integer!');
end
if (round(B) ~= B | size(B(:)) > 1 | B < 0)
   error('B is not a single, positive, integer!');
end

%% exhaustive search, starting from the largest possible,
%% which is the smallest of the two numbers

for N = min(A,B):-1:1
   if (mod(A,N) == 0 & mod(B,N) ==0)
      break
   end
end


function I = integrate(t,y)
%% compute the integral of y(t) from t(1) to t(end)

I = (y(2:end) + y(1:end-1))/2;
I = I.*diff(t);
I = sum(I);
```

These functions can be run at the MATLAB command prompt. For example, to run the last function you can go to the MATLAB prompt and issue the following commands to get the the value of the integral returned to you.

```
>> t = [0:100]/100;
>> y = sin(2*pi*t);
>> integrate(t,y)
```

A very useful feature of functions in MATLAB is that comments placed at the top of the file, with the comment marker in the first column are available as help from the command line. This feature allows you to document the usage of your functions. This is very important so that you may return to the function at a later date and remember what it does. For example, using the code written above,

```
>> help gdivisor

Find the greatest common divisor of A and B and return answer.
A and B must be positive integers or an error is returned.?
A and B are only single numbers, not arrays.
```

As a final note, MATLAB requires that each function you create exist in a separate file where the function and file name match. You may use multiple functions in a single file, but then those functions are only available inside that file. When breaking a specific problem into smaller, convenient pieces, it is common

to keep all the related functions in one file. When creating general-purpose functions, then one strategy for keeping your work straight is to keep a special directory containing one file per function. You can keep all these files in a special directory that is part of the MATLAB search path. With that strategy you can simply call the function as you would any built in MATLAB function. You should also be aware that all the good functions are already written. MATLAB contains most of the common mathematical functions that you can think of.

15   Problems

1. Create a vector `t` that ranges from `0` to `1` with `1000` equally spaced points. Plot the functions $\sin(2\pi t)$, e^{-5t}, and $\cos(t)e^{-t}$ on the same graph.

2. Create a vector `t` that ranges from `0` to `1` with `1000` equally spaced points. Plot the functions $f(t) = e^{5t}$ with a saturation limit of $f = 10$. This means that if f is greater than or equal to ten, then set $f = 10$. The `find` command may be useful.

3. Alias error refers to taking digital samples of a signal at a rate that is slower than the signal itself. The result is false measurement. One way to visualize this error is to create a high frequency sine wave with a low number of points. Create two vectors `t1` and `t2`. Each array should range from `0` to `1`, but `t1` should have `21` equally space points (i.e., `t1 = [0:20]/20`) and —t2— should have `1000` equally spaced points. On the same graph plot the function $f = sin(38\pi t)$ using `t1` and `t2`. Plot the function with `t1` as discrete points and not a connected curve, while you should plot the function with `t2` as a connected curve. Try changing the number of points in `t1` and see what happens. Note that the command hold on will hold the current figure to allow you to plot a new curve on top of it and `plot(t,f,'.')` will plot the function as discrete points rather than a connected line.

4. Modify the `integrate` program to compute the following integrals over $0 < t < 1$: $\sin(2\pi t)$, e^{-5t}, and $\cos(t)e^{-t}$. Use 100 data points in your definition of `t`. Compare your result to the built in MATLAB function `trapz`. Read the help on the MATLAB function to see how to use the function.

5. Numerically solve the simple differential equation $dy/dt = -y$, with initial condition $y(0) = 1$. Use Euler's method to solve the equation. Euler's method simply takes the slope of the solution, dy/dt, and propagates it forward a small increment in time, Δt. Therefore the value of y at the first time step is

$$y(\Delta t) = y(0) + \Delta t \frac{dy}{dt}|_{t=0}. \tag{4}$$

This procedure is then used to propagate forward in time to the next time step and so on. This is the simplest numerical method to integate ordinary differential equations.

6. In the early 1800's an English botanist, Robert Brown, observed that pollen grains contained in a drop of water wiggled about in a jagged, random path. Later this behavior was attributed to molecular fluctuations in the water. Water molecules impinge on the small particle causing random forces acting in random directions. If the particle is small enough then the particle will move about in response to these random forces.

A simple model of Brownian motion in two dimensions is as follows: Take a particle and place it at the origin $x = y = 0$. Use a for loop to take steps in time. At each time step add a normally distributed (i.e. bell-curve) random number to the horizontal and vertical coordinate. Plot the position of the particle at each step. The MATLAB command `randn` will generate a single normal random number. The command `plot(x,y,'.')` will plot a single data point and the command `hold on` will hold the graphics "on" so that the next iteration will plot the point on the same graph.

Write a program that plots the path (in $x - y$ space) of one particle over 10,000 iterations. Try running the program a few times and see what patterns you develop. You will notice that each run will look very unique. Type `clf` between runs to clear the figure for each new time you run the program.

7. In order to do this problem, we need an image to process. Any image will do. You can check the file types that are supported by the function imread. Most common formats such as tiff and jpeg are

supported. The function imread will import the image file into the MATLAB workspace. If the image is color you will get a three dimensional array, two-dimensions for the picture and the third for RGB scale. If the original picture is gray-scale then you will get a two-dimensional array. The format for the numbers is `uint8`, which is not suitable for ordinary arithmetic, so we transform the image data type to a default for numbers, double. The command imagesc is useful for plotting images; the function performs self scaling. Try the following sequence of commands

```
>> A = imread('picture.jpg');
>> image(A);
>> whos
  Name        Size              Bytes  Class      Attributes

  A           1728x2304x3   11943936  uint8

>> A = double(A(:,:,1)); %% take only one color level.
>> whos
  Name        Size              Bytes  Class      Attributes

  A           1728x2304     31850496  double

>> imagesc(A); %% this will look a little funny since we are not decoding the color
```

Now you can manipulate the two-dimensional array `A`. Write a program that applies a "binary" threshold filter to the image data array. The threshold filter sets some value and everything above the filter is "on" and everything below is "off". Have your program work as a function. The input value is a two-dimensional data array. The program should find the average value of the input image and use that value as the threshold. The new image should be the same size as the input and consist of one and zero depending on if you are above or below the threshold. The output of the program should be the new image data file. Plot the output using `imagesc`. Try plotting the element by element multiplication of the input and output data arrays.