CS5350: Machine Learning *Piyush Rai*

Matlab for Machine Learning

This tutorial is intended to provide you with a basic introduction to MATLAB but it also touches upon certain issues that you may come across while writing machine learning code using MATLAB. Towards the end, two simple examples (on machine learning problems) are also provided to reinforce the concepts. Finally, some useful pointers are given at the end that should help you explore, on your own, more about MATLAB.

1 Starting up and exiting

Windows: On Windows, you can find MATLAB by going to Start → Programs → Matlab → R2007b. This assumes R2007b to be the installed version of MATLAB which could be something else on your machine, depending on the actual installation you may have.

Linux/Unix: You can start MATLAB by just typing `matlab` on an `xterm` window.

`$matlab &`: Starts MATLAB in the GUI mode.
`$matlab -nojvm -nodesktop &`: Starts MATLAB without the GUI mode.

MATLAB has its own command window interface and the command prompt look like `>>`. You can execute various MATLAB commands from the command window itself. However, to write your own MATLAB code, you would need some text-editor. You can use any editor of your own choice (VIM/EMACS or whatever you are comfortable with). MATLAB also provides its own editor which can be started by typing `>>edit` on the MATLAB command window. To open a specific file (say `foo.m`) with the editor, just type `>>edit foo.m`

Diary: Typing the `diary` command starts logging in all the commands executed during a session. By default, it saves them to a file named '`diary`'. To turn this off at the end of a session, you can just type `diary off`. You can see the transcript of a session in the `diary` file present in the current directory.

Help: MATLAB comes with an excellent documentation and help system. The following `help` commands can be used to get help on specific topics: `help general` - General purpose commands. `help ops` - Operators and special characters. `help lang` - Programming language constructs. `help elmat` - Elementary matrices and matrix manipulation. `help elfun` - Elementary math functions. `help specfun` - Specialized math functions. Note that any of these commands can splash out lots of information on the screen. To see information one screenful at a time, use `'more on'` before typing `help`. For example:

```
>>more on
>>help elmat
```

For help on a specific function (say `funcName`), just type `>>help funcName`.

Exiting MATLAB: Typing `>>exit` from the MATLAB command window quits the current MATLAB session.

2 Writing and executing programs

MATLAB, essentially, is a command interpreter. In its basic mode, it acts as a calculator (or rather an algebraic calculator!). This mean that you can execute commands from its command window interface. Commands can be executed like (% below just indicates a comment):

```
>>x = 10 % assignment of a value (with printing)
>>x = 10; % assignment of a value (semicolon used to suppress printing)
>>y = exp(x);
>>z = x + y;
```

However, to do something more substantial, MATLAB offers features such as *script files*, and files that can contain user-defined functions. These files usually consist of a bunch of well-defined MATLAB commands/statements.

Script files: If your program does not require passing input arguments or returning results, you can just write all the statements in a file and save it with a ".m" extension. To execute it, you can press F5 key or type the file name on MATLAB command prompt followed by hitting the enter key.

Function files: Often you do need to pass input arguments and return results of various computations. Functions help you achieve this. A typical function in MATLAB looks like:

```
function [out1, out2,.., outn] = foo(in1, in2,...,inm)
% function body and return statements
```

The input and output arguments are optional. Also note that comments in a MATAB code are preceded by %. The function is saved in a .m file. One can define several functions in a .m file. Typically, the file is named after the main function since that is the one you would be calling from the MATLAB command prompt. For examples, your .m file could have a main function named `foo` (such as the one defined above) and several other functions (`foo2`, `foo3`, etc) called by `foo`. In such a case, you save the file as `foo.m` and, to execute it, you would call foo from the MATLAB command prompt.

```
>>[out1, out2,.., outn] = foo(in1, in2,...,inm);
```

Variables names: Legal variable names consist of any combination of letters and digits, starting with a letter. Variable names cannot consist of special characters. Reserved MATLAB keywords (such as `eps`, `pi`, etc) should not be used.

Mathematical Operators: MATLAB allows the use of all standard mathematical operators (`+, -, *, /,` ∧, *etc*). Often, while working with matrices, we may want to use some of these operators element-wise. Such an operator can be preceded by a dot operator to achieve this. For example, to do an element-wise square of a matrix `X`, you should use: `X.∧2`.

3 Creating and accessing matrices

The most basic primitive for data representation in MATLAB is a matrix. A 1-D matrix is also called a *vector*. A scalar can be thought of as a 1x1 matrix. In principle, you can create matrices having arbitrary number of dimensions, but mostly you would have to deal with only matrices having dimensions 2 (and, very rarely, sometimes 3).

3.1 Creating matrices

There are several ways to create matrices in MATLAB. **The general rule:** to create a row, enter entries separated by spaces (or commas, although not necessary). To start a new row, use a semi-colon (;). Some simple examples:

```
>>X = [5 3 7 1]; % elements separated by space or comma: creates a 1x4 matrix
>>X = [5;3;7;1]; % elements separated by semicolon: creates a 4x1 matrix
>>X = [1 2 3; 4 5 6; 7 8 9]; % creates a 3x3 matrix
>>Z = X'; % Z is the transpose matrix of X
```

Besides, MATLAB provides several functions that let you create vectors/matrices with user-specified properties. Examples: `linspace`: evenly separated vectors with pre-defined separation, `eye`: identity matrix, `ones`: vector/matrix of all ones, `zeros`: vector/matrix of all zeros, `magic`: magic square matrix, `rand`: vector/matrix of uniformly distributed random values between 0 and 1, `randn`: vector/matrix of Gaussian distributed random values with mean 1 and variance 1. Note that you need to specify the size of the matrix you want to create (e.g. `randn(3,4)` would create a matrix of 3 rows and 4 columns). Refer to the MATLAB help to know more about these. Using these functions, matrices can also be built from blocks. For example, if `A` is a 3-by-3 matrix, then `B = [A, zeros(3,2); ones(2,3), eye(2)]` will build a certain 5-by-5 matrix. While creating matrices using blocks, you need to make sure that the dimensions of individual blocks are consistent with each other (so that the concatenation is actually possible). Otherwise, the operation will give an error.

MATLAB also provides a way to create a matrix by reading data from a file. For example, if we have a file named `data` containing some data (in row-column format), we can read it and store in a matrix.

```
>>X = load('data');
```

Important: If the data in the file is not stored in matrix format, you will need to do some extra processing to open and read it (using the `fopen` and `fscanf`/`fread` functions, for example). Refer to the MATLAB help for more information on these.

3.2 Accessing matrix elements

MATLAB follows a subscript notation to access matrix (or vector) entries. Some examples:

`X(i,j)` : element at i^{th} row, j^{th} column in a 2-D matrix
`X(i)` : i^{th} element of a 1-D vector

There are also special symbols and keywords helpful for accessing the matrix entries: ":" for an index position, in general, means the entire dimension (i.e. all elements) associated with that index. `end` keyword means the last element along a particular dimension. Some examples:

`X(i,:)` means ALL entries in i^{th} row. `X(i,end)` means the LAST entry in i^{th} row.
`X(:,j)` means ALL entries in j^{th} column. `X(end,j)` means the LAST entriy in j^{th} column.

The colon operator ":" can also be used to specify a range for matrix (or vector) entries. `X(r1:r2,j)` means all entries from row `r1` to `r2` in the jth column (using ":" in place of j, i.e. `X(r1:r2,:)`, would mean ALL entries from row `r1` to `r2`). On similar lines, think what `X(i,c1:c2)` and `X(:,c1:c2)` will do. Also, think what replacing `r2` (or `c2`) with `end` would do for the previous cases?

3.3 Operations on matrices

All standard mathematical operators work on matrices too. The important thing to remember, however, is that the sizes of operand matrices should be compatible. For example, you cannot add or subtract two matrices with mismatching dimensions. Also, you cannot multiply two matrices together unless the number of columns in the first matrix equals the number of rows in the second. Otherwise, MATLAB will show an error.

3.4 Important operations on vectors and matrices

Dot product of two equally sized vector: `X*Y'`
Dot product of a vector with itself: `X*X'` (same as the squared *norm*)
Length of a vector from origin (also called the *norm*): `norm(X)`
Distance between two equally sized vectors: `norm(X-Y)` (norm of the difference of two vectors gives their Euclidean distance)

A note on the inner (dot) and outer products: Note above the way how the dot product of a vector with itself (or of two different vectors) is calculated. Dot product of two vectors always yields a scalar number (it is basically component wise multiplication followed by a sum; that is why inner product of a vector with itself yields its squared distance from origin). Also, in MATLAB, dot product of two vectors `X` and `Y` can be computed as `X'*Y` or `X*Y'` depending on whether they are column or row vectors (if you do it the other way round, you will get something which is called an *outer* product which is *not* a scalar number but a matrix).

3.5 Important Built-in Functions

Listed below are some important built-in functions that operate on vectors and matrices. Note that the output may differ depending on whether they are operating on vectors or matrices.

Scalar Functions: Certain MATLAB functions operate essentially on scalars, but operate element-wise when applied to a matrix. The most common such functions are: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `exp`, `log`(natural log), `rem` (remainder), `abs`, `sqrt`, `sign`, `round`, `floor`, `ceil`

Vector Functions: Other MATLAB functions operate essentially on a vector (row or column), but act on an mxn matrix in a column-by-column fashion to produce a row vector containing the results of their application to each column. Row-by-row action can be obtained by using transpose. For example, mean(A')'. A few of these functions are: `max`, `sum`, `median`, `any`, `min`, `prod`, `mean`, `all`, `sort`, `std`

For example, the maximum entry in a matrix `A` is given by `max(max(A))` rather than `max(A)`. Similarly, the sum of all entries in a matrix `A` is given by `sum(sum(A))` rather than `sum(A)`.

Matrix Functions: Much of MATLAB's power comes from its matrix functions. The most useful ones are:

`eig`: eigenvalues and eigenvectors, `chol`: cholesky factorization, `svd`: singular value decomposition, `inv`: inverse, `lu`: LU factorization, `qr`: QR factorization, `hess`: hessenberg form, `schur`: schur decomposition, `rref`: reduced row echelon form, `expm`: matrix exponential, `sqrtm`: matrix square root, `poly`: characteristic polynomial, `det`: determinant, `size`: size, `norm`: 1-norm, 2-norm, F-norm, infinity-norm, `cond`: condition number in the 2-norm, `rank`: rank

Note: MATLAB functions may have single or multiple output arguments. For example, `y = eig(A)`, or simply `eig(A)` produces a column vector containing the eigenvalues of A while `[U,D] = eig(A)` produces a matrix U whose columns are the eigenvectors of A and a diagonal matrix D with the eigenvalues of A on its diagonal. Thus you must make sure that you capture all the outputs if you need them. When unsure, look up the MATLAB help to know more about the function input and output parameters.

Some other important functions:

Listed below are some other useful functions that are often used and you should make sure that you understand their usage.

`nargin`: Checks the number of input arguments to a program
`size`: Returns the size (i.e. the number of rows and columns) of a matrix
`length`: Returns the length of a vector
`randperm`: randperm(n) generates a random permutation of the integers from 1 to n
`find`: Searches for an element in a vector/matrix and returns the indices of matches found
`repmat`: Replicates a vector/matrix by a specified number of times along rows and columns. Example: Let Y be a 1xD (row) vector. Then `X = repmat(Y,N,1);` would replicate Y N times along the rows and 1 times (i.e. no effect) along the columns, returning an NxD matrix X.

3.6 A note on data representation in machine learning

Although it depends on the context, in many cases a NxD matrix `X` can be thought of as a set of N *examples*, each having D *features* (or *attributes*). The matrix `X` is usually called the data or feature matrix. Under this representation, each row (e.g. `X(i,:)`) is an example and the individual elements are the features of that example. To facilitate the understanding, let us also associate a vector `Y` of size Nx1 such that each entry (say y_i) of `Y` is +1 or -1. We can think of y_i as the *label* of example i (i.e. `X(i,:)`). Now we shall see some useful tricks that can allow us to manipulate or extract elements from the matrix `X`.

3.7 Logical expressions as indices for the matrix entries

A logical expression applied to a vector (or matrix) gives us a vector (or matrix) of 0s and 1s. The logical expression can consist of standard relational operators (>, <, >=, <=, ==) or may use some other built-in MATLAB functions (for example, `find`). We can use the result of such an expression to select specific entries from a matrix (and mask out the rest). Let us see some examples:

Extracting submatrices:
`>>Z = X(Y>0,:);` % Z is a matrix containing all the positive examples from X
`>>Z = X(X>0);` % Z is a *vector* containing all the positive *entries* from X

Replacing elements:
`>>X(X<0)=0;` % Replace all negative entries of X with 0

Some other examples: Suppose, we want to find the average Y value for all examples with feature number 5 having value greater than zero (recall that the convention assumes data to be in example × features format). The following MATLAB statement does this: `mean(Y(X(:,5)>0)`.

Suppose, we want to do some preprocessing of the data (given examples `X` and labels `Y`) and get rid of all examples that have all features as zero (assume non-zero features to be positive). This means the sum of all features for each such example would also be zero. The following statement does this:

`nz = sum(X,2)>0;` % creates a 0/1 mask
`X = X(nz,:);` % selects the desired examples
`Y = Y(nz);` % selects the corresponding labels

Depending on the situation, you may need to construct and use even more sophisticated logical expressions (involving other matlab built-in functions). The basic idea, however, remains the same.

Exercise:

1. Given examples `X` and labels `Y` (+1/-1), select only those examples that have positive labels.

4 Flow Control

Just like any other programming language, MATLAB provides several constructs (such as `for`, `while`, `switch`, etc) for flow control in programs. The usage are similar to other language such as C. There is a minor difference that each block using these constructs must be closed using an `end` keyword.

For loop: `x = []; for i =1:n, x=[x,i∧2], end`
While loop: `n = 0; while (2∧n <= a) n = n + 1; end`
If statement: `if n < 0 parity = 0; elseif rem(n,2) == 0 parity = 2; else parity = 1 end`

The statements above are written in a single line to save space. They can also be made to span several lines, if it enhances readability.

Flow control statements often also require relational and bitwise operators.
Relational operators: < less than, > greater than, <= less than or equal to, >= greater than or equal to, == equal, ∼= not equal
Bitwise operators: & and, | or, ∼ not

5 Probability Routines

While writing machine learning code based on probabilistic methods, you would often need to generate random numbers from various probability distributions. MATLAB provides built-in functions that let you do this. The most common operation is generating random numbers.

`rand`: Generates uniformly distributed (between 0 and 1) random numbers
`randn`: Generates Gaussian distributed random numbers with mean 0 and variance 1

You can specify size of the generated matrix by passing arguments to these functions. For example, `rand` or `rand(1,1)` generates a *single* random number between 0 and 1; `rand(1,n)` generates a `1xn` *vector* of random numbers bewteen 0 and 1; `rand(n)` or `rand(n,n)` both generate a `nxn` *matrix* of random numbers between 0 and 1. The `randn` function behaves similarly with these arguments but instead generates Gaussian distributed random numbers.

Exercise:

1. Using the `rand` function, how would you generate a uniformly distributed random number random number between 0 and 10? How would you generate a random *integer* between 1 and 10 (**Hint:** use `ceil` function)?
2. Using the `randn` function, how would you generate a Gaussian distributed random number with mean μ and variance σ^2 (**Hint:** A Gaussian random variable X with mean μ and variance σ^2 can be transformed into a Gaussian random variable Z with mean 0 and variance 1 by using $Z = (X - \mu)/\sigma$)?

Note: Depending on the toolboxes available on your MATLAB installation, there may be probability routines that allows you generate random numbers from other probability distributions (Binomial, Beta, Gamma, etc). Besides, there are other functions that let you compute several other quantities of interests such as the *probability density function* at any point. Refer to the MATLAB help for more on this.

Sampling vs computing probability density: There is often a confusion regarding generating/sampling a value from a distribution, and computing the density of some point according to that distribution. So I think an example would help here.

Consider the Binomial distribution (coin toss outcomes follow this distribution). Let us assume π to be the probability of a heads and x to be the outcome (which could be heads and tails). Now, in maths notation (not MATLAB), we typically write $\mathcal{B}in(x|\pi)$ to mean "x is distributed according to a binomial distribution with parameter π". This has density $\mathcal{B}in(x|\pi) = \pi^x(1-\pi)^{(1-x)}$.

On the other hand, we often also want to sample x from a binomial with parameter π, usually written "$x \sim \mathcal{B}in(\pi)$". In this case, x will be zero or one; it will be one with probability π.

So how do I go about doing these in MATLAB?: Let us assume π in the above example is 0.4.

To sample a random variable x distributed according to a Binomial:
```
>>x = binornd(0.4);
```

To compute the probability of the point x under the same Binomial distribution:
```
>>z = binopdf(x,0.4);
```

So you see that `matlab` provides simple functions to help to do these but you should be aware of the differences. Note: The above assumes that your MATLAB installation comes with the Statistics toolbox.

6 Importing and exporting data

MATLAB provides functions to load data from files and store it in some marix using the `load` function. You pass on the name of file containing data to be loaded from as the argument to the `load` function. Similarly, the `save` function lets you save a matrix to a file (file name is gives as an argument to the `save` function). For example, `X = load('filename');` loads the data from the file filename to an array X. `save('filename', 'X');` saves the array X from the current workspace (notice the quotes symbols around X) to the file `filename`.

The load and store functions have other usages as well. You can also save the variables in your current workspace to a file (the default name `matlab.mat` or some other name that you choose) by using the `save` command. The same variables can be retrieved by using the `load` command. For more, refer to MATLAB help.

7 Plotting

MATLAB provides a rich set of functions to draw 2-D as well as 3-D plots. Most of the plotting functions allow you to plot vectored values on various axes (all vectors must have the same dimensions) with plots representing their relationships. Examples of such functions are `plot` (for 2-D), `plot3` for 3-D), `meshgrid`/`surf`(used in conjunction for plotting 3-D surfaces). Let us see a simple example.

```
>>X = [-pi:pi/12:pi];
>>Y = sin(X);
>>plot(X,Y);
```

MATLAB also provides functions to label plots, controlling scales for axes, and also provides options to choose color/style etc for the plots. Refer to the MATLAB help for more on this. Also, if you just want to visualize how the plot of a function looks like, you can use the *easy* versions (`ezplot`, `ezsurf`, etc), that do not require you to pass vectored inputs. For example:

```
>>ezplot('sin(x)');
```

Some useful plotting functions: `figure`: Create Figure (graph window). `clf`: Clear current figure. `close`: Close figure. `subplot`: Create axes in tiled positions. `axis`: Control axis scaling and appearance. `hold`: Hold current graph. `figure`: Create figure window. `text`: Create text. `print`: Save graph to file. `plot`: Linear plot. `loglog`: Log-log scale plot. `semilogx`: Semi-log scale plot. `semilogy`: Semi-log scale plot.

Functions for plot annotation: `title`: Graph title. `xlabel`: X-axis label. `ylabel`: Y-axis label. `text`: Text annotation. `gtext`: Mouse placement of text. `grid`: Grid lines. `contour`: Contour plot. `mesh`: 3-D mesh surface. `surf`: 3-D shaded surface. `waterfall`: Waterfall plot. `view`: 3-D graph viewpoint specification. `zlabel`: Z-axis label for 3-D plots.

For more details on these, refer to MATLAB help.

8 Sparse Data Format

Sometimes, you come across matrices that are extremely large in size but have most of the entries as zero. Storing all the entries in such cases is wasteful. MATLAB provides an efficient storage alternative of sparse matrices which let you store only the non-zero entries. You can convert a full matrix to sparse (using the `sparse` keyword) and vice-versa (`full`). There are also variants of various standard MATLAB functions that let you directly create sparse matrices. Some important related functions are `issparse, spalloc, spones, speye, spconvert`.

9 Structures and Cell Arrays

Often, we want to store dissimilar kinds of data in some object. MATLAB offers structures and cell arrays to provide a hierarchical storage mechanism for such kinds of data.

Structures let us store different kinds of data organized by named fields. A structure has a name and it consists of fields that can take certain values. For example: `s = struct('strings',{'hello','yes'},'lengths',[5 3])` creates a structure with name `s` having two fields 'strings' and 'lengths'.

```
s =
strings:  {'hello' 'yes'}
lengths:  [5 3]
```

We access fields within a structure by simply using the field name. For example, `s.strings` or `s.length` for the previous example.

The cell array is a general purpose matrix. Each of the elements can contain data of a different type, size and dimension. Cell arrays are created using the `cell` command or by using curly braces, for example `cellArray{row,col} = data;`. Here is an example:

```
>>A= {rand(3,4,5), 'February', 10.28}
```

In cell arrays, data is accessed through matrix indexing operations. For example, it the above cell array A, `A{1}` means the 3x4x5 matrix, `A{2}` means `'February'` and `A{3}` means the value `10.28`.

For more on structures:
`www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/f2-88951.html`
For more information on cell arrays:
`www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/f2-67323.html`

10 Debugging

Often you want to monitor program behavior or inspect values of specific variables, on the fly. This can be done by setting breakpoints at specific locations in your code. For this purpose, MATLAB provides very useful debugging features that can be extremely handy when debugging complex code.

Graphical Mode Debugging: If you have opened your code in MATLAB's editor, you can just click on the 'Debug' tab and see all the keyboard shortcuts and ways to set/clear breakpoints. Also note that, in debugging mode, the MATLAB command prompt changes from `'>>'` to `'K>>'`. When the program has paused at a breakpoint, a green arrow appears in the editor window indicating the breakpoint location. During a breakpoint pause, you can inspect the current values of variables by simply moving the mouse or cursor over them

Command Line Debugging: You can also set/clear breakpoints at various locations from the command prompt itself. For example, the `dbstop` command can be used from command line to set breakpoints at specific locations in the program (e.g. line number, in the beginning of subfunctions, or even at exceptions such as when infinity or NaN values are encountered, etc). Try `help debug` for more information on various debug functions.

Finally, there are some other useful commands that turn out to be quite handy:

`keyboard`: The `keyboard` command can prove to be very useful while debugging code. When placed at some point in an M-file, it stops execution at that point and gives control to the keyboard. The command prompt also changes to `K>>`. At this prompt, you can examine or change variables, or use other MATLAB commands to manipulate variables. `return`: Typing the `return` command at `K>>` gets you out of the keyboard mode and resumes execution normally. `dbquit`: Typing the `dbquit` command also stops the debug mode but is different in the sense that the M-file is not processed any further and no results are returned.

11 Putting it all together with some examples

Nearest Neighbor Classification: Assume we have an NxD data set `X` (N *examples* having D *features* each), an Nx1 vector of labels `Y` (one for each example), and a 1xD test example `T`. We want to assign the label to the new example `T`. The nearest neighbor principle tells us that its label should be the same as the label of the example most similar to it (i.e. the nearest/closest example). The Euclidean distance between two D-dimensional vectors X_1 and X_2 is given by: $L = \sqrt{\sum_{d=1}^{D}(X_1(d) - X_2(d))^2}$. With this in mind, let us now see how we can use the nearest neighbor principle to find the label for `T`, using MATLAB:

```
function [label] = nnclass(X,Y,T)

[N,D]=size(X);
Xdist=X-repmat(T,N,1);
Xdist2=sum(Xdist.∧2,2);
[minX,nn]=min(Xdist2);
label = Y(nn);
```

Line 1: `[N,D]=size(X);` computes the size of matrix X.
Line 2: : Computes the element-wise difference of the test example with all other examples and stores the result in `Xdist`. The `repmat` function replicates the example `T` N many times along the rows since we want to compute its distance from *all* N examples.
Line 3: Computes the actual squares distances of T from each of the examples from X
Line 4: Uses the `min` function to find the example nearest to the test example and its index.
Line 5: Finds the label for the test example.

Thus `X(nn,:)` is the nearest neighbor for our test example `T`. Its label is `Y(nn)`.

Principal Component Analysis: Assume we have an NxD dataset X. Often we want to reduce the dimension D of the data to a smaller number (say K) in order to have a compact representation. Essentially, we want to *project* the data onto a lower dimensional space. This is an eigenvalue decomposition problem. Let us see how we can do it in MATLAB. The code is reasonably self-explanatory.

```
function [Z,vecs,vals] = PCA(X,K)

[N D] = size(X);

if K > D,
error('PCA: you are trying to *increase* the dimension!');
end;

mu = mean(X);
X = X - repmat(mu,N,1); % center the data (i.e. subtract off the mean)

C = cov(X); % compute the covariance matrix of the centered data

% eigenvalue decomposition of C
[vecs,vals] = eigs(C, K); % choose the top K eigenvalues and vectors
vals = diag(vals); % vals are the eigenvalues associated with the dimensions

Z = X*vecs; % Z is the projected data
```

12 References

To explore further, here are some links that you may find useful.

An Introduction to MATLAB:
`http://www.maths.dundee.ac.uk/~ftp/na-reports/MatlabNotes.pdf`

MATLAB quick reference:
`http://www.math.umd.edu/~jeo/matlab_quickref.pdf`

MATLAB array manipulation tips and tricks:
`http://home.online.no/~pjacklam/matlab/doc/mtt/doc/mtt.pdf`

Analyzing and Improving Performance of your MATLAB code:
`http://www.mathworks.com/access/helpdesk/help/techdoc/index.html?/access/helpdesk/help/techdoc/matlab_prog/f8-705906.html`

Some memory related issues:
`http://www.caspur.it/risorse/softappl/doc/matlab_help/techdoc/matlab_prog/ch8_pr17.html#31126`

The MATLAB Debugger:
`www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/f10-60570.html`

MATLAB FAQs:
`http://www.mit.edu/~pwb/cssm/matlab-faq.html`