Lecture `matlab2`: More MATLAB

August 22, 2022

Summary: Programming control: `if`, `for`, and `while`; scripts and functions; plots and tables

References: Appendix B of T. Sauer's textbook *Numerical Analysis*; Chapter 1 of Cleve Moler's textbook *Numerical Computing with MATLAB*; Monika Nitsche's tutorial (posted on the `UNM Learn` page)

Elements of programming

Three of the basic programming control structures are 'if' constructs, and 'for' and 'while' loops. Let's consider how these structures are implemented in MATLAB.

`if` statement

Suppose we want to define a variable `y` to be the absolute value of another (already defined) variable `x`. Ordered relations in MATLAB are implemented via the operators `>, <, >=,` and `<=`. In addition, the equality test operator is `==`. We can define `y` via the following code snippet.

```
if x >= 0
  y = x;  % Semicolons suppress command window output.
else
  y = -x;
end
```

The indentation of the lines `y = x;` and `y = -x;` has been artificially introduced. Although not necessary for functionality, such indentation enhances readability. The `end` statement here indicates when the `if` construct is over.

We can use the `elseif` statement to obtain a more complex `if` construct. Suppose `x` is a scalar, and we wish to define the following scalar function.

$$y = \begin{cases} x^2 & \text{if} \quad x > 0 \\ x & \text{if} \quad -1 \le x \le 0 \\ -x & \text{if} \quad x < -1, \end{cases}$$

The following code in MATLAB will so define this function.

```
if x>0
  y = x^2;
elseif x<-1
  y = -x;
else
  y = x;
end
```

The above code does not explicitly use the condition $-1 \leq x \leq 0$, rather implements it via exclusion of all others. Should we want to explicitly test whether or not `x` lies in the interval $[-1, 0]$, then would have to test both `x>=-1` and `x<=0`. That is, we would need a logical 'and' operator. In MATLAB, this is the ampersand `&`. Thus, our condition can be explicitly written as `(x>=-1)&(x<=0)`. In addition, MATLAB's logical 'or' operator is the pipe or vertical bar `|`.

`for` loop

In the previous lecture, we discussed how to sum the integers from 1 to 100 inclusive. The first (and simplest) way of doing this is `sum(1:100)`, making use of the `sum` function and the incremental colon operator `:`. The following code achieves the same end via a `for` loop:

```
>> x = 0;
>> for q = 1:100
x = x + q;
end
>> x
x =
         5050
```

The above example is a very simple case and not very useful; why would we want to complicate `sum(1:100)` by writing four or five lines of code? This is a familiar pattern in MATLAB: a task which would seem amenable to a `for` loop can often be implemented in a much shorter (and usually computationally faster) way by clever uses of some of MATLAB's advanced matrix–vector operators (e.g. the colon operator).

However, `for` loops still have important applications. For example, suppose we have a 2×2 matrix A and a 2×1 vector v defined as

$$A = \begin{pmatrix} 3 & 4 \\ 5 & 2 \end{pmatrix}, \qquad v = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Further suppose we need the 10 vectors Av, A^2v, A^3v, …, $A^{10}v$. The following code will output these vectors.

```
A = [3 4; 5 2];
v = [1;1];
B = [1 0; 0 1]; % The identity matrix.
for q = 1:10
  B = A*B;  % Now B = A^q.
  B*v  % No semicolon: answer will be printed.
end
```

`while` loop

The `while` loop is very similar in spirit to the `for` loop. Here is a simple example.

```
>> x = 0;
>> while x<10
x = x+1;
end
>> x
x =
     10
```

The code stops running when the condition `x<10` is no longer satisfied. However, keep in mind that this construction can cause troubles, as seen in the following.

```
x = 3;
while x>0
  x = x+4;
end
```

If one were to run the above code, the condition `x>0` would never be false, and the `while` loop would continue indefinitely.

Scripts and functions

Not all of your work can be done in the command window. You will need to write your own scripts and functions. Each is a collection of commands, bundled together. Unlike scripts, functions have take definite inputs (arguments) and return outputs.

Scripts

Often you will want to collect a series of commands into a program or *script*, usually since you want to run the collection repeatedly and don't want to retype the commands afresh into the command window. Here is an example.

```
%
%  Simple test script: SimpleScript.m
%  Adds two explicitly given vectors of the same size.
%
   u = [10:0.5:20]; % Semicolons suppress command window output.
   v = [ 1:0.5:11];
   w = v+u
```

If this script is saved as the file `SimpleScript.m`, then, so long as the MATLAB path includes the folder (or directory) where the file resides, we can type `SimpleScript` into the command window.

```
>> SimpleScript
w =
  Columns 1 through 15
    11    12    13    14    15    16    17    18    19    20    21    22    23    24    25
```

```
  Columns 16 through 21
    26    27    28    29    30    31
```

In the example, the vectors `u` and `v` don't appear in the output, since their output has been suppressed with the semicolon notation. This example script is only 3 lines long, but a script can be as long as you need.

Functions

Earlier in this lecture, we used the `if` clause to define the function

$$y = \begin{cases} x^2 & \text{if} \quad x > 0 \\ x & \text{if} \quad -1 \leq x \leq 0 \\ -x & \text{if} \quad x < -1, \end{cases}$$

Recall that the code used to evaluate this function was

```
if x>0;
  y = x^2;
elseif x<-1;
  y = -x;
else
  y = x;
end
```

Of course, we do not want to type this whole mess into the command line every time an evaluation of $y(x)$ is required. To avoid doing so, we can define our own custom MATLAB function `eval_y`, which takes one argument. Indeed, creating functions is the basic technique of *modular programming* forming the basis for any programming language. Let's describe how to do this in MATLAB.

Type `pwd` at the MATLAB command prompt, and navigate into the displayed folder on your hard drive. Create a new file, and name it `eval_y.m`. Place the following code into `eval_y.m`:

```
      function [ y ] = eval_y ( x )
%     function [ y ] = eval_y ( x )
%
%     This is the help text for my function.
%
if x>0;
  y = x^2;
elseif x<-1;
  y = -x;
else
  y = x;
end
```

Save this file, and return to the MATLAB command prompt. Create a scalar `x` with some value. Typing `eval_y(x)` then evaluates the function, using the input provided by the value of `x`. What happens if you type `help eval_y` in the command line? Is there a typo in the function above?

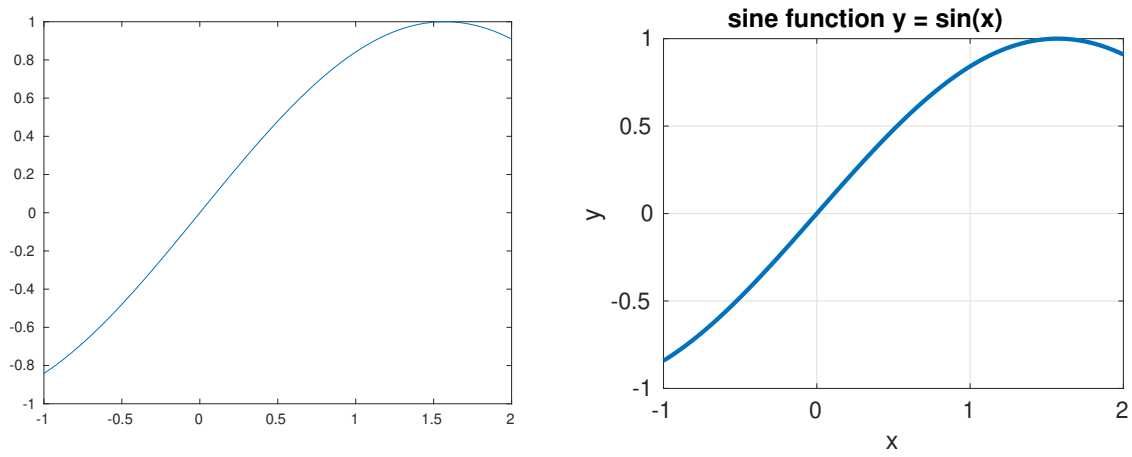

Figure 1: Plots of the function $y = \sin x$ for $x \in [-1, 2]$. The left plot stems from MATLAB's default output, the right one from the script `MakeSimplePlot`.

You can define any number of functions this way by creating separate `.m` files (or m-files) for each function. Each function must reside in its own individual file, and if you name the function file `foo.m`, it is good practice to name the function itself `foo`. In the example above, the function file is `eval_y.m`, and the function declaration reads `function[y] = eval_y(x)`. To write a function of `n` inputs and `m` outputs, the grammar is

```
function[out1, out2, ..., outm] = foo(in1, in2, ..., inn)
```

and to call the function, one would type

```
[out1, out2, ..., outm] = foo(in1, in2, ..., inn)
```

at the command line.

Plots and tables

Plots

One of MATLAB's strengths is its graphical capabilities. There are bells and whistles, but we only consider the fundamentals. Let us graph the function $y = \sin x$ on the interval $x \in [-1, 2]$. We choose the domain points as `x = [-1:0.01:2]`, which is a 1×301 row vector, enough points for a smooth plot, but not so many that `x` is a huge vector. Defining `y=sin(x)`, we can then use MATLAB's `plot` function:

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

The left panel in Fig. 1 shows the output. Many other options are available, and you can type `help plot` at the command line for more information, and more advanced examples. The figure shows the default MATLAB output. For your assignments (and other professional work) you will almost certainly want to suitably modify the output; for example, often MATLAB's default fonts are too small. Here is a script which makes the better plot shown in the right panel of Fig. 1.

```
%
%  Script: MakeSimplePlot
%  Makes a plot of y=sin(x) with larger font and labels.
%
x = [-1:0.01:2];
y = sin(x);
plot(x,y,'linewidth',3)
set(gca, 'fontsize', 16);
xlabel('x')
ylabel('y')
title('sine function y = sin(x)')
grid on
```

Plots in this course should be legible and convey information. Always label axes and include an appropriate title, legend, or figure caption.

Tables

You will also need to prepare tables in this course, and the following example describes a simple procedure for doing so. Modification of this example will suffice for most of your assignments. Consider the following expressions

$$E_1(x) = \frac{1-\cos x}{\sin^2 x}, \qquad E_2(x) = \frac{1}{1+\cos x}.$$

Since $\sin^2 x = 1 - \cos^2 x = (1-\cos x)(1+\cos x)$, the two expressions are equivalent for $x \neq 0$. However, the first is an indeterminate form of type $0/0$. Using the second expression, we see that $\lim_{x\to 0} E_1(x) = \frac{1}{2} = \lim_{x\to 0} E_2(x)$, but let's see how the well the computer does in evaluating these expressions for small x. The following script makes a table of values for inputs $x = 10^0, 10^{-1}, \ldots, 10^{-12}$.

```
%  Matlab script: IndeterminateFormTable.
%  Generates a table of values for the expressions E1 and E2 as x-->0+.
%  Example from "Numerical Analysis," Timothy Sauer; p18, 2nd edition.
%  For more about format control see the following URL
%  http://www.cplusplus.com/reference/cstdio/fprintf/
%
x = logspace(0,-12,13);        % Creates the array x = [10^0 10^(-1) ... 10^(-12)]
E1 = (1-cos(x))./(sin(x).^2);  % Note the use of ./ for elementwise division
E2 = 1./(1+cos(x));
fid=fopen('IndeterminateFormTable.txt','w');  % filename for output
fprintf(fid,'-------------------------------------------------------------------------------------\n');
fprintf(fid,'|        x         |        E1        | error(E1) |        E2        | error(E2) |\n');
fprintf(fid,'-------------------------------------------------------------------------------------\n');
for k = 1:length(x)
  % See URL above for explanation of these format statements. Here 1.14f means fixed
  % (decimal) point format, with 14 digits past the decimal point, while 1.3e means
  % scientific notation, with 3 digits past the decimal point.
  errE1=abs(E1(k)-0.5);
  errE2=abs(E2(k)-0.5); % Define these errors here to shorten next line.
  fprintf(fid,'| %1.14f | %1.14f | %1.3e | %1.14f | %1.3e |\n',x(k),E1(k),errE1,E2(k),errE2);
end
fprintf(fid,'-------------------------------------------------------------------------------------\n');
```

```
fprintf(fid,'Compares absolute errors of E1=(1-cos(x))/sin^2(x) and E2=1/(1+cos(x)) for small x\n');
fclose(fid);
```

The output from the script is shown below. Notice that the table output is lined up, and that numbers are presented with an appropriate number of digits. The output shows that $E_2(x)$ behaves well numerically. Indeed, it approaches $\frac{1}{2}$ as expected. But $E_1(x)$ does not. The error $|E_1(x) - \frac{1}{2}|$, with $E_1(x)$ viewed as an approximation to $\frac{1}{2}$, should go to zero as x becomes small. As we'll learn later, here round-off error spoils the calculation. The error $|E_1(x) - \frac{1}{2}|$ does go to zero. Notice that errors have been reported with just a few digits. Errors should not be reported like `1.252304304004e-03`; that's too much information.

```
-------------------------------------------------------------------------------
|        x         |        E1        | error(E1) |        E2        | error(E2) |
-------------------------------------------------------------------------------
| 1.00000000000000 | 0.64922320520476 | 1.492e-01 | 0.64922320520476 | 1.492e-01 |
| 0.10000000000000 | 0.50125208628857 | 1.252e-03 | 0.50125208628857 | 1.252e-03 |
| 0.01000000000000 | 0.50001250020848 | 1.250e-05 | 0.50001250020834 | 1.250e-05 |
| 0.00100000000000 | 0.50000012499219 | 1.250e-07 | 0.50000012500002 | 1.250e-07 |
| 0.00010000000000 | 0.49999999862793 | 1.372e-09 | 0.50000000125000 | 1.250e-09 |
| 0.00001000000000 | 0.50000004138685 | 4.139e-08 | 0.50000000001250 | 1.250e-11 |
| 0.00000100000000 | 0.50004445029134 | 4.445e-05 | 0.50000000000013 | 1.250e-13 |
| 0.00000010000000 | 0.49960036108132 | 3.996e-04 | 0.50000000000000 | 1.221e-15 |
| 0.00000001000000 | 0.00000000000000 | 5.000e-01 | 0.50000000000000 | 0.000e+00 |
| 0.00000000100000 | 0.00000000000000 | 5.000e-01 | 0.50000000000000 | 0.000e+00 |
| 0.00000000010000 | 0.00000000000000 | 5.000e-01 | 0.50000000000000 | 0.000e+00 |
| 0.00000000001000 | 0.00000000000000 | 5.000e-01 | 0.50000000000000 | 0.000e+00 |
| 0.00000000000100 | 0.00000000000000 | 5.000e-01 | 0.50000000000000 | 0.000e+00 |
-------------------------------------------------------------------------------
Compares absolute errors of E1=(1-cos(x))/sin^2(x) and E2=1/(1+cos(x)) for small x
```