Programming in MATLAB

Julie Blackwood

Williams College

January 30, 2015

Contents

Chapter 1

Introduction to MATLAB

MATLAB (**MAT**rix **LAB**oratory) is, according to the MathWorks homepage (the developer of MATLAB):

> "a high-level language and interactive environment for numerical computation, visualization, and programming. Using MATLAB , you can analyze data, develop algorithms, and create models and applications. The language, tools, and built-in math functions enable you to explore multiple approaches and reach a solution faster than with spreadsheets or traditional programming languages..."

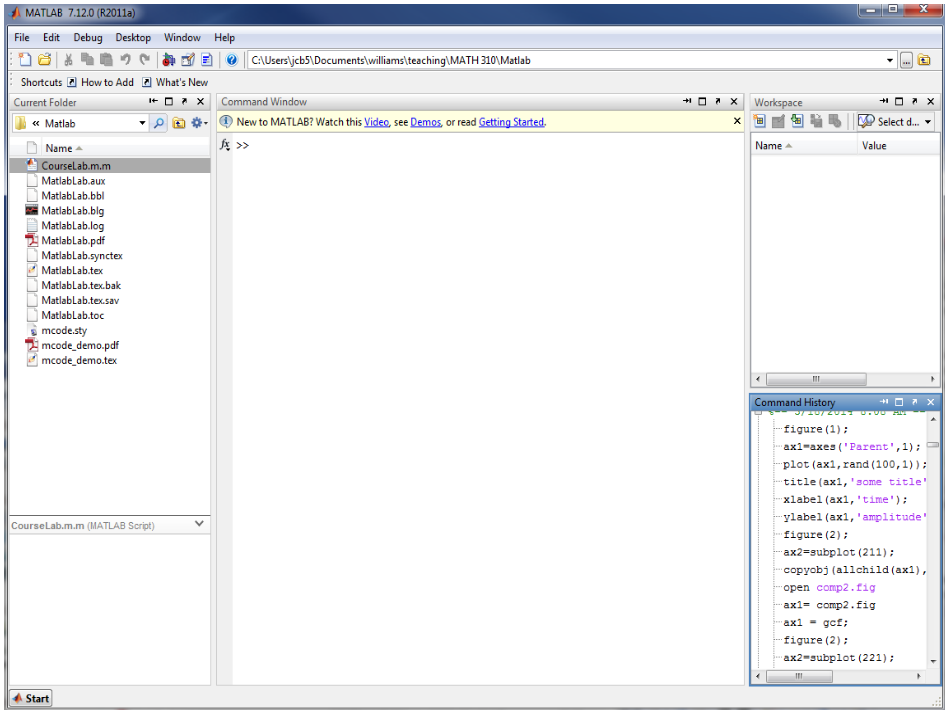
MATLAB has a great user interface that is easy to work with and (as indicated by its name) it generally works with vectors and matrices. MATLAB is used for research across many disciplines to run numerical simulations, create plots, etc. In fact, it is my program of choice! In this document, we will first learn some basic programming in MATLAB and we will then see how it is used in research.

1.1 Getting started

The computers in the computer lab on the first floor of BSC have MATLAB installed, but you can also download it to your own computer. MATLAB is available for download through OIT at `https://oit.williams.edu/software/`. Click on "MatLab 2013a (MathWorks)" and it will guide you from there. If you have a Mac, all operations in MATLAB are identical.

When using MATLAB , it is easiest to save all files to one directory and then make sure that same directory is your current directory (see figure below). Open MATLAB by clicking on the Windows start button → All Programs → Matlab R2013a → Matlab R2013a icon. This will open the MATLAB GUI (graphical user interface) and it will look something like Figure 1.1. *Note*: there might be slight differences in appearance depending on the version of MATLAB installed on your computer.

Figure 1.2 labels the components of Figure 1.1. The large window in the center is the "command window"



```
figure(1);
ax1=axes('Parent',1);
plot(ax1,rand(100,1));
title(ax1,'some title'
xlabel(ax1,'time');
ylabel(ax1,'amplitude'
figure(2);
ax2=subplot(211);
copyobj(allchild(ax1),
open comp2.fig
ax1= comp2.fig
ax1 = gcf;
figure(2);
ax2=subplot(221);
```

Figure 1.1

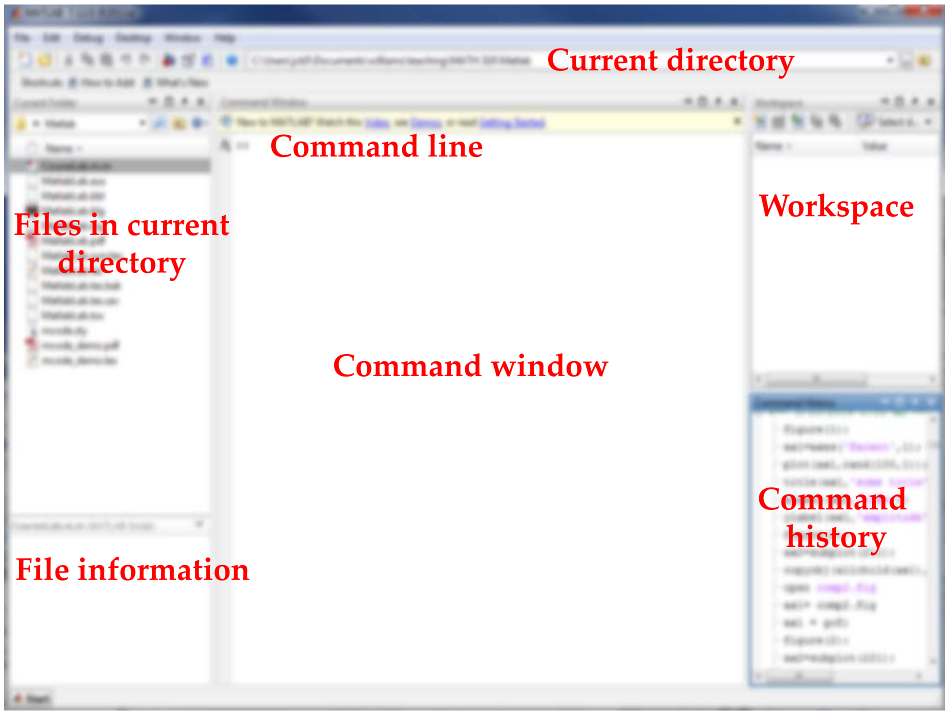


Figure 1.2

and the line at the top beginning with >> is the "command line." In MATLAB , there are two ways to perform operations. One is through the command line which we will discuss in the next section (the other is with m files, which we will get to in Chapter 1.4).

At the top there is a line identifying the current directory. You can change the current directory by clicking on the three dots to the right of it. Click on this and browse to open the "Math310" folder we just created. The top window to the left lists the files that are in the current directory (i.e. the Math130 file). Files in this folder can be opened in MATLAB by double clicking on the file name in this window. The box below this provides file information – I usually ignore this box.

The top box on the right is the "Workspace." It shows all current variables under "Name" as well as their "Value," "Min," and "Max." "Value" will usually state the value of each given variable but if the variable is too long (e.g. a large matrix), it will simply state its dimensions.

Below this window is the "Command History." It lists what commands have been made and can be useful if you need to recover something you did earlier.

1.2 Basic commands in MATLAB

In this section, we will learn some basic commands in MATLAB using the command line. First, we can use it like a calculator. For example, type

```
2.45*632
```

and hit "enter." Notice that MATLAB automatically labels it "ans" and it shows up in the Workspace. Suppose that we want to store this value as a constant a. If you press the up arrow key, the previous line that you typed will show up (to save time) or you can just type it again. We will define `a` in two ways: first, type

```
a = 2.45*632
```

and then type

```
a = 2.45*632;
```

Notice that using a semicolon suppresses the Command Window from printing the value of `a`. However, MATLAB has still stored this value of `a` and if you type "a" into the command line the value you assigned it will appear. When performing long simulations, simple things like preventing variables to constantly print out can drastically speed up the computational time!

Now that we've stored the constant `a`, we can manipulate it. Play around with this by multiplying `a` by another number, etc. You can also overwrite `a` with a new value or define a new parameter

```
 b = a/10
```

Suppose now that we want to get rid of `a`. We can do this using the "clear" command:

```
 clear a
```

Next to clear you can just list one value to clear or you can list multiple. If you want to get rid of everything, just type

```
 clear all
```

Another useful command is

```
clc
```

This clears the Command Window.

1.3 Vectors and matrices

Manual definitions

Now, let's learn how to define vectors and matrices. There are several ways to do this and we will first learn how to manually create a vector. The command

```
 v = [1; 2; 3]
```

will create a vector `v` that is 3x1 (three rows, one column). Here, the usage of semicolons forces MATLAB to create a new row in the vector. Alternatively, we can create a 1x3 vector by leaving out the semicolons:

```
 v2 = [1 2 3]
```

Suppose we want to access the second element of `v2`. We can do this by typing

```
v2(1,2)
```

Because this is a vector and the number of either rows or columns is one, this can more succinctly be found by typing

```
v2(2)
```

We can now figure out how to create a matrix; we need to use semicolons to create a new row and spaces to move to the next column. For example

```
 m = [1 2; 3 4; 5 6]
```

will create a 3x2 matrix whereas

```
 m2 = [1 2 3; 4 5 6]
```

will create a 2x3 matrix. You can actually check that this produces the dimensions 2x3 by typing

```
 size(m2)
```

This will tell you the number of rows and the number of columns, respectively.

Suppose we want to access the element in the second row of `m2` and we want to name it `q`. We can do this by simply typing

```
q = m2(2,3)
```

Again, in all of these examples we can always add a semicolon if we do not want to see the actual value.

Compact definitions

Suppose we want to create a vector storing the numbers one through 100 in order. This is clearly a pain to manually define, so we can instead do the following:

```
  v3 = 1:1:100
```

The first 1 tells MATLAB to start with the value one, the second 1 tells MATLAB to increment by one, and the last value tells it to stop at 100. By default, the creates a 1x100 vector (also known as a column vector). If we instead want this to be a 100x1 vector we can define

```
  v4 = v3'
```

For those of you familiar with linear algebra, the `'` command takes the transpose of a matrix (in this case, a vector).

For a more less straight forward example, suppose we want to create a vector that starts at 0.1 and each element adds 0.2 to the previous row until 2.1 is reached. This can be found by typing

```
 v5 = 0.1:0.2:2.1
```

If we want to access the fifth element, just type

```
 v5(5)
```

Now, suppose we want to create a 10x1 vector of all zeros with the exception that the eighth element is 2. MATLAB has a nice command `zeros(m,n)` that will create a matrix with `m` rows and `n` columns. In this example, we can then define

```
v6 = zeros(10,1)
```

and then

```
v6(8) = 2
```

Notice that this can be used to create a matrix of any size that has all zeros. A similar command is `ones(m,n)` which creates a matrix of all ones.

One more convenient command is `eye(n)`. This will create the nxn identity matrix. For example

```
m3 = eye(2)
```

will create the 2x2 identity matrix. Suppose we want to multiply this matrix (or any vector or matrix) by a constant. This is done exactly as you'd expect. For example

```
m4 = m3*5
```

will simply multiply every element in `m3` by 5.

Exercises

1. Perform the following operations using MATLAB
   (a) Create a vector `E1` that starts with 2 and ends at 32 with steps of 3.
   (b) What is the fifth element of `E1`?
2. (a) Manually create a matrix `E2a` with 3 rows and 3 columns. Row 1 should contain all 1's, row 2 should contain all 2's, and row 3 should contain all 3's.
   (b) Manually create a matrix `E2b` with 3 rows and 3 columns. Column 1 should contain all 1's, column 2 should contain all 2's, and column 3 should contain all 3's.
3. Create a 3x3 matrix of all ones with the exception that the element in the second row and second column is 2.

1.4 Creating m files

While working from the command line is a quick way to perform simple computations, it has many drawbacks. For example, it's difficult to edit. Instead, we can use m files (m is for MATLAB , clever eh?).

To create an m file, either click on File $\rightarrow$ New $\rightarrow$ Script *or* click on the upper-left icon of a sheet of paper.

For the remainder of this chapter, we will use the m file we just created. We first have to save it to our current directory (the Math310 folder). Click on File $\rightarrow$ Save As, and for "File name" type "BasicPlots.m" (we will use this file in the next section when we learn some plotting techniques). Double check that you are in the correct directory.

When we are ready to run a file, either type the file name BasicPlots (without the .m) in the command line, or click on the icon at the top of the m file that looks like a green "play" button.

After running a file, if there are any errors, they will be displayed in the Command Window.

One important good habit to develop in programming is ***commenting*** your code. A comment is something that you can write in your program that does not have any effect on the program itself. This is useful for creating a description of the program at the beginning of every m file, stating what your variable definitions mean or why you performed a particular computation, etc. In MATLAB , comments are preceded by a percent symbol. Let's put a comment at the top of the m file describing the purpose of the file:

```
% This file is used as an introduction to plotting in MATLAB
```

1.5 Creating plots

Now that we have a handle on vectors and matrices, we will move on to creating plots. In general, plotting in MATLAB is much different than programs such as Mathematica in that it requires the usage of vectors. To create these plots, we will use our file "BasicPlots.m" At the top, make a comment specifying what this file will do.

As a first example, suppose that we want to plot the graph of $y = x^2$ from $x = -2$ to $x = 2$. To do this, we need to create a vector containing all of the x-values and a second vector containing all of the corresponding y-values. Notice that this requires the the length of the x and y vectors must be the same. MATLAB will then plot the series of x and y coordinates and by default will connect the points with a blue line.

Let's first create the x-vector. This requires choosing x-values that are close enough together so that the plot does not look "choppy." Sometimes, this requires a bit of playing around to make the plot look nice, but let's choose values of x that are spaced at intervals of 0.1:

```
x = -2:0.1:2;
```

Notice that this forces the vector to start at -2 and run up through 2.

Now, we want to create the y-vector. There are two ways to do this. One is using loops (see Chapter 1.6), but we will not cover this in class (at least for now!). The second is simply to create a new vector by manipulating the x vector. Specifically, we can define a vector `y` by

```
y = x.^2;
```

The "dot" in the command essentially tells MATLAB to square the vector `x` element by element. Now we are set to plot using the `plot` command. The first input is the x-values, second input is the y-values (there are many additional commands you can give, but for now we will ignore them. Therefore, our line of code is

```
plot(x,y)
```

To summarize, to plot $y = x^2$ we need the code

```
x = -2:0.1:2;    % Define a vector of x-values

y = x.^2;        % Find the corresponding y-values

plot(x,y)        % Plot
```

and the resulting plot is show in Figure 1.3. If you want to save this figure, in the figure window click on File $\rightarrow$ Save As. Then, browse to the directory you want to save in and chose a file name. For "Save as type" you can save it as a MATLAB figure file (using .fig) so that you can open it in MATLAB . You can also save it as an image (for example, a JPEG). Suppose that you want to copy the figure and paste it into a document (e.g. using Microsoft Word). To do this, click on the icon in the figure window that is an arrow. Next, click on Edit $\rightarrow$ Copy Figure. You can then paste this into a document.

Now, suppose that we want to overlay the graph of $y = x^3$ onto our plot of $y = x^2$ and we want $y = x^3$ to be in red. Well, we can use the same x-vector as we used before but we can create a new y vector to store the values of x^3. Similar to the last plot, we can do this with:

```
y2 = x.^3;
```

Notice that we need to call this y vector something new (here, `y2`) so that we do not overwrite our initial vector. To plot $y = x^3$ without erasing the graph of $y = x^2$ we have to use a special command:

```
hold on
```

The last thing that we need to do is specify that we want the color of $y = x^3$ to be red. We can add an additional input to the `plot` command indicating the color:

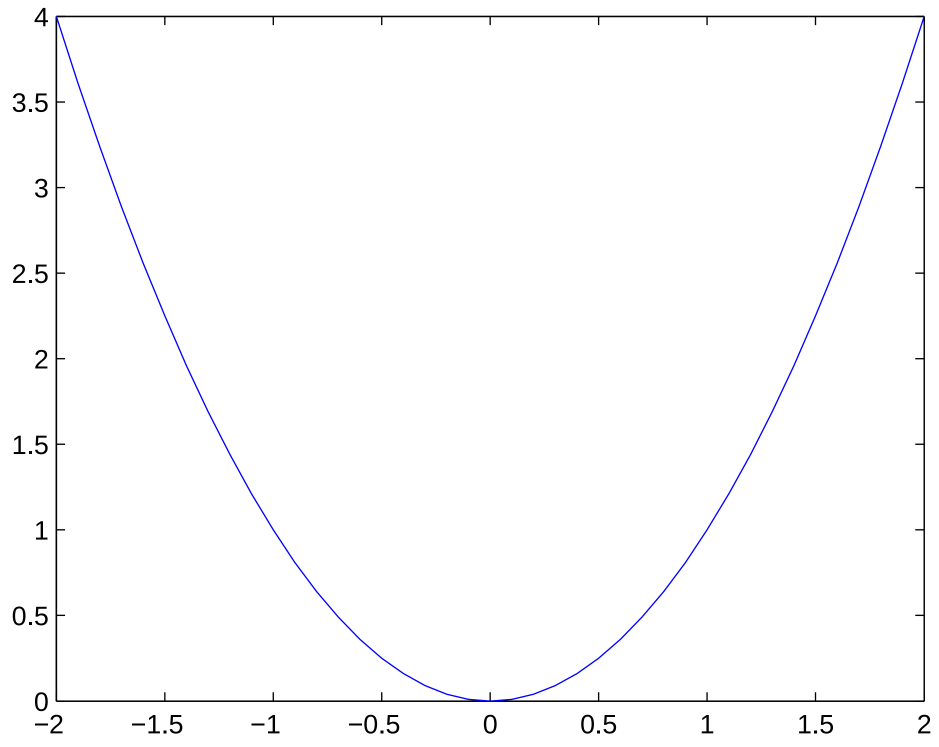


Figure 1.3. Resulting plot for $y = x^2$

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

where the `'r'` indicates the color red. Other common colors are `'k'` (black), `'p'` (purple), `'g'` (green), and `'y'` (yellow). Again, to summarize our second bit of code should look like:

```
y2 = x.^3;       % Define a new vector y2 storing x^3
hold on          % Prevents current graph of y = x^2 from being erased
plot(x,y2,'r')   % Plot command specifying that y = x^3 should be red
```

and the resulting plot is in Figure 1.4

Now, suppose that we want to graph these results on two separate plots within the same figure. We need to figure out how to put separate plots within the same figure. The first thing we need to do is open a new figure so that we do not overwrite our initial figure. This is done with the `figure` command:

```
figure()
```

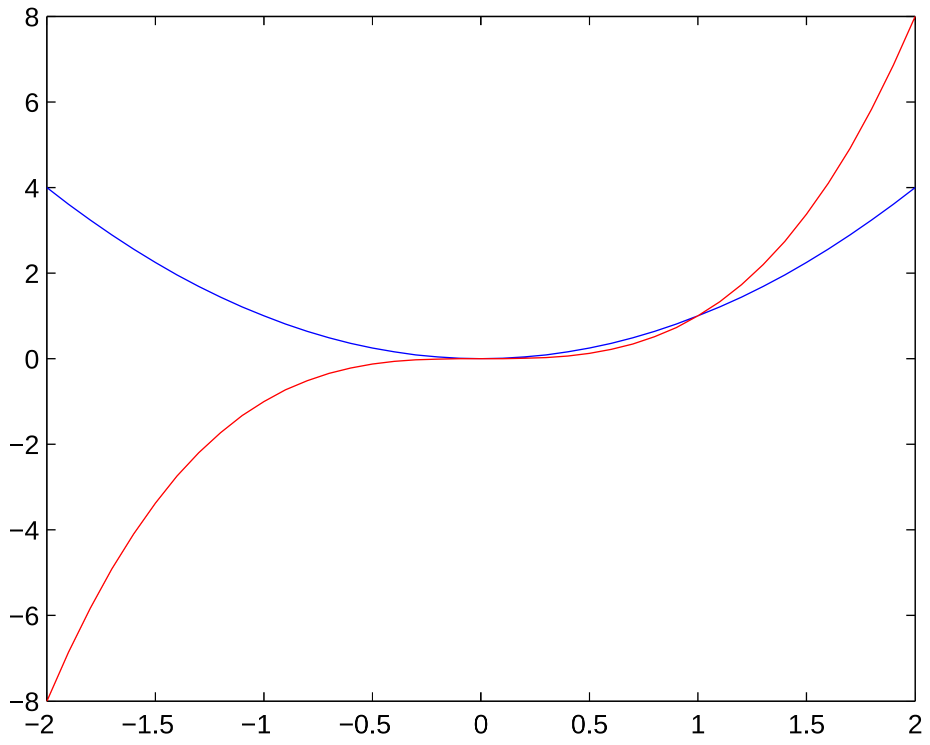


Figure 1.4. Resulting plot when overlaying $y = x^3$ on our previous plot with $y = x^2$.

To create multiple plots within this graph we use the `subplot` command. `subplot` in some sense is treated like a matrix; the first input is the number of *rows* of plots and the second input is the number of *columns* of plots. The third command specifies which plot within the figure you are currently working with. In our case, for example, suppose that we want to create two side by side plots. In other words, there is one row and two columns. To access the first plot, we would type

```
subplot(1,2,1)
```

and the second plot would be

```
subplot(1,2,2)
```

Each plot within the subplot is labeled sequentially across rows. If we instead had a second row also with two plots, the first plot on the second row would be accessed by typing

```
subplot(2,2,3)
```

Now back to our problem. The first thing we need to do is create a new figure – otherwise, we would write over the previous plot. To do this, we use the `figure()` command. On the left hand plot, we want to show $y = x^2$ and on the right hand plot we want to show $y = x^3$ (in red). We have already created the necessary vectors, so we use the following code to plot:

```
figure()          % Open a new figure

subplot(1,2,1)    % Specify that we want to work with the left hand plot

plot(x,y)         % Plot y = x^2

subplot(1,2,2)    % Specify that we want to work with the right hand plot

plot(x,y2,'r')    % Plot y = x^3 in red
```

The resulting plots are shown in Figure 1.5.

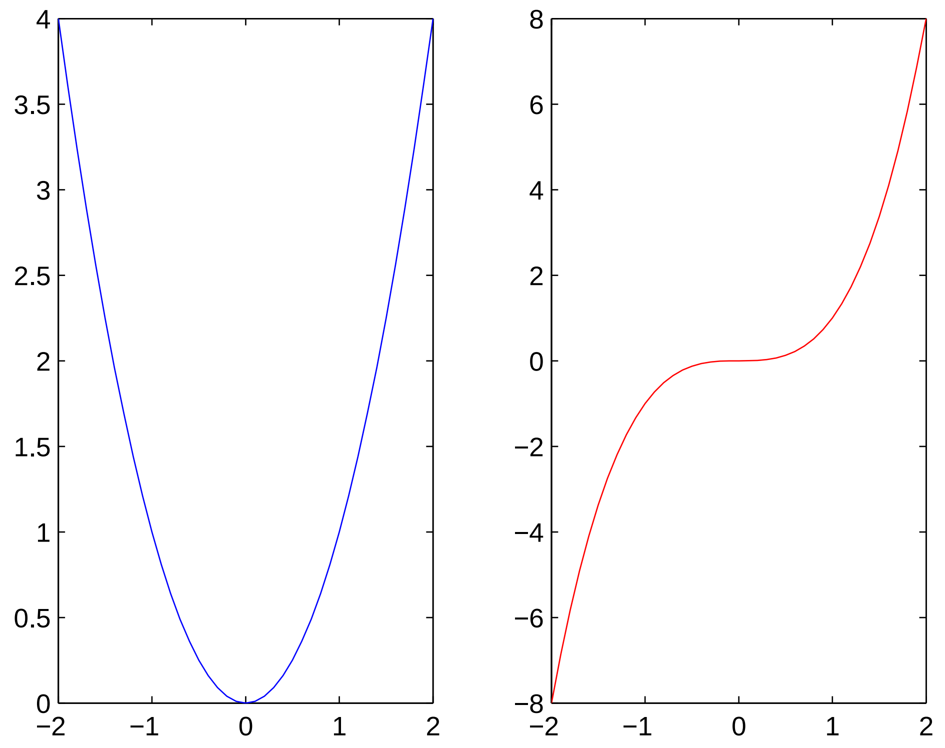


Figure 1.5. Resulting plot when using `subplot` to plot $y = x^2$ and $y = x^3$ side-by-side.

To edit plots, there are a couple of ways to do it. This can be done manually, using "File Properties" and "Axes Properties" (click on "Edit" in the Figure window and go to these options). There are also a few quick ways to customize your plots in your m file. Here we will list a few. First, let's create a new figure that again plots $y = x^2$:

```
figure()        % Open a new figure
plot(x,y)       % Plot y = x^2
```

Suppose we want to title this figure "$y = x^2$." Here we use the `title` command:

```
title('y = x^2')    % Title the plot. Note that single quotation marks are required
```

We can also label the axes using the commands `xlabel` and `ylabel`:

```
xlabel('x')     % Label the x-axis
ylabel('y')     % Label the y-axis
```

The resulting plot is shown in Figure 1.6.

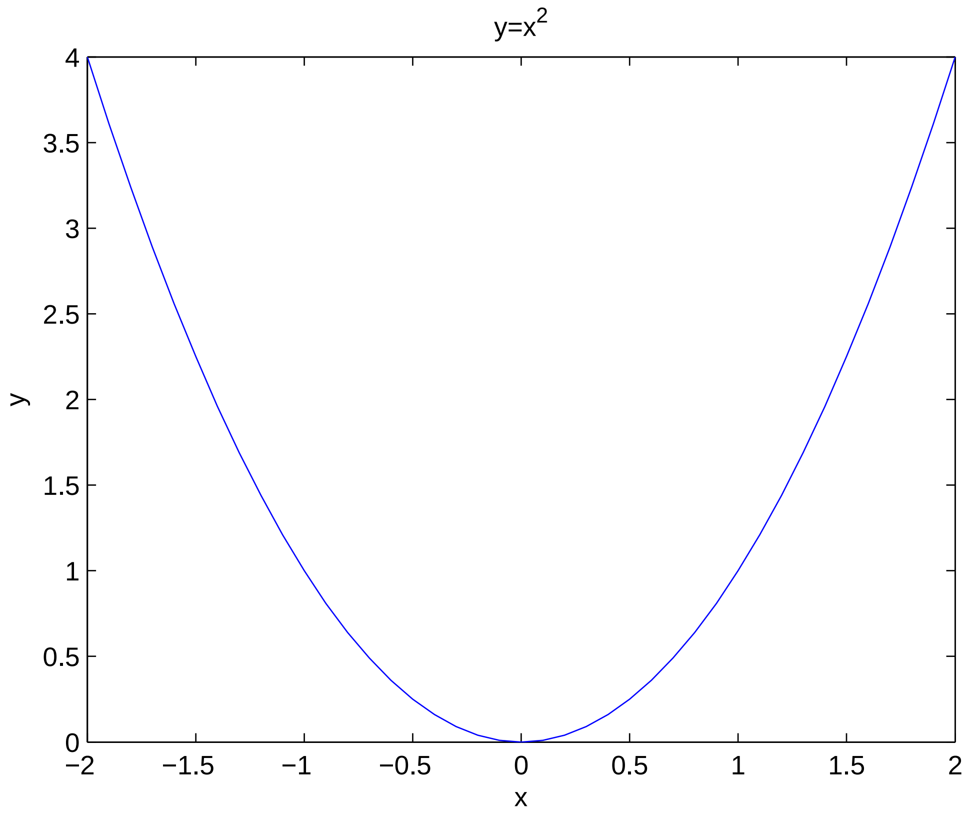


Figure 1.6. Plot of $y = x^2$ with a title and labels.

There are a couple of other useful commands when plotting in MATLAB . For example, you can specify the x and y axis bounds that are used in the figure. Suppose that we want to be the lower x-bound to be some specified value `xmin`, the upper x-bound to be `xmax`, the lower y-bound to be `ymin` and the upper y-bound to be some specified value `ymax`. Then, after we create a plot we can use the `axis` command as follows:

```
axis([xmin xmax ymin ymax])
```

Another useful plotting tool is that we can adjust the thickness of a curve. The default thickness is 0.5 in MATLAB . Suppose that we want to simply plot $y = x^2$ as we did above, but now we want the curve to be displayed in red and we want the thickness of the curve to be 2. Here, we use the `LineWidth` command as follows:

```
plot(x,y,'r','LineWidth',2)
```

Exercises

1. Create the following plots:

   (a) Plot $y = \sqrt{x+1}$ for $x = -1$ to $x = 5$ using intervals of 0.1 between consecutive x-values. Use a red line to plot, label the x-axis `x` and the y-axis `y`.

   (b) Repeat (a) but chose the interval between x-values to be 1. Notice the difference in quality of the plots

   (c) Create a 2x2 subplot. Plot $y = x$ on the upper left, $y = x^2$ on the upper right, $y = x^3$ on the bottom left, and $y = x^4$ on the bottom right subplot. For each of these, let x range from -5 to 5.

Solution to Exercises

```
% Code for Chapter 1.5 exercises

% Exercise 1a

x = -1:0.1:5;
y = sqrt(x + 1); % Alternatively, use y = (x + 1).^(1/2)

plot(x,y,'r')
xlabel('x')
ylabel('y')


% Exercise 1b

figure()
x2 = -1:1:5;
y2 = sqrt(x2 + 1);

plot(x2,y2,'r')
xlabel('x')
ylabel('y')


% Exercise 1c

figure()
x = -5:.1:5;
y1 = x;
```

```
y2 = x.^2;
y3 = x.^3;
y4 = x.^4;

subplot(2,2,1)
plot(x,y1)

subplot(2,2,2)
plot(x,y2)

subplot(2,2,3)
plot(x,y3)

subplot(2,2,4)
plot(x,y4)
```

1.6 for loops

Creating loops is a very important tool in programming. Here we will introduce one type of loop. Suppose that we want a program to create a 10x1 vector `vec` such that each element within the vector `i` is defined as `i+4`. First, open and save a new m file.

Now, we want to do is define a 10x1 vector. Let's initially define this as a vector of all zeros and then we will learn a way to fill in the elements of the vector without having to do it manually.

```
vec = zeros(10,1);
```

Notice that I used the semicolon to prevent the program from printing out the vector.

Now, we will now overwrite each zero in `vec` using what is known as a ***for loop***. In a for loop, you create an index variable (most commonly these are denoted by `i`, `j`, or `k`). You then assign an initial and final value to this index variable. This is the first part of the loop and is written as

```
for i = 1:10
```

Note that the colon tells MATLAB to iterate from 1 to 10. Next, for each given `i` we want to tell MATLAB what to do. For us, the next line is then

```
vec(i) = i+4;
```

In our case, this is the only operation we want to perform. Therefore, the next step is to provide MATLAB an indicator that this is the final (and only – but you can also have multiple) operation we want to perform. In MATLAB , this is simply given by

```
end
```

In summary, the loop will start at the initial value, perform whatever operations are written within the loop (the operations are terminated when the `end` command is reached), `i` will increment by one, and repeat. Once the final value of `i` has been reached and the relevant operations have been performed, MATLAB will exit the loop. In our example, the final loop should look like:

```
for i = 1:10              % i is the index, start at i=1 and stop at i=10
    vec(i) = i + 4;       % for each value of i, set vec(i) = i + 4
                          % all operations have been computed, i will increment by 1 and the ...
                              loop repeats
end                       % ``end'' is required to close the loop
```

Now, we can run this program. In the Workspace you will see that `vec` has been created. You can print the vector by typing `vec` in the command line.

Let's make this a little more complex. Suppose now that we want to create a 10x10 matrix `mat` such that each element within the vector `(i,j)` is defined as the product of `i` (i is the "row index") and `j` (j is the "column index"). We'll begin as before, and create a matrix of all zeros:

```
mat = zeros(10,10);
```

Now, we have to create what's known as a ***nested*** loop, or a loop within a loop. In other words, we have two indexes we care about (the `i` and `j` indices). For each row `i` of `mat`, we need to define each column `j` as `i*j`.

```
for i = 1:10              % i is for our row index
    for j = 1:10          % j is our column index
        mat(i,j) = i*j;   % for each combination of i &j, set mat(i,j) = i*j
    end                   % this ``end'' closes the ``j'' loop
end                       % this ``end'' closes the ``i'' loop
```

The loop

```
    for j = 1:10
        mat(i,j) = i*j;
    end
```

is called the *inner loop* and

```
for i = 1:10
end
```

is called the *outer loop*. Again, we can run our program to see that the correct matrix is produced.

Before moving on, there is one more item that we will introduce. Suppose we want to examine the element `(4,8)` in the matrix we just created (`mat`). If `mat(4,8)` is greater than 10, we want to set it equal to 0. Lastly, if it equals 10 we want to set it to 1. Otherwise (i.e. if it is less than 10), we want to set it to -1. We can perform this test using a ***if*** statement. The way to construct this statement is through a series of conditions. First, we begin by stating

```
if mat(4,8) > 10
```

If this is true, we tell MATLAB to set `mat(4,8)= 0`:

```
mat(4,8) = 0;
```

If `mat(4,8)` is not greater than 10, MATLAB moves to the next *elseif* statement:

```
elseif mat(4,8) == 10
```

Here, if `mat(4,8)` is not greater than 10 but is exactly equal to 10 (this requires a double equal sign), then we tell MATLAB to set `mat(4,8)= 1`:

```
mat(4,8) = 1;
```

We have one final condition. We can do this as above and the next line of code is:

```
elseif mat(4,8) < 10
```

Similar to above, the next line is

```
mat(4,8) = -1;
```

Now, we need to close the loop using the `end` command. In the last “ifelse” statement, if `mat(4,8)` is not greater than or equal to 10 then the only option is for it to be less than 10. Therefore, the last “ifelse” statement can be simplified to just say “else” (i.e. in all other cases).

The above lines of code are summarized here:

```
if mat(4,8) > 10         % First condition. If this is true perform the next line. If not, move
                         % to the next part of the statement
    mat(4,8) = 0;        % If first condition is true, set mat(4,8) to zero

elseif mat(4,8) == 0     % If mat(4,8) did not satisfy the first condition but it is exactly
                         % equal to zero, perform the next line. If not, move to the next part
                         % of the statement
    mat(4,8) = 1;        % If second condition is true, set mat(4,8) to 1

elseif mat(4,8) < 0      % If first two conditions were not true and this is, perform next line

    mat(4,8) = -1;       % If third condition is true, set mat(4,8) to -1

end                      % Close the loop
```

We could have done this manually without an if statement, but in practice these are extremely useful.

Exercises

1. Open and save a new m file. Name it whatever you'd like. Use this file to create the loops specified below. When finished with each part of the problem, run the program to make sure that it returns the correct information.
   (a) Create a 10x1 vector `V1` of all zeros. Then, use a for loop to overwrite each element in the vector. For each row `i`, `V1(i)` should be equal to two times `i`.
   (b) Create a 10x1 matrix `M1` of all zeros. Then, use a for loop to overwrite each element in the vector. For each row `i` and column `j`, `M1(i,j)` should be equal to `i+j`.
   (c) Create a 3x3 identity matrix `M1`. If a given element `(i,j)` of `M1` is equal to zero, set it to -1. To complete this problem, use nested for loops as well as an if statement.

1.7 Functions, ODE solving goodness, and the predator prey model

In this section, we will use the Lotka-Volterra predator prey model to learn how to solve ODEs in MATLAB . To do this, MATLAB has a few built-in programs that will numerically integrate ODEs. Without going in to details on the specific numerical methods, the basic idea is that given a model and set of initial conditions MATLAB will return a matrix that stores the values of each state variable over a specified time frame. We will use the built-in program `ode45` which is a standard and reliable choice for most systems.

Before learning about `ode45`, let's quickly go over the Lotka-Volterra model. Let V be the population size (or density) of prey, P be the population size (or density) of predators. Then the model is given by:

$$\begin{aligned}\frac{dV}{dt} &= bV - aVP \\ \frac{dP}{dt} &= wVP - dP\end{aligned}$$

where *b* is the *per capita* growth rate of prey, *a* is the per predator capture rate, *w* is "conversion" rate of prey into predators, and *d* is the *per capita* death rate of predators. It has two equilibria given by $(V^*, P^*) = (0,0), (d/w, b/a)$. It can be shown that the non-trivial (non-zero) equilibrium is a *center* and that stable oscillations between predator and prey occur through time.

Now let's learn how to simulate this model (i.e. plot the time dynamics of V and P) using MATLAB . To do this, we will use ***functions***. In MATLAB , a function is a file that will return some value. We are going to create two m-files: one will be a "main" file where we define the parameter values, initial conditions, run `ode45` and plot the results. This file will access our second file which will be written as a function. This second file will actually define the model. This most likely sounds confusing, so let's just dive in and do it – this will help it all come together.

Open an m file and title it "Main.m" Put a comment at the top indicating the purpose of the file. Again, we are going to use `ode45` to simulate the model. This first requires the initial conditions of V and P, all parameter values, and the time frame to simulate the mode. The parameters can be any positive numbers you'd like, but for now let's define the following:

```
% This is the main file for simulating the predator-prey model

b = 1;      % Define the birth rate
a = 2;      % Define the capture rate
w = 1;      % Define the conversion rate
d = 0.5;    % Define the death rate
```

Notice that this implies the non-trivial equilibrium is given by $(V^*, P^*) = (0.5, 0.5)$. Assuming that we start at time zero, we now want to choose a maximum time. Let's simulate the model for 50 years:

```
MaxTime = 50;      % Length of time to simulate for
```

Finally, we need to specify the initial conditions of V and P which we will call `V0` and `P0`, respectively.

```
V0 = 1;      % Initial prey density
P0 = 1;      % Initial predator density
```

Now, we will use `ode45` to simulate the model. We will later create an m file that specifies the model and name it "PredPrey.m," but for right now let's assume that we already have this. "PredPrey.m" will return two things: a vector storing the times and a matrix such that each column stores the corresponding values for each state variable at each time in the time vector. The code for this is given by:

```
[t, pop]=ode45(@PredPrey,[0 MaxTime],[V0 P0],[],[b a w d]);
```

Here, `[t, pop]` are the outputs from `ode45` as described above, the first entry `@PredPrey` tells `ode45` to access that file for the model, `[0, MaxTime]` is a vector storing the start and stop time, `[V0 P0]` is a vector storing the initial V and P values, `[]` is an empty vector (it is used for additional options, for our purposes we will ignore this), and the final entry `[b a w d]` is a vector storing all parameter values.

Assuming that we have written "PredPrey.m" we now want to plot the the predator and prey through time. Let's first isolate the population dynamics for each state variable which are again the columns of `pop`:

```
V=pop(:,1); % The first column of pop stores V
P=pop(:,2); % The second column of pop stores P
```

Now, let's plot the prey in black and the predators in red:

```
figure()        % Open a new figure
plot(t,V,'k')   % Plot prey density in black
hold on;        % Don't erase the first plot
plot(t,P,'r')   % Plot predator density in red
```

Now, we can add some axes labels

```
xlabel('Time, t')
ylabel('Predator (black) / Prey (red) densities')
```

Now, we just have to write a separate file that codes up the model itself. Open a *new* m file and save it as "PredPrey.m". As mentioned previously, this will be a `function`. For functions, the first line of code has to specify that it is a function, what it outputs, and then set that equal to the function name and in parentheses put all input values. `ode45` will access this function and the output will be the change in population size (we will call this `popchange`) and requires time, the population sizes/densities, and the parameter values. It should look like:

```
function popchange = PredPrey(t,pops,parameter)
```

All inputs will be specified by `45`. `pops` will be a vector of the population size for V and P, respectively, and `parameter` will be a vector of all parameters that we initially specified. To make things simpler, let's redefine `pops` and `parameter` according to our original definitions:

```
b=parameter(1);
a=parameter(2);
w=parameter(3);
d=parameter(4);

V=pops(1);
P=pops(2);
```

Now, we want to define a vector `popchange` (i.e. the output of this function as specified in the function definition). It will be a vector of the change in V and P:

```
popchange = zeros(2,1);
```

Now, we find these values by simply writing out the original ODEs:

```
popchange(1)= b*V - a*V*P;
popchange(2)= w*V*P - d*P;
```

Save this file, and run "Main.m" and your results should look exactly like Figure 1.7! Play around with parameter values and initial conditions to see how the output changes.

When learning how to do this for the first time, it can seem confusing. However, as you practice it will become more and more clear. Also, now that you have a template for solving a system of ODEs this code can be easily modified to model other systems.

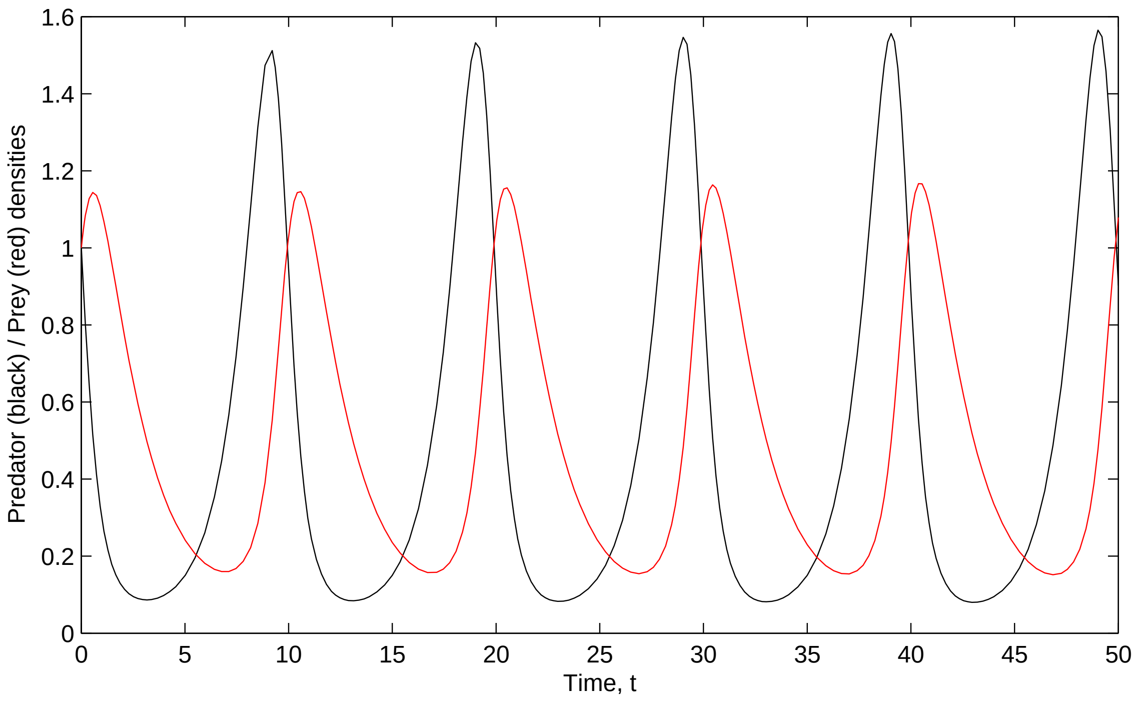


Figure 1.7. Plot of the Lotka-Volterra predator prey model.