Making MatLab work for you, quickly

Utkarsh Upadhyay,
Indian Institute of Technology Kanpur

May 21, 2009

1 The basics

1.1 Things to notice

- Workspace : The place where you keep your variables
- Command History : To repeat your mistakes easily
- Command Window : The real arena
- Editor : Source of code

1.2 Things to remember

- Matlab is only a matrix calculator, and a programming language, it is not magic
- There are more than one way of doing things, but only one fastest way
- Use tab-completion whenever it works
- *help <name-of-function>*
- To output, remove a semicolon from the end of a line

1.3 Things to understand

- A *.m* file can be a script, just like typed from the terminal.
- A *.m* file can also be a function of (usually) the same name as the filename.

2 Matlab sheet

This list is non exhaustive. Remember *help* <*name*> for more help.

2.1 Cheat sheet

INPUT	To take input from the user R = INPUT('Any Expression:') will input a string, find its *value* using variables in the workspace R = INPUT('Your name:', 's') will input a string and return it
dlmread	Read a file directly into a matrix
edit	Editing a *.m* file in the source editor
save	for saving variables from the workspace to a file
load	for reading variables from a file
CUMPROD	Provide a cumulative product of elements
CUMSUM	Provide a cumulative sum of elements
SUM	Provide juat a sum of elements
DIAG	Create diagonal matrices and diagonals of a matrix
GALLERY	Generate standard matrixes
EYE	Produce an identity matrix
ZEROS	Produce a matrix of zeros
ONES	Produce an matrix of 1s
MAX	Get the maximum
MIN	Get the minimum
SORT	Sorting arrays
UNIQUE	Find the unique elements in a array
SIZE	Finds the size of a $n \times m \times l \times \ldots$ matrix
FIND	Locate all non-zero elements in an array
TIC	Starts an internal stopwatch
TOC	Stops the stopwatch and gives the time elapsed

Table 1: A small list of standard library

3 Concatenating Matrices

Notice the behaviour, try out your own and then see the questions below:

```
>> A = [ [1 2]; [3 4] ]

A =

     1     2
     3     4

>> B = [ [5 6]; [7 8] ]'

B =

     5     7
     6     8

>> C = [ A B ]

C =

     1     2     5     7
     3     4     6     8
```

3.1 Q1

Observe and find the conditions when MatLab *should* be able to 'join' (technically *concatenate*) matrices. (Hint: Use the 'size' function)

4 Scripts, functions and anonymity

Scripts	are just a collection of statements of Matlab. Can be run with just the name on the command window.
Functions	can be called for different arguments to obtain differnt outputs. Called by the function name (filename).
Anonymous Functions	can be made for ready use. eg. *>> f = @(x,y) max([x y] * [[1 2]; [3 4]] + [1 5]);* Now, f is a function which takes two values and returns the maximum of $x + 3y + 1$ and $2x + 4y + 5$.

Table 2: Scripts v/s Functions

4.1 Syntax titbits

IFs	IF *condition* . . . ; ELSE . . . ; END
Ranges	start:step:end *>> 1:0.3:2* returns: *[1 1.3 1.6 1.9]*
FOR	FOR ii = range . . . ; END
FUNCTIONs	*function* [OUT1 OUT2] = funk_tion(IN1, IN2) OUT1 = . . . ; OUT2 = . . . ; END
WHILE	WHILE *condition* . . . ; END

Table 3: Syntax

The loops modifiers:

- *break* : Breaking the closest loop
- *continue* : Skipping the rest of the loop and continuing with the next value

4.2 Question 2

Write a function in file: *checkCat.m* which would check whether *[X Y]* is a valid operation or not. Assume the matrices to be 2D only.

```
>> isAlright = checkCat( A, B )

isAlright =

     1

>> isAlright = checkCat( A, [1 3] )

isAlright =

     0
```

4.3 Question 3

Suppose one does not have the *step* option in making a range array (the *2* in *1:2:10* is not allowed). Write a function *my_step.m* which would serve the same purpose, e.g. (*range = my_step(start, finish, stepsize);*)

Hints:

- You can still use *start:end* form, if you want.
- An array can grow: *X = [X 12];*

```
>> range = my_step(0,9,2)

range =

     0     2     4     6     8
```

5 . . . quickly

tic and *toc* can be used to time operations using the internal stopwatch. And they can be used in a *short-cut* way.

```
>> tic, my_step(0,100000,2);, toc
Elapsed time is 3.728453 seconds.

>> tic, my_step(0,10000,2);, toc
Elapsed time is 0.105028 seconds.
```

5.1 Question 4.1

Preallocation can shave down a running time. One can tell how *large* and resulting array is going to be. Try to preallocate enough space for X instead of allowing it to grow, and see the time differences.
Hint: Use *zeros* to preallocate space.

5.2 Question 4.2

Write the *my_better_step.m* which is a one line function which just returns *start:stepSize:finish*. And compare the running time for the two versions for arrays of size 10, 1000 and 1000000.

5.3 Question 5.1

Write a function $B = greater\_than\_n(A,n)$ which will return the elements in A which are greater than n.
Hint: B will grow in the loop.

```
>> B = greater_than_n(A, 2)

B =

     3     4

>> B = greater_than_n(A', 2)

B =

     3     4
```

6 Logical indexing

Used to *pick* out values from an array.

```
>> A

A =

     1     2
     3     4

>> A( [[true false]; [false true]] )

ans =

     1
     4

>> true

ans =

     1

>> false

ans =

     0
```

Now *true* is *1* and *false* is *0*, but 1 and 0 are not *true* and $false$. Logical indexes, however, *look* like numbers:

```
>> A > 2

ans =

     0     0
     1     1
```

Where is actually: *[[false false]; [true true]]* .

6.1 Question 5.2

Write the *greater_than_n_better.m* which has the same behaviour as *greater_than_n.m* but uses Logical Indexing.
Compare the time taken by the two for large arrays.
Hint: Use *gallery* for *large* arrays.

6.2 Question 6.1

Write a function which takes a 1D array, and then produces an array which is the product of adjacent elements of the input array. Call it *adj_mul.m*

```
>> adj_mul([1,2,3])

ans =

     2     6

>> cumsum(ones(1,10))

ans =

     1     2     3     4     5     6     7     8     9    10

>> adj_mul(cumsum(ones(1,10)))

ans =

     2     6    12    20    30    42    56    72    90
```

7 Vectorization

Indexes of an array can be in form of a range.
Also, all arrays are stored in a column major form.

```
>> A

A =

     1     2
     3     4

>> A( [1 2 3 4] )

ans =

     1     3     2     4

>> A( [1 3] )

ans =

     1     2

>> C = 1:.5:5

C =

    1.0000    1.5000    2.0000    2.5000    3.0000    3.5000    4.0000    4.5000    5.0000

>> C(1:length(C)-1)

ans =

    1.0000    1.5000    2.0000    2.5000    3.0000    3.5000    4.0000    4.5000

>> C(2:length(C))

ans =

    1.5000    2.0000    2.5000    3.0000    3.5000    4.0000    4.5000    5.0000
```

The idea is that loops and repeated function calls are very slow, hence, should be avoided. Array indexing can avoid loops completely. Also, useful are *dot (.)* operators, note the element-wise operation. The . can be placed before $*$ and / to get desired results.

```
>> A

A =

     1     2
     3     4
```

```
>> B

B =

     5     7
     6     8

>> A * B

ans =

    17    23
    39    53

>> A .* B

ans =

     5    14
    18    32
```

7.1 Question 6.2

Rewrite the program *adj_mul_better.m* using vectorization and dot operators and compare timings.

8 Final trick

Sliced allocations also work.

```
>> A

A =

     1     2
     3     4

>> A( [1 3] ) = [-1 200]

A =

    -1   200
     3     4
```

This is a tough one.

8.1 Question 7

Visit: http://www.mathworks.com/support/tech-notes/1100/1109.html for the solution. Repeat a vector value when the following value is zero. This is called *zero order hold*.

```
a=2; , b=3; , c=5; , d=15; , e=11;
x = [a 0 0 0 b 0 0 c 0 0 0 0 d 0 e 0 0 0 0 0];
```

into this:

```
>> y = zero_hold(x);
y = [a a a a b b b c c c c c d d e e e e e e];

>> x

x =

     1     0     0   -10     0    18     0     0    -1     0

>> y = zero_hold(x)

y =

     1     1     1   -10   -10    18    18    18    -1    -1
```

Hints:

- *find* can tell you the indexes you will be interested in.
- *diff* can find differences in successive elements.
- $(b - a) + a == b$