MATLAB language skills and debugging
Aarthi Sridhar, Ph.D., Jenni Rinker, Ph.D., Kim Bourne, Henri P. Gavin, Ph.D.
November 16, 2022

1 Orientation

MATLAB and its open-source derivative Octave provide a computing environment and a programming language that is efficient at calculations involving vectors and matrices. As a computing environment, Matlab / Octave allows users to interactively carry out calculations by manually entering code or calculations in the Command Window. The Command Window looks like a terminal window with a >> prompt. As a programming language, Matlab / Octave allows users to write algorithms into scripts or functions saved as .m-files. Users may conveniently see the dimensions and values of variables stored in the Workspace Window.

MATLAB code looks like matrix equations. For example, if A and x are vectors or matrices of compatible dimensions, they can be multiplied just like this ... b = A*x. This and other features of the MATLAB language present opportunities for writing terse code, and often eliminate the need for writing for-loops, especially when computing with matrices.

2 A matlab webinar

This video https://www.youtube.com/watch?v=T_ekAD7U-wU is more than four hours long. Refer to the indexed Time Stamps in the description. For general programming consider watching the first 1:55, and for people interested in probability and statistics, 3:16 - 4:03

3 Useful tips

1. **Comment your code**: Start one-line comments with % or ... Text and code following the % or ... and going to the end of the line is not executed. It’s fine and good to comment every line.

   Blocks of text or code starting with %{ as the first characters of the line at the top of the block and ending with %} as the first characters of the line at the bottom of the block are not executed.

2. **Suppress output**: End every line with a ; to keep intermediate results from being displayed to the MATLAB Command Window. To display the value of any variable of interest to the Command Window, remove the ; from the end of the line.

3. **Abort**: Hit CTRL-C in the command window to stop a calculation for any reason.

4. **Re-run a previous command**: Scroll through your previous command-line inputs by selecting the Command Window and hitting the up arrow and the down arrow on your keyboard.
5. **Clear Command Window**: Type `clc` into the Command Window.

6. **Clear variables from the Workspace**: Type `clear <varname1> <varname2> ...` to clear certain variables from the workspace, and the memory allocated to them. Doing so, you will see these variables disappear from the Workspace Window. Clear all variables by typing `clear all`.

7. **Break up long lines of code in your .m-files**: If a line of code is longer than 80 characters (one page width of text), you may continue an expression onto subsequent lines by ending by the line with `(...). This is not a strict rule, and there are some instances in which a line of code should be very long. Note that ellipsis for line continuation may not be followed by any blank space.

   An ellipses is unnecessary with matrices; you may enter matrix row by row on different lines like this ...

   ```
   M = [ 1 2 3 ;
            4 5 6 ];
   ```

4. **Assigning values to scalar, vector, matrix, and cell variables**

   Use square brackets `[ ]` to begin and end vectors and matrices. Commas or blank space separate elements in a single row. Semicolons or new-lines separate rows.

   1. Scalar: `a = 4`

   2. Row vector: `r = [ 1 2 3 ]` or `r = [ 1 , 2 , 3 ]` or `r = [ 1 ; 2 ; 3 ]`

   3. Column vector: `c = [ 1 2 3 ]'` or `c = [ 1 ; 2 ; 3 ]` or `c = [ 1 , 2 , 3 ]`

   4. Matrix: `M = [ 1 3 5 ; 2 4 6 ]`

   5. Cell Array: (a set of dissimilar quantities, such as a set including character strings and numerical quantities)

   ```
   S = { 'hello' , [ 3 5 ; 7 11 ] , [ 2 4 6 8 ] , 'appreciate' }
   S{5} = 'today'
   ```

   6. Dimensions of variables are shown in the Workspace Window or can be found by Command Window commands:

   ```
   [nr,nc] = size(X) ... nr = size(X,1) ... nc = size(X,2) ... n = length(x)
   ```

5. **Here are just some of the many built-in math functions**

   ```
   sin(r) , cos(r) , tan(r) , cot(r) ... r in radians
   sind(d) , cosd(d) , tand(d) , cotd(d) ... d in degrees
   asin(y) , acos(y) , atan(y) , acot(y) ... result in radians
   asind(y) , acosd(y) , atand(y) , acotd(y) ... result in degrees
   exp(x) , expm(X) , log(y) , log10(y) , log2(y) , rank(A) , eig(A) , svd(A) , qr(A)
   ```
6 Shortcuts to assign values to vectors and matrices

1. **Define vectors of uniformly spaced values using the colon:**
   \[ v = \begin{bmatrix} \text{start} : \text{increment} : \text{end} \end{bmatrix} \]
   represents a row vector from \text{start} up to and including \text{end} with \text{increment} between sequential values.
   - For example, \( v = [3 : 3 : 24] \) represents \( v = [3, 6, 9, 12, 15, 18, 21, 24] \) and \( v = [2 : 10] \) represents \( v = [2, 3, 4, 5, 6, 7, 8, 9, 10] \) and \( v = [10 : -1 : 2] \) represents \( v = [10, 9, 8, 7, 6, 5, 4, 3, 2] \)

2. **Define vectors using linspace and logspace**
   \[ v = \text{linspace}(\text{start}, \text{end}, N) \]
   represents a row vector of \( N \) values from \text{start} up to and including \text{end} with a uniform increment between values.
   \[ v = \text{logspace}(\text{start}, \text{end}, N) \]
   represents a row vector of \( N \) values from \( 10^{\text{start}} \) up to and including \( 10^{\text{end}} \) with log-increment between values.
   - For example, \( v = \text{linspace}(1, 4, 5) \)
     represents the row vector \( v = [1.00, 1.75, 2.50, 3.25, 4.00] \)
   - and \( v = \text{logspace}(-1, 1, 3) \)
     represents the row vector \( v = [0.1, 1.0, 10.0] \)

3. **Common Values:**
   Common Values like \( \pi \), \( e \), and \( i \) are provided by default, so there is normally no need to write \( \text{pi} = 3.14159 \).

4. **Vector or Matrix of zeros:**
   \( \text{zeros}(m,n) \)
   creates an \( m \)-by-\( n \) matrix of zeros

5. **Vector or matrix of ones:**
   \( \text{ones}(m,n) \)
   creates an \( m \)-by-\( n \) matrix of ones

6. **Identity matrix:**
   \( \text{I} = \text{eye}(n) \)
   creates an \( n \)-by-\( n \) identity matrix.
   The identity matrix is all zeros except for the main diagonal which has values of 1.

7. **Diagonal matrix:**
   \( D = \text{diag}(v) \)
   creates a diagonal matrix whose diagonal elements are the elements of the vector \( v \)
   \( D = \text{diag}(v,1) \)
   creates a diagonal matrix whose first diagonal above the main diagonal are the elements of the vector \( v \)
   \( D = \text{diag}(v,-2) \)
   creates a diagonal matrix whose second diagonal below the main diagonal are the elements of the vector \( v \)

8. **Matrices of Random Values:**
   \( M = \text{rand}(m,n) \)
   creates a \( m \)-by-\( n \) matrix of random values uniformly distributed between 0 and 1.
   \( M = \text{randn}(m,n) \)
   creates a \( m \)-by-\( n \) matrix of random values normally distributed with mean of 0 and standard deviation of 1.
   \( M = \text{randi}(\text{min}, \text{max}, m, n) \)
   creates a \( m \)-by-\( n \) matrix of random integers uniformly distributed between \( \text{min} \) and \( \text{max} \).
9. **Assigning values to other kinds of matrices:**
   Other built-in shortcuts used to assign values to certain kinds of matrices include: `tril`, `triu`, `pascal`, `magic`, `hadamard`, `compan`, `vander`, `toeplitz`, and `hankel`.
   You can read about these, and any other function, by typing (for example) `help tril` at the Command Window prompt >>

7 **Vector and Matrix operations**

1. **Select elements from vector:**
   - `v(i)` selects the *i*th entry of *v*
   - `v(idx)`, where *idx* is a vector of integers, selects the elements of *v* whose indices correspond to *idx*. For example, if *v* = [5 1 3 7] and *idx* = [1 3 4] then `v(idx)` represents [5 3 7]

2. **Select to end of vector:**
   - `v(i:end)` selects the *i*th entry to the last entry of *v*.
   - `v(i:end-1)` selects up to the second-to-last entry of *v*.
   This works for any number up to the length of the vector.

3. **Select elements from matrix:**
   This follows the same principle as vectors, but indices for row and column are separated by a comma.
   - `M(i,j)` is the scalar in the *i*th row and *j*th column of matrix *M*.
   - `M(i,:)` represents the vector in the *i*th row of matrix *M*.
   - `M(:,j)` represents the vector in the *j*th column of matrix *M*.
   If `idc = [3:2:9]` then `M(:,idc)` represents the odd column of matrix *M* from column 3 to column 9.
   If `idr = [2:2:4]` and `idc = [3:2:9]` then `M(idr,idc)` represents the values of *M* in the even rows from row 2 to row 4 and the odd columns from column 3 to column 9.

4. **Transpose:**
   - If *v* is a row vector, *v'* is a column vector. If *v* is a column vector, *v'* is a row vector.
   - If *M* is a *m*-by-*n* matrix then *M'* represents a *n*-by-*m* matrix.
   - Row *i* of *M'* is the transpose of column *i* of *M*.
   If some values in *v* or *M* are complex, the transpose can also involve taking the complex-conjugate of the values in *v* or *M*. If *v* is a complex row (column) vector, then *v'* represents the column (row) vector containing the complex conjugates of the elements of *v*. If *v* is a complex row (column) vector, then *v.'* represents the the column (row) vector with the same elements of *v*.
   If *v* and *M* are real, the transposes *v'* and *v.'* are equivalent, and the transposes *M* and *M.'* are equivalent.

5. **Tiling Matrices:**
   - If `A = zeros(3); B = ones(3,4);` and `D = randn(4);` then `M = [A, B ; -B', D]`
is a 7-by-7 matrix in which the first three rows of the first three columns are all zero, the first three rows of columns 4 through 7 are all ones, the last four rows of the first three columns are all negative ones, and the last four rows of the last four columns are random.

\[
M = \begin{bmatrix}
0.00 & 0.00 & 0.00 & 1.00 & 1.00 & 1.00 & 1.00 \\
0.00 & 0.00 & 0.00 & 1.00 & 1.00 & 1.00 & 1.00 \\
0.00 & 0.00 & 0.00 & 1.00 & 1.00 & 1.00 & 1.00 \\
-1.00 & -1.00 & -1.00 & -0.21 & 2.10 & 0.89 & -0.18 \\
-1.00 & -1.00 & -1.00 & 0.78 & -0.13 & -0.90 & -0.17 \\
-1.00 & -1.00 & -1.00 & 1.51 & -1.10 & -0.68 & 0.65 \\
-1.00 & -1.00 & -1.00 & -0.29 & -0.11 & -1.57 & -1.08
\end{bmatrix}
\]

6. Addition, Subtraction, Multiplication, and Division of a matrix (or a vector) with a scalar:

If \( M \) is a matrix and \( s \) is a scalar, then:
- \( A = M + s \) is a matrix of the sum of each element of \( M \) and \( s \) ... \( A(i,j) = M(i,j) + s \)
- \( A = M - s \) is a matrix of the difference of each element of \( M \) and \( s \) ... \( A(i,j) = M(i,j) - s \)
- \( A = M \times s \) is a matrix of the product of each element of \( M \) and \( s \) ... \( A(i,j) = M(i,j) \times s \)
- \( A = M / s \) is a matrix of the ratio of each element of \( M \) over \( s \) ... \( A(i,j) = M(i,j) / s \)

7. Vector Addition and Subtraction:

If \( u \) and \( v \) are vectors the same dimension (both row vectors or both column vectors, and of the same length) then \( a = u + v \) represents a vector of the sum of each element of \( u \) with the corresponding element of \( v \), and has the same dimensions as \( u \) and \( v \)

\[
a_i = u_i + v_i
\]

If \( u \) and \( v \) are both column vectors (or both row vectors) of different dimensions, they cannot be added or subtracted, and MATLAB will return the error:

**Matrix dimensions must agree.**

This is a very common error in MATLAB programming, and it’s useful to know what it means. To fix matrix dimension errors, check the dimensions of the involved variables, either by looking at the **Workspace Window** or entering the line \([\text{n\_row,n\_col}] = \text{size}(X)\) (without a semi-colon ‘;’) in the **Command Window** or in your .m-file.

If \( u \) is a column vector of length \( m \) and \( v \) is a row vector of length \( n \), then \( M = u + v \) represents a matrix of dimension \( m \)-by-\( n \), where

\[
M_{ij} = u_i + v_j
\]

8. Matrix Addition and Subtraction:

If \( P \) and \( Q \) are both \( m \)-by-\( n \) matrices then \( M = P + Q \) is also a \( m \)-by-\( n \) matrix where

\[
M_{ij} = P_{ij} + Q_{ij}
\]

If \( P \) and \( Q \) are matrices of different dimension, they cannot be added, and MATLAB will return the error: **Matrix dimensions must agree.**
9. **Vector Multiplication:**
In MATLAB there are four kinds of vector multiplication: *, *, .* , and \texttt{cross}.

If \( u \) and \( v \) are both column vectors of length \( n \), then \( w = u' \ast v \) is a scalar-valued \textit{inner product} (or \textit{dot product})
\[
w = \sum_{i=1}^{n} u_i v_i
\]

If \( u \) is a column vector of dimension \( m \)-by-1 and \( v \) is a column vector of dimension \( n \)-by-1, then \( M = u \ast v' \) is a matrix-valued \textit{outer product}
\[
M_{ij} = u_i v_j
\]

The matrix \( M \) has dimension \( m \)-by-\( n \).

If \( u \) and \( v \) are vectors of the same dimension (\( 1 \)-by-\( n \) or \( n \)-by-\( 1 \)), then \( w = u \ast v \) is a vector of the same dimension with elements
\[
w_i = u_i v_i
\]

If \( u \) and \( v \) are both vectors of dimension 3, then \texttt{cross} \((u,v)\) is the \textit{cross product} of \( u \) and \( v \) defined by the determinant
\[
u \times v = \det \begin{pmatrix}
\vec{i} & \vec{j} & \vec{k} \\
u_1 & u_2 & u_3 \\
v_1 & v_2 & v_3
\end{pmatrix}
\]

10. **Matrix Multiplication:**
In the MATLAB language there are three kinds of matrix multiplication: *, *, and .*

If \( P \) has dimension \( p \)-by-\( q \) and \( Q \) has dimension \( q \)-by-\( r \) then \( M = P \ast Q \) is the \textit{inner product} (or \textit{dot product}) ans is a \( p \)-by-\( r \) matrix.
\[
M_{ij} = \sum_{k=1}^{q} P_{ik} Q_{kj}
\]

Note that the \textit{inner dimension} of the two matrices being multiplied must be equal.

If \( P \) has dimension \( p \)-by-\( q \) and \( Q \) has dimension \( q \)-by-\( r \) and \( p \) is not the same as \( r \), then \( M = Q \ast P \) can not be computed since the inner dimensions do not agree, and MATLAB will display the error message: \texttt{Matrix dimensions must agree}.

If \( u \) is a column vector of dimension \( m \)-by-1 and \( v \) is a column vector of dimension \( n \)-by-1, then \( M = u \ast v' \) is a matrix-valued \textit{outer product}
\[
M_{ij} = u_i v_j
\]

The matrix \( M \) has dimension \( m \)-by-\( n \).

If \( P \) and \( Q \) are both \( m \)-by-\( n \) matrices then \( M = P \ast Q \) is also a \( m \)-by-\( n \) matrix where
\[
M_{ij} = P_{ij} Q_{ij}
\]
Matlab language skills and debugging

8. `<`, `>`, `<=`, `>=`, `==`, `~=`

1. The conditions `<`, `>`, `<=`, `>=`, `==`, `~=` return a 1 if the condition is true and return a 0 otherwise.

Examples with vectors

- For `a = [ 1 : 3 : 11 ]` and `b = [ 3 : 2 : 10 ]`;
- `x = a < 5;` evaluates to `x = [ 1 1 0 0 ]`
- `x = a == 7;` evaluates to `x = [ 0 0 1 0 ]`
- `x = a ~= 5;` evaluates to `x = [ 1 1 1 1 ]`
- `x = a <= 0;` evaluates to `x = [ 0 0 0 0 ]`
- `x = a <= b;` evaluates to `x = [ 1 1 1 0 ]`
- `x = a == b;` evaluates to `x = [ 0 0 1 0 ]`

Examples with matrices

For `A = a' * b` and `B = pascal(4)`
- `X = A < 5;` evaluates to
  
  \[
  X = \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  \end{bmatrix}
  \]

- `X = A == 7;` evaluates to
  
  \[
  X = \begin{bmatrix}
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  \end{bmatrix}
  \]

- `X = A ~= 5;` evaluates to
  
  \[
  X = \begin{bmatrix}
  1 & 0 & 1 & 1 \\
  1 & 1 & 1 & 1 \\
  1 & 1 & 1 & 1 \\
  1 & 1 & 1 & 1 \\
  \end{bmatrix}
  \]

- `X = A <= 0;` evaluates to
  
  \[
  X = \begin{bmatrix}
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  \end{bmatrix}
  \]

- `X = A <= 9 * B;` evaluates to
  
  \[
  X = \begin{bmatrix}
  1 & 1 & 1 & 1 \\
  0 & 0 & 0 & 1 \\
  0 & 0 & 1 & 1 \\
  0 & 0 & 1 & 1 \\
  \end{bmatrix}
  \]
2. \( q = \text{any}(x); \) returns 1 if any element of \( x \) is non-zero, 0 otherwise.

3. \( q = \text{all}(x); \) returns 1 if all elements of \( x \) are non-zero, 0 otherwise.

4. \( \text{idx} = \text{find}(x); \) finds the indices of vector \( x \) that are non-zero.
   
   \[ \text{[IDr,IDc]} = \text{find}(X); \] finds the row,column indices of matrix \( X \) that are non-zero.

   Examples with vectors ... for \( a = [1 : 3 : 11]; \) and \( b = [3 : 2 : 10]; \)
   
   \( \text{idx} = \text{find}(a < 5); \) evaluates to \( \text{idx} = [1 2] \)
   
   \( \text{idx} = \text{find}(a == 7); \) evaluates to \( \text{idx} = 3 \)
   
   \( \text{idx} = \text{find}(a ~= 5); \) evaluates to \( \text{idx} = [1 2 3 4] \)
   
   \( \text{idx} = \text{find}(a <= 0); \) evaluates to \( \text{idx} = [0x1] \)
   
   \( \text{idx} = \text{find}(a == b); \) evaluates to \( \text{idx} = [1 2 3] \)
   
   \( \text{idx} = \text{find}(a == b); \) evaluates to \( \text{idx} = 3 \)

   Examples with matrices ... for \( A = a'*b \) and \( B = \text{pascal}(4) \)
   
   \[ \text{[IDr,IDc]} = \text{find}(A < 5); \] evaluates to \( \text{IDr} = 1 \) and \( \text{IDc} = 1 \)
   
   \[ \text{[IDr,IDc]} = \text{find}(A == 7); \] evaluates to \( \text{IDr} = 1 \) and \( \text{IDc} = 3 \)
   
   \[ \text{[IDr,IDc]} = \text{find}(A ~= 5); \] evaluates to
   
   \( \text{IDr} = [1 2 3 4 1 2 3 4 1 2 3 4] \)
   
   \( \text{IDc} = [1 1 1 2 2 3 3 3 4 4 4 4] \)
   
   \[ \text{[IDr,IDc]} = \text{find}(A <= 0); \] evaluates to \( \text{IDr} = [0x1] \) and \( \text{IDc} = [0x1] \)
   
   \[ \text{[IDr,IDc]} = \text{find}(A <= 9*B); \] evaluates to
   
   \( \text{IDr} = [1 1 1 3 4 1 2 3 4] \)
   
   \( \text{IDc} = [1 2 3 3 4 4 4 4] \)

5. Find the indices of the elements of vector \( u \) that are greater than \( u\_\text{limit} \)

   \[ \text{idx}_u = \text{find}(u > u\_\text{limit}); \]

   Find the fraction of elements in the vector \( u \) that are greater than \( u\_\text{limit} \)

   \[ P = \text{sum}(u > u\_\text{limit}) / \text{length}(u); \]

6. Use \text{find} to confirm mathematical equalities, for example for any square matrix \( A \)

   \( A = \text{rand}(5); \)
   
   \( \text{find}(A^2 - A*A) \) evaluates to \( [0x1] \)

9 Solving Linear Equations

Given a full rank matrix \( A \) and a vector \( b \) in which \( A \) and \( b \) have the same number of rows, solve \( Ax = b \) for \( x \). \( x = A \ld \ b \). This is called a matrix left divide and is conceptually the same as \( x = A^{-1}b \), or in the MATLAB language, \( x = \text{inv}(A) * b \)
10  **&&**,  **||**,  **&**,  **|**  (with scalars and vectors)

1. **&&** is used as **AND** to combine logical statements.
   For example, for scalars $a, b, c, d$
   
   $x = (a < b) \&\& (c < d)$
   
   evaluates to
   
   $x = 1$
   
   if both $a$ is less than $b$ **AND** $c$ is less than $d$.

2. **||** is used as **OR** to combine logical statements.
   For example
   
   $x = (a < b) \lor (c < d)$
   
   evaluates to
   
   $x = 1$
   
   if either $a$ is less than $b$ **OR** $c$ is less than $d$.

3. **&** is used as a bitwise **AND** to combine vector-valued logical statements.
   For example for vectors of equal length $a, b, c, d$
   
   $x = (a < b) \& (c < d)$
   
   evaluates to a vector with each element corresponding to the elements of
   
   $(a < b)$ **AND** $(c < d)$.
   
   If $a = [1\ 2\ 3\ 4]$; $b = [4\ 3\ 2\ 1]$; $c = [4\ 1\ 2\ 3]$; and $d = [1\ 2\ 3\ 4]$;
   
   $(a < b)$ evaluates to $[1\ 1\ 0\ 0]$ and $(c < d)$ evaluates to $[0\ 1\ 1\ 0]$ so
   
   $x = (a < b) \& (c < d)$ evaluates to $x = [0\ 1\ 0\ 0]$

4. **|** is used as a bitwise **OR** to combine vector-valued logical statements.
   For example for vectors of equal length $a, b, c, d$
   
   $x = (a < b) \lor (c < d)$
   
   evaluates to a vector with each element corresponding to the elements of
   
   $(a < b)$ **OR** $(c < d)$.
   
   With the same vectors in the previous example ...
   
   $(a < b)$ evaluates to $[1\ 1\ 0\ 0]$ and $(c < d)$ evaluates to $[0\ 1\ 1\ 0]$ so
   
   $x = (a < b) \lor (c < d)$ evaluates to $x = [1\ 1\ 1\ 0]$
max, min, sort, sum, diff (with vectors and matrices)

1. \[ [a_{\text{max}}, \text{idx}] = \text{max}(x) \] ... the most positive value of vector \( x \) and its index
\[ [A_{\text{max}}, \text{IDr}, \text{IDc}] = \text{max}(X) \] ... the max value of matrix \( X \) and its row, column index

2. \[ [a_{\text{min}}, \text{idx}] = \text{min}(x) \] ... the most negative value of vector \( x \) and its index
\[ [A_{\text{min}}, \text{IDr}, \text{IDc}] = \text{min}(X) \] ... the min value of matrix \( X \) and its row, column index

3. \[ [a_{\text{sort}}, \text{idx}] = \text{sort}(x) \]
... sorted values of vector \( x \) in numerically increasing order and the sorted index
\[ [A_{\text{sort}}, \text{IDX}] = \text{sort}(X) \]
... sorted columns of matrix \( X \) and the sorted index of each column

Example with a vector ... for \( a = [ 5 \ 7 \ 5 \ 3 \ 9 \ 1 \ 4 ] \);
\[ [\text{sort}_a, \text{idx}] = \text{sort}(a) \] evaluates to
\[ \text{sort}_a = [ 1 \ 3 \ 4 \ 5 \ 5 \ 7 \ 9 ] \]
\[ \text{idx} = [ 6 \ 4 \ 7 \ 1 \ 3 \ 2 \ 5 ] \]
\[ \text{find}(a(\text{idx}) - \text{sort}_a) \] evaluates to \([()](1x0)\)

4. \[ \text{sum}_x = \text{sum}(x) \] ... the sum of the elements of the vector \( x \)
\[ \text{sum}_A = \text{sum}(X) \] ... the sum down the columns of matrix \( X \)
\[ \text{sum}_A = \text{sum}(X,2) \] ... the sum across the rows of matrix \( X \)

Example with a matrix ... for \( M = \text{magic}(5) \);

\[
M = \begin{bmatrix}
 17 & 24 & 1 & 8 & 15 \\
 23 & 5 & 7 & 14 & 16 \\
 4 & 6 & 13 & 20 & 22 \\
 10 & 12 & 19 & 21 & 3 \\
 11 & 18 & 25 & 2 & 9 \\
\end{bmatrix}
\]

\[ \text{sc} = \text{sum}(M) \] evaluates to \[ \text{sc} = [ 65 \ 65 \ 65 \ 65 \ 65 ] \]
\[ \text{sc} = \text{sum}(M,2) \] evaluates to

\[ \text{sc} = [ 65 \\
 65 \\
 65 \\
 65 \\
 65 ] \]

5. \[ \text{diff}_x = \text{diff}(x) \] ... the differences between adjacent elements of the vector \( x \)
\[ \text{diff}_A = \text{diff}(X) \] ... the adjacent differences down the columns of matrix \( X \)
\[ \text{diff}_A = \text{diff}(X,2) \] ... the adjacent differences across the rows of matrix \( X \)

Example with a vector... for \( a = [ 5 \ 7 \ 5 \ 3 \ 9 \ 1 \ 4 ] \)
\[ \text{da} = \text{diff}(a) \]; evaluates to \[ \text{da} = [ 2 \ -2 \ -2 \ 6 \ -8 \ 3 ] \]
12 Scripts and functions:

To perform lots of operations in a single go, write your operations into a .m-file as scripts or functions. With scripts, numerical values for all variables are already in the Workspace or are provided within the script. With functions, none of the internal variables are stored in the Workspace. Basically, scripts are an assembly of lines you would type into the Command Window, whereas functions compute a set of zero or more output results using a set of zero or more input values. To define a function in a .m-file, begin the function with a line like:

```
function [outputA , outputB ] = functionName( inputA , inputB )
```

Save the function into a .m-file named functionName.m ... the same as the function named at the beginning of the file. The name of the .m-file must match the name of the main function in the file. To call the function (either in the Command Window or within other functions) enter the line of code following the function keyword in the function's .m-file. For example to call the example function above, type the following lines (either in a different function, a script, or on the command line)

```
inputA = 7;
inputB = 8;
[ outputA , outputB ] = functionName(inputA , inputB);
```

More than one function can be defined in a single .m-file. Other functions defined in the .m-file can only be helper functions to the first function included in functionName.m and are not accessible outside of the .m-file. For example, assume I have an .m-file mainFunction.m with functions mainFunction, helper1, helper2, etc. If I type mainFunction into the Command Window, then function mainFunction will run and can make use of functions helper1 and helper2 written within the .m-file mainFunction.m. However, if I type helper1 on the command line I will get an error stating that the function helper1 is undefined, since the Command Window cannot see helper1 inside mainFunction.m.

As you write, download and use more and more .m-files, you will develop a library of .m-files that may be of use in a variety of contexts. It is convenient to store all your general purpose .m-files in one place where MATLAB knows to look for them. Here's how to do it:

1. Create a directory (aka “folder”) on your computer called something like my-m-files.
2. Move and/or save your general purpose .m-files to that directory (aka “folder”).
3. With the MATLAB IDE open, click on the Set Path icon, near the middle at the top of the screen.
4. Click Add Folder and navigate to your “my-m-files” folder
5. Click Open. Click Save.
6. When asked Do you want to save pathdef.m to another location? click Yes, click Close.

.... aaaand Bob’s your uncle.

You may wish to save .m-files for a particular project elsewhere, like in a separate myProject directory (aka “folder”). That’s fine and good. When you start MATLAB just click the Browse for folder icon on the upper left of your screen and navigate to your myProject directory (aka “folder”). Then you can work with the .m-files in myProject while also having access to the .m-files in my-m-files.
13 if else

```matlab
if (condition is true)
    (run this code)
else
    (run alternative code)
end  % close if–else with end
```

14 while loops

```matlab
while (condition is true)
    (run this code)
end  % close while loops with end
```

For example, using nested while-loops to evaluate

\[ S(n) = \frac{\sum_{k=1}^{n} (-1)^{k-1}/(2k - 1)}{2} \]

for three values of \( n = [5, 10, 15] \);

```matlab
n = [5, 10, 15];
S = zeros(1, length(n));
i = 1;
while (i <= length(n))
    k = 1;
    while (k <= n(i))
        % accumulate the sum for the i-th value of vector n and summing over values of k
        S(i) = S(i) + (-1)^(k-1) / (2*k - 1)
        k = k+1;
    end
    i = i+1;
end
```

Using a while-loop, the index variables (\( i \) and \( k \)) need to be initialized and incremented using additional lines of code. The command break exits a loop before the end condition (e.g., \( i \) equals \( n \)) is met.

15 for loops

```matlab
for i = 1 : n
    (run this code)
end  % close for loops with end
```

For example, using nested for-loops to evaluate the same sum as above

```matlab
n = [5, 10, 15];
S = zeros(1, length(n));
for i = 1 : length(n)  % loop over the values in the vector n
    for k = 1 : n(i)  % what does n(i) represent?
        % accumulate the sum for the i-th value of vector n and summing over values of k
        S(i) = S(i) + (-1)^(k-1) / (2*k - 1)
    end
end
```
16 Loops should be avoided whenever possible

1. Evaluate $M = P \cdot Q$ where $P$ is a $p \times n$ matrix and $Q$ is a $n \times q$ matrix.

**Don’t do this:**

```matlab
for i = 1 : p
    for j = 1 : q
        M(i,j) = 0;
        for k = 1 : n
            M(i,j) = M(i,j) + P(i,k) * Q(k,j);
        end
    end
end
```

**Just do this:**

```matlab
A = P * Q;
```

2. Evaluate $y_i = \sin(\pi x_i)$ for $x = [0 \ 0.01 \ 0.02 \ \ldots \ 10]$

**Don’t do this:**

```matlab
x = [0 : 0.01 : 10];
N = length(x);
for i = 1 : N
    y(i) = sin(pi*x(i));
end
```

**Just do this:**

```matlab
dx=0.01; N=10/dx; x=[0:N]*dx; % all values of x in a vector
y = sin(pi*x); % evaluate in one line without a loop
```

3. Evaluate $y_i = \sum_{j=1}^{5} a_j \cos(\pi (2j - 1) x_i)$

for $a = [5 \ -4 \ 3 \ -2 \ 1]$ and $x = [0 \ 0.01 \ 0.02 \ \ldots \ 10]$

**Don’t do this:**

```matlab
a = [5, -4, 3, -2, 1];
x = [0 : 0.01 : 10];
M = length(a);
N = length(x);
y = zeros(1,N);
for i = 1 : N
    for j = 1 : M
        y(i) = y(i) + a(j) * cos(pi*(2*j-1)*x(i));
    end
end
```

**Just do this:**

```matlab
% coefficient values, ROW vector
a = [5, -4, 3, -2, 1];
% all values of x in a ROW vector
dx=0.01; N=10/dx; x=[0:N]*dx;
% odd integers 1, 3, 5, 7, 9 COL vector
j_odd = [1 : 2 : 9]';
y = a * cos(pi*j_odd*x);
% evaluate in one line without a loop
y = a * cos(pi*j_odd*x);
```
Note:

- $\pi j_{\text{odd}} x$ is a matrix with 5 rows and 1001 columns,
- $\cos(\pi j_{\text{odd}} x)$ is also a matrix with 5 rows and 1001 columns, and
- $a \cdot \cos(\pi j_{\text{odd}} x)$ is a vector-matrix inner product resulting in a row vector with 1001 columns.

17 2-D and 3-D Plots

To start with, here are two commented examples of 2-D plotting of parametric equations.

```matlab
Plots = 1; % 1: draw plots, 0: don't
dpdfPlots = 0; % 1: export plots to .pdf files, 0: don't

t = [0:0.01:10]; % points in the form: [start : stepSize : end]
x1 = cos(t).*exp(t/10); % compute some data for the 1st curve ... why use .* here?
y1 = sin(t).*exp(t/10);
x2 = sin(t).*(t/5).^2; % compute some data for the 2nd curve ... why use .^ here?
y2 = cos(t).*(t/5).^2;

if Plots
    figure(1); % plot x vs t and y vs t
    lw = 5; % line width value
clf % clear the plot
    subplot(2,1,1) % 1st of 2 sub-plots in this figure
    plot(t,x1, 'LineWidth', lw) % plot the first curve
    hold on % hold the first plot on the axes before plotting the next curve
    plot(t,x2, 'LineWidth', lw) % plot the second curve
    ylabel('x(t)') % y-axis label
    legend('x_1(t)', 'x_2(t)')
    legend('location','SouthWest')
    subplot(2,1,2) % 2nd of 2 sub-plots in this figure
    plot(t,y1, 'LineWidth', lw) % plot the first curve
    hold on % hold the first plot on the axes before plotting the next curve
    plot(t,y2, 'LineWidth', lw) % plot the second curve
    ylabel('y(t)') % y-axis label
    xlabel('t') % x-axis label
    legend('y_1(t)', 'y_2(t)')
    legend('location','SouthWest')
    set(gca, 'FontSize', 14) % increase the font size
    if dpdfPlots, print('MSDB-1.pdf', '-dpdfcrop'); end % in matlab just use '-dpdf'
end

figure(2); % plot y vs x
clf % clear the plot
plot(x1,y1, 'LineWidth', 5) % plot the first curve
hold on % hold the first plot on the axes before plotting the next curve
plot(x2,y2, 'LineWidth', 5) % plot the second curve
xlabel('x(t)') % axis labels
ylabel('y(t)')
axis('square') % equal spacing in x and y axes
grid on % add a grid to the plot
%legend('spiral #1', 'spiral #2')

text(-2.3,-1.7, 'Spiral #1', 'FontSize', 17, 'color', 'b'); % put some text at -2.3,-1.7

text(-2.2,-3.1, 'Spiral #2', 'FontSize', 17, 'color', 'r'); % put some text at -2.2,-3.1

set(gca, 'FontSize', 17) % increase the font size
if dpdfPlots, print('MSDB-2.pdf', '-dpdfcrop'); end % in matlab just use '-dpdf'
end % Plots
```
And here are commented examples of plotting random data.

```matlab
plots = 1; % 1: draw plots, 0: don’t
pdfplots = 0; % 1: export plots to .pdf files, 0: don’t

N = 1000;
rU = rand(N,1); % a vector of N random numbers uniformly distributed from 0 to 1
rN = randn(N,1); % a vector of N random numbers normally distributed, mean=0, std.dev=1

if plots
    figure(3) % scatter plot of rN vs rU
    clf
    plot(rU,rN, ’o’, ’MarkerSize’, 9)
    xlabel(’uniformly distributed X’)
    ylabel(’normally distributed Y’)
    set(gca, ’FontSize’, 17) % increase the font size
    if pdfplots,
        print(’MSDB-3.pdf’, ’-dpdfcrop’);
    end % in matlab just use ’-dpdf’

    figure(4) % PDF and CDF of rU and rN
    nBin = 20; % number of bins in the histogram
    clf
    subplot(2,2,1) % plot the histogram (pdf) of rU
    hist( rU, nBin, nBin/(max(rU)-min(rU)) );
    ylabel(’PDF’)
    subplot(2,2,2) % plot the histogram (pdf) of rN
    hist( rN, nBin, nBin/(max(rN)-min(rN)) );
    subplot(2,2,3) % plot the ogive (cdf) of rU
    stairs( sort(rU), ([1:N]-0.5)/N, ’LineWidth’, 3 );
    xlabel(’uniformly distributed X’)
    ylabel(’CDF’)
    subplot(2,2,4) % plot the ogive (cdf) of rN
    stairs( sort(rN), ([1:N]-0.5)/N, ’LineWidth’, 3 );
    xlabel(’normally distributed Y’)
    if pdfplots,
        print(’MSDB-4.pdf’, ’-dpdfcrop’);
    end % in matlab just use ’-dpdf’
end % Plots
```
And here are two commented examples of 3-D plotting surfaces and contours of surfaces.
18 Importing and Exporting Data

It is useful to know how to import data into MATLAB code, in order to analyze the data, maybe make some plots, and then to export the processed data in a particular format. The .mat data format is MATLAB’s own file format for exporting and importing MATLAB arrays and structures. For example, to save the three MATLAB variables called $A$, testData, and results into a single file called myWork.mat:

```
save myWork.mat A testData results
```

And to reload the contents of a .mat file called myWork.mat back into MATLAB,

```
load myWork.mat
```

The `save` and `load` commands can import and export data in ASCII (plain text) or binary format, and is compatible with multiple versions of MATLAB. ASCII files with numerical data stored in multiple tab or space delimited columns of equal length, preceded by some lines of header text, in which the first character of each header line is the comment character # may also be imported into a matrix using `load`. For example, to import the numerical data in an ASCII file called test-20201017.173529.dat into a five-row and four-column matrix, use ...

```
data = load('test-20201017.173529.dat');
```

Note here that every row and every column have the same number of values and that all the information in the header is ignored.
To import/export numerical data from/to an ASCII file of delimited columns, use `dlmread` or `dlmwrite`. And to import/export numerical data from/to an ASCII file of comma-delimited columns, (i.e., `.csv` files) use `csvread` or `csvwrite`. In `dlmread` and `csvread`, missing values are replaced by `0`. For example, to import the numerical data in a file called `test-20201017.173529.dat` with comma-separated columns

```
1 2 3 4
| filename   | test-20201017.173529.dat               |
| Sat Oct 17 17:35:29 |
| Experimental Data |
| Time | Sensor 1 | Sensor 2 | Sensor 3 |
| 0.01 | 5.02     | 6.2384   | -7.2304  |
| 0.02 | 8.22     | 7.2484   | 1.3324   |
| 0.03 | 8.42     | -6.2284  | -4.2311  |
| 0.04 | 7.32     | 9.2584   | 3.4378   |
| 0.05 | 5.18     |          | 9.3470   |
```

into a five-row and four-column matrix, skipping the first four lines (rows) of plain text data, skipping over zero columns, and replacing the missing values with zeros, use ...

```
data = dlmread( 'test-20201017.173529.dat' , ',' , 4 , 0 );
```

The most general purpose function for importing ASCII data (text and numerical values) is `textscan`. The `textscan` function scans a data file line by line and saves the results in to a Matlab structure, having one structure element for each of the format-specified variables. For example, to import the numerical data in contained in the ASCII file called `test-20201017.173529.dat`

```
1 2 3 4
% filename   test-20201017.173529.dat
% Sat Oct 17 17:35:29
% Experimental Data
% Time | Sensor 1 | Sensor 2 | Sensor 3 |
% 0.01 | 5.02     | 6.2384   | -7.2304  |
% 0.02 | 8.22     | 7.2484   | 1.3324   |
% 0.03 | 8.42     | -6.2284  | -4.2311  |
% 0.04 | 7.32     | 9.2584   | 3.4378   |
% 0.05 | 5.18     |          | 9.3470   |
```

into a structure of four vectors (one vector for each column), and skipping the first four lines of header information, use ...

```
1 fid = fopen( 'test-20201017.173529.dat' , 'r' ); % file pointer
2 data = textscan( fid , '%f %f %f %f' , 'HeaderLines', 4 );
3 fclose(fid);
```

in which `data` is a structure with four elements: `data{1}` is a column vector of the first column of the file, `data{2}` is a column vector of the second column of the file, and so on. An alternative usage of the `textscan` command simply skips over all text from `%` to the end of the line, and doesn’t require knowing the number of header lines.

```
data = textscan( fid , '%f %f %f %f' , 'CommentStyle', '+' );
```

The MATLAB language supports the C-standard file input/output functions like `fopen()`, `fclose()`, `fprintf()`, and `fscanf()`.

To automate the processing of a number of data files stored in a particular directory, use the `dir` command to create a list of the data files in the directory. For example, to
import all the data from a set of five-column data files that have header lines and other comments starting with the # character, into a structure of structures of five column vectors,

```matlab
dataDir = '/home/userID/CoolProject/SpecialData/';
files = dir(dataDir); % file names in dataDir
nFiles = size(files,1); % number files in the SpecialData folder

for ff = 3:nFiles    % loop over all files in dataDir skipping . and ..
    fn = files(ff).name;
    fid = fopen(fullfile(dataDir,fn), 'r'); % file pointer
    dataIn{ff} = textscan( fid, '%f %f %f %f %f', 'CommentStyle', '#');
    fclose(fid);
end
```
19 Skills

1. Understand the differences between a scalar (1-by-1), a row-vector (1-by-n), a column-vector (n-by-1), a matrix (m-by-n), and a multi-dimensionally array (m-by-n-by-r-by-...)
By default vectors are row-vectors, unless pre-initialized by a command like:
```matlab
my_vector = zeros(100,1);
```

2. Understand the need-for, and proper application of element-wise multiplications and vector multiplication:
```matlab
X = A * B  ... vector multiplication
X = A .* B  ... element wise multiplication
```

<table>
<thead>
<tr>
<th>size of A</th>
<th>1 x 1</th>
<th>m x n</th>
<th>m x k</th>
<th>m x n</th>
</tr>
</thead>
<tbody>
<tr>
<td>size of B</td>
<td>m x n</td>
<td>1 x 1</td>
<td>k x n</td>
<td>m x n</td>
</tr>
<tr>
<td>use:</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>.*</td>
</tr>
<tr>
<td>size of X</td>
<td>m x n</td>
<td>m x n</td>
<td>m x n</td>
<td>m x n</td>
</tr>
</tbody>
</table>

3. Understand the need-for, and proper application of left divide vs. right divide in matrix equations:
```matlab
X = A \ B  ... left divide solves ... AX = B ... for ... X
X = B / A  ... right divide solves ... XA = B ... for ... X
```

<table>
<thead>
<tr>
<th>size of A</th>
<th>1 x 1</th>
<th>m x n</th>
<th>m x n</th>
<th>m x n</th>
</tr>
</thead>
<tbody>
<tr>
<td>size of B</td>
<td>m x n</td>
<td>1 x 1</td>
<td>k x n</td>
<td>m x n</td>
</tr>
<tr>
<td>use:</td>
<td>\</td>
<td>.\</td>
<td>\</td>
<td>.\</td>
</tr>
<tr>
<td>size of X</td>
<td>m x n</td>
<td>m x n</td>
<td>n x k</td>
<td>m x n</td>
</tr>
</tbody>
</table>

```matlab
X = A .^ B  ... vector exponentiation
X = A.^ B  ... element wise exponentiation
```

<table>
<thead>
<tr>
<th>size of A</th>
<th>1 x 1</th>
<th>m x n</th>
<th>m x n</th>
<th>n x n</th>
<th>m x n</th>
</tr>
</thead>
<tbody>
<tr>
<td>size of B</td>
<td>m x n</td>
<td>1 x 1</td>
<td>1 x 1</td>
<td>m x n</td>
<td></td>
</tr>
<tr>
<td>use:</td>
<td>^</td>
<td>^</td>
<td>or</td>
<td>^</td>
<td>^</td>
</tr>
<tr>
<td>size of X</td>
<td>m x n</td>
<td>m x n</td>
<td>n x n</td>
<td>m x n</td>
<td></td>
</tr>
</tbody>
</table>

5. Understand the need-for, and proper application of scripts vs. functions.
6. Understanding how to concatenate row-vectors into a matrix and column-vectors into a matrix, or, more generally, how to tile sub-matrices into a larger matrix. Try to answer the following questions with pencil and paper.

Given ...

```matlab
1 a = [ 1 2 3 4 ]
2 b = [ 5 6 7 8 ]
3 x = [ 5 6 7 8 9 ]
```

... what are ...

```matlab
1 c = [ a , b ]
2 d = [ a' ; b' ]
3 C = [ a ; b ]
4 D = [ a' , b' ]
5 X = [ a , b' ] ... or ... X = [ a ; b' ] ... or ... [ a ; x ] ... etc.
```

Given ...

```matlab
1 A = [ 1 2 3 ; 4 5 6 ; 7 8 9 ]
2 B = [ 10 11 ; 12 13 ; 14 15 ]
3 C = [ 16 17 18 ; 19 20 21 ]
4 D = [ 22 23 ; 24 25 ]
```

... what is ...

```matlab
1 H = [ A B ; C D ]
```

7. Use descriptive names for variables, functions, and scripts

8. Comment every line

9. Organize your code into sections in this order:

   (a) within scripts, initialize variables with numerical values, or NaN (not a number)

   (b) within functions, pull out vector- or structure- input variables into scalar or matrix variables meaningful names

   (c) calculations

   (d) plots

10. Start programming problems by writing pseudo-code on paper.
20 Debugging

Computer programs almost never work as intended at first. It is very helpful to know how to locate the lines of code with errors. Fortunately, MATLAB error messages show the lines of code involved in errors.

Matlab errors often errors result from trying to add, or multiply matrices of incompatible dimension, or trying to divide by zero, or indexing a matrix beyond its dimensions.

Other times, the program will run but will give results that don’t make sense, or are otherwise not as intended.

Here are tips on how to find the lines of code that could be the source of these errors.

1. Check the dimensions and numerical values of a variable by omitting the ; at the end of the line computing the variable’s value
2. A very (very) common programming error is to attempt to multiply, divide, add, or subtract vectors or matrices with incompatible dimensions. A (3-by-4) matrix cannot be multiplied by a (5-by-7) matrix. Dimensions of variables are shown in the Workspace Window or can be found by commands:
   \[ \text{size}(X) \] 
   \[ \text{size}(X,1) \] 
   \[ \text{size}(X,2) \]
   \[ \text{length}(x) \] 
   without a semi-colon, so that the result is displayed in the Command Window, just for de-bugging purposes.
3. Use the built-in help function to understand how a function is to be used.
4. Save variables into a .mat file in order to read them back in again, later
% Purpose: illustrate the use of helpful matlab commands
% Kim Bourne 9/07/2018

Getting Started
% Either set your working directory manually or use the command cd (this is
% my file path on Windows, replace with your own)
cd 'C:\Users\Kimberly Bourne\Box Sync\EGR 305'
clear % Always begin code by clearing your workspace
close all % Also consider closing all figures
clc % Totally optional, but I also like to clear the command window

Basic Matrix Commands
a = [1 9 4] % create a vector
b = [3 2 1; 1 2 3]; % Create a matrix with multiple rows
z = zeros(2,3);
a(1,3)

% To perform multiplication of two matrices of compatible dimensions,
% just use the * operator.
% In the example below c is 1-by-3 and d = 3-by-2, so the result is 1-by-2
c = [1 2 3];
d = [4 5; 6];
e = c*d;

% If you want to perform element wise operations on matrices, insert a
% period before the operand.
f = d.^2;
g = d .* [1 2; 3 4; 2 1];

% To transpose a matrix, just use an apostrophe
aprime = a';

Creating masks
% ex. if you want to know which elements of f are greater than
% 30, you can create a matrix of ones and zeros where ones indicate
% positions where f(i,j) > 30
mask = f > 30;

% Read in data from an excel file
% If you have an excel file with character and numeric variables, you will
% want to store those in different matrices in MATLAB using the following
% code.
[num, txt, raw] = xlsread('MatlabTutorial_PracticeData.xlsx');

% for loops and if statements
% For loops are useful when you want to code iterative processes or
% manipulate a matrix in a way that isn't convenient or possible using
% matrix operations. Below is something you wouldn't actually want to use a
% for loop for, it's just to give you an example of how to write them.
for i = 1:1:4
    a(1,i) = a(1,i) + i*3;
end

% For loops can often be avoided in MATLAB.
% Doing so leads to code that looks cleaner and runs faster.
% For example, consider evaluating the sum of (1/n) for n = 1 to N
% This can be done with a for loop ...
N = 5;
z = 0;
for n = 1:N
    z = z + 1/n;
end
% ... or it can be done in one line, without a for loop.
z = sum(1./(1:N));

% If statements are useful for many reasons such as sorting data or creating
% specific matrices.
% Ex. Say a road trip is either local (0–50mi), short (51–300mi), or long
% (301+) and I wanted to rank my trips with a numerical version of this
% (1, 2, or 3, respectively) based on mileage.
for i = 1:size(num,1)
    if num(i,1) < 50
        num(i,3) = 1;
    elseif num(i,1) > 50 && num(i,1) < 300
        num(i,3) = 2;
    else
        num(i,3) = 3;
    end
end

22 References

1. The Complete MATLAB Course: Beginner to Advanced!,
   https://www.youtube.com/watch?v=T_ekAD7U-wU

2. Introduction to MATLAB, CS 229 Machine Learning, Stanford
   http://cs229.stanford.edu/section/cs229-matlab.pdf