Chapter 3

Functions and Scripts

3.1 Built-in (Intrinsic) Mathematical Functions

A simple function in mathematics, `f(x)`, associates a unique number to each value of `x`. The function can be expressed in the form `y = f(x)`, where `f(x)` is usually a mathematical expression in terms of `x`. A value of `y` (output) is obtained when a value of `x` (input) is substituted in the expression. Many functions are programmed inside MATLAB as built-in functions, and can be used in mathematical expressions simply by typing their name with an argument; examples are `sin(x)`, `cos(x)`, `sqrt(x)`, and `exp(x)`.

MATLAB has a plethora of built-in functions for mathematical and scientific computations. Remember that the arguments to trigonometric functions are given in radians (same as with `C++`). A function has a name and an argument list provided in the parentheses. For example, the function that calculates the square root of a number is `sqrt(x)`. Its name is `sqrt`, and the argument is `x`. When the function is used, the argument can be a number, a variable that has been assigned a numerical value, or an expression. Functions can also be included in arguments, as well as in expressions.

```
>> sqrt(4)                    % argument is a number

ans =

      2


>> sqrt(25 + 12*7)            % argument is an expression

ans =

   10.4403

>> sqrt(23 + 9*sqrt(64))      % argument includes a function

ans =

    9.7468


>> (12 + 500/4)/sqrt(136)      % function is included in an
                               %   argument

ans =

   11.7477
```

Lists of commonly used elementary MATLAB mathematical built-in functions are listed below.

A complete list of functions organized by name of category can be found in the `Help` Window.
`HELP elfun`

Trigonometric Math Functions	
Function	**Description**
`sin(x), sinh(x)`	sine and hyperbolic sine
`asin(x),` `asinh(x)`	inverse sine and inverse hyperbolic sine
`cos(x), cosh(x)`	cosine and hyperbolic cosine
`acos(x),` `acosh(x)`	inverse cosine and inverse hyperbolic cosine
`tan(x), tanh(x)`	tangent and hyperbolic tangent
`atan(x),` `atanh(x)`	inverse tangent and inverse hyperbolic tangent
`atan2(y,x)`	four-quadrant inverse tangent
`sec(x), sech(x)`	secant and hyperbolic secant
`asec(x),` `asech(x)`	inverse secant and inverse hyperbolic secant
`csc(x), csch(x)`	cosecant and hyperbolic cosecant
`acsc(x),` `acsch(x)`	inverse cosecant and inverse hyperbolic cosecant
`cot(x), coth(x)`	cotangent and hyperbolic cosecant
`acot(x),` `acoth(x)`	inverse cotangent and inverse hyperbolic cotangent

Exponential	
Function	**Description**
`exp(x)`	exponential of the elements of X, e to the X. For complex Z=X+i*Y, EXP(Z) = EXP(X)*(COS(Y)+i*SIN(Y))
`log(x)`	natural logarithm

Exponential	
Function	**Description**
log2(x)	base 2 logarithm and dissect floating-point numbers into exponent and mantissa
log10(x)	common (base 10) logarithm
pow2	Base 2 power and scale floating point number
realpow	Power that will error out on complex result
realllog	Natural logarithm of real number
realsqurt	Square root of number greater than or equal to zero
sqrt	Square roo
nextpow2	Next higher power of 2

Complex	
Function	**Description**
abs(x)	absolute value and complex magnitude
angle(H)	returns the phase angles, in radians, of a matrix with complex elements
complex(a,b)	construct complex data from real and imaginary compo-nents return a + bi
conj(x)	complex conjugate of X. For a complex X, CONJ(X) = REAL(X) - i*IMAG(X)
imag(x)	imaginary part of a complex number
real(x)	real part of complex number
unwrap	Unwrap phase angle
isreal	True for real array
xplxpair	Sort numbers into compiles conjugate pairs

Rounding and Remainder	
Function	**Description**
`ceil(x)`	`round x to nearest integer toward infinity`
`fix(x)`	`rounds the elements of X to the nearest integers towards zero`
`floor(x)`	`rounds the elements of X to the nearest integers towards minus infinity`
`gcd(a,b)`	`is the greatest common divisor of corresponding elements of A and B`
`lcm(a,b)`	`the least common multiple of corresponding elements of A and B`
`mod(x,y)`	`modulus (signed remainder after division)`
`nchoosek(n,k)`	`binomial coefficient or all combinations`
`rem(x,y)`	`remainder after division`
`round(x)`	`round to nearest integer`
`sign(x)`	`signum function`

3.2 Scripts and Functions

When you work in MATLAB, you are working in an interactive environment that stores the variables you have defined and allows you to manipulate them throughout a session. You do have the ability to save groups of commands in files that can be executed many times. Actually MATLAB has two kinds of command files, called M-files. The first is a script M-file. If you save a bunch of commands in a script file called `MYFILE.m` and then type the word MYFILE at the MATLAB command line, the commands in that file will be executed just as if you had run them each from the MATLAB command prompt (assuming MATLAB can find where you saved the file). A good way to work with MATLAB is to use it interactively, and then edit your session and save the edited commands to a script file. You can save the session either by cutting and pasting or by turning on the **`diary`** feature (use the on-line help to see how this works by typing **`help diary`**).

3.2.1 User-Defined Functions

Frequently, in computer programs, there is a need to calculate the value of functions that are not built-in. When a function expression is simple and needs to be calculated only once, it can be typed as part of the program. However, when a function needs to be evaluated many times for different values of arguments it is convenient to create a "user-defined" function. Once the new function is created (saved) it can be used just like the built-in functions.

One of the most important aspects of MATLAB is the ability to write your own functions, which can then be used and reused just like intrinsic MATLAB functions. A function file is a file with an

m extension (e.g., `MYFUNC.m`) that begins with the word `function`. Basically, functions are M-files that can accept input arguments and return output arguments. The name of the M-file and of the function should be the same. Functions operate on variables within their own workspace, separate from the workspace you access at the MATLAB command prompt.

A good example is `height.m`. (Note: To view a file: type filename, e.g., `type height`)

```
function h = height(g, v, time)
%   HEIGHT(g,v,time) returns height a ball reaches when subject to
%   constant gravitational acceleration.
%   Arguments:
%     g - gravitational acceleration
%     v - initial velocity
%     time - time at which height is desired
h = v * time - g * time * time * 0.5;
```

The first line of a function M-file starts with the keyword function. It gives the function name and order of arguments. In this case, there are up to two input arguments and one output argument.

In the `Command Window`, if you run the function using 9.81 for g, 10 for v and 1 for time, the result is:

```
>> height (9.81, 10, 1)

ans =

    5.0950
```

3.2.2 Creating a Function File

Function files are created and edited in the `Editor/Debugger Window`. This window is opened from the `Command Window`. In the `File` menu, select `New`, and them select `M-file`. once the `Editor/Debugger Window` opens it looks like that below. The commands of the file can be typed line after line. The first line in a function file must be the function definition line.

3.2.3 Function Definition Line

The first executable line in a function must be the function definition line. Otherwise the file is considered a script file. The function definition line:

- Defines the file as a function file
- Defines the name of the function
- Defines the number and order of the input and output arguments

The form of the function definition line is:

```
function [output arguments] = function_name(input arguments)
```

The word function must be the 1st word and must be typed in lower-case letters

A list of output arguments typed inside brackets.

The name of the function

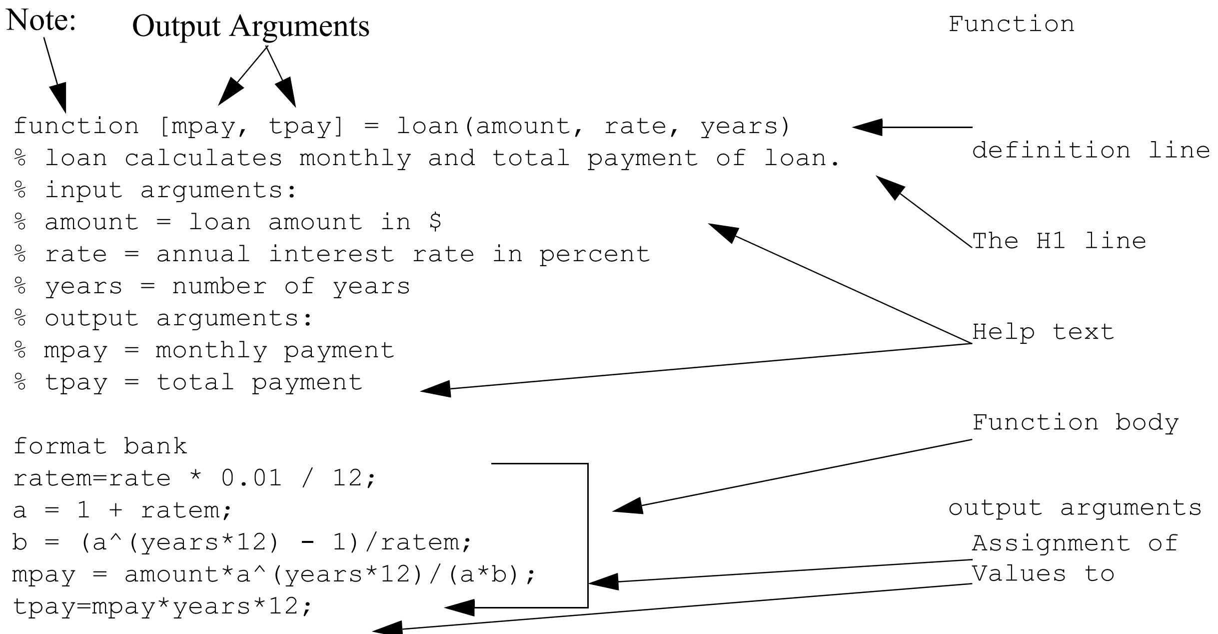
A list of input arguments typed inside parentheses.

The input and output arguments are used to transfer data into and out of the function. The input arguments are listed inside parentheses following the function name. Usually, there is at least one input argument, although it is possible to have a function that has no input arguments. If there are more than one, the input arguments are separated with commas. The computer code that performs the calculations within the function file is written in terms of the input arguments and assumes that the arguments have assigned numerical values. This means that the mathematical expressions in the function file must be written according to the dimensions of the arguments. In the example below, there are three input arguments (amount, rate, years), and in the mathematical expressions they are assumed to be scalars. The actual values of the input arguments are assigned when the function is used (called). Similarly, if the input arguments are vectors or arrays, the mathematical expressions in the function body must be written to follow linear algebra or element-by-element calculations

The output arguments, which are listed inside brackets on the left side of the assignment operator in the function definition line, transfer the output from the function file. Function files can have none, one, or several output arguments. If there are more than one, the output arguments are separated with commas. If there is only one output argument it can be typed without brackets. In order for the function file to work, the output arguments must be assigned values in the computer program that is in the function body. In the example below, there are two output arguments `[mpay, tpay]`. When a function does not have an output argument, the assignment operator in the function definition line can be omitted. A function without an output argument can, for example, generate a plot or print data to a file.

3.2.4 loan Example in MATLAB:

.



3.2.5 loan Example in C++

```
void function loan(double amount, double rate, int years,
                   double & mpay, double & tpay)
{
  /*
  * loan calculates monthly and total payment of loan.
  *  input arguments:
  *  amount = loan amount in $
  *  rate = annual interest rate in percent
  *  years = number of years
  *  output arguments:
  *  mpay = monthly payment
  *  tpay = total payment
 */
   cout << fixed << showpoint << setprecision(2);
   double ratem = rate * 0.01 / 12;
   double a = 1 + ratem;
   double b = (pow(a,(years*12)) - 1)/ratem;
   mpay = amount*pow(a,(years*12))/(a*b);
   tpay = mpay*years*12;
}
```

It is also possible to transfer string into a function file. This is done by typing the string as part of the input variables (text enclosed in single quotes). Strings can be used to transfer names of other functions into the function file.

Usually, all the input to, and output from, a function file are transferred through the input and output arguments. In addition, however, all the input and output features of script files are valid and can be used in function files. This means that any variable that is assigned a value in the code of the function file will be displayed on the screen unless a semicolon is typed at the end of the command. In addition, the input command can be used to input data interactively, and the **`disp`**, **`fprintf`**, and **`plot`** commands can be used to display information on the screen, save to a file, or plot figures just as in a script file. The following are examples of function definition lies with different combinations of input and output arguments.

Function Definition Line MATLAB	C++

`function[mpay,tpay] = loan(amount, rate,years)`
 	
`% Three input arguments, two output arguments`
	`void loan(double amount,`
	`double rate, double years,`
	`double & mpay,`
	`double & tpay)`

`function[A] = rectArea(a,b)`
 	
`% Two input arguments, one output argument`
	`double rectArea(double a, double b)`
	`or`
	`void rectArea (double a, double b,`
	`double & A)`

`function A = rectArea(a,b)`
 	
`% same as the above`
	`same as the above`

`function [V,S] = sphereVolArea(r)`
 	
`% One input variable, two output variables`
	`void sphereVolArea(double r,`
	`double & V,`
	`double & S)`

`function trajectory(v,h,g)`
 	
`% Three input arguments, no output arguments`
	`void trajectory(double v, double h,`
	`double g)`

3.3 Documentation is Important

The `H1` line and help text lines are comment lines (lines that begin with the percent `%` sign) following the function definition line. They are optional, but frequently used to provide information about the function. The **`H1`** line is the first comment line and usually contains the name and a short definition of the function. When a user types (in the Command Window) **lookfor a_word**,

MATLAB searches for **a_word** in the H1 lines of all the functions, and if a match is found, the H1 line that contains the match is displayed.

The **`help`** text lines are comment lines that follow the H1 line. These lines contain an explanation of the function and any instructions related to the input and output arguments. The comment lines that are typed between the function definition line and the first non-comment line (the H1 line and the **help** text) are displayed when the user types help function_name in the Command Window. This is true for MATLAB build-in functions as well as the user-defined functions.

```
>> lookfor height
height.m: %   HEIGHT(g,v,time) returns height a ball reaches when subject to
```

The next several lines, up to the first blank or executable line, are comment lines that provide the **`help`** text. These lines are printed when you type **help height**.

```
>> help height

    HEIGHT(g,v,time) returns height a ball reaches when subject to
    constant gravitational acceleration.
    Arguments:
      g - gravitational acceleration
      v - initial velocity
      time - time at which height is desired
```

3.4 Using the Function File

A user-defined function is used in the same way as a built-in function. The function can be called from the `Command Window`, or from another function. To use the function file, the directory where it was saved **`must`** either be in the current directory or be in the search path. You may need to alter where your current directory is pointing to.

A function can be used by assigning its output to a variable (or variables), as a part of a mathematical expression, as an argument in another function, or just by typing its name in the `Command Window` or in a script file. In all cases the user must know exactly what the input and output arguments are. An input argument can be a number, an expression, or it can be a variable that has an assigned value. The arguments are assigned according to their position in the input and output argument lists in the function definition line.

Two of the ways that a function can be used are illustrated below with the user-defined loan function that calculates the monthly and total payment (two output arguments) of a loan. The input arguments are the loan amount, annual interest rate, and the length (number of years) of the loan. In the first illustration, the loan function is used with numbers as input arguments:

```
>> [month,total] = loan(25000,7.5,4)

month =

        600.72
```

```
total =

      28834.47
```

In the second example, the loan function is used with two preassigned variables and a number as the input arguments:

```
>> a = 70000
a =
      70000.00

>> b = 6.5
b =
          6.50

>> [month, total] = loan(a,b,30)
month =
        440.06

total =
     158423.02
>>
```

In the third example, the loan function is called, but only one return value is captured. Note that only "month" is captured.

```
>> month = loan(a,b,30)
month =
        440.06

>> total
??? Undefined function or variable 'total'.

>>
```

Note: It is very easy to get unexpected results if you give the same name to different functions, or if you give a name that is already used by MATLAB. Prior to saving a function that you write, it is useful to use the **`which`** command to see if the name is already in use.

3.4.1 Number of Arguments to Functions

MATLAB is very flexible about the number of arguments that are passed to and from a function. This is especially useful if a function has a set of predefined default values that usually provide good results. For example, suppose you write a function that iterates until a convergence criteria is met or a maximum number of iterations has been reached. One way to write such a function is to make the convergence criteria and the maximum number of iterations be optional arguments. The following function attempts to find the value of x such that `ln(x)=ax`, where `a` is a parameter.

```
function x=SolveIt(a,tol,MaxIters)
if nargin<3 | isempty(MaxIters), MaxIters=100; end
```

```
if nargin<2 | isempty(tol), tol=sqrt(eps); end
x=a;
for i=1:MaxIters
    lx=log(x);
    fx=x.*lx-a;
    x=x-fx./(lx+1);
    disp([x fx])
    if abs(fx)<tol, break; end
end
```

In this example, the command `nargin` means "number of input arguments" and the command `isempty` checks to see is a variable is passed but is empty (an empty variable is created by setting it to []). An analogous function for the number of output arguments is `nargout`; many times it is useful to put a statement like

```
if nargout<2, return; end
```

into your function so that the function does not have do computations that are not requested.

It is possible that you want nothing or more than one thing returned from a procedure. For example

```
function [m,v]=MeanVar(X)
% MeanVar Computes the mean and variance of a data matrix
% SYNTAX
%     [m,v]=MeanVar(X);
n=size(X,1);
m=mean(X);
if nargout>1
    temp=X-m(ones(n,1),:);
    v=sum(temp.*temp)/(n-1);
end
```

To use this procedure call it with `[mu,sig]=MeanVar(X)`. Notice that is only computes the variance if more than one output is desired. Thus, the statement `mu=MeanVar(X)` is correct and returns the mean without computing the variance.

3.5 Handling Name Conflicts

Suppose that you do not know that sum is a built-in function and type the MATLAB satement

```
>> x = 1;
>> y = 2;
>> z = 3;
>> sum = x + y + z;
```

The name sum now represents a variable and MATLAB's built-in sum function is hidden (you can check this with the command who).

When a name is typed at the prompt or used in an arithmetic expression the MATLAB interpreter evaluates the name using the following in the order they are given:
1) looks to see if it is the name of a variable
2) look to see if it is the name of a built-in function

3) looks in the current directory to see if it is the name of a script file
4) looks in the MATLAB search path for a script file matching the name.

Clearing the variable sum (clear sum) reactivates the built-in function sum.

3.6 Error Checking:

Good documentation is very important but it is also useful to include some error checking in your functions. This makes it much easier to track down the nature of problems when they arise. For example, if some arguments are required and/or their values must meet some specific criteria (they must be in a specified range or be integers) these things are easily checked.

The command error(‘`message`’) inside a function or script aborts the execution, displays the error message, and returns control to the keyboard.

```
function a = areaCircle(r)
if nargin ~= 1
    error('Sorry, need one argument')
end
if length(r) == 1
    ...
end
```

For example, consider the function `areaCircle` listed above. This is intended for a scalar. The input is needed. The command `error` in a function file prints out a specified error message and returns the user to the MATLAB command line.

An important feature of MATLAB is the ability to pass a function to another function. For example, suppose that you want to find the value that maximizes a particular function, say `f(x)= x*exp(-0.5x2)`. It would useful not to have to write the optimization code every time you need to solve a maximization problem. Instead, it would be better to have solver that handles optimization problems for arbitrary functions and to pass the specific function of interest to the solver. For example, suppose we save the following code as a MATLAB function file called MYFUNC.m

```
function fx=myfunc(x)
fx=x.*exp(-0.5*x.^2);
```

Furthermore suppose we have another function call MAXIMIZE.m which has the following calling syntax

```
function x=MAXIMIZE(f,x0)
```

The two arguments are the name of the function to be maximized and a starting value where the function will begin its search (this is the way many optimization routines work). One could then call MAXIMIZE using

```
x=maximize(‘myfunc’,0)
```

and, if the MAXIMIZE function knows what it’s doing, it will return the value 1. Notice that the word `myfunc` is enclosed in single quotes. It is the name of the function, passed as a string variable, that is passed in. The function MAXIMIZE can evaluate MYFUNC using the `feval` command. For example, the code

```
fx=feval(f,x)
```

is used to evaluate the function. It is important to understand that the first argument to `feval` is a string variable (you may also want to find out about the command `eval`).

It is often the case that functions have auxiliary parameters. For example, suppose we changed MYFUNC to

```
function fx=myfunc(x,mu,sig)
fx=x.*exp(-0.5*((x-mu)./sig).^2);
```

Now there are two auxiliary parameters that are needed and MAXIMIZE needs to be altered to handle this situation. MAXIMIZE cannot know how many auxiliary parameters are needed, however, so MATLAB provides a special way to handle just this situation. Have the calling sequence be

```
function x=MAXIMIZE(f,x0,varargin)
```

and, to evaluate the function, use

```
fx=feval(f,x,varargin{:})
```

The command `varargin` (variable number of input arguments) is a special way that MATLAB has designed to handle variable numbers of input arguments. Although it can be used in a variety of ways the simplest is shown here, where it simply passes all of the input arguments after the second on to feval. Don't worry too much if this is confusing at first. Until you start writing code to perform general functions like MAXIMIZE you will probably not need to use this feature in your own code, but it is handy to have an idea of what its for when you are trying to read other peoples' code.

3.7 Debugging

Bugs in your code are inevitable. Learning how to debug code is very important and will save you lots of time and aggravation. Debugging proceeds in three steps. The first ensures that your code is syntactically correct. When you attempt to execute some code, MATLAB first scans the code and reports an error the first time it finds a syntax error. These errors, known as compile errors, are generally quite easy to find and correct (once you know what the right syntax is). The second step involves finding errors that are generated as the code is executed, known as run-time errors. MATLAB has a built-in editor/debugger and it is the key to efficient debugging of run-time errors. If your code fails due to run time errors, MATLAB reports the error and provides a trace of what was being done at the point where the error occurred. Often you will find that an error has occurred in a function that you didn't write but was called by a function that was called by a function that was called by a function (etc.) that you did write. A safe first assumption is that the problem lies in your code and you need to check what your code was doing that led to the eventual error.

The first thing to do with run-time errors is to make sure that you are using the right syntax in calling whatever function you are calling. This means making sure you understand what that syntax is. Most errors of this type occur because you pass the wrong number of arguments, the arguments you pass are not of the proper dimension or the arguments you pass have inappropriate values. If the source of the problem is not obvious, it is often useful to use the debugger. To do this, click on File and the either Open or New from within MATLAB. Once in the editor, click on Debug, then on Stop if error. Now run your procedure again. When MATLAB encounters an error, it now enters a debugging mode that allows you to examine the values of the variables in the various

functions that were executing at the tie the error occurs. These can be accessed by selecting a function in the stack on the editor's toolbar. Then placing your cursor over the name of a variable in the code will allow you to see what that variable contains. You can also return to the MATLAB command line and type commands. These are executed using the variables in the currently selected workspace (the one selected in the Stack). Generally a little investigation will reveal the source of the problem (as in all things, it becomes easier with practice).

There is a third step in debugging. Just because your code runs without generating an error message, it is not necessarily correct. You should check the code to make sure it is doing what you expect. One way to do this is to test it one a problem with a know solution or a solution that can be computed by an alternative method. After you have convinced yourself that it is doing what you want it to, check your documentation and try to think up how it might cause errors with other problems, put in error checks as appropriate and then check it one more time. Then check it one more time.

3.7.1 A few last words of advice on writing code and debugging.

(1) Break your problem down into small chunks and debug each chunk separately. This usually means write lots of small function files (and document them).

(2) Try to make functions work regardless of the size of the parameters. For example, if you need to evaluate a polynomial function, write a function that accepts a vector of values and a coefficient vector. If you need such a function once it is likely you will need it again. Also if you change your problem by using a fifth order polynomial rather than a fourth order, you will not need to rewrite your evaluation function.

(3) Try to avoid hard-coding parameter values and dimensions into your code. Suppose you have a problem that involves an interest rate of 7%. Don't put a lot of 0.07s into your code. Later on you will want to see what happens when the interest rate is 6% and you should be able to make this change in a single line with a nice comment attached to it, e.g.,

```
rate=0.07;              % the interest rate
```

(4) Avoid loops if possible. Loops are slow in MATLAB. It is often possible to do the same thing that a loop does with a vectorized command. Learn the available commands and use them.

(5) RTFM – internet lingo meaning Read The (F-word of choice) Manual.

(6) When you just can't figure it out, check the MATLAB technical support site (MathWorks), the MATLAB discussion group (comp.soft-sys.matlab) and DejaNews for posting about your problem and if that turns up nothing, post a question to the discussion group. Don't overdo it, however; people who abuse these groups are quickly spotted and will have their questions ignored. Also don't ask the group to solve your homework problems; you will get far more out of attempting them yourself then you'll get out of having someone else tell you the answer. You are likely to be found out anyway and it is a form of cheating.

3.8 Extended Example

3.8.1 MATLAB: ball4.m

```
function h = height(g, v, time)
%   HEIGHT(g,v,time) returns height a ball reaches
%   when subject to constant gravitational acceleration.
%   FILE:  height.m
%   Arguments:
%     g - gravitational acceleration
%     v - initial velocity
%     time - time at which height is desired
h = v * time - g * time * time * 0.5;

function vf = velocity(g, v, time)
%   VELOCITY(g,v,time) returns velocity ball reaches
%   when subject to constant gravitational acceleration.
%   FILE:  velocity.m
%   Arguments:
%     g - gravitational acceleration
%     v - initial velocity
%     time - time at which height is desired
vf = v - g * time ;


% ============================================================
% ball4.m
% Code to compute the height and velocity of a ball
% launched with given initial velocity on Earth and on Mars
% ============================================================

% define accelerations due to gravity
gEarth = 9.81;  %acceleration on Earth
gMars = 3.63;   % acceleration on Mars

v0 = input ('Input the initial velocity (m/s):> ');

t = input ('Input the time (s):> ');

% Results on Earth
disp('===========================================')
disp('Earth:')
fprintf('The velocity at time %4.2f is %6.2f',  t, velocity(gEarth, v0,t))

fprintf('\n and the height is %9.4f \n\n', height(gEarth, v0, t))

% Now do Mars
disp('===========================================')
disp('Mars:')
fprintf('The velocity at time %4.2f is %6.2f',  t, velocity(gMars, v0,t))

fprintf('\n and the height is %9.4f', height(gMars, v0, t))
```

3.8.2 MATLAB: Run

```
>> ball4
Input the initial velocity (m/s):> 10.0
Input the time (s):> 1.0
==========================================
Earth:
The velocity at time 1.00 is   0.19
 and the height is    5.0950

==========================================
Mars:
The velocity at time 1.00 is   6.37
 and the height is    8.1850
```

3.8.3 C++: ball4.cpp

```
// ==============================================================
// ball4.cpp
// Code to compute the height and velocity of a ball
// launched with given initial velocity on Earth and on Mars
// ==============================================================
#include <iostream>
using namespace std;

double height(double g, double v, double time);
double velocity(double g, double v, double time);

// ========================>> main <<========================
int main()
{
  // define accelerations due to gravity
  double gEarth = 9.81; // acceleration on Earth
  double gMars = 3.63;  // acceleration on Mars

  cout << "Input the initial velocity (m/s):> ";
  double v0;
  cin >> v0;

  cout << "Input the time (s):> ";
  double t;
  cin >> t;


  // Results on Earth
  cout << "==========================================" << endl;
  cout << "Earth:" << endl;
  cout << "The velocity at time " << t << " is "
       << velocity(gEarth, v0, t) << endl;
  cout << "and the height is "
       << height(gEarth, v0, t) << endl;

  // Now do Mars
  cout << "==========================================" << endl;
```

```
  cout << "Mars:" << endl;
  cout << "The velocity at time " << t << " is "
       << velocity(gMars, v0, t) << endl;
  cout << "and the height is "
       << height(gMars, v0, t) << endl;

  return 0;
}

// ==================>> height <<==========================
// Return height a ball reaches when subject to constant
// gravitational acceleration.
// Arguments:
//   g - gravitational acceleration
//   v - initial velocity
//   time - time at which height is desired
// ========================================================
double height(double g, double v, double time)
{
  double h;   // height ball will reach
  h = v * time - g * time * time * 0.5;
  return h;
}

// ==================>> velocity <<========================
// Return velocity ball reaches when subject to constant
// gravitational acceleration.
//   g - gravitational acceleration
//   v - initial velocity
//   time - time at which height is desired
// ========================================================
double velocity(double g, double v, double time)
{
  double vf;  // final velocity of ball
  vf = v - g * time;
  return vf;
}
```

3.8.4 C++: Run

```
Input the initial velocity (m/s):> 10.0
Input the time (s):> 1.0
==========================================
Earth:
The velocity at time 1 is 0.19
and the height is 5.095
==========================================
Mars:
The velocity at time 1 is 6.37
and the height is 8.185
```

3.9 References:

[1] MatLab an Introduction with Applications, Amos Gilat, John Wiley and Sons, Inc., 2004

[2] A MATLAB Primer, http://www4.ncsu.edu/unity/users/p/pfackler/www/MPRIMER.htm