

4.3 Functions in Matlab

You've encountered function notation in your mathematics courses, perhaps in algebra, trigonometry, and/or calculus. For example, consider the function defined by

$$f(x) = 2x^2 - 3. \quad (4.1)$$

This function expects a single input x , then outputs a single number $2x^2 - 3$. Sometimes an alternative notation is used to reinforce this concept that the input x is mapped to the output $2x^2 - 3$, namely $f : x \rightarrow 2x^2 - 3$. For example,

$$f : 3 \longrightarrow 2(3)^2 - 3.$$

That is, $f : 3 \rightarrow 15$. Using the notation in (4.1), we proceed in a similar manner.

$$f(3) = 2(3^2) - 3.$$

That is, $f(3) = 15$. Note that there is one input, 3, and one output, 15. The goal of this section is to learn how Matlab emulates this input-output behavior of functions.

Anonymous Functions

As we've seen in previous work, we can use *anonymous functions* in Matlab to emulate the behavior of mathematical functions like $f(x) = 2x^2 - 3$. The general syntax of an anonymous function is

```
fhandle = @(arglist) expr
```

where **expr** is the body of the function, the part that churns your input to produce the output. It can consist of any **single** valid Matlab expression. To the left of **expr** is **(arglist)**, which should be a comma separated list of all input variables to be passed to the function. To the left of the argument list is Matlab's **@** symbol, which creates a *function handle* and stores it in the variable **fhandle**.

You execute the function associated with the function handle with the following syntax.

```
fhandle(arg1, arg2, ..., argN)
```

¹ Copyrighted material. See: <http://msenux.redwoods.edu/Math4Textbook/>

That is, you specify the variable holding the function handle, followed by a comma separated list of arguments in parentheses.

► **Example 1.** *Create a function to emulate the behavior of the mathematical function $f(x) = 2x^2 - 3$.*

We use the syntax **handle = @(arglist) expr** to create and store a function handle in the variable *f*.

```
>> f = @(x) 2*x^2 - 3
f =
    @(x) 2*x^2 - 3
```

We now execute the function by specifying the function handle associated with it, followed by a single argument (the input) in parentheses.

```
>> f(3)
ans =
    15
```

However, note what happens when we pass an array as input to this function.

```
>> x=0:5
x =
     0     1     2     3     4     5
>> f(x)
??? Error using ==> mpower
Matrix must be square.

Error in ==> @(x) 2*x^2 - 3
```

We've failed to make our function “array smart.” You cannot square a vector, as we are reminded in the error message “Matrix must be square.” We need to use array operators to make our anonymous function “array smart.”

```
>> f = @(x) 2*x.^2 -3
f =
    @(x) 2*x.^2 -3
```

Note that this time we used the “dot raised to” operator \wedge which will operate elementwise and square each entry of the input. Note that using an array as input is now successful, and the function is evaluated at each entry of the input vector $\mathbf{x}=0:5$.

```
>> f(x)
ans =
    -3    -1     5    15    29    47
```

Even though we’ve made our function array smart, it will still operate on individual scalar entries.

```
>> f(5)
ans =
    47
```



Making Functions Array Smart. Many of Matlab’s routines require that a function passed as an argument be “array smart.” You need to be aware of this if you pass a function to one of these routines.

Let’s look at another example.

► **Example 2.** Create a function to emulate the mathematical behavior of the function defined by $f(x, y) = 9 - x^2 - y^2$.

Note that there are two inputs to the function f . As an example, we evaluate the function at the point $(x, y) = (2, 3)$.

$$f(2, 3) = 9 - 2^2 - 3^2.$$

Thus, $f(2, 3) = -4$.

We define an anonymous Matlab function to emulate the behavior of f . Note that the $@$ symbol is followed by a comma delimited list of two variables x and y , contained in parentheses.

```
>> f = @(x,y) 9 - x^2 - y^2
f =
    @(x,y) 9 - x^2 - y^2
```

We evaluate the function f at $(x, y) = (2, 3)$ as follows.

```
>> f(2,3)
ans =
    -4
```

Note that this agrees with our hand calculation above.



It's possible to use constants in your anonymous function that are previously defined in your current workspace.

► **Example 3.** Write a function to evaluate $f(x, y, z) = Ax + By + Cz$ at the point $(-1, 2, -3)$, where $A = 2$, $B = 3$, and $C = -4$ are previously defined in the current workspace.

Set the constants $A = 2$, $B = 3$, and $C = -4$.

```
>> A=2; B=3; C=-4;
```

Now, define the anonymous function as follows.

```
>> f = @(x,y,z) A*x + B*y + C*z
f =
    @(x,y,z) A*x + B*y + C*z
```

Now, evaluating the function at $(x, y, z) = (-1, 2, -3)$ by hand,

$$f(-1, 2, -3) = A(-1) + B(2) + C(-3) = (2)(-1) + (3)(2) + (-4)(-3) = 16.$$

Checking in Matlab.

```
>> f(-1,2,-3)
ans =
    16
```

It's important to note that changing the values of A , B , and C after the function definition will have no effect on the anonymous function.

```
>> A=1; B=1; C=1;
>> f(-1,2,-3)
ans =
    16
```

If you want these new constants to be used in the function, you must reevaluate the anonymous function, as the anonymous function retains the values of any constants as they existed in the workspace at the moment of the anonymous function's conception.

```
>> f = @(x,y,z) A*x + B*y + C*z
f =
    @(x,y,z) A*x + B*y + C*z
>> f(-1,2,-3)
ans =
    -2
```

We'll leave it to our readers to check the accuracy of this last result.



Finally, it is possible to construct an anonymous function with no input.

```
>> t = @() datestr(now)
t =
    @() datestr(now)
```

You must call such a function with an empty argument list.

```
>> t()
ans =
04-Mar-2007 00:34:07
```

Function M-Files

Note that the anonymous function syntax **handle = @(args) expr** only allows for a single Matlab expression in its definition body. Moreover, only one output variable is allowed. In this section we will look at *Function M-files*, which are similar to scripts, but allow for both multiple input and output and as many lines as needed in the body of the function.

Like script files, you write function m-files in the Matlab editor. The first line of the function m-file must contain the following syntax.

```
function [out1, out2, ...] = funname(in1, in2, ...)
```

The keyword **function** must be the first word in the function m-file. This alerts the Matlab interpreter that the file contains a function, not a script. Next comes a comma delimited list of output arguments (surrounding brackets are required if more than one output variable is used), then an equal sign, followed by the function name. The function name **funname** must be string composed of letters, digits, and underscores. The first character of the name must be a letter. A valid length for the function name varies from machine to machine, but can be determined by the command **namelengthmax**.

```
>> namelengthmax
ans =
    63
```

If you name your function **funname**, as above, you must save the function m-file as **funname.m**. That is, you must attach **.m** to the function name when saving the function m-file.

After the function name comes a comma delimited list of input variables, surrounded by required parentheses. Input arguments are passed by value and all variables in the body of the function are “local” to the function (we’ll have more to say about local and global variables later). In the body of the function m-file,

you must assign a value to each output variable before the function definition is complete.

Before getting too ambitious, let's start with some simple examples of function M-files that emulate the behavior of the anonymous functions in [Examples 1](#) through [3](#).

► **Example 4.** Write a function m-file to emulate the behavior of the function $f(x) = 2x^2 - 3$.

Open the Matlab editor and enter the following lines.

```
function y=f(x)
y=2*x^2-3;
```

Note how we've followed the syntax rules outlined above.

- i. The first word on the first line is the keyword **function**.
- ii. We are using y as the output variable. Because there is only a single output variable, the surrounding brackets are not required.
- iii. The function name is **f**.
- iv. Finally, surrounded in required parentheses is a single input variable x .

Save the file as **f.m**. Note how we've attached **.m** to the function name **f**.

In [Example 1](#), we found that $f(3) = 15$. Let's test the current function. Return to the command window and enter the following at the Matlab prompt.

```
>> f(3)
ans =
    15
```

As in [Example 1](#), the current function is not “array smart,” so attempting to evaluate the function on an array will fail.

```
>> x=0:5
x =
    0    1    2    3    4    5
>> f(x)
??? Error using ==> mpower
Matrix must be square.

Error in ==> f at 2
y=2*x^2-3;
```

Return to the editor and make the following change to the function m-file.

```
function y=f(x)
y=2*x.^2-3;
```

Note that we have replaced the regular exponentiation symbol \wedge with the array exponentiation symbol \wedge (“dot raised to”). This makes our function “array smart” and we can now evaluate the function at each entry of the vector \mathbf{x} .

```
>> f(x)
ans =
   -3   -1    5   15   29   47
```



In the next example, we use a function m-file to emulate the behavior in **Example 2**.

► **Example 5.** Write a function m-file for $f(x, y) = 9 - x^2 - y^2$. Evaluate the function at $(x, y) = (2, 3)$.

Open the Matlab editor and enter the following lines.

```
function z=f(x,y)
z=9-x^2-y^2;
```

Note that this time we have two input variables, x and y . The function name is **f**, so we again save the file as **f.m**. Evaluate the function at $(x, y) = (2, 3)$ as the Matlab prompt in the command window.

```
>> f(2,3)
ans =
    -4
```

This agrees with the result in [Example 2](#).

Let's stretch this example a bit further. Move to the command window and set the following variables at the Matlab command prompt.

```
>> dogs=3; cats=3;
```

Now, evaluate the function at these variables and store the result in the variable **pets**.

```
>> pets=f(dogs,cats)
pets =
    -4
```

Note that this produces an identical result, highlighting two important points.

- i. The names of the variables in command window do not have to match the names of the input arguments in the function definition. In this example, the *values* of the variables **dogs** and **cats** are passed to the arguments **x** and **y** in the function m-file.
- ii. The value of the output variable in the command workspace does not need to match the value of the output variable in the function definition. In this example, the value of the output variable **z** is returned and stored in the command workspace variable **pets**.



In [Example 3](#), there were three constants involved, A , B , and C . We can pass these additional constants to the function if we add them to the list of input arguments in the function definition.

► **Example 6.** Evaluate the function $f(x, y, z) = Ax + By + Cz$ at $(x, y, z) = (-1, 2, -3)$, where $A = 2$, $B = 3$, and $C = -4$.

Enter the following lines in the Matlab editor.

```
function w=f(x,y,z,A,B,C)
w=A*x+B*y+C*z;
```

Save the file as **f.m**.

Set $A = 2$, $B = 3$, and $C = -4$ in the command workspace at the Matlab prompt.

```
>> A=2; B=3; C=-4;
```

We evaluate the function by passing the appropriate values to the function arguments x , y , and z . In addition, we pass the values of A , B , and C in the command workspace to the functions arguments A , B , and C .

```
>> f(-1,2,-3,A,B,C)
ans =
    16
```

Note that this matches the result in **Example 3**.

Again, it is not required that the names of the command workspace variables match those of the input and output arguments in the function definition. We could just as easily use the current function definition to compute the cost of a purchase at a pet store.

```
>> guppies=10; tadpoles=12; goldfish=4;
>> per_guppy=0.50; per_tadpole=0.35; per_goldfish=0.85;
>> cost=f(guppies,tadpoles,goldfish,per_guppy,
per_tadpole,per_goldfish)
cost =
    12.6000
```

Global Variables. Alternatively, we could declare the constants A , B , and C as global variables at the command prompt in Matlab's workspace. You should declare the variables as global before you assign values to them. As we already have values assigned to variables A , B , and C in the command window workspace, we will clear them, declare them to be global, then assign values.

```
>> clear A B C
>> global A B C
>> A=2; B=3; C=-4;
```

You must also declare these constants as global in the function m-file.

```
function w=f(x,y,z)
global A B C
w=A*x+B*y+C*z;
```

Resave the function m-file as **f.m**. Note that we've made two changes.

- i. We've removed A , B , and C as input arguments. The first line of the function file now reads **function w=f(x,y,z)**.
- ii. We've declared the variables A , B , and C as global. Now, any changes to the global variables in the command workspace will be recognized by the variables in the function workspace, and vice-versa, any change to the global variables in the function's workspace will be recognized by the command workspace. In effect, the two workspaces share the values assigned to the global variables.

Evaluate the function at $(x, y, z) = (-1, 2, -3)$.

```
>> f(-1,2,-3)
ans =
    16
```



More Than One Output

To this point, anonymous functions can seemingly do everything that we've demonstrated via function m-files in **Examples 4-6**. However, two key facts about anonymous functions explain why we should continue to explore function m-files in more depth.

1. Anonymous functions are allowed only one output.
2. The body of an anonymous function is allowed exactly one valid Matlab expression.

Let's look at a function m-file that sends two outputs back to the calling program or Matlab command workspace.

► **Example 7.** Write a function m-file that will compute the area and circumference of a circle of radius r .

Open the editor and enter the following lines.

```
function [area,circumference]=area_and_circumference(r)
area=pi*r^2;
circumference=2*pi*r;
```

Save the file as **area_and_circumference.m**. Set the radius of a particular circle at the Matlab prompt.

```
>> radius=12;
```

Call the function **area_and_circumference** with input argument **radius** and store the resulting area and circumference in the workspace variables A and C .

```
>> [A,C]=area_and_circumference(radius)
A =
  452.3893
C =
  75.3982
```

Functions return output variables in the order in which they are listed in the function definition line within the m-file. Thus,

```
[area,circumference]=area_and_circumference(r)
```

declares that **area** will be the first output returned and **circumference** will be the second.

It is not required that the calling statement (whether from the command line, a script, or another function) contain the same number of output variables as defined by the function m-file. If you specify fewer output variables in the calling statement than are defined in the function m-file, the output variables in the

calling statement will be filled in the order defined in the function m-file. For example, the following command returns and stores the area.

```
>> myarea=area_and_circumference(radius)
myarea =
    452.3893
```

The order of the output variables in the function definition control the order in which results are returned to the calling statement, not the names of the output variables in the calling statement.

```
>> mycircum=area_and_circumference(radius)
mycircum =
    452.3893
```

This could be a crucial error and illustrates why function m-files should contain documentation explaining their use.

Documenting Help in Function M-Files. The first contiguous block of comment files in a function m-file are displayed to the command window when a user enters **help funname**, where **funname** is the name of the function. You terminate this opening comment block with a blank line or an executable statement. After that, any later comments do not appear when you type **help funname**.

The first line of the comment block is special and is used by Matlab's **lookfor** and should therefore contain the function name and a brief description of the function. Ensuing lines should contain syntax of use and sometimes it is useful to provide examples of use. Type **help svd** for an example of a well crafted help file and then type **lookfor svd** to see how the first line of the help file is special.

With these thoughts in mind, let's craft some documentation for our function **area_and_circumference**. Add the following lines to the m-file and save.

```
function [area,circumference]=area_and_circumference(r)
%AREA_AND_CIRCUMFERENCE finds the area and circumference
%of a circle.
% [A,C]=AREA_AND_CIRCUMFERENCE(R) returns the area A and
% circumference C of a circle of radius R.
area=pi*r^2;
circumference=2*pi*r;
```

Type the following to view the help.

```
>> help area_and_circumference
AREA_AND_CIRCUMFERENCE finds the area and circumference
of a circle.
[A,C]=AREA_AND_CIRCUMFERENCE(R) returns the area A and
circumference C of a circle of radius R.
```

“Look For” functions that have the the word “area” in the first line of their help blocks.

```
>> lookfor area
AREA_AND_CIRCUMFERENCE finds the area and circumference
POLYAREA Area of polygon.
...
```

Note that our **area_and_circumference** function is listed along with a number of other function m-files.



No Output

It is not required that a function m-file produce output variables. There are times when we will want to write functions that take input arguments, but then are self contained. The remainder of work needed to be processed is accomplished within the function m-file itself.

► **Example 8.** *The equation $ax^2 + bx + c = 0$, called a quadratic equation, has solutions provided by the quadratic formula.*

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (4.2)$$

Write a function m-file that takes as input the coefficients a , b , and c of the quadratic equation, then, based on the result of the discriminant $D = b^2 - 4ac$, produces the solutions of the quadratic equation.

We begin with the function line and a short help block.

```
function quadratic(a,b,c)
%QUADRATIC finds solutions of ax^2+bx+c=0.
% QUADRATIC(A,B,C) uses the value of the discriminant
% D=B^2-4AC to determine if the quadratic equation
% has two real solutions, 1 real solution, or no real
% solutions, and prints the solutions to the screen.
```

Note the first line: **function quadratic(a,b,c)**. The function **quadratic** requires three input arguments A , B , and C (the coefficients of the quadratic equation $ax^2 + bx + c = 0$), but provides no output.

First, calculate the value of the discriminant.

```
D = b^2 - 4*a*c; %the discriminant
```

If the discriminant is positive, there are two real solutions, given by the quadratic formula (4.2). If the discriminant is zero, then the quadratic formula provides a single solution, namely $x = -b/(2a)$. If the discriminant is negative, then there are no real solutions (we cannot take the square root of a negative number). This calls for an **if..elseif..else** conditional.

```
if D>0
    x1=(-b-sqrt(D))/(2*a);
    x2=(-b+sqrt(D))/(2*a);
    fprintf('The discriminant is %d.\n',D)
    fprintf('Hence, there are two solutions.\n')
    fprintf('The first solution is: %f\n',x1)
    fprintf('The second solution is: %f\n',x2)
elseif D==0
    x=-b/(2*a);
    fprintf('The discriminant is %d.\n',D)
    fprintf('Hence, there is exactly one solution.\n')
    fprintf('The solution is: %f\n',x)
else
    fprintf('The discriminant is %d.\n',D)
    fprintf('Hence, there are no real solutions.\n')
end
```

Save the function as **quadratic.m**.

As a first test, consider quadratic equation $x^2 - 2x - 3 = 0$. Note that the discriminant is $D = (-2)^2 - 4(1)(-3) = 16$ and predicts two real roots. Indeed, the quadratic factors as $(x + 1)(x - 3) = 0$. Thus, the equation has solutions $x = -1$ or $x = 3$.

```
>> quadratic(1,-2,-3)
The discriminant is 16.
Hence, there are two solutions.
The first solution is: -1.000000
The second solution is: 3.000000
```

So far, so good. As a second test, consider the quadratic $x^2 - 4x + 4 = 0$. The discriminant is $D = (-4)^2 - 1(1)(4) = 0$ and predicts exactly one real root. Indeed, the quadratic factors as $(x - 2)^2 = 0$. Thus, the equation has a single real root $x = 2$.

```
>> quadratic(1,-4,4)
The discriminant is 0.
Hence, there is exactly one solution.
The solution is: 2.000000
```

Finally, consider the quadratic equation $x^2 - 2x + 2 = 0$. The discriminant is $D = (-2)^2 - 4(1)(2) = -4$, so the quadratic equation has no real solutions.

```
>> quadratic(1,-2,2)
The discriminant is -4.
Hence, there are no real solutions.
```



4.3 Exercises

In **Exercises 1-6**, write an anonymous function to emulate the given function. In each case, evaluate your anonymous function at the given value(s) of the independent variable.

1. $f(x) = x^2 - 3x - 4$, $f(1)$.
2. $f(x) = 5 - x - x^2 - 2x^4$, $f(1)$.
3. $f(t) = e^{-0.25t}(2 \cos t - \sin t)$, $f(1)$.
4. $g(t) = e^{0.10t}(\cos 2t - 3 \sin 2t)$, $g(1)$.
5. $h(u, v) = \cos u \cos v$, $h(1, 2)$.
6. $f(u, v) = \frac{\sin(u^2 + v^2)}{u^2 + v^2}$, $f(1, 2)$.

In **Exercises 7-12**, write a function M-file to emulate the function in the given exercise. In each case, evaluate the given function at the given value of the independent variable.

7. The function and evaluation given in **Exercise 1**.
8. The function and evaluation given in **Exercise 2**.
9. The function and evaluation given in **Exercise 3**.
10. The function and evaluation given in **Exercise 4**.
11. The function and evaluation given in **Exercise 5**.

12. The function and evaluation given in **Exercise 6**.

13. Consider the anonymous function

```
f=@(x) x.*exp(-C*x.^2);
```

Write a **for** loop that will plot this family of functions on the interval $[-1, 1]$ for C in $\{-1, -0.8, -0.6, \dots, 1\}$.

14. Consider the anonymous function

```
f=@(x) C*exp(t-t.^2/2);
```

Write a **for** loop that will plot this family of functions on the interval $[-2, 4]$ for C in $\{-1, -0.8, -0.6, \dots, 1\}$.

15. Consider the function

$$f(x) = xe^{-Cx^2}.$$

Write a function M-file that will emulate this function and allow the passing of the parameter c . That is, your first line of your function should look as follows:

```
function y=f(x,c)
```

In cell mode (or a script M-file), write

² Copyrighted material. See: <http://msenux.redwoods.edu/IntAlgText/>

a **for** loop that will plot this family of functions on the interval $[-1, 1]$ for C in $\{-1, -0.8, -0.6, \dots, 1\}$.

16. Consider the function

$$f(x) = ce^{x-x^2/2}.$$

Write a function M-file that will emulate this function and allow the passing of the parameter c . That is, your first line of your function should look as follows:

```
function y=f(x,c)
```

In cell mode (or a script M-file), write a **for** loop that will plot this family of functions on the interval $[-2, 4]$ for C in $\{-1, -0.8, -0.6, \dots, 1\}$.

17. The sequence

$$1, 1, 2, 3, \dots, a_{n-2}, a_{n-1}, a_n, \dots$$

where $a_n = a_{n-2} + a_{n-1}$, is called a *Fibonacci* sequence. Write a function that will return the n th term of the Fibonacci sequence. The first line of your function should read as follows.

```
function term=fib(n)
```

Call this function from cell mode (or a script M-file) and format the output.

18. Adjust the function M-file created in **Exercise 17** so that it returns a vector containing the first n terms of the Fibonacci sequence. Call this function from cell mode (or a script M-file) and format the output.

19. Euclid proved that there were an infinite number of primes. He also believed that there was an infinite number of *twin primes*, primes that differ by 2 (like 5 and 7 or 11 and 13), but no one has been able to prove this conjecture for the past 2300 years. Write a function that will produce all twin primes between two inputs, integers a and b .

20. In mathematics, a *sexy prime* is a pair of primes $(p, p+6)$ that differ by 6. Write a function that will produce all sexy primes between two inputs, integers a and b .

21. The factorial of nonnegative integer n is denoted by the mathematic notation $n!$ and is defined as follows.

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n(n-1)(n-2)\cdots 1, & \text{if } n \neq 0 \end{cases}$$

Write a function M-file **myfactorial** that returns the factorial when a nonnegative integer is received as input. Use cell mode (or a script M-file) to test your function at $n = 0$ and $n = 5$. *Hint: Consider using Matlab's **prod** function.*

22. The *binomial coefficient* is defined by

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Write a function M-file called **binom** that takes two inputs, n and k , then returns the binomial coefficient. Use the **myfactorial** function written in **Exercise 21**. Use cell mode to test your function for $n = 5$ and $k = 2$.

23. The first few rows of *Pascal's Triangle* look like the following.

1				
1	1			
1	2	1		
1	3	3	1	
1	4	6	4	1

The n th row holds the binomial coefficients

$$\binom{n}{k}, \quad k = 0, 1, \dots, n.$$

Write a function to produce Pascal's Triangle for an arbitrary number r of rows. Your first line should read as follows.

```
function pascalTriangle(r)
```

This function should use nested **for** loops to produce the rows of Pascal's Triangle. The function should also use the **fprintf** command to print each row to the screen. Use cell mode to test your function by entering the following line in your cell enabled editor.

```
pascalTriangle(8)
```

Your function should employ the **binom** function written in **Exercise 20**. *Hint: The format to use for the **fprintf** command is the tricky part of this exercise. Before entering the inner loop to compute the coefficients for a particular row, set **format=**' (the empty string), then build the string in the in-*

*ner loop with **format=[format,'%6d']**. Once the inner loop completes, you'll need to add a line return character before executing **fprintf(format,row)**.*

24. Rewrite the **quadratic** function m-file of **Example 8** by replacing the **if..elseif..else** conditional with Matlab's **switch..case..otherwise** conditional.

25. Rewrite the **quadratic** function m-file of **Example 8** to produce two complex solutions of the quadratic equation $ax^2 + bx + c = 0$ in the case where the discriminant $D = b^2 - 4ac < 0$.

26. Write a function called **mysort** that will take a column vector of numbers as input, sort them in ascending order, then return the sorted list as a column vector. Test the speed of your routine with the following commands.

```
in=rand(10000,1);
>> tic,mysort(in); toc
```

Compare the speed of your sorting routine with Matlab's.

```
>> tic,sort(in); toc
```

Note: This is a classic problem in computer science, a ritual of passage, like crossing the equator for the first time. There are very sophisticated algorithms for attacking this problem, but try to see what you can do with it on your own instead of searching for a sort routine on the internet.

27. Write a function called **mymax** that will take a column vector of numbers as input and find the largest entry. The routine should spit out two numbers, the largest entry m and its position k in the vector. Test the speed of your routine with the following commands.

```
in=rand(1000000,1);
>> tic, [m,k]=mymax(in); toc
```

Compare the speed of your maximum routine with Matlab's.

```
>> tic, [m,k]=max(in); toc
```

28. Write a function called **mymin** that will take a column vector of numbers as input and find the smallest entry. The routine should spit out two numbers, the smallest entry m and its position k in the vector. Test the speed of your routine with the following commands.

```
in=rand(10000,1);
>> tic, [m,k]=mymin(in); toc
```

Compare the speed of your maximum routine with Matlab's.

```
>> tic, [m,k]=min(in); toc
```

29. According to *Wikipedia*, the solutions to the *cubic equation*

$$ax^3 + bx^2 + cx + d = 0$$

are found as follows. First, compute

$$q = \frac{3ac - b^2}{9a^2}$$

and

$$r = \frac{9abc - 27a^2d - 2b^3}{54a^3}.$$

Next, compute

$$s = \sqrt[3]{r + \sqrt{q^3 + r^2}}$$

and

$$t = \sqrt[3]{r - \sqrt{q^3 + r^2}}.$$

The solutions are, first,

$$x_1 = s + t - \frac{b}{3a},$$

then

$$x_2 = -\frac{s+t}{2} - \frac{b}{3a} + \frac{\sqrt{3}}{2}(s-t)i,$$

and

$$x_3 = -\frac{s+t}{2} - \frac{b}{3a} - \frac{\sqrt{3}}{2}(s-t)i,$$

where $i = \sqrt{-1}$. Write a function to compute the roots of the cubic equation. The first line should read as follows.

```
function [x1,x2,x3]=cardano(a,b,c,d)
```

Set up a test to verify that your implementation of the solution of the cubic equation is working correctly.

4.3 Answers

1.

```
>> f=@(x) x^2-3*x-4;
>> f(1)
ans =
    -6
```

3.

```
>> f=@(t) exp(-0.25*t)*(2*cos(t)-sin(t));
>> f(1)
ans =
    0.1862
```

5.

```
>> h=@(u,v) cos(u)*cos(v);
>> h(1,2)
ans =
   -0.2248
```

7. The function M-file:

```
function y=f(x)
y=x^2-3*x-4;
```

Calling the function from the command line.

```
>> f(1)
ans =
    -6
```

9. The function M-file:

```
function y=f(t)
y=exp(-0.25*t)*(2*cos(t)-sin(t));
```

Calling the function from the command line.

```
>> f(1)
ans =
    0.1862
```

11. The function M-file:

```
function z=h(u,v)
z=cos(u)*cos(v);
```

Calling the function from the command line.

```
>> h(1,2)
ans =
   -0.2248
```

13. Set the domain.

```
x=linspace(-1,1);
```

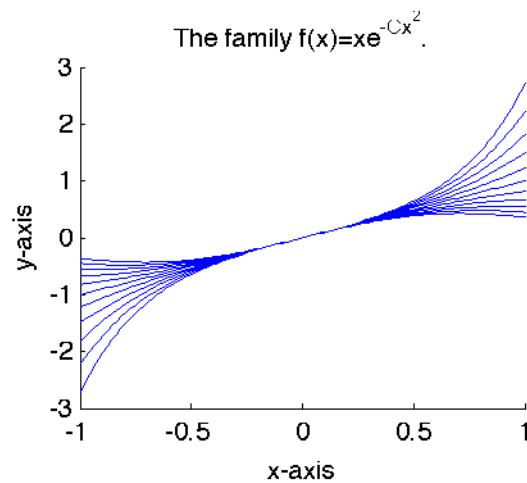
Write the **for** loop.

```

for C=-1:0.2:1
    f=@(x) x.*exp(-C*x.^2);
    y=f(x);
    line(x,y)
end

```

Annotating the plot produces the following image.



15. Write the function M-file.

```

function y=f(x,c)
y=x.*exp(-c*x.^2);

```

Set the domain.

```

x=linspace(-1,1);

```

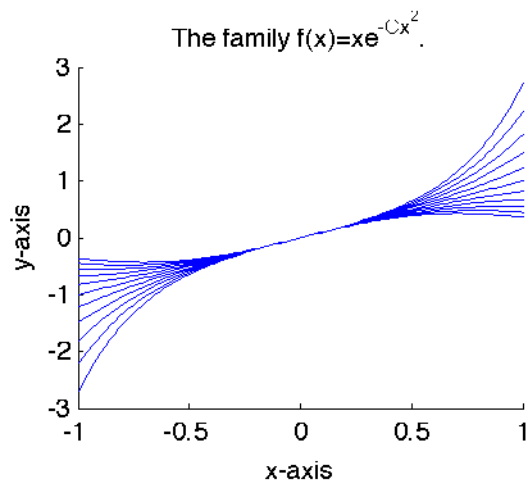
Write the **for** loop.

```

for C=-1:0.2:1
    y=f(x,C);
    line(x,y)
end

```

Annotating the plot produces the following image.



17. Save the following as **fib.m**.

```
function c=fib(n)
if n==1
    term=1;
    return
elseif n==2
    term=1;
    return
else
    a=1;
    b=1;
    for k=3:n
        c=a+b;
        a=b;
        b=c;
    end
end
end
```

At the command line or in the editor with cell mode enabled, enter and execute the following as a test.

```
n=7;
nterm=fib(n);
fprintf('The n = %d term of the sequence is: %d\n', n,nterm)
```

The output is correct.

```
The n = 7 term of the sequence is: 13
```

Readers should check the behavior for other values of n .

19. Save these lines as **twinprimes.m**. The `||` is Matlab's "short-circuit or." If the first the first part of the "or" expression evaluates as true, then the interpreter doesn't bother with the second part of the "or" expression, as it is already true. Note that we've also included an error message, which when executed, displays to the command window and stops program execution.

```
function twins=twinprimes(a,b)
if round(a)~=a || round(b)~=b
    error('The inputs a and b must be integers.')
end
candidates=a:b;
twins=[];
candidates=candidates(isprime(candidates));
len=length(candidates);
for k=1:(len-1)
    if candidates(k+1)-candidates(k)==2
        twins=[twins;candidates(k:k+1)];
    end
end
```

Test the function.

```
>> twins=twinprimes(1,100)
twins =
     3     5
     5     7
    11    13
    17    19
    29    31
    41    43
    59    61
    71    73
```

Looks good. We leave to the reader to check that we have all of the twin primes between 1 and 100.

21. Save the following lines as **myfactorial.m**. Again we're error checking. If n is not an integer or if it is not the case that n is nonnegative, we send an error message to the command window and halt program execution. The **&&** is Matlab's "short-circuit and." If the first part of the and statement evaluates as false, the interpreter doesn't bother with the second part of the "and" statement, saving some time.

```
function fact=myfactorial(n)
if round(n)~=n or ~(n>=0)
    error('n must be a nonnegative integer')
end
if n==0
    fact=1;
else
    fact=prod(1:n);
end
```

Checking.

```
>> fact=myfactorial(0)
fact =
     1
>> fact=myfactorial(5)
fact =
    120
```

Looks correct.

- 23.** Enter the following lines and save a **pascalTriangle.m**.

```
function pascalTriangle(rows)
for n=0:rows
    row=[];
    format='';
    for k=0:n
        row=[row,binom(n,k)];
        format=[format,'%6d'];
    end
    format=[format,'\n'];
    fprintf(format,row)
end
```

This function will only work if you have written a successful binomial coefficient function **binom.m** (see **Exercise 22**), either with a personalized **myfactorial.m** (see **Exercise 21**) or Matlab's built in **factorial.m**. Execute the following at the command prompt.

```
>> pascalTriangle(5)
 1
 1  1
 1  2  1
 1  3  3  1
 1  4  6  4  1
 1  5 10 10  5  1
```

- 27.** Enter the following code in the editor and save as **mymax.m**.

```
function [m,k]=mymax(v)
m=v(1);
k=1;
for i=1:length(v)
    if v(i)>m
        m=v(i);
        k=i;
    end
end
end
```

Test it on something where you can easily spot the maximum, as is the case in the 5th position of the following vector.

```
>> v=[2 4 7 8 11 8 4 2 5]
v =
     2     4     7     8    11     8     4     2     5
>> [m,i]=mymax(v)
m =
    11
i =
     5
```

Now for a comparison. First, **mymax**.

```
>> tic, [m,i]=mymax(v); toc
Elapsed time is 0.107614 seconds.
```

Now using Matlab's **max** routine.

```
>> tic, [m,i]=max(v); toc
Elapsed time is 0.040215 seconds.
```