# MATLAB Tutorial

## Chapter 6. Writing and calling functions

In this chapter we discuss how to structure a program with multiple source code files. First, an explanation of how code files work in MATLAB is presented. In compiled languages such as FORTRAN, C, or C++, code can be stored in one or more source files that are linked together to form a single executable at the time of compilation. MATLAB, being an interpreted language, deals with multiple source files in a more open-ended manner. MATLAB code is organized into ASCII files carrying the extension .m (also known as m-files). MATLAB 6 has an integrated word processing and debugging utility that is the preferred mode of editing m-files, although other ASCII editors such as vi or emacs may also be used.
There are two different kinds of m-files. The simplest, a script file, is merely a collection of MATLAB commands. When the script file is executed by typing its name at the interactive prompt, MATLAB reads and executes the commands within the m-file just as if one were entering them manually. It is as if one were cutting and pasting the m-file contents into the MATLAB command window. The use of this type of m-file is outlined in section 6.1.

The second kind of m-file, discussed in section 6.2, contains a single function that has the same name as that of the m-file. This m-file contains an independent section of code with a clearly defined input/output interface; that is, it can be invoked by passing to it a list of dummy arguments arg1, arg2, ... and it returns as output the values out1, out2, .... The first non-commented line of a function m-file contains the function header, which is of the form :
**function [out1,out2,...] = filename(arg1,arg2,...);**
The m-file ends with the command return, which returns the program execution to the place where the function was called. The function code is executed whenever, either at the interactive command prompt or within another m-file, it is invoked with the command :
**[outvar1,outvar2,...] = filename(var1,var2,...)**
with the mapping of input to dummy arguments : arg1 = var1, arg2 = var2, etc. Within the function body, output values are assigned to the variables out1, out2, etc. When return is encountered, the current values of out1, out2, ... are mapped to the variables outvar1, outvar2, ... at the point where the function was called. MATLAB allows much latitude in writing functions with variable length argument and output variable lists. For example, the function could also be invoked by the command :
**outvar1 = filename(var1,var2,...)**
in which case only a single output variable is returned, containing on exit the value of the function variable out1. The input and output arguments may be strings, scalar numbers, vectors, matrices, or more advanced data structures.

Why use functions? As is well known from every computer science course, splitting a large program into multiple procedures that perform each a single well defined and commented task, results in programs that are easier to read, easier to modify, and that are more resistant to error. In MATLAB, one writes first a master file for the program, either a script file or better yet a function m-file that returns a single integer (that might return 1 for program success, 0 for incomplete program execution, or a negative value to indicate a run-time error), that is the point of entry to the program. This program file then calls upon code in other m-files by invoking them as functions. But if there is no compilation process to link all of the source code files together, how does MATLAB know where to look for a function when it is called?

MATLAB's program memory contains a search path list, the contents of which can be viewed with the command path, that stores the names of the directories it has been told contain function m-files. Initially, the path lists only the directories that hold the built-in MATLAB functions such as **sin(), exp(),** etc.. As demonstrated in section 6.2, one uses the command addpath to add to this list the name of each directory that contains a m-file for the present project. Then, when the MATLAB code interpreter encounters a function, say with the name **filename**, it starts at the top of the path list and works its way down searching in each directory for a file **filename.m**. When it finds it, it executes the file's code in the manner

described above. For this reason, it is imperative that the names of the m-file and of the function agree; in fact it is only the filename that counts.

### 6.1. Writing and running m-files

While MATLAB can be run interactively from the command line, you can write a MATLAB program by composing a text file that contains the commands you want MATLAB to perform in the order in which they appear in the file. The standard file suffix for a text file containing a MATLAB program is .m. In the MATLAB command window, selecting the pull-down menu File -> New -> M-file opens the integrated MATLAB text editor for writing a m-file. This utility is very similar to word processors, so the use of writing and saving m-files is not explained in detail here.

As an example, use this secion as a file "MATLAB_tutorial_c6s1.m" that has only the following executable commands.

```
file_name = 'MATLAB_tutorial_c6s1.m';
disp(['Starting ' file_name ]);
j = 5;
for i=1:5
j = j - 1;
disp([int2str(i) ' ' int2str(j)]);
end
disp(['Finished ' file_name]);
```

We can run this m-file from the prompt by typing its name
>> MATLAB_tutorial_c6s1

If we type "whos" now, we see that the variables that are in the memory at the end of the program also remain in memory after the m-file is done running. This is because we have written the m-file as a script file where we have simply collected together several commands in a file, and then the code executes them one-by-one when the script is run, as if we were merely typing them into the interactive session window. A more common use for m-files is to isolate a series of commands in an independent function, as explained in the following section.

### 6.2. Structured programming with functions
### Unstructured programming approach

### File unstructured.m

In this section, let us demonstrate the use of subroutines to write structured, well-organized programs. We do so for a particularly simple and familiar case, the simple 1-D PDE problem that we encounted in section 4.1. First, in this m-file, we solve the problem with a program the combines all of the commands into a single file. This "unstructured" approach is fine for very small programs, but rapidly becomes confusion as the size of the program grows.

```
num_pts = 100; # of grid points
x = 1:num_pts; grid of x-values
```

We now set the values for the matrix discretizing the PDE with Dirichlet boundary conditions.

```
nzA = 3*(num_pts-2) + 2; # of non-zero elements
A = spalloc(num_pts,num_pts,nzA); allocate memory
```

set values

```
A(1,1) = 1;
A(num_pts,num_pts) = 1;
```

**for i=2:(num_pts-1)**
**A(i,i) = 2;**
**A(i,i-1) = -1;**
**A(i,i+1) = -1;**
**end**

Next, we set the values of the function at each boundary.
**BC1 = -10;** value of f at x(1);
**BC2 = 10;** value of f at x(num_pts);

We now create the vector for the right hand side of the problem.
**b_RHS = linspace(0,0,num_pts)';** create column vector of zeros
**b_RHS(1) = BC1;**
**b_RHS(num_pts) = BC2;**
**b_RHS(2:(num_pts-1)) = 0.05;** for interior, b_RHS is source term

Now, we call the standard MATLAB solver.
**f = A\b_RHS;**

Then, we make a plot of the results.
**figure; plot(x,f);**
**title('PDE solution from FD-CDS method (sparse matrix)');**
**xlabel('x'); ylabel('f(x)');**

While this approach of putting all of the commands together works for this small program, it becomes very unwieldy for large programs.

**clear all**

## Structured programming approach

## File structured.m

NOTE: BEFORE RUNNING THIS FILE, THE OTHER M-FILES CONTAINING THE SUBROUTINES MUST ALREADY EXIST.
First, we define the number of points
**num_pts = 100;**

We now create a vector containing the grid points.
**x = 1:num_pts;**

In MATLAB, each function is stored in a separate m-file of the same name. When you call the function at the interactive session prompt or in another script or funtcion m-file, MATLAB searches through a list of directories that it has been told contain functions until it finds an m-file with the appropriate name. Then, it executes the MATLAB code contained within that m-file. When we write m-files that contain a functions, before we can use them we have to tell MATLAB where they are; that is, we have to add the name of their directory to the search **path.**

We can check the current contents of the search path with the command "path".
**path**

The command "pwd" returns the current directory.
**pwd**

We use the command "addpath" to add the directory with our subroutines to this search list. We can remove a directory from the path using "rmpath".

```
addpath(pwd);
path
```

The following function calculates the A matrix. A function call has the following syntax : [out1,out2,...] = func_name(in1,in2,...), where the input arguments are the variables in1,in2,... and the output from the function is stored in out1,out2,... In our case, the input is the dimension of the matrix A, num_pts, and the output variables are A and iflag, an integer that tells us if the code was performed sucessfully.

```
[A,iflag] = c6s2_get_A(num_pts);
if(iflag ~= 1) then error
disp(['c6s2_get_A returned error flag : ' int2str(iflag)]);
end
```

We also see from the code below that the existence of a local variable in the function named i does nothing to alter the value of i at the point of calling.

```
i = 1234;
[A,iflag] = c6s2_get_A(num_pts);
i
```

Next, we ask the user to input the function values at the boundaries.

```
BC1 = input('Input the function value at x = 1 : ');
BC2 = input('Input the function value at x = num_pts : ');
source = input('Input the value of the source term : ');
```

We now call upon another subroutine that calculates the vector for the RHS.

```
[b_RHS,iflag] = c6s2_get_b_RHS(num_pts,BC1,BC2,source);
```

We now solve the system.

```
f = A\b_RHS;
```

Then, we make plots of the output.

```
figure; plot(x,f);
phrase1 = ['PDE solution with source = ' num2str(source)];
phrase1 = [phrase1 ', BC1 = ' num2str(BC1)];
phrase1 = [phrase1 ', BC2 = ' num2str(BC2)];
title(phrase1); xlabel('x'); ylabel('f(x)');
```

Then, to clean up, we clear the memory

```
clear all
```

## File c6s2_get_A.m

The first executable line of the m-file declares the name and input/output structure of the subroutine using the "function" command.

```
function [A,iflag] = c6s2_get_A(Ndim);
```

**iflag = 0;** signifies job not complete

If Ndim < 1, then we have an error, since we can't have a matrix with a dimension less than 1.
if(Ndim<1) signify error

**A=-1;**
**iflag = -1;**
we return control to the m-file that called this subroutine without executing the rest of the code.
**return;**
**end**

First, we declare A using sparse matrix format.
**nzA = 3*(Ndim-2) + 2;** # of non-zero elements
**A = spalloc(Ndim,Ndim,nzA);** allocate memory
**A(1,1) = 1;**
**A(Ndim,Ndim) = 1;**
**for i=2:(Ndim-1)**
**A(i,i) = 2;**
**A(i,i-1) = -1;**
**A(i,i+1) = -1;**
**end**

**iflag = 1;** signify job complete and successful

**return;** return control to the m-file that called this routine

**File c6s2_get_b_RHS.m**
**function [b_RHS,iflag] = c6s2_get_b_RHS(num_pts,BC1,BC2,source);**
**iflag = 0**; declares job not completed

**if(num_pts < 3)** not enough points
**iflag = -1;**
**b_RHS = -1;**
**return;**
**end**

We allocate space for b_RHS and initialize to zeros.
**b_RHS = linspace(0,0,num_pts)';**

Now, we specify the first and last components from the boundary conditions.
**b_RHS(1) = BC1;**
**b_RHS(num_pts) = BC2;**

Next, we specify the interior points.
**for i=2:(num_pts-1)**
**b_RHS(i) = source;**
**end**

**iflag=1;** signifies successful completion

**return;**

## 6.3. Inline functions

Sometimes, we do not want to go through the bother of writing a separate m-file to define a function. For these times, we can define an inline function. Let us say that we want to define the function
f1(x) = 2*x + 3*x^2

We can define this function using
**f1 = inline('2*x + 3*x^2');**

Then, we can call this function directly
**f1(1), f1(23)**

We can also define functions using vectors and matrices as input.
**invariant2 = inline('(trace(A)*trace(A) - trace(A*A))/2');**
**A = rand(3);**
**invariant2(A)**

We can check the definition of the function by typing its name
**invariant2**

While this is convenient, the execution of inline functions is rather slow.

```
clear all;
try
invariant2
catch
disp('We see that inline functions are cleared also');
end
```

### 6.4. Functions as function arguments

The function, trig_func_1, listed below,
returns the value of
f(x) = a*sin(x) + b*cos(x)
for given values of a, b, and x.
The function, plot_trig_1, listed below, plots a function on the domain 0 to 2*pi.

The following code asks the user to input values of a and b, and then uses plot_trig to plot trig_func_1 by
including the function name as an argument in the list.

```
disp('Plotting a*sin(x) + b*cos(x) ...');
a = input('Input a : ');
b = input('Input b : ');
func_name = 'trig_func_1';
```

make sure current direction is in the path

```
addpath(pwd)

plot_trig_1(func_name,a,b);

clear all
```

**File trig_func_1.m**

```
function f_val = trig_func_1(x,a,b);
f_val = a*sin(x) + b*cos(x);

return;
```

**File plot_trig_1.m**

```
function iflag = plot_trig_1(func_name,a,b);
```

**iflag = 0;** signifies no completion

First, create an x vector from 0 to 2*pi

```
num_pts = 100;
x = linspace(0,2*pi,num_pts);
```

Next, make a vector of the function values. We evaluate the argument function indirectly using the "feval" command.

```
f = linspace(0,0,num_pts);
for i=1:num_pts
f(i) = feval(func_name,x(i),a,b);
end
```

Then, we make the plot.

```
figure;
plot(x,f);
xlabel('Angle (radians)');
ylabel('Function value');

return;
```