

# Programming with MATLAB

By Gilberto E. Urroz, August 2002

## What is programming?

Programming the computer is the art of producing code in a standard computer language, such as Fortran, BASIC, Java, etc., to control the computer for calculations, data processing, or controlling a process. For details on programming structures, use of flowcharts and pseudo code, and some simple coding in MATLAB, Scilab, and Visual Basic, see the document entitled *Programming the Computer* (updated August 2004).

## What do I need to program the computer?

To type code you need an editor, which is simply a program where you can edit text files. Modern programming environments such as Matlab, Scilab, or Visual Basic include an editor for typing code.

Once the code is typed and the corresponding file saved, you need to have either a compiler (as in the case of Fortran) or an interpreter as in the case of Visual Basic (although, compilation is also allowed). A compiler translates the contents of a program's code into machine language, thus making the operation of the program faster than interpreting it. An interpreter collects as much code as necessary to activate an operation and performs that portion of the code before seeking additional code in the source file.

MATLAB programs are called also *functions*. MATLAB functions are available within the MATLAB environment as soon as you invoke the name of a function, but cannot be run separated from the MATLAB environment as are Visual Basic compiled programs.

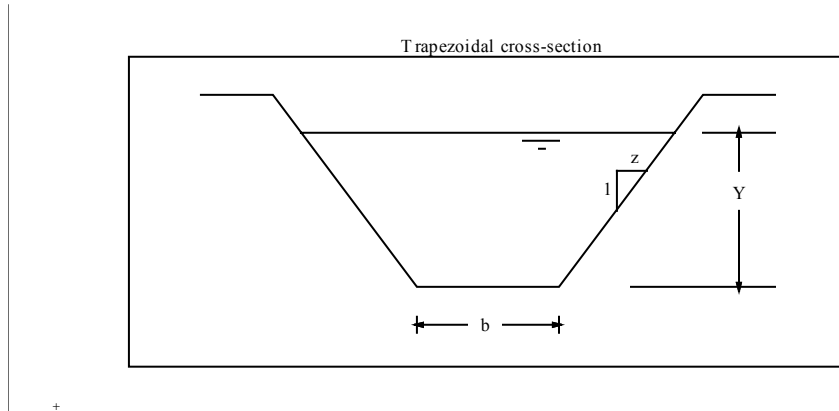
To produce code that works, you need, of course, to know the basic language syntax. If using MATLAB, you need to be familiar with its basic programming commands, and have some knowledge of its elementary functions. You also need to have some knowledge of basic program structures (`if ... else ... end`, `for ... end`, `while ... end`, `switch ... select`, etc.), of string manipulation, and input and output. Knowledge of matrix operations within Matlab is imperative since the numerical operation of the software is based on matrix manipulation.

## Scripts and functions

For details on the differences between scripts and functions in MATLAB see 2 - *Matlab Programming, IO, and Strings*. Both scripts and functions are stored as so-called *m files* in Matlab. A script is a series of interactive MATLAB commands listed together in an *m* file. Scripts are not programs properly. MATLAB functions, on the other hand, are self-contained programs that require arguments (input variables) and, many a time, an assignment to output variables. Some MATLAB functions can be called from other MATLAB functions (thus acting like subroutine or function subprograms in Fortran, or Sub or Function procedures in Visual Basic). To call a function, we use the name of the function followed by either an empty set of parentheses or parentheses containing the function arguments in the proper order. The function call can be done with or without assignment. Some examples of user-defined functions (as opposite to those pre-programmed functions such as *sin*, *exp*) are presented in this document.

## Examples of simple MATLAB functions

Herein we present an example of a simple function that calculates the area of a trapezoidal cross-section in an open channel, given the bottom width,  $b$ , the side slope  $z$ , and the flow depth  $y$  (see the figure below).



```
function [A] = Area(b,y,z)
%|-----|
%| Calculation of the area of a trapezoidal |
%| open-channel cross-section.             |
%|                                         |
%| Variables used:                         |
%| =====|
%|      b = bottom width (L)               |
%|      y = flow depth (L)                 |
%|      z = side slope (zH:1V, dimensionless) |
%|      A = area (L*L, optional in lhs)    |
%|-----|
A = (b+z*y)*y;
```

The file function turns out to have 13 lines, however, it contains only one operational statement,  $A = (b+z*y)*y$ . The remaining 12 lines are a fancy set of comments describing the function. A function like this could be easily defined as a MATLAB inline function through the command *inline*, e.g.,

```
Area = inline('(b+z*y)*y','b','y','z')
```

Use inline functions only for relatively simple expressions, such as the case of function *Area*, shown above. An example of application of the function *Area* is shown next:

```
» b = 2; z = 1.5; y = 0.75; A = Area(b,y,z)
```

```
A =
```

```
2.3438
```

If the inputs to the function will be vectors and you expect the function to return a vector of values, make sure that operations such as multiplication and division are term-by-term operations (*.\**, *./*), e.g.,

```
Area = inline('(b+z.*y).*y','b','y','z')
```

An application of this function with vector arguments is shown next:

```
» b = [1, 2, 3]; z = [0.5, 1.0, 1.5]; y = [0.25, 0.50, 0.75];
» A = Area(b,y,z)

A =

    0.2813    1.2500    3.0938
```

The following function, called *CC*, calculates the Cartesian coordinates  $(x,y,z)$  corresponding to the Cylindrical coordinates  $(\rho,\theta,\phi)$ :

```
function [x,y,z] = CC(rho,theta,phi)
%|-----|
%|This function converts the cylindrical |
%|coordinates (rho, theta, phi) into the |
%|Cartesian coordinates (x,y,z).        |
%|-----|
%Check consistency of vector entries
nr = length(rho);
nt = length(theta);
np = length(phi);
if nr ~= nt | nr ~= np | nt ~= np
    error('Function CC - vectors rho, theta, phi have incompatible
    dimensions')
    abort
end

%Calculate coordinates if vector lengths are consistent
x=rho.*sin(phi).*cos(theta);
y=rho.*sin(phi).*sin(theta);
z=rho.*cos(phi);
```

Type this function into an *m* file called *CC.m*. Here is an example of function *CC* applied to the cylindrical coordinates  $\rho = 2.5$ ,  $\theta = \pi/6$ , and  $\phi = \pi/8$ .

```
» [x,y,z] = CC(2.5,pi/6,pi/8)

x =

    0.8285

y =

    0.4784

z =

    2.3097
```

If no assignment is made to a vector of three elements, the result given by *CC* is only one value, that belonging to *x*, e.g.,

```
» CC(2.5,pi/6,pi/8)
```

ans =

0.8285

Notice that in the definition of the function term-by-term multiplications were provided. Thus, we could call the function using vector arguments to generate a vector result, e.g.,

```
EDU> [x,y,z] = CC(rho,theta,phi)
```

x =

0.2985 0.2710 0.1568

y =

0.0800 0.1564 0.2716

z =

0.9511 1.9754 2.9836

Consider now a vector function  $\mathbf{f}(\mathbf{x}) = [f_1(x_1, x_2, x_3) \ f_2(x_1, x_2, x_3) \ f_3(x_1, x_2, x_3)]^T$ , where  $\mathbf{x} = [x_1, x_2, x_3]^T$  (The symbol  $[\ ]^T$  indicates the transpose of a matrix). Specifically,

$$f_1(x_1, x_2, x_3) = x_1 \cos(x_2) + x_2 \cos(x_1) + x_3$$

$$f_2(x_1, x_2, x_3) = x_1 x_2 + x_2 x_3 + x_3 x_1$$

$$f_3(x_1, x_2, x_3) = x_1^2 + 2x_1 x_2 x_3 + x_3^2$$

A function to evaluate the vector function  $\mathbf{f}(\mathbf{x})$  is shown below.

```
function [y] = f(x)
%|-----|
%| This function calculates a vector |
%| function f = [f1;f2;f3] in which |
%| f1,f2,f3 = functions of (x1,x2,x3)|
%|-----|
y = zeros(3,1); %create output vector
y(1) = x(1)*cos(x(2))+x(2)*cos(x(1))+x(3);
y(2) = x(1)*x(2) + x(2)*x(3) + x(3)*x(1);
y(3) = x(1)^2 + 2*x(1)*x(2)*x(3) + x(3)^2;
```

Here is an application of the function:

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

x =

1  
2  
3

```
>> f(x)
```

ans =

```
3.6645
11.0000
22.0000
```

A function can return a matrix, for example, the following function produces a 2x2 matrix, defined by

$$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} x_1 + x_2 & (x_1 + x_2)^2 \\ x_1 x_2 & \sqrt{x_1 + x_2} \end{bmatrix},$$

with

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

```
function [J] = JJ(x)
%|-----|
%| This function returns a 2x2 matrix |
%| J = [f1, f2; f3, f4], with f1, f2, |
%| f3, f4 = functions of (x1,x2) |
%|-----|
J = zeros(2,2);
J(1,1) = x(1)+x(2);
J(1,2) = (x(1)+x(2))^2;
J(2,1) = x(1)*x(2);
J(2,2) = sqrt(x(1)+x(2));
```

An application of function *JMatrix* is shown below:

```
>> x = [2;-1]

x =

     2
    -1

>> JJ(x)

ans =

     1     1
    -2     1
```

### Examples of functions that include decisions

Functions including decision may use the structures `if ... end`, `if ... else ... end`, or `if ... elseif ... else ... end`, as well as the `switch ... select` structure. Some examples are shown below.

Consider, for example, a function that classifies a flow according to the values of its Reynolds (*Re*) and Mach (*Ma*) numbers, such that if  $Re < 2000$ , the flow is laminar; if  $2000 < Re < 5000$ , the flow is transitional; if  $Re > 5000$ , the flow is turbulent; if  $Ma < 1$ , the flow is sub-sonic, if  $Ma = 1$ , the flow is sonic; and, if  $Ma > 1$ , the flow is super-sonic.

```

function [] = FlowClassification(Re, Ma)
%|-----|
%| This function classifies a flow      |
%| according to the values of the      |
%| Reynolds (Re) and Mach (Ma).       |
%| Re <= 2000, laminar flow           |
%| 2000 < Re <= 5000, transitional flow|
%| Re > 5000, turbulent flow         |
%| Ma < 1, sub-sonic flow            |
%| Ma = 1, sonic flow                |
%| Ma > 1, super-sonic flow          |
%|-----|
%Classify according to Re
if Re <= 2000
    class = 'The flow is laminar ';
elseif Re > 2000 & Re <= 5000
    class = 'The flow is transitional ';
else
    class = 'The flow is turbulent ';
end
%Classify according to Ma
if Ma < 1
    class = strcat(class , ' and sub-sonic. ');
elseif Ma == 1
    class = strcat(class , ' and sonic. ');
else
    class = strcat(class , ' and super-sonic. ');
end
%Print result of classification
disp(class)

```

Notice that the function statement has no variable assigned in the left-hand side. That is because this function simply produces a string specifying the flow classification and it does not returns a value. Examples of the application of this function are shown next:

```

» FlowClassification(1000,0.5)
The flow is laminar and sub-sonic.
» FlowClassification(10000,1.0)
The flow is turbulent and sonic.
» FlowClassification(4000,2.1)
The flow is transitional and super-sonic.

```

An alternative version of the function has no arguments. The input values are requested by the function itself through the use of the *input* statement. (Comment lines describing the function were removed to save printing space).

```

function [] = FlowClassification()
%Request the input values of Re and Ma
Re = input('Enter value of the Reynolds number: \n');
Ma = input('Enter value of the Mach number:\n');
%Classify according to Re
if Re <= 2000
    class = 'The flow is laminar ';
elseif Re > 2000 & Re <= 5000
    class = 'The flow is transitional ';
else
    class = 'The flow is turbulent ';
end
%Classify according to Ma
if Ma < 1
    class = strcat(class , ' and sub-sonic. ');
elseif Ma == 1
    class = strcat(class , ' and sonic. ');
else
    class = strcat(class , ' and super-sonic. ');
end
%Print result of classification
disp(class)

```

An example of application of this function is shown next:

```

>> FlowClassification
Enter value of the Reynolds number:
14000
Enter value of the Mach number:
0.25
The flow is turbulent and sub-sonic.

```

The structures `if ... end`, `if ... else ... end`, or `if ... elseif ... else ... end` are useful, for example, in the evaluation of multiply defined functions (see 2 - *Matlab Programming, IO, and Strings*).

The following example presents a function that orders a vector in increasing order:

```

function [v] = MySort(u)
%|-----|
%|This function sorts vector u in |
%|increasing order, returning the |
%|sorted vector as v.           |
%|-----|
%Determine length of vector
n = length(u);
%Copy vector u to v
v = u;
%Start sorting process
for i = 1:n-1
    for j = i+1:n
        if v(i)>v(j)
            temp = v(i);
            v(i) = v(j);
            v(j) = temp;
        end
    end
end
end

```

An application of this function is shown next:

```
» u = round(10*rand(1,10))
u =
     0     7     4     9     5     4     8     5     2     7
» MySort(u)
ans =
     0     2     4     4     5     5     7     7     8     9
```

Of course, you don't need to write your own function for sorting vectors of data since MATLAB already provides function *sort* for this purpose, e.g.,

```
» sort(u)
ans =
     0     2     4     4     5     5     7     7     8     9
```

However, a function like *MySort*, shown above, can be used for programming an increasing-order sort in other computer languages that do not operate based on matrices, e.g., Visual Basic or Fortran.

To sort a row vector in decreasing order, use *sort* and *fliplr* (flip left-to-right), e.g.,

```
» sort(u)
ans =
     0     2     4     4     5     5     7     7     8     9
» fliplr(ans)
ans =
     9     8     7     7     5     5     4     4     2     0
```

Function *MySort* can be modified to perform a decreasing order sort if the line `if v(i)>v(j)` is modified to read `if v(i)<v(j)`. Alternatively, you can modify the function *MySort* to allow for increasing or decreasing sort by adding a second, string, variable which can take the value of 'd' for decreasing order or 'i' for increasing order. The function file is shown below:

```
function [v] = MySort(u,ittype)
%|-----|
%|This function sorts vector u in |
%|increasing order, returning the |
%|sorted vector as v.             |
%|If itype = 'd', decreasing order |
%|If itype = 'i', increasing order |
%|-----|
%Determine length of vector
n = length(u);
%Copy vector u to v
```



```

v = u;
%Start sorting process
if itype == 'i'
    for i = 1:n-1
        for j = i+1:n
            if v(i)>v(j)
                temp = v(i);
                v(i) = v(j);
                v(j) = temp;
            end
        end
    end
else
    for i = 1:n-1
        for j = i+1:n
            if v(i)<v(j)
                temp = v(i);
                v(i) = v(j);
                v(j) = temp;
            end
        end
    end
end
end

```

The following is an application of this function

```
» u = round(10*rand(1,10))
```

```
u =
```

```
8    0    7    4    8    5    7    4    3    2
```

```
EDU» MySort(u,'i')
```

```
ans =
```

```
0    2    3    4    4    5    7    7    8    8
```

```
EDU» MySort(u,'d')
```

```
ans =
```

```
8    8    7    7    5    4    4    3    2    0
```

In the following example, the function is used to swap the order of the elements of a vector. Thus, in a vector  $v$  of  $n$  elements,  $v_1$  is swapped with  $v_n$ ,  $v_2$  is swapped with  $v_{n-1}$ , and so on. The general formula for swapping depends on whether the vector has an even or an odd number of elements. To check whether the vector length,  $n$ , is odd or even we use the function *mod* (modulo). If  $\text{mod}(n,2)$  is equal to zero,  $n$  is even, otherwise, it is odd. If  $n$  is even, then the swapping occurs by exchanging  $v_j$  with  $v_{n-j+1}$  for  $j = 1, 2, \dots, n/2$ . If  $n$  is odd, the swapping occurs by exchanging  $v_j$  with  $v_{n-j+1}$  for  $j = 1, 2, \dots, (n-1)/2$ . Here is the listing of the function:

```

function [v] = swapVector(u)
%|-----|
%| This function swaps the elements |
%| of vector u, v(1) with v(n), v(2) |

```

```

%| with v(n-1), and so on, where n |
%| is the length of the vector.    |
%|-----|
n = length(u); %length of vector
v = u;        %copy vector u to v
%Determine upper limit of swapping
if modulo(n,2) == 0 then
    m = n/2;
else
    m = (n-1)/2;
end
%Perform swapping
for j = 1:m
    temp = v(j);
    v(j) = v(n-j+1);
    v(n-j+1) = temp;
end

```

Applications of the function, using first a vector of 10 elements and then a vector of 9 elements, is shown next:

```

» u = [-10:1:-1]

u =
    -10     -9     -8     -7     -6     -5     -4     -3     -2     -1

» swapVector(u)

ans =
     -1     -2     -3     -4     -5     -6     -7     -8     -9    -10

» u = [1:9]

u =
     1     2     3     4     5     6     7     8     9

» swapVector(u)

ans =
     9     8     7     6     5     4     3     2     1

```

Of course, these operations can be performed by simply using function *fliplr* in Matlab, e.g.,

```

EDU» fliplr(u)

ans =
     9     8     7     6     5     4     3     2     1

```

The `switch ... select` structure can be used to select one out of many possible outcomes in a process. In the following example, the function *Area* calculates the area of different open-channel cross-sections based on a selection performed by the user:

```

function [] = Area()
%|-----|
disp('=====')
disp('Open channel area calculation')
disp('=====')
disp('Select an option:')
disp('  1 - Trapezoidal')
disp('  2 - Rectangular')
disp('  3 - Triangular')
disp('  4 - Circular')
disp('=====')
itype = input('');
switch itype,
    case 1
        id = 'trapezoidal';
        b = input('Enter bottom width:');
        z = input('Enter side slope:');
        y = input('Flow depth:');
        A = (b+z*y)*y;
    case 2
        id = 'rectangular';
        b = input('Enter bottom width:');
        z = 0;
        y = input('Flow depth:');
        A = (b+z*y)*y;
    case 3
        id = 'triangular';
        b = 0;
        z = input('Enter side slope:');
        y = input('Flow depth:');
        A = (b+z*y)*y;
    otherwise
        id = 'circular';
        D = input('Enter pipe diameter:');
        y = input('Enter flow depth (y<D):');
        beta = acos(1-2*y/D);
        A = D^2/4*(beta+sin(beta)*cos(beta)) ;
end
%Print calculated area
fprintf('\n\nThe area for a %s cross-section is %10.6f.\n\n',id,A)

```

An example of application of this function is shown next:

```

» Area
=====
Open channel area calculation
=====
Select an option:
  1 - Trapezoidal
  2 - Rectangular
  3 - Triangular
  4 - Circular
=====
4
Enter pipe diameter:1.2
Enter flow depth (y<D):0.5

The area for a circular cross-section is    0.564366.

```

## Simple output in MATLAB functions

Notice that we use function *disp* in function *Area*, above, to print the table after the function is invoked. On the other hand, to print the value of the resulting area we use function *fprintf*. This function is borrowed from C, while *disp* is a purely MATLAB function. Function *disp* will print a string used as the argument, as in the examples listed in the function *Area*. The following are two examples of using function *disp* interactively in MATLAB:

- Displaying a string:

```
» disp('Hello, world')
Hello, world
```

- Displaying a string identifying a value and the corresponding value:

```
» A = 2.3; disp(strcat('A = ', num2str(A)))
A =2.3
```

Function *Area* also utilizes the function *fprintf* to produce output. This function includes as arguments a string and, optionally, a list of variables whose values will be incorporated in the output string. To incorporate those values in the output string we use *conversion strings* such as *%10.6f*. The conversion string *%10.6f* represents an output field of for a floating-point result, thus the *f*, with 10 characters including 6 decimal figures. An integer value can be listed in a conversion string of the form *%5d*, which will reserve 5 characters for the output. The following interactive example illustrates the use of *fprintf* to print a floating point and an integer value:

```
» A = 24.5; k = 2;
» fprintf('\n\nThe area is %10.6f for iteration number %5d.\n',A,k)

The area is          25 for iteration number      2.
```

The characters `\n` in the output string of function *fprintf* produce a new line. For example:

```
» fprintf('\n\nThe area is %10.6f \nfor iteration number %5d.\n',A,k)

The area is          25
for iteration number      2.
```

Function *fprintf* can be used to produce an output table, as illustrated in the following function:

```
function [] = myTable()
fprintf('=====\n');
fprintf('  a      b      c      d  \n');
fprintf('=====\n');
for j = 1:10
    a = sin(10*j);
    b = a*cos(10*j);
    c = a + b;
    d = a - b;
    fprintf('("%+6.5f  %+6.5f  %+6.5f  %+6.5f \n',a,b,c,d);
end
fprintf('=====\n');
```

The application of this function is shown next:

```

» myTable
=====
      a      b      c      d
=====
-0.54402  +0.45647  -0.08755  -1.00049
+0.91295  +0.37256  +1.28550  +0.54039
-0.98803  -0.15241  -1.14044  -0.83563
+0.74511  -0.49694  +0.24817  +1.24206
-0.26237  -0.25318  -0.51556  -0.00919
-0.30481  +0.29031  -0.01451  -0.59512
+0.77389  +0.49012  +1.26401  +0.28377
-0.99389  +0.10971  -0.88418  -1.10360
+0.89400  -0.40058  +0.49342  +1.29457
-0.50637  -0.43665  -0.94301  -0.06972
=====

```

Notice that the conversion string was changed to `%+6.5f`, which allows for the inclusion of a sign in the values printed.

The function *myTable* is modified next to produce output using scientific notation. The conversion string used here is `%+6.5e`.

```

function [] = myTable()
fprintf('=====\\n');
fprintf('      a      b      c      d  \\n')
fprintf('=====\\n');
for j = 1:10
    a = sin(10*j);
    b = a*cos(10*j);
    c = a + b;
    d = a - b;
    fprintf('%+6.5e  %+6.5e  %+6.5e  %+6.5e\\n',a,b,c,d);
end
fprintf('=====\\n');

```

The application of the function with the new conversion strings produces the following table:

```

» myTable
=====
      a      b      c      d
=====
-5.44021e-001  +4.56473e-001  -8.75485e-002  -1.00049e+000
+9.12945e-001  +3.72557e-001  +1.28550e+000  +5.40389e-001
-9.88032e-001  -1.52405e-001  -1.14044e+000  -8.35626e-001
+7.45113e-001  -4.96944e-001  +2.48169e-001  +1.24206e+000
-2.62375e-001  -2.53183e-001  -5.15558e-001  -9.19203e-003
-3.04811e-001  +2.90306e-001  -1.45050e-002  -5.95116e-001
+7.73891e-001  +4.90120e-001  +1.26401e+000  +2.83771e-001
-9.93889e-001  +1.09713e-001  -8.84176e-001  -1.10360e+000
+8.93997e-001  -4.00576e-001  +4.93420e-001  +1.29457e+000
-5.06366e-001  -4.36649e-001  -9.43014e-001  -6.97170e-002
=====

```

The table examples presented above illustrate also the use of the *for ... end* structure to produce programming loops. Additional examples of programming loops are presented in the following examples.

### Examples of functions that include loops

MATLAB provides with the *while ... end* and the *for ... end* structures for creating programming loops. Programming loops can be used to calculate summations or products. For example, to calculate the summation

$$S(n) = \sum_{k=0}^n \frac{1}{k^2 + 1},$$

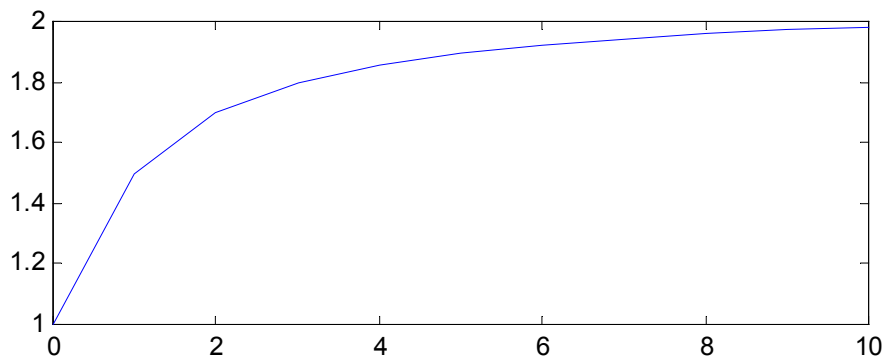
we can put together the following function:

```
function [S] = sum1(n)
% |-----|
% | This function calculates the summation |
% |-----|
% |                                     n |
% |                                     |
% |                                     |
% |                                     |
% |                                     |
% | S(n) = \-----/ |
% |                                     |
% |                                     |
% |                                     |
% |                                     |
% |                                     |
% |                                     |
% |                                     |
% |-----|
S = 0;           % Initialize sum to zero
k = 0;           % Initialize index to zero
% loop for calculating summation
while k <= n
    S = S + 1/(k^2+1); % add to summation
    k = k + 1;        % increase the index
end
```

Within the MATLAB environment we can produce a vector of values for the sum with values of  $n$  going from 0 to 10, and produce a plot of values of the sum versus  $n$ :

```
>> nn = [0:10];           % vector of values of n (nn)
>> m = length(nn);       % length of vector nn
>> SS = [];               % initialize vector SS as an empty matrix
>> for j = 1:m
    SS = [SS sum1(nn(j))]; % This loop fills out the vector SS
    end                    % with values calculated
>> plot(nn,SS)            % from function sum1(n)
                           % show results in graphical format
```

The resulting graph is shown below:

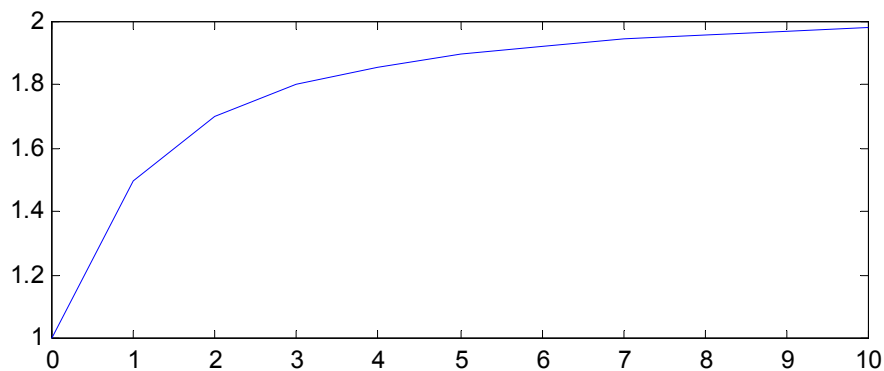


If we need to calculate a vector, or matrix, of values of the summation we can modify the original function *sum1* to handle matrix input and output as follows (the function is now called *sum2*):

```
function [S] = sum2(n)
%|-----|
%| This function calculates the summation |
%| |
%|          n
%|          \-----|
%|          \      1
%| S(n) =  /-----|
%|          /      2
%|          /      k + 1
%|          /-----|
%|          k = 0
%| |
%| Matrix input and output is allowed. |
%|-----|
[nr,nc] = size(n); %Size of input matrix
S = zeros(nr,nc); %Initialize sums to zero
%loop for calculating summation matrix
for i = 1:nr      %sweep by rows
    for j = 1:nc  %sweep by columns
        k = 0;   %initialize index to zero
        while k <= n(i,j)
            S(i,j) = S(i,j) + 1/(k^2+1);
            k = k + 1;
        end
    end
end
end
```

Notice that both *n* and *S* are now treated as matrix components within the function. The following MATLAB commands will produce the same results as before:

```
>>nn = [0:10]; SS = sum2(nn); plot(nn,SS)
```



Notice that function *sum2* illustrates the use of the two types of loop control structures available in MATLAB:

- (1) *while ... end* (used for calculating the summations in this example) and,

(2) *for ... end* (used for filling the output matrix  $S$  by rows and columns in this example).

Nested *for ... end* loops can be used to calculate double summations, for example,

$$S(n, m) = \sum_{i=0}^n \sum_{j=0}^m \frac{1}{(i+j)^2 + 1}.$$

Here is a listing of a function that calculates  $S(n, m)$  for single values of  $n$  and  $m$ :

```
function [S] = sum2x2(n,m)
%|-----|
%| This function calculates the double |
%| summation:                          |
%|          n          m                |
%|          \----- \-----        |
%|          \         \         1      |
%| S(n,m) = \         \         2      |
%|          /         /         (i+j) + 1 |
%|          /----- /-----        |
%|          i=0       j=0                |
%|-----|
%|Note: the function only takes scalar |
%|values of n and m and returns a single |
%|value. An error is reported if n or m |
%|are vectors or matrices.              |
%|-----|
%Check if m or n are matrices
if length(n)>1 | length(m)>1 then
    error('sum2 - n,m must be scalar values')
    abort
end
%Calculate summation if n and m are scalars
S = 0; %initialize sum
for i = 1:n %sweep by index i
    for j = 1:m %sweep by index j
        S = S + 1/((i+j)^2+1);
    end
end
end
```

A single evaluation of function *sum2x2* is shown next:

```
» sum2x2(3,2)
ans = 0.5561
```

The following MATLAB commands produce a matrix  $S$  so that element  $S_{ij}$  corresponds to the *sum2x2* evaluated at  $i$  and  $j$ :

```
» for i = 1:3
    for j = 1:4
        S(i,j) = sum2x2(i,j);
    end
end
```



```
» S
```

```
S =
```

```
    0.2000    0.3000    0.3588    0.3973  
    0.3000    0.4588    0.5561    0.6216  
    0.3588    0.5561    0.6804    0.7659
```

As illustrated in the example immediately above, nested *for ... end* loops can be used to manipulate individual elements in a matrix. Consider, for example, the case in which you want to produce a matrix  $z_{ij} = f(x_i, y_j)$  out of a vector  $x$  of  $n$  elements and a vector  $y$  of  $m$  elements. The function  $f(x, y)$  can be any function of two variables defined in MATLAB. Here is the function to produce the aforementioned matrix:

```
function [z] = f2eval(x,y,f)  
%|-----|  
%| This function takes two row vectors |  
%| x (of length n) and y (of length m) |  
%| and produces a matrix z (nxm) such |  
%| that z(i,j) = f(x(i),y(j)).      |  
%|-----|  
%Determine the lengths of vectors  
n = length(x);  
m = length(y);  
%Create a matrix nxm full of zeros  
z = zeros(n,m);  
%Fill the matrix with values of z  
for i = 1:n  
    for j = 1:m  
        z(i,j) = f(x(i),y(j));  
    end  
end
```

Here is an application of function *f2solve*:

```
» f00 = inline('x.*sin(y)', 'x', 'y')
```

```
f00 =
```

```
    Inline function:  
    f00(x,y) = x.*sin(y)
```

```
» x = [1:1:3]
```

```
x =
```

```
    1    2    3
```

```
» y = [2:1:5]
```

```
y =
```

```
    2    3    4    5
```

```
» z = f2eval(x,y,f00)
```

```
z =
```

```

0.9093    0.1411   -0.7568   -0.9589
1.8186    0.2822   -1.5136   -1.9178
2.7279    0.4234   -2.2704   -2.8768

```

Notice that MATLAB provide functions to produce this evaluation as will be illustrated next. First, the values of the  $x$  and  $y$  vectors produced above must be used to produce grid matrices  $X$  and  $Y$  for the evaluation of the function  $f00$ . This is accomplished by using function *meshgrid* as follows:

```
» [X,Y] = meshgrid(x,y)
```

```
X =
```

```

1     2     3
1     2     3
1     2     3
1     2     3

```

```
Y =
```

```

2     2     2
3     3     3
4     4     4
5     5     5

```

Then, use the following call to function  $f00$ :

```
» f00(X,Y)
```

```
ans =
```

```

0.9093    1.8186    2.7279
0.1411    0.2822    0.4234
-0.7568   -1.5136   -2.2704
-0.9589   -1.9178   -2.8768

```

Alternatively, you could use function *feval* as follows:

```
» feval(f00,X,Y)
```

```
ans =
```

```

0.9093    1.8186    2.7279
0.1411    0.2822    0.4234
-0.7568   -1.5136   -2.2704
-0.9589   -1.9178   -2.8768

```

MATLAB functions *meshgrid* and *feval* (or the user-defined function *f2eval*) can be used for evaluating matrix of functions of two variables to produce surface plots. Consider the following case:

```
» f01 = inline('x.*sin(y)+y.*sin(x)', 'x', 'y')
```

```
f01 =
```

```

Inline function:
f01(x,y) = x.*sin(y)+y.*sin(x)

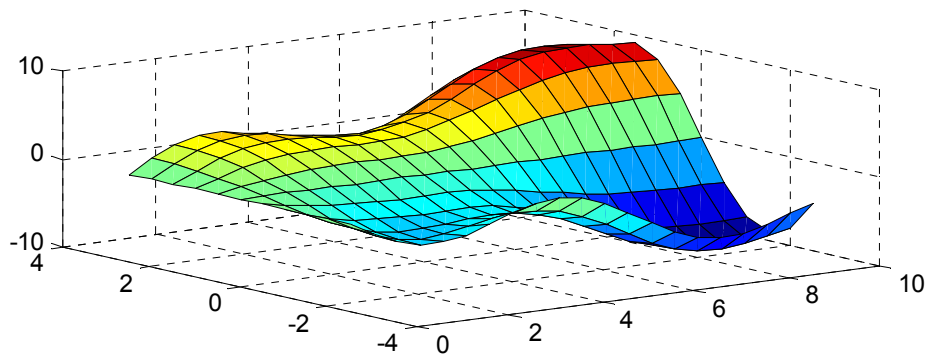
```

```

» x = [0:0.5:10]; y = [-2.5:0.5:2.5];
» z = f2eval(x,y,f01);
» surf(x,y,z')

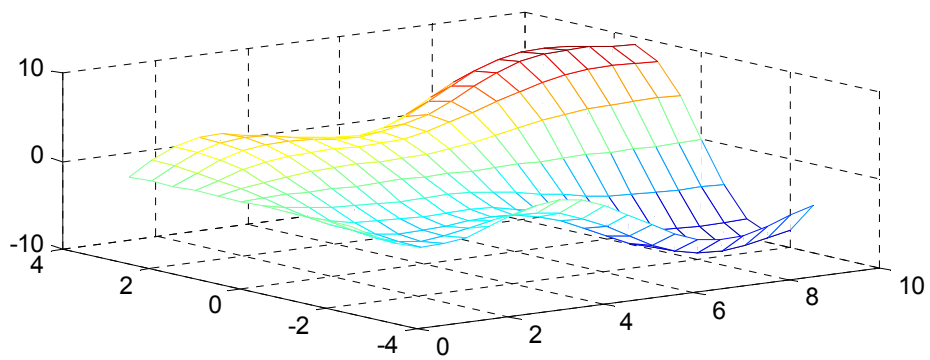
```

The resulting graph is shown next:



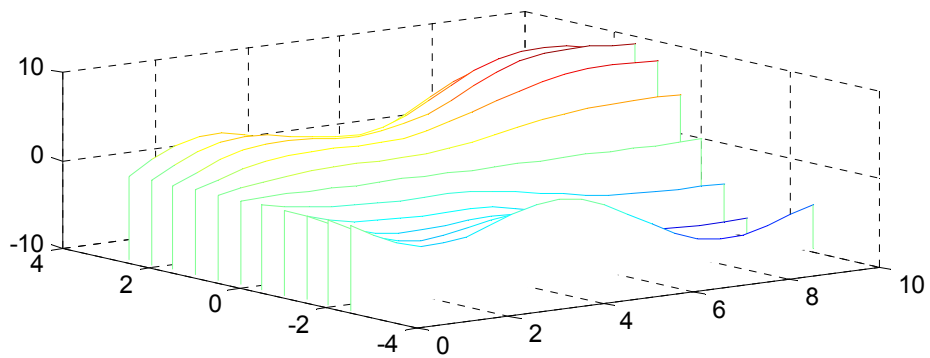
The function can also be plotted using *mesh*:

```
--> mesh(x, y, z')
```



or,

```
--> waterfall(x, y, z')
```



Nested loops can be used for other operations with matrices. For example, a function that multiplies two matrices is shown next:

```

function [C] = matrixmult(A,B)
%|-----|
%| This function calculates the matrix |
%| C(nxm) = A(nxp)*B(pxm)           |
%|-----|
%First, check matrices compatibility
[nrA,ncA] = size(A);
[nrB,ncB] = size(B);
if ncA ~= nrB
    error('matrixmult - incompatible matrices A and B');
    abort;
end
%Calculate matrix product
C = zeros(nrA,ncB);
for i = 1:nrA
    for j = 1:ncB
        for k = 1:ncA
            C(i,j) = C(i,j) + A(i,k)*B(k,j);
        end
    end
end
end

```

An application of function *matrixmult* is shown next. Here we multiply  $A_{3 \times 4}$  with  $B_{4 \times 6}$  resulting in  $C_{3 \times 6}$ . However, the product  $B_{4 \times 6}$  times  $A_{3 \times 4}$  does not work.

```
» A = round(10*rand(3,4))
```

```
A =
```

```

     2     5     4     6
     7     2     9     5
     3     7     9     9

```

```
EDU» B = round(10*rand(4,6))
```

```
B =
```

```

     8     3     7     4     7     5
     6     3     3     7     6     9
     8     3     8     5     8     2
     7     5     6     4    10    10

```

```
EDU» C = matrixmult(A,B)
```

```
C =
```

```

    120    63    97    87    136    123
    175    79   157   107   183   121
    201   102   168   142   225   186

```

```
» D = matrixmult(B,A)
```

```

??? Error using ==> matrixmult
matrixmult - incompatible matrices A and B

```

Of course, matrix multiplication in MATLAB is readily available by using the multiplication sign (\*), e.g.,

```
» C = A*B
```

C =

```
120    63    97    87   136   123
175    79   157   107   183   121
201   102   168   142   225   186
```

```
» D = B*A
??? Error using ==> *
Inner matrix dimensions must agree.
```

## Reading a table from a file

The first example presented in this section uses the command *file* to open a file for reading, and the function *read* for reading a matrix out of a file. This application is useful when reading a table into MATLAB. Suppose that the table consists of the following entries, and is stored in the file *c:\table1.txt*:

2.35	5.42	6.28	3.17	5.23
3.19	3.87	3.21	5.18	6.32
1.34	5.17	8.23	7.28	1.34
3.21	3.22	3.23	3.24	3.25
8.17	4.52	8.78	2.31	1.23
9.18	5.69	3.45	2.25	0.76

The following function reads the table out of file *c:\table1.dat* using function *fgetl*:

```
function [Table] = readtable()
%|-----|
%| This is an interactive function that |
%| requests from the user the name of a |
%| file containing a table and reads the |
%| table out of the file. |
%|-----|
fprintf('Reading a table from a file\n');
fprintf('=====\n');
fprintf(' \n');
filename = input('Enter filename between quotes:\n');
u = fopen(filename,'r'); %open input file
%The following 'while' loop reads data from the file
Table = []; %initialize empty matrix
while 1
    line = fgetl(u); %read line
    if ~ischar(line), break, end %stop when no more lines
available
    Table = [Table; str2num(line)]; %convert to number and add to
matrix
end
%Determine size of input table (rows and columns)
[n m] = size(Table);
fprintf('\n');
fprintf('There are %d rows and %d columns in the table.\n',n,m);
fclose(u); %close input file
```

The *fopen* command in function *readtable* uses two arguments, the first being the filename and the second ('r') indicates that the file is to be opened for input, i.e.,

reading. Also, the left-hand side of this statement is the variable *u*, which will be given an integer value by MATLAB. The variable *u* represents the *logical input-output unit* assigned to the file being opened.

Function *readtable* uses the command *fgetl* within the *while* loop to read line by line. This function reads the lines as strings, thus, function *str2num* is used to convert the line to numbers before adding that line to matrix *Table*.

As shown in function *readtable*, it is always a good idea to insert a *fclose* statement to ensure that the file is closed and available for access from other programs.

Here is an example of application of function *readtable*. User response is shown in bold face.

```

>> T = readtable
Reading a table from a file
=====
Enter filename between quotes:
'c:\table1.txt'

There are 6 rows and 5 columns in the table.

T =
    2.3500    5.4200    6.2800    3.1700    5.2300
    3.1900    3.8700    3.2100    5.1800    6.3200
    1.3400    5.1700    8.2300    7.2800    1.3400
    3.2100    3.2200    3.2300    3.2400    3.2500
    8.1700    4.5200    8.7800    2.3100    1.2300
    9.1800    5.6900    3.4500    2.2500    0.7600

```

## Writing to a file

The simplest way to write to a file is to use the function *fprintf* in a similar fashion as function *fprintf*. As an example, we re-write function *myTable* (shown earlier) to include *fprintf* statements that will write the results to a file as well as to the screen:

```

function [] = myTableFile()
fprintf('Printing to a file\n');
fprintf('=====\n');
filename = input('Enter file to write to (between quotes):\n');
u = fopen(filename,'w'); %open output file
fprintf('=====\n');
fprintf(u, '\n=====\r');
fprintf('  a      b      c      d  \n');
fprintf(u, '\n  a      b      c      d  \r');
fprintf('=====\n');
fprintf(u, '\n=====\r');
for j = 1:10
    a = sin(10*j);
    b = a*cos(10*j);
    c = a + b;
    d = a - b;
    fprintf('%+6.5f %+6.5f %+6.5f %+6.5f\n', a,b,c,d);
    fprintf(u, '\n%+6.5f %+6.5f %+6.5f %+6.5f\r', a,b,c,d);
end;
fprintf('=====\n');
fprintf(u, '\n=====\r');
fclose(u); %close output file

```

Notice that the strings that get printed to the file, i.e., those lines starting with *fprintf(u,...)*, end with the characters `\r` rather than with `\n`. The characters `\n` (*newline*) work fine in the screen output to produce a new line, but, for text files, you need to use the characters `\r` (*carriage return*) to produce the same effect.

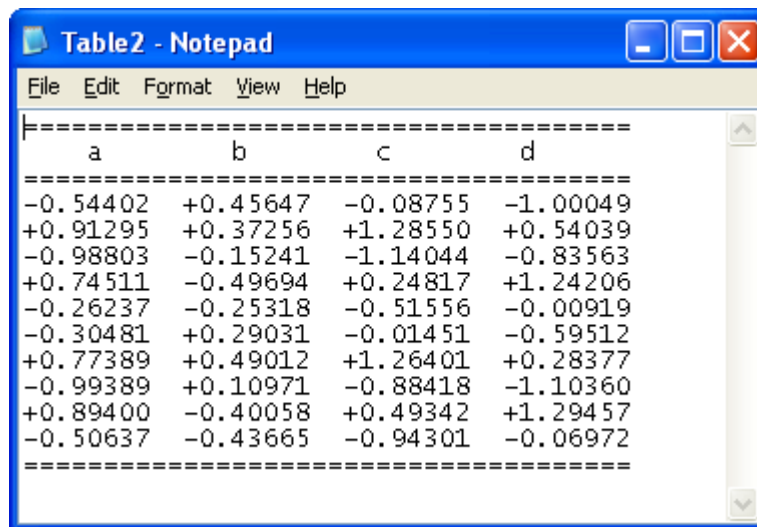
Here is an application of function *myTableFile*. The function prints a table in MATLAB, while simultaneously printing to the output file. The user response is shown in bold face:

```

>> myTableFile
Printing to a file
=====
Enter file to write to (between quotes):
'c:\Table2.txt'
=====
      a      b      c      d
=====
-0.54402 +0.45647 -0.08755 -1.00049
+0.91295 +0.37256 +1.28550 +0.54039
-0.98803 -0.15241 -1.14044 -0.83563
+0.74511 -0.49694 +0.24817 +1.24206
-0.26237 -0.25318 -0.51556 -0.00919
-0.30481 +0.29031 -0.01451 -0.59512
+0.77389 +0.49012 +1.26401 +0.28377
-0.99389 +0.10971 -0.88418 -1.10360
+0.89400 -0.40058 +0.49342 +1.29457
-0.50637 -0.43665 -0.94301 -0.06972
=====

```

With a text editor (e.g., *notepad*) we can open the file *c:\Table2.txt* and check out that the same table was printed to that text file.



## Summary

This document shows a number of programming examples using decision and loop structures, input-output, and other elements of programming. Advanced MATLAB programming includes the use of menus and dialogs, as well as advanced mathematical, string, and file functions.