Introduction to Matlab

Andreas C. Kapourani

(Credit: Steve Renals & Iain Murray)

25 January 2017

1 Introduction

MATLAB is a programming language that grew out of the need to process matrices. It is used extensively in science and engineering. It now has many features and toolboxes to support numerical programming (including machine learning) and visualization. Here you will meet only a fraction of MATLAB features. **Important**: Data analysis, programming, and mathematics are not spectator sports! You can only learn to munge data by doing it. Unless you spend a reasonable amount of time per lecture (around 2hrs) reproducing the material for yourself, you won't really understand it.

1.1 Getting started

Open the terminal and type:

```
s1263191@chatton: matlab
```

This will bring up the main working window of MATLAB which includes the following elements:

- **Current Folder** - Access your files.
- **Command Window** - Enter commands at the command line, indicated by the prompt (`>>`).
- **Workspace** - Explore data that you create or import from files.

You can type MATLAB expressions to the `>>` prompt, eg:

```
>> 57*64.3 % multiply two numbers
ans =
     3.6651e+03

>> power(10, pi/2) % call a function
ans =
     37.2217
```

The second of these makes use of a built-in function, `power`, and a predefined constant `pi`. If you want to see what a MATLAB function does you can go to the help window and find it using the index, the contents, or by searching. (You can also type `>> help power` at the command line.) The MATLAB help system is very good, and worth exploring.

MATLAB handles complex numbers (using the symbol `j`) so it is possible to do things such as:
>> exp(j*pi) % using complex numbers
ans =
    -1.0000 + 0.0000i

We can set up some variables in MATLAB and perform some computations:

>> x = 3.4; % defining x and initializing it with a value
>> y = 5.7;
>> z = x*y;
>> z
z = 19.3800
>> a = 2; b = 7; % multiple assignments on the same line

Variables x, y and z persist in the MATLAB workspace until they are changed. Note that a semicolon (';') at the end of a line prevents the result of an expression being printed to the screen, useful when dealing with big vectors and matrices.

If you forgot the variables you can type who at the command line. The who command displays all the variable names you have used.

2 Vectors

In MATLAB 1D arrays don’t really exist. A vector is actually a 2D array or matrix. MATLAB allows creating two types of vectors: 1) row vectors and 2) column vectors.

A row vector is defined by placing a sequence of numbers within square braces:

v = [5, 1, 8, 5, 4]; % The commas are optional
v = [5 1 8 5 4]; % Same as above

>> v % v is a row vector, 1x5
v =
    5    1    8    5    4

To create a column vector you could use the transpose operator ’:

>> v'
ans =
    5
    1
    8
    5
    4

or you can explicitly create a column vector by using semicolon(;) to delimit the elements:

>> v = [5; 1; 8; 5; 4]; % semicolon(;) signifies where rows break

We can get the size of the vector by typing:

>> size(v) % size of column vector
ans =
    5    1
Useful vector operations include:

**Vector sum** \( v+w \)

**Vector difference** \( v-w \)

**Multiplication by a scalar** \( 10*v \)

**Scalar dot (inner) product** \( \text{dot}(v,w) \)

**Vector cross product** \( \text{cross}(v,w) \)

**Vector magnitude (norm)** \( \text{norm}(v) \)

**Vector sum** example:

\[
\begin{align*}
\text{>> } & \quad v = [5 \ 1 \ 8 \ 5 \ 4] ; \\
\text{>> } & \quad w = [2 \ 3 \ 4 \ 5 \ 6] ; \\
\text{>> } & \quad v+w \quad \% \text{ add two vectors} \\
\text{ans} & = \\
& \quad 7 \ 4 \ 12 \ 10 \ 10
\end{align*}
\]

Some operations on vectors work on the elements one-by-one, returning the answer as a vector of the same dimension:

\[
\begin{align*}
\text{>> } & \quad \text{sqrt}([1 \ 2 \ 3 \ 4]) \quad \% \text{ take the square root of vector elements} \\
\text{ans} & = \\
& \quad 1.0000 \ 1.4142 \ 1.7321 \ 2.0000
\end{align*}
\]

The colon (’:’) operator generates equally-spaced points between its first and last inputs:

\[
\begin{align*}
\text{>> } & \quad 1:4 \quad \% \text{ gap is 1 (default)} \\
\text{ans} & = \\
& \quad 1 \ 2 \ 3 \ 4 \\
\text{>> } & \quad 1:0.5:2 \quad \% \text{ gap is 0.5} \\
\text{ans} & = \\
& \quad 1.0000 \ 1.5000 \ 2.0000 \\
\text{>> } & \quad 4:-1:1 \quad \% \text{ gap is -1} \\
\text{ans} & = \\
& \quad 4 \ 3 \ 2 \ 1
\end{align*}
\]

When there are three arguments, the second one specifies the gap between the outputs (defaults to 1 when there are only two arguments).

If we want to look only at the first three elements of a vector:

\[
\begin{align*}
\text{>> } & \quad v(1:3) \quad \% \text{ take the first 3 element of } v' \\
\text{ans} & = \\
& \quad 5 \ 1 \ 8
\end{align*}
\]
2.1 Exercise

Create a vector vec, e.g. vec = [1 5 6 2 3 9].
Then evaluate vec+vec, vec/2, vec+3, vec.*vec, vec.^2 etc. Some operators (+, −) and most functions (exp, log, sin, etc.) work element-wise on all the numbers in the array. * and / work elementwise with scalars, but for matrices do matrix multiplication (and division). For elementwise versions of multiplication, division, and raising to the power, use .*, ./ and .^ (note the dots are important).

3 Matrices

MATLAB really comes into its own when you want to do computations with matrices, such as for machine learning and pattern recognition. Most variables in Matlab are matrices of double floating point numbers. If you write x=1 you will get a 1x1 matrix of doubles containing a 1.0.
Defining a matrix is similar to defining a vector. To define a matrix, you can treat it like a column of row vectors (note that the spaces are required!):

```matlab
>> m = [1 2 3; 4 5 6; 7 8 9] % create 3x3 matrix
m =
    1  2  3
    4  5  6
    7  8  9
>> size(m) % get the size of the matrix which is 3x3
ans =
    3  3
```

Semi-colon (;) is used to separate rows.

Here is another matrix:

```matlab
>> n = [1 0 0; 0 1 0; 0 0 1] % this is the identity matrix I
n =
    1  0  0
    0  1  0
    0  0  1
```

And we can do the usual matrix operations of add and multiply:

```matlab
>> m+n % matrix addition
ans =
    2  2  3
    4  6  6
    7  8 10
>> m*n % matrix multiplication
ans =
    1  2  3
    4  5  6
    7  8  9
```
\( \mathbf{n} \) is the 3x3 identity matrix. An easier way to generate an identity matrix is to use the MATLAB command `eye`:

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]

MATLAB overloads its operators (when it is not ambiguous). For example, **add 1** to each matrix element:

\[
\begin{bmatrix}
2 & 3 & 4 \\
5 & 6 & 7 \\
8 & 9 & 10 \\
\end{bmatrix}
\]

MATLAB can distinguish between matrix operations and element-by-element operations. The **dot** syntax is used for element-by-element operations. For example, the following multiplies corresponding elements of two matrices:

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 5 & 0 \\
0 & 0 & 9 \\
\end{bmatrix}
\]

You should be careful when doing matrix operations, since your matrices need to have the right size!

\[
\begin{bmatrix}
1 & 2; & 3 & 4 \end{bmatrix} \quad \text{\texttt{m} is 3x3 matrix and is multiplied with a 2x2 matrix}
\]

Error using * 
Inner matrix dimensions must agree.

Other useful functions include `zeros(i,j)` which generates an \( i \times j \) zero matrix, and `ones` to generate a matrix of ones. The `size` command returns the dimensions of a matrix:

\[
\begin{bmatrix}
0 & 0 \\
0 & 0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 1 & 1 \\
1 & 1 & 1 \\
\end{bmatrix}
\]

\[
\begin{array}{ccc}
\text{rw} & = & 2 \\
\text{cl} & = & 3 \\
\end{array}
\]
And of course it is possible to combine functions:

```matlab
>> [rw, cl] = size(ones(2,3)) % get the size of a 2x3 matrix
rw =
    2
cl =
    3
```

To access the $i,j$th element of a matrix $m$, use $m(i,j)$.

```matlab
>> m(2,3)
ans =
    6
```

The colon operator `:` can be used to select a submatrix. For example to select a submatrix containing rows 2 and 3 and columns 1 and 2 of $m$:

```matlab
>> m(2:3,1:2)
ans =
    4   5
    7   8
```

A colon on its own selects all rows (or columns):

```matlab
>> m(1,:)
ans =
    1   2   3
>> m(:,)
ans =
    1   2   3
    4   5   6
    7   8   9
```

Finally, a colon on its own turns a matrix into a 1D column vector, working column by column:

```matlab
>> m(:)
ans =
    1
    4
    7
    2
    5
    8
    3
    6
    9
```

Relation operators such as `<` and `>` work with matrices too, e.g.:

```matlab
>> m > 5 % which elements are bigger than 5 (returns boolean)
an =
    0   0   0
    0   0   1
    1   1   1
```
These are not elements of \( m \), but \textbf{boolean} values (MATLAB calls them logical indices) which indicate the positions where the test is true. To access the actual elements that are greater than 5:

\[
\texttt{>> m.*(m > 5)} \\
\texttt{ans} = \\
\begin{array}{ccc}
0 & 0 & 0 \\
0 & 0 & 6 \\
7 & 8 & 9
\end{array}
\]

We can also use these logical indices to extract just those values that obey the condition:

\[
\texttt{>> m(m > 5)} \\
\texttt{ans} = \\
7 \\
8 \\
6 \\
9
\]

Note that the result is a vector containing only those values that obey the condition. See also the function \texttt{find} (look at the help pages, i.e. type \texttt{help find}).

\section*{3.1 Other matrix operations}

Once you are able to create and manipulate a matrix, you can perform many standard operations on it. MATLAB has all the matrix operations you might expect. However, you should be careful, since the operations are numerical manipulations done on digital computers!

Given a matrix \( a \):

\[
\texttt{>> a = [ 1 4 2 ; 4 2 -1 ; 2 -1 3] } \quad \text{\textbackslash% 3}\text{x}\text{3 matrix} \\
\texttt{a} = \\
\begin{array}{ccc}
1 & 4 & 2 \\
4 & 2 & -1 \\
2 & -1 & 3
\end{array}
\]

we can take its transpose \( a' \), compute the matrix norm \texttt{norm(a)}, the determinant \texttt{det(a)} and the matrix inverse \texttt{inv(a)}:

\[
\texttt{>> a'} \quad \text{\textbackslash% matrix transpose} \\
\texttt{ans} = \\
\begin{array}{ccc}
1 & 4 & 2 \\
4 & 2 & -1 \\
2 & -1 & 3
\end{array}
\]

\[
\texttt{>> inv(a)} \quad \text{\textbackslash% matrix inverse} \\
\texttt{ans} = \\
\begin{array}{ccc}
-0.0746 & 0.2090 & 0.1194 \\
0.2090 & 0.0149 & -0.1343 \\
0.1194 & -0.1343 & 0.2090
\end{array}
\]

\[
\texttt{>> det(a)} \quad \text{\textbackslash% matrix determinant} \\
\texttt{ans} = \\
-67
\]
There are also routines that let you find solutions to linear equations. For example, if $Ax = b$ and you want to find $x$, a slow way to find $x$ is to simply invert $A$ and perform a left multiply on both sides. However, this approach is not efficient and unstable (another approach is to perform the L/U decomposition with pivoting, for example). Matlab has special commands that will do this for you using the operator \\:

$\text{>> b = [1 ; 2; 3]} \%\text{ vector containing the right sides of the linear equation}$

\[
b = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}
\]

$\text{>> x = a\backslash b} \%\text{ denotes the solution to the matrix equation } Ax = b$

\[
x = \begin{bmatrix} 0.7015 \\ -0.1642 \\ 0.4776 \end{bmatrix}
\]

$\text{>> a*x} \%\text{ confirm that } Ax = b$

\[
\begin{bmatrix} 1.0000 \\ 2.0000 \\ 3.0000 \end{bmatrix}
\]

Some other very useful commands are \texttt{sum} which computes the sum of each column, and \texttt{mean} which computes the mean of each column:

$\text{>> sum(m)} \%\text{ compute the sum of each column}$

\[
\text{ans} = \begin{bmatrix} 12 & 15 & 18 \end{bmatrix}
\]

$\text{>> mean(m)} \%\text{ compute the mean of each column}$

\[
\text{ans} = \begin{bmatrix} 4 & 5 & 6 \end{bmatrix}
\]

How to take the mean of each row? One way would be to compute the transpose:

$\text{>> mean(m')} \%\text{ compute the mean of each row of matrix } m$

\[
\text{ans} = \begin{bmatrix} 2 & 5 & 8 \end{bmatrix}
\]

Another way would be to read the Help documentation on mean and find that it takes an optional second argument:

$\text{>> mean(m,2)} \%\text{ compute the mean of each row of matrix } m$

\[
\text{ans} = \begin{bmatrix} 2 & 5 & 8 \end{bmatrix}
\]
Note that here the answer is a column vector.

And \texttt{reshape}(m, r, c) reshapes matrix (or vector) \texttt{m} to have \texttt{r} rows and \texttt{c} columns:

\begin{verbatim}
>> reshape(m, 1, 9) % reshape matrix \texttt{m} to 1x9 vector
ans =
   1   4   7   2   5   8   3   6   9
\end{verbatim}

\section*{4 Control structures}

MATLAB has the usual set of control structures: \texttt{for}, \texttt{if-else}, etc.

The \texttt{for} loop allows us to repeat certain commands. All of the loop structures in matlab are started with the keyword \texttt{for} and they all end with the word \texttt{end}. A \texttt{for} loop requires a vector of loop variable values, eg:

\begin{verbatim}
>> for (i=1:5) % repeat 5 times
    i % print the iteration in the console
end
i =
   1
i =
   2
i =
   3
i =
   4
i =
   5
\end{verbatim}

or

\begin{verbatim}
>> points = [3 5 7 9 11];
>> for i = points % iterate over the values of \texttt{points'}
    i % print the value of \texttt{points'} in \texttt{i}th iteration
end
i =
   3
i =
   5
i =
   7
i =
   9
i =
  11
\end{verbatim}

Use the help system to find out more.

To control the sequence of your program, the \texttt{if-else} control structure can be used. The way to do that is to put the code within an \texttt{if} statement (and possibly in an \texttt{else} statement) and finish with the \texttt{end} statement. For example:
5 Turning maths in matrix operations: Vectorization

If you write MATLAB code with lots of explicit loops (even worse, lots of nested loops), then your code is unlikely to be taking advantage of the facilities for efficient matrix-vector computation (vectorization). Writing your code in terms of matrix-vector operations will result in code that is easier to read and that runs an order of magnitude faster.

Given a mathematical expression like:

\[ R = \sum_{i=1}^{I} fn(x_i, x_j) val(z_i) \]  

The results of \( fn \) could be put in an \( I \times J \) matrix, and the results of \( val \) in a \( I \times 1 \) vector. The sum is then a matrix-vector multiply:

\[
\begin{aligned}
>> \text{fn} &= \begin{bmatrix} 2 & 3 \\ 4 & 6 \\ 7 & 8 \end{bmatrix}; \quad \text{\( fn \) is a 3x2 matrix} \\
>> \text{val} &= \begin{bmatrix} 4 \\ 7 \\ 8 \end{bmatrix}; \quad \text{\( val \) is a 3x1 matrix} \\
>> R &= \text{fn'} * \text{val} \quad \text{\( R \) is a 2x1 matrix}
\end{aligned}
\]

If we used a for loop, we should write something like the following:

\[
\begin{aligned}
>> \text{R} &= \text{zeros}(2,1); \quad \text{\% initialize result matrix to zeros} \\
>> \text{for } i &= 1:3 \quad \text{\% iterate over } i \\
&\quad \text{for } j &= 1:2 \quad \text{\% iterate over } j \\
&\quad \text{R}(j,1) &= \text{R}(j,1) + \text{fn}(i,j) * \text{val}(i); \\
&\quad \text{end} \\
&\text{end} \\
>> \text{R} \\
\text{R} &= \\
92 \\
118
\end{aligned}
\]

**Exercise:** Given another expression:

\[ R = \sum_{i=1}^{I} fn(x_i, x_j) weights(z_i, z_j) \]

use vectorization to compute the result vector (hint: first do the multiplication inside the sum for all \( i \) and \( j \) at once, and then sum out the index \( i \)).
Sometimes repmat or bsxfun is required to make the dimensions of matrices match. Check online or Matlab help to understand what they do, we will also have an example of these function in the next lab.

In general, if you see code that involves tricks you don’t understand, try to write a for-loop version based on what you think it should do, and see if you get the same answer. If you are writing code with lots of for loops, keep going, but then afterwards try to replace parts of it with a vectorized version, checking that your answers don’t change. There are more references in the Matlab notes at: http://homepages.inf.ed.ac.uk/imurray2/compnotes/matlab_octave_efficiency.html

6 Next lab session

In the next lab, we will continue with some more sophisticated matrix operations, we will learn how to program in MATLAB using scripts and functions. Also, we will go through data visualization using plots and how to handle text files.

If you want to clear the command window you can type:

```matlab
>> clc % clear commands shown in 'Command Window'
```

and if you want to clear all your data and files that are stored in the workspace, you can use the clear command. Be careful though, it does not ask you for a second opinion!

```matlab
>> clear % clear data stored in 'Workspace'
```