Table of Contents

More on Basic MATLAB Workshop

This document describes the steps covered to teach MATLAB Fundamentals after receiving the Introduction to MATLAB Workshop and prepare attendees with the remainder of the material for the MathWorks Certified MATLAB Associate Exam.

```
% Written by: Michael Hayashi
% Purdue IEEE Student Branch
% 17 November 2017
% Version: MATLAB(R) R2015b
close('all')
clearvars
clc
```

Summary of Introduction to MATLAB Workshop

MATLAB (matrix laboratory) is a numerical computing engine and very high-level programming language created by MathWorks. It is found in many corporations and excels at linear algebra and very high-level tasks while lacking the nimbleness of smaller programming languages and numeric solvers. It is assumed that you are familiar with the graphical user interface (GUI) of the MATLAB main window and script editor.

Remember that `help` followed by a function name displays the basic information about that function in the Command Window. Using `doc` followed by the function name opens a browser in MATLAB with a detailed explanation of how the function works with related topics. The Command History reminds you of your recent inputs to the Command Window, and the Workspace shows all variables currently saved in memory. Clicking on a variable opens the *Variable Editor* for inspecting and adjusting variables in the uncommon situation where that feature is needed.

Scalar mathematics are straightforward: `=` for assignment, `;` at the end for output suppression, `true` and `false` as the `logical` class values, `function handle` class being created by putting `@` before a function, and nearly everything else being stored as some sort of array of double-precision numbers. Parentheses `()` are used for grouping, and addition `+`, subtraction `-`, multiplication `*`, right division `/`, left division `\`, and exponentiation `^` are available. The relational operators use the standard symbols of `=` (two of them meaning "equal to"), the left carat for less than, and the right carat for greater than. NOT in MATLAB is specified with ~. The logical operators use & or && (AND) as well as a vertical pipe or two vertical pipes (OR). The doubled logical operators use short-circuiting to deliver an output sooner if possible and should be used in most Boolean expressions. The complex unit is either `1j` or `1i`. Avoid using the letters `i` and `j` by themselves as variables to avoid overloaded behavior. Lines are continued with three periods `...` at the end.

Vectors are single-dimensional arrays, either rows or columns (preferred), that have their own associated mathematics. Matrices are two-dimensional arrays with similar mathematics. Array literals are created with square brackets `[]` with commas `,` separating entries in the same row and semicolons `;` separating rows vertically. The functions `length()` (longest dimension of array), `numel()` (total number of elements in an array), and `size()` (array with the length of each dimension as entries) have different purposes when measuring arrays. The colon operator `:` can be used to create vectors spanning from a start point to stop point with a specified difference between entries. The functions `linspace()` and `logspace()` are used to create a given number of linearly spaced entries or logarithmically spaced entries between two numbers. Transposition is accomplished with `.'`, and conjugate transposition/Hermitian transposition that is used more frequently with complex matrices is accomplished with `'`. The dot operator allows element-wise multiplication `.*` (also known as the Schur product), element-wise right division `./`, element-wise left division `.\`, and element-wise exponentiation `.^`.

Matrices have a few additional quirks that vectors lack. A system of equations in matrix form can be solved quickly with Gaussian elimination using left division `\`. Pseudorandom matrices can be created using `randn()` and similar functions. The identity element for matrix addition is created with `zeros()`, the identity element for matrix multiplication is created with `eye()`, and the identity element for element-wise matrix multiplication is created with `ones()`. When requested, MATLAB can save matrices as **sparse matrices** using `sparse()`.

There are many statistics functions avaiable, working along the first non-unity dimension.

- Central tendency: `mean()`, `median()`, `mode()`
- Other means: `geomean()`, `harmmean()`, `rms()`
- Dispersion measures on the sample: `min()`, `max()`, `range()`, `iqr()`, `var()`, `std()`
- Moments: `skewness()` and `kertosis()`
- Multiple distributions: `cov()` and `corrcoef()`

We will cover more on strings later. The MATLAB format specification syntax is useful when printing to the Command Windows or files with `fprintf()` and when printing to strings `sprintf()`. (https://www.mathworks.com/help/matlab/matlab_prog/formatting-strings.html)

Variables can be saved to MATLAB data files (.mat) with `save()` and loaded into the Workspace by drag-and-drop or by the `load()` command. Other files types can be loaded with `importdata()` or `uiimport()` if that fails. There are more specific functions available in case those fail to work as intended for other files.

Low-level file input/output (I/O) is good for reading and writing simple text files. The file identifier is obtained with `fopen()` using the appropriate permissions. Writing is simple with `fprintf()` as long as you remember to use the newline character `'\n'` to separate each line. **Always** remember to close your file with `fclose()`, even if an error stops the script midway through. A number of functions such as `fgetl()`, `fgets()`, `fscanf()`, `ftell()`, `fseek()`, `feof()`, and `frewind()` allow you to move around the file with precise control, especially when reading data.

The `for` loop has a special `for variable = vector` ... `end` syntax that behaves more like "for each" to iterate over a given set of data. The `if` statement has a syntax like `if boolean1` ... `elseif boolean2` ... `else` ... `end` for making decisions. These are the two flow of control structures that encompass the vast majority of MATLAB use cases. The `while` loop has a `while boolean` ... `end` syntax useful when the iteration goes on for an unknown duration. The `switch expression` ... `case num` ... `otherwise` ... `end` is an alternative to `if` that works for some complicated expressions to evaluate.

Matrices are one-indexed first by rows numbered vertically, then by columns numbered horizontally. Slices and strides of the array are taken with `start:stride_pitch:stop` in the desired index position, with the default stride pitch of `1`. Arrays also accept logical indexing by accepting a same-sized array of logicals in the indexing parentheses. The `find()` function also can return the indices that meet a condition the programmer imposes.

The best practices for writing scripts and functions will be covered again by virtue of example. Cell mode makes scripts easier to read and develop by using `%%` to denote sections. A function is a MATLAB .m file that has all code contained with one or more function blocks. The function block begins with the keyword `function` and ends with the keyword `end` just like the control of flow blocks. Following the keyword `function` is the group of outputs (enclosed in square brackets `[]` and separated by commas `,`). After the outputs comes a single equal sign `=`. The name of the custom function follows with comma-separated inputs as arguments in parentheses `()`. User-defined functions support a commenting style that produces documentation with `help` just like standard functions. Make sure that all scripts and functions necesssary for a project are part of the current path.

Breakpoints are used to assist debugging by stopping script interpretation once one is reached. Scripts can be read line-by-line after each breakpoint if needed to pinpoint issues. The MATLAB Code Analyzer will help you write better code by flagging syntax errors and warnings as they arise. Try to get the green square every time.

Plotting can be handled systematically in a script or function:

1. Create a figure window with a specific number using `figure()`.
2. Split the figure window into subplots with their own axes using `subplot()` if desired.
3. Use `plot()` to put any number of curves onto a 2D figure. Similar syntax is used for `semilogx()`, `semilogy()`, and `loglog()`.
4. Adjust the appearance of the graph with `axis()`, `grid()`, and related functions.
5. Label the plots and the axes with `title()`, `xlabel()`, and `ylabel()`.
6. Callout certain features using `sprintf()` within `text()`.
7. Add a legend using `legend()` if desired before moving on the the next subplot axes or figure.

Matrices Revisited: Plotting

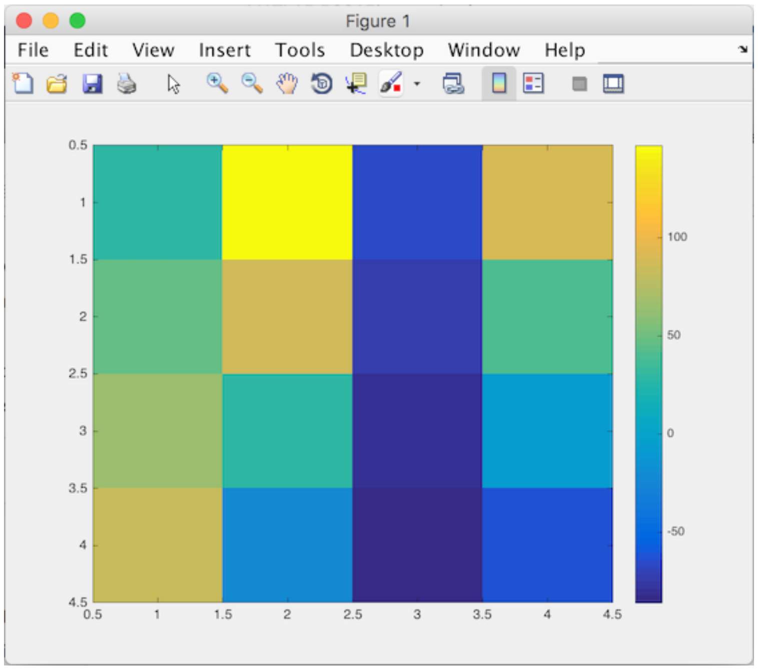
It can be difficult to keep track of the features of larger matrices. Sometimes a plot of the matrix helps a user understand the relationship between the entries contained within. (https://www.mathworks.com/help/matlab/stem-and-stair-plots.html)

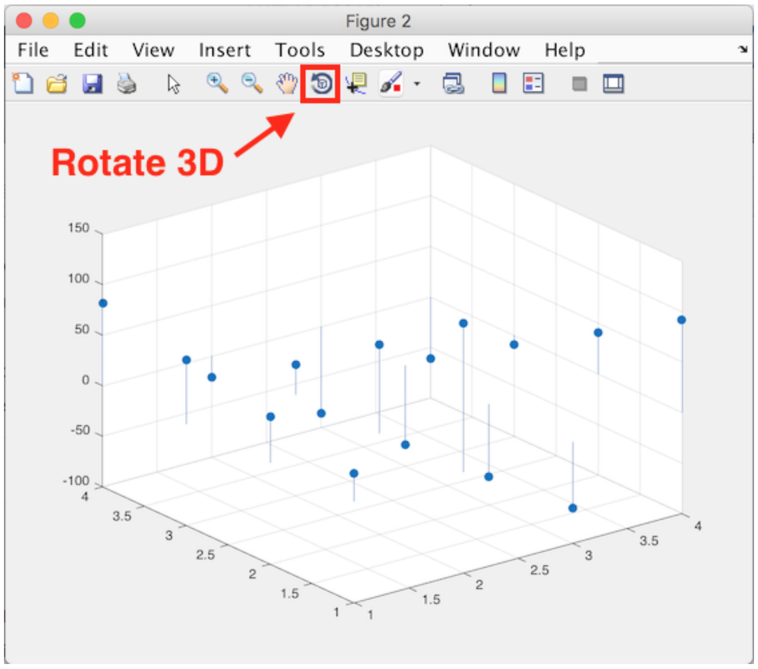
Type the following into a script and save it as "workshop2.m", inserting your own information into the header for your own records:

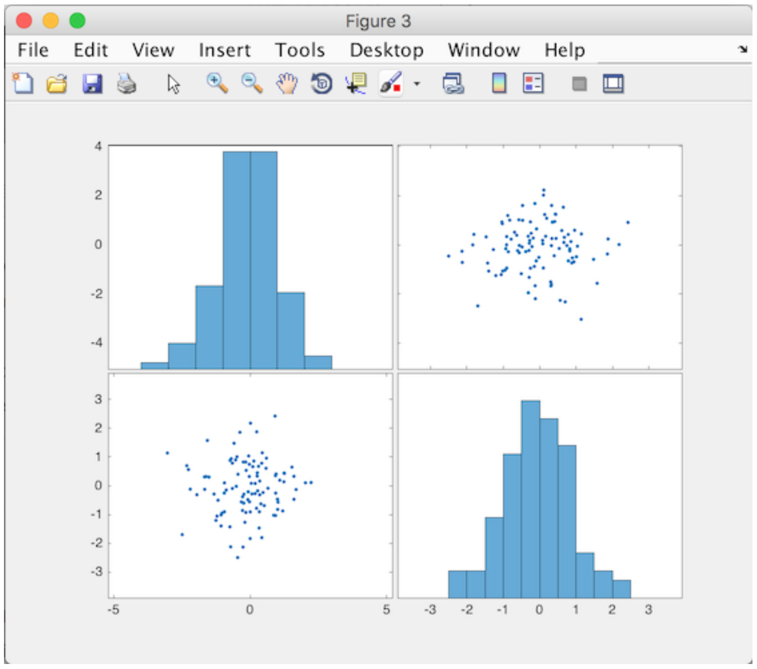
```
%% Header
% Script to follow along with the More on Basic MATLAB Workshop.

% Written by: <First> <Last>
% <Affiliation>
% <Day> <Month> <Year>
% Version: MATLAB(R) <Version>
close('all')
clearvars
clc

%% Exploration

mat1 = [28, 147, -65, 92; 46, 88, -72, 41; ...
    64, 30, -79, -9; 82, -21, -86, -61];

figure(1)
imagesc(mat1)
colorbar
% heatmap(mat1)

figure(2)
stem3(mat1, 'filled')

figure(3)
plotmatrix(randn(100, 2))
```

MATLAB Comprehension Question: What do you expect to see in each figure?

Run the script by pressing the green "Run" button (F5 in Windows). Let's make some observations on **matrix plotting**:







- The first figure treats the matrix as an image in the default **colormap**. The function `imagesc()` creates an image from a matrix, scaling the values to use the full range of the colormap. By contrast, the function `image()` assigns specific colors to values based on the range specified in the colormap, hard-limiting values that are out-of-bounds to either the maximum or minium value in the colormap.

- The default colormap in modern MATLAB is `parula()`, a vibrant gradation of colors that is meant to be easy to see on graphs. The name comes from a distinctive species of songbird (http://www.audubon.org/field-guide/bird/northern-parula). The colormap may be changed by passing the name of a recognized colormap to `colormap()`. Other colormaps include `jet()` (rainbow order), `hsv()` (the complete hue-saturation-value model of color), `hot()` and `cool()` (used for weather stations), `autumn()` and other seasons (festive), `gray()` (grayscale images), `bone()` (when viewing x-rays of mammals), and `prism()` (repeating sequence of major rainbow colors).

- The function `colorbar()` creates a small axis on the side that shows the mapping between the displayed color and the underlying numerical value.
- The second figure uses `stem3()` to create a stem plot of discrete data in three dimensions, as opposed to normal plotting functions such as `semilogy()` that are meant to show continuous data. Each data point has a marker (chosen to be `'filled'`) at the matrix coordinates located at the value in the z-direction with the start of the stem at the zero value.
- It may be difficult to get a complete picture of 3D data from the default viewing angle in MATLAB. Press the button in the *figure window* with the blue cube surrounded by a counterclockwise arrow to activate the *Rotate 3D* tool. By clicking and dragging the cursor in the 3D axes, you may change the view of the data. In the rare case where the default view does not work for a particular data set, use the `view()` function that accepts a row vector of length two with the the azimuth angle in degrees (-37.5? default) as the first entry and the elevation angle in degrees (+30? default) as the second entry.
- The third figure uses `plotmatrix()` to create a matrix of subplots showing the distribution of the entries within a sizable matrix of values. The subplots along the main diagonal are histograms of the data within a given column. The subplots in the i^{th} row and j^{th} column have a scatter plot of the data in the i^{th} column of the matrix as the y-axis values against the j^{th} column of the matrix as the x-axis values.

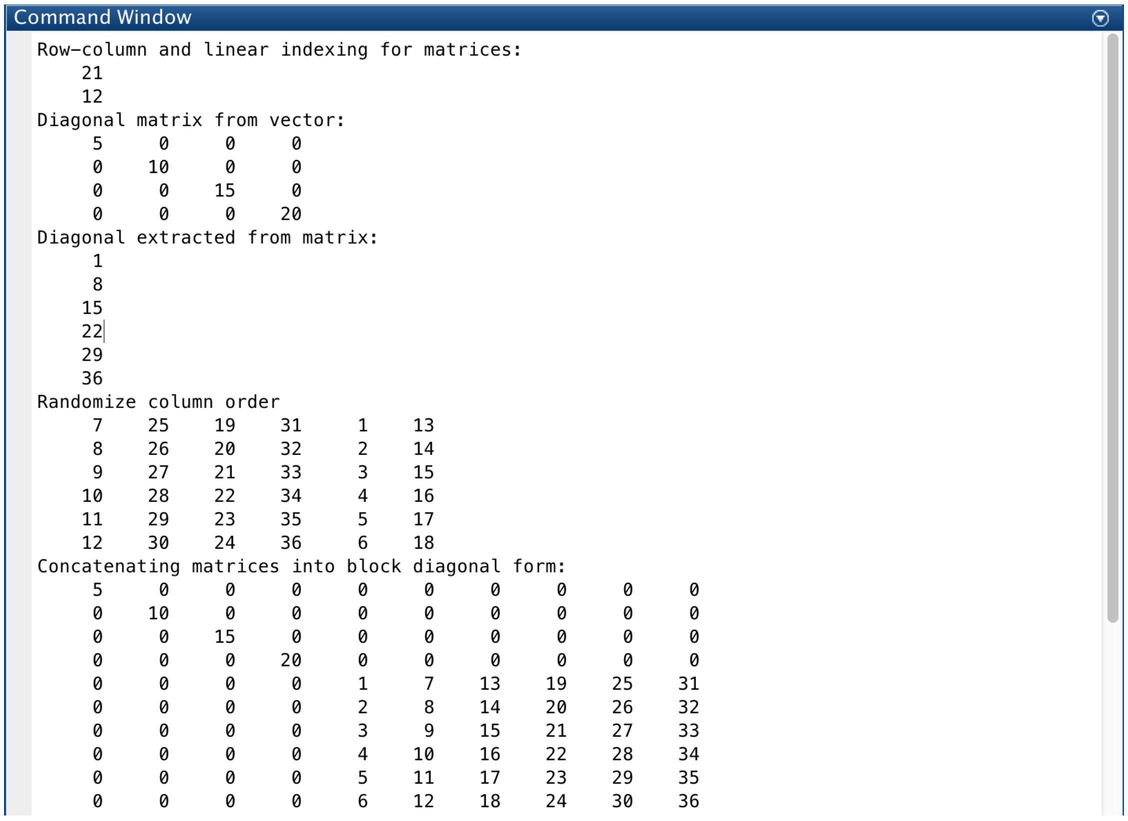
Matrices Revisited: Intermediate Manipulation

There are some odds and ends to cover in order to have full control over how matrices are indexed, created, and concatenated. Ensure that all active lines below the "Exploration" section in "workshop2.m" are commented out, by pressing the green percent sign "Comment" (Ctrl + R in Windows) in the "Edit" section of the "Editor" ribbon or some other means. Go to a blank line below the commented out script. It is time to move on to **intermediate matrix manipulation**.

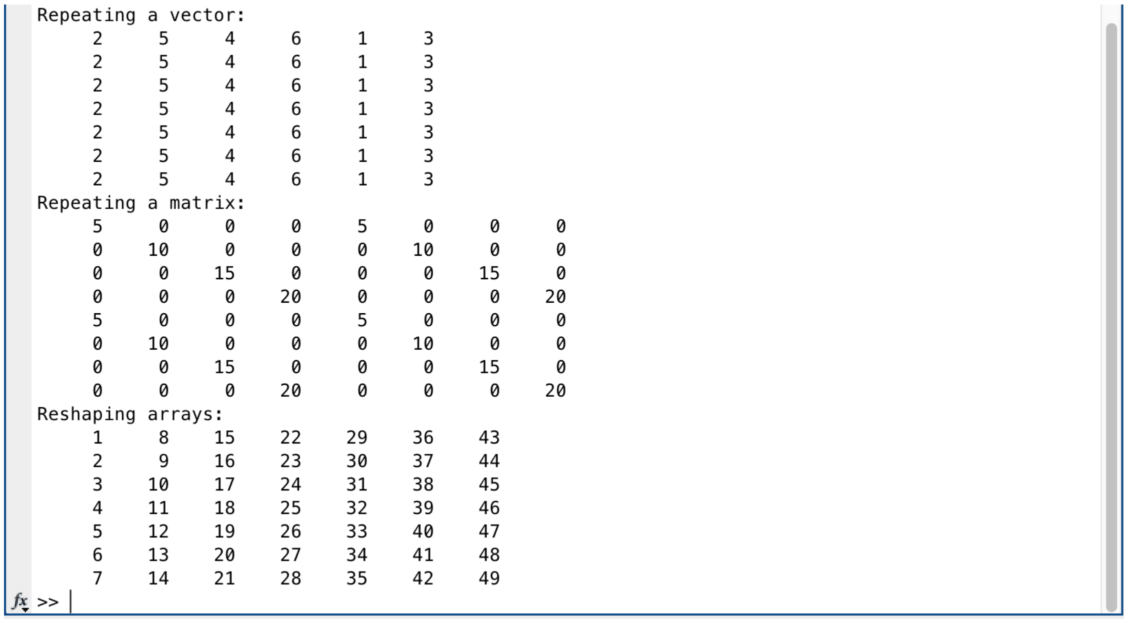
```
mat1 = [(1:6).', (7:12).', (13:18).', (19:24).', (25:30).',
 (31:36).'];
vec1 = randperm(size(mat1, 2));
vec2 = (5:5:20).';
mat2 = diag(vec2);
mat3 = blkdiag(mat2, mat1);
mat4 = reshape([mat1(:); (37:49).'], 7, 7);

disp('Row-column and linear indexing for matrices:')
disp(mat1(3,4))
disp(mat1(12))
disp('Diagonal matrix from vector:')
disp(mat2)
disp('Diagonal extracted from matrix:')
disp(diag(mat1))
disp('Randomize column order')
disp(mat1(:,vec1))
disp('Concatenating matrices into block diagonal form:')
disp(mat3)
disp('Repeating a vector:')
disp(repmat(vec1, 7, 1))
disp('Repeating a matrix:')
disp(repmat(mat2, 2, 2))
disp('Reshaping arrays:')
disp(mat4)
```

MATLAB Comprehension Questions: Which values will display from indexing? How will the function `diag()` react to a vector input versus a matrix input? Why can the columns be rearranged with the function `randperm()`? What does it mean for a matrix to be *block diagonal*? Whem might `repmat()` and `reshape()` be useful?

Run the script by pressing the green "Run" button (F5 in Windows). Let's make some observations on **linear indexing, matrix creation, matrix concatenation, and reshaping**:



```
Row-column and linear indexing for matrices:
    21
    12
Diagonal matrix from vector:
     5     0     0     0
     0    10     0     0
     0     0    15     0
     0     0     0    20
Diagonal extracted from matrix:
     1
     8
    15
    22
    29
    36
Randomize column order
     7    25    19    31     1    13
     8    26    20    32     2    14
     9    27    21    33     3    15
    10    28    22    34     4    16
    11    29    23    35     5    17
    12    30    24    36     6    18
Concatenating matrices into block diagonal form:
     5     0     0     0     0     0     0     0     0     0
     0    10     0     0     0     0     0     0     0     0
     0     0    15     0     0     0     0     0     0     0
     0     0     0    20     0     0     0     0     0     0
     0     0     0     0     1     7    13    19    25    31
     0     0     0     0     2     8    14    20    26    32
     0     0     0     0     3     9    15    21    27    33
     0     0     0     0     4    10    16    22    28    34
     0     0     0     0     5    11    17    23    29    35
     0     0     0     0     6    12    18    24    30    36
```



```
Repeating a vector:
     2     5     4     6     1     3
     2     5     4     6     1     3
     2     5     4     6     1     3
     2     5     4     6     1     3
     2     5     4     6     1     3
     2     5     4     6     1     3
     2     5     4     6     1     3
Repeating a matrix:
     5     0     0     0     5     0     0     0
     0    10     0     0     0    10     0     0
     0     0    15     0     0     0    15     0
     0     0     0    20     0     0     0    20
     5     0     0     0     5     0     0     0
     0    10     0     0     0    10     0     0
     0     0    15     0     0     0    15     0
     0     0     0    20     0     0     0    20
Reshaping arrays:
     1     8    15    22    29    36    43
     2     9    16    23    30    37    44
     3    10    17    24    31    38    45
     4    11    18    25    32    39    46
     5    12    19    26    33    40    47
     6    13    20    27    34    41    48
     7    14    21    28    35    42    49
fx >> 
```

- *Row-column indexing* or *array subscripts indexing* uses one number in each dimension of the array (rows numbered up-down before columns numbered left-right). *Linear indexing* is an alternative that uses a single number to access any element in the array. The following statements are equivalent, demonstrating how the linear index may be found: `mat1(indi,indj)` and `mat1((indj - 1) * size(mat1, 1) + indi)`. *This potentially conflicts with how one might read entries in a matrix naturally.*

- There are two uses of the function `diag()`. If the argument is a vector, then the output is a diagonal matrix that is vector-length-by-vector-length with the vector values on the *main diagonal*. If the argument is a square matrix, then the output is a column vector with a length equal to the square root of the number of elements and the values along the main diagonal. A second argument is optionally added to work along a superior diagonal (positive integer) or inferior diagonal (negative integer).

- Creation of pseudorandom matrices was covered in the previous workshop with functions such as `randi()` (uniform discrete distribution of integers), `rand()` (uniform continuous distribution), and `randn()` (normal distribution). The function `randperm()` creates a random permutation from the number one to the integer passed to it. An optional second argument (a smaller integer than the first) allows for only that many elements to be selected from the permutation. Your answers for `vec1` and `mat3` will likely differ from mine.

- The variable `vec1` contains the random permutation of integers from 1 to 6, the number of columns in `mat1`. By passing the vector to the column index of `mat1` and using the colon `:` to represent the entire range of values along the row, a *slice* of the matrix is taken such that the result is a random arrangement of the columns of `mat1`. This is useful in double-blind studies meant to anonymize data. A methodical way to rearrange matrix columns involves the function `circshift()`.

- Concatenation of matrices was covered in the previous workshop with functions such as `horzcat()` or the comma for literals `,` and `vertcat()` or the semicolon for literals `;`. A more exotic concatenation of matrices ends in a *block diagonal* matrix by passing multiple matrices as arguments to `blkdiag()`. The following statements are equivalent: `blkdiag(A, B)` and `[A, zeros(size(A, 1), size(B, 2)); zeros(size(B, 1), size(A, 2)), B]`. Note that this form follows the idea of a *diagonal matrix*, but with the entries being matrices themselves including rectangular zero matrices in off-diagonal positions.

- The function `repmat()` is useful for expanding vectors or matrices in situations where data needs to be repeated or made cyclic, a recurring type of concatenation. The first argument is the vector or matrix serving as the basis for the *tiling*. The subsequent two or more arguments are the number of times to replicate the basis in each dimension in turn.

- When data arrives in a weird way, it may be useful to use `reshape()` to force the array into a more familiar form. The first argument is an array whose entries will be read down each column first before reading across the row. The subsequent arguments are the sizes along each dimension. For the function to avoid errors, the product of the subsequent arguments **must** equal the number of elements in the array as determined by `numel()`. **Reshaping an array with a prime number of elements will always fail.**`mat4` unpacks `mat1` with linear indexing, concatenates the resulting column vector with more values, and then reshapes the result into a 7-by-7 matrix.

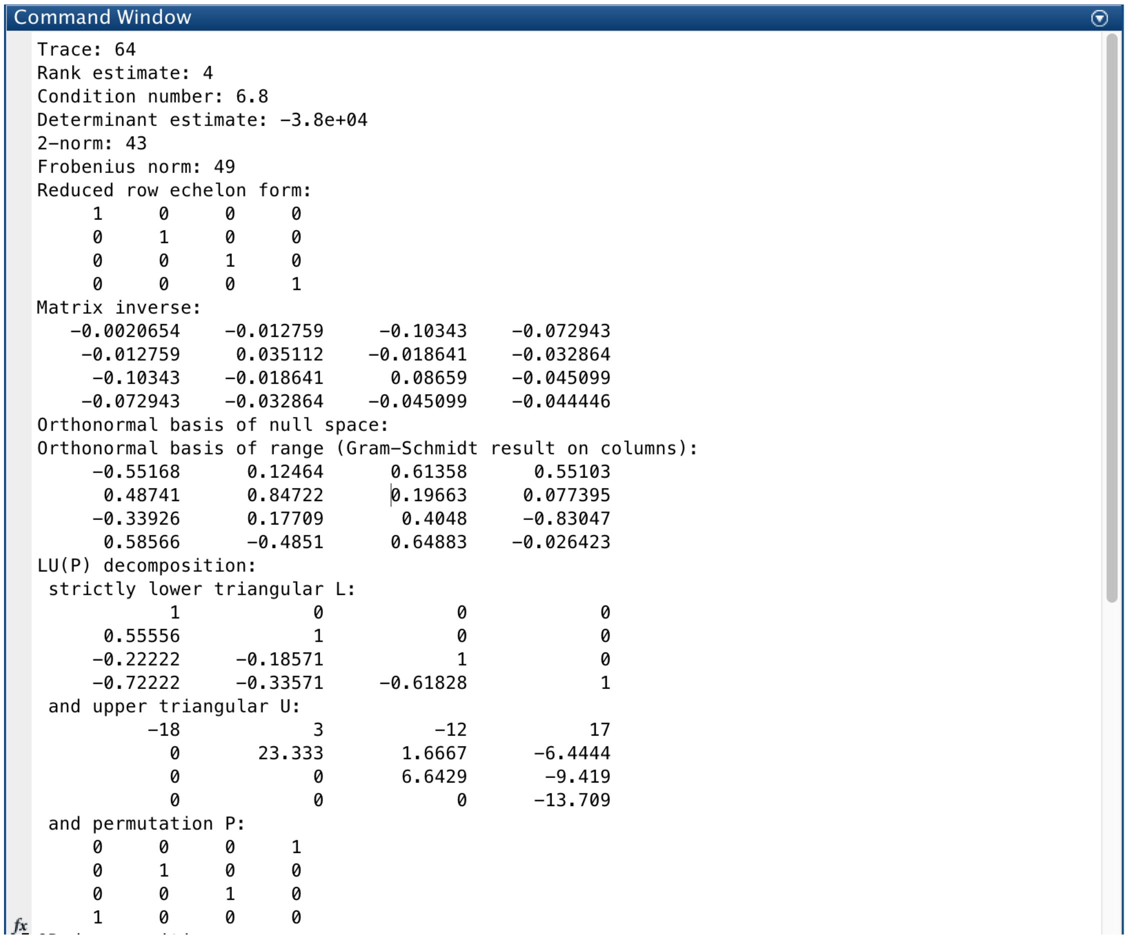
Matrices Revisited: Linear Algebra Metrics and Decompositions

A good course on linear algebra will cover such topics as determinants, traces, reduced row echelon form, and LU decomposition. MATLAB naturally has functions meant to perform these calculations quickly. Since a course on linear algebra is not a prerequisite for this workshop, please feel free to skip over any unfamiliar terms and concepts. Ensure that all active lines below the "Exploration" section in "workshop2.m" are commented out, by pressing the green percent sign "Comment" (Ctrl + R in Windows) in the "Edit" section of the "Editor" ribbon or some other means. Go to a blank line below the commented out script. It is time to explore **linear algebra metrics and decompositions**.

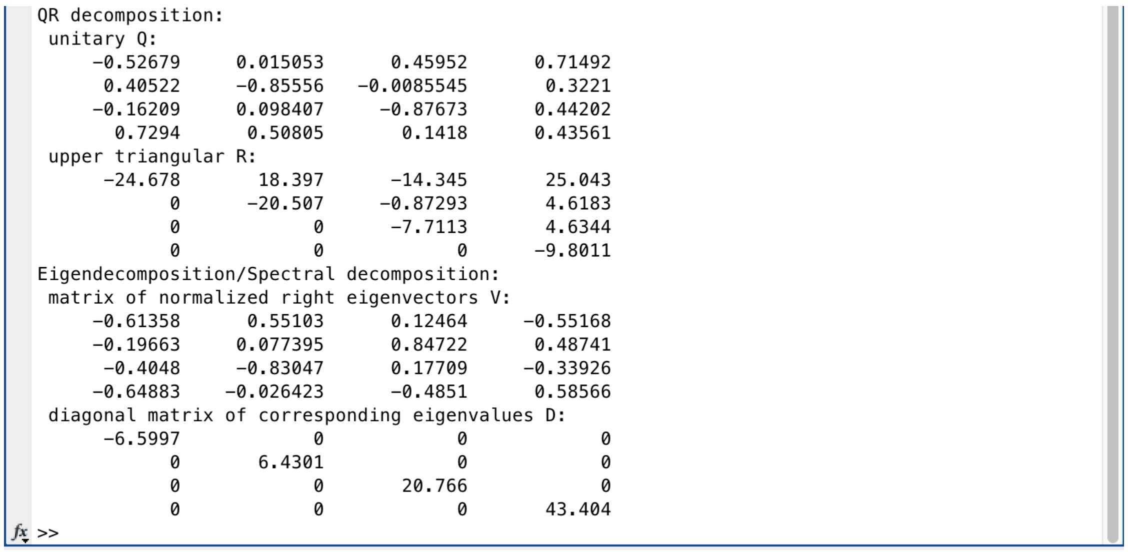
```
mat1 = [13, -10, 4, -18; -10, 25, -5, 3; ...
    4, -5, 9, -12; -18, 3, -12, 17];
% mat2 = [13, -10, 4, -8; -10, 25, -5, 3; ...
%     4, -5, 9, -2; -8, 3, -2, 17];
[mat1_L, mat1_U, mat1_P] = lu(mat1);
[mat1_Q, mat1_R] = qr(mat1);
[mat1_V, mat1_D] = eig(mat1);

fprintf(1, 'Trace: %1.2g\n', trace(mat1));
fprintf(1, 'Rank estimate: %1.2g\n', rank(mat1));
fprintf(1, 'Condition number: %1.2g\n', cond(mat1));
fprintf(1, 'Determinant estimate: %1.2g\n', det(mat1));
fprintf(1, '2-norm: %1.2g\n', norm(mat1, 2));
fprintf(1, 'Frobenius norm: %1.2g\n', norm(mat1, 'fro'));
disp('Reduced row echelon form:')
disp(rref(mat1))
disp('Matrix inverse:')
disp(inv(mat1))
disp('Orthonormal basis of nullspace:')
disp(null(mat1))
disp('Orthonormal basis of range (Gram-Schmidt result on columns):')
disp(orth(mat1))
disp('LU(P) decomposition:')
disp(' strictly lower triangular L:')
disp(mat1_L)
disp(' and upper triangular U:')
disp(mat1_U)
disp(' and permutation P:')
disp(mat1_P)
disp('QR decomposition:')
disp(' unitary Q:')
disp(mat1_Q)
disp(' upper triangular R:')
disp(mat1_R)
disp('Eigendecomposition/Spectral decomposition:')
disp(' matrix of normalized right eigenvectors V:')
disp(mat1_V)
disp(' diagonal matrix of corresponding eigenvalues D:')
disp(mat1_D)
```

MATLAB Comprehension Questions: Given the complicated nature of these calculations and the fact that MATLAB cannot work with fractions in most cases, to what degree is each function only an estimate of the ideal result? Without getting too deep into the linear algebra, what superficial things do you notice when running these functions on `mat2` or other example matrices?

Run the script by pressing the green "Run" button (F5 in Windows). Let's make some observations on **linear algebra on real, square matrices**:



```
Trace: 64
Rank estimate: 4
Condition number: 6.8
Determinant estimate: -3.8e+04
2-norm: 43
Frobenius norm: 49
Reduced row echelon form:
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
Matrix inverse:
   -0.0020654    -0.012759     -0.10343    -0.072943
    -0.012759     0.035112    -0.018641    -0.032864
     -0.10343    -0.018641      0.08659    -0.045099
    -0.072943    -0.032864    -0.045099    -0.044446
Orthonormal basis of null space:
Orthonormal basis of range (Gram-Schmidt result on columns):
     -0.55168      0.12464      0.61358      0.55103
      0.48741      0.84722      0.19663     0.077395
     -0.33926      0.17709       0.4048     -0.83047
      0.58566      -0.4851      0.64883    -0.026423
LU(P) decomposition:
 strictly lower triangular L:
            1            0            0            0
      0.55556            1            0            0
     -0.22222     -0.18571            1            0
     -0.72222     -0.33571     -0.61828            1
 and upper triangular U:
          -18            3          -12           17
            0       23.333       1.6667      -6.4444
            0            0       6.6429       -9.419
            0            0            0      -13.709
 and permutation P:
     0     0     0     1
     0     1     0     0
     0     0     1     0
     1     0     0     0
```



```
QR decomposition:
 unitary Q:
    -0.52679     0.015053      0.45952      0.71492
     0.40522     -0.85556   -0.0085545       0.3221
    -0.16209     0.098407     -0.87673      0.44202
      0.7294      0.50805       0.1418      0.43561
 upper triangular R:
     -24.678       18.397      -14.345       25.043
           0      -20.507     -0.87293       4.6183
           0            0      -7.7113       4.6344
           0            0            0      -9.8011
Eigendecomposition/Spectral decomposition:
 matrix of normalized right eigenvectors V:
    -0.61358      0.55103      0.12464     -0.55168
    -0.19663     0.077395      0.84722      0.48741
     -0.4048     -0.83047      0.17709     -0.33926
    -0.64883    -0.026423      -0.4851      0.58566
 diagonal matrix of corresponding eigenvalues D:
     -6.5997            0            0            0
           0       6.4301            0            0
           0            0       20.766            0
           0            0            0       43.404
>>
```

- The *trace* is an important metric of a square matrix, defined as the sum of the entries on the main diagonal. The function `trace()` handles this.

- The *rank* of a matrix is the maximum number of *linearly independent* columns in the matrix (linearly independent meaning that no element of a set can be created by adding together scaled versions of other nonzero elements). The function `rank()` estimates this using a tolerance controlled by an optional second argument. If a square matrix has a rank equal to either of its dimensions, it is called *full rank*. A full rank matrix has properties that make it possible to solve linear equations uniquely.

- The *condition number* of a matrix is related to the rank. Small condition numbers mean that the matrix is well-conditioned when solving matrix equations. Large condition numbers mean the matrix is ill-conditioned and require backwards stable algorithms. Unfortunately, there may be nothing possible to improve the condition number of some problems. (My rule of thumb: condition number between 1 and 10 means that the numerical approximations can be believed, condition number between 10 and 100 means that the numberical approximations should be treated with skepticism and believable results should only be used for guidance, and condition number above 100 means that everything should be thrown out.)

- *Determinants* are even more important than traces as a square matrix metric. An interpretation of these metrics can be found if sought by envisioning the matrix as a linear transformation. Computing them is complicated, but the function `det()` will do it. An ill-conditioned matrix will not return a sensible result for the determinant.

- Just like with vectors, matrices have a *norm* associated with them. Any order *induced norm* (2-norm in example), as well as the *Frobenius norm* (using `'fro'`), may be found using `norm()`. Again, interpretation of these ideas is way beyond the scope of this workshop.

- Every real matrix has a unique *reduced row echelon form (RREF)* found using `rref()` found using Gaussian elimination that encodes how to solve a system of linear equations. Recall that RREF is useful for *computers*, it requires an extra round of calculations to find, meaning that **it is slower than matrix left division \ and is an indication of computer help on homework**.

- A square matrix is *invertible* if it is full rank. Mathematically, the inverse of a matrix $\mathbf{A}$ is found $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$, where $\mathbf{A}^{-1}$ represents the inverse matrix and $\mathbf{I}$ represents the identity matrix (for matrix multiplication). The function `inv()` finds this inverse matrix. **The inverse matrix should only be found this way for well-conditioned matrices that are used to solve a matrix equation for several inputs.** Use matrix left division if you can.

- The *Fundamental Theorem of Linear Algebra* and *rank-nullity theorem* that follows from it govern the relationships between the *vector subspaces* of a matrix. The two fundamental subspaces getting the most attention are the *column space* (also called *range* or *image* of a matrix_) and the *nullspace* (also called *kernal* of a matrix). An orthonormal basis (direction vectors of unit magnitude all normal to each other) for these subspaces may be found using `orth()` and `null()`, respectively.

- There are several matrix decompositions that MATLAB has available. (https://en.wikipedia.org/wiki/Matrix_decomposition) Each one has multiple outputs, each one being a matrix factor. The *LUP decomposition*$\mathbf{PA} = \mathbf{LU}$ is useful when solving systems of linear equations since $\mathbf{L}$ and $\mathbf{U}$ summarize the steps in Gaussian elimination and make subsequent solves easier. $\mathbf{P}$ shows how the matrix rows need to be permuted to make the decomposition work and helps make the algorithm more stable.

- The *QR* decomposition helps solve systems of linear equations numerically since the orthogonal matrix $\mathbf{Q}$ satisifies $\mathbf{Q}^{\mathrm{T}}\mathbf{Q} = I$ (the transpose is equal to the inverse). Thus, $\mathbf{A} = \mathbf{QR}$ in $\mathbf{A}\vec{x} = \vec{b}$ can be solved more easily as $\mathbf{R}\vec{x} = \mathbf{Q}^{\mathrm{T}}\vec{b}$, often as an intermediate step in other algorithms.

- MATLAB finds eigenvalues and associated right eigenvectors a lot faster than you can using `eig()`. The first output is a matrix of right eigenvectors of magnitude 1 in each column. The second output is a diagonal matrix with the associated eigenvalues in the same place. This *eigendecomposition* (or *spectral decomposition*) is $\mathbf{A} = \mathbf{VDV}^{-1}$. Unfortunately, enough things in engineering use eigenvalues to make this function used in practice.

Linear algebra has plenty of complicated ways to describe rectangular (not square) matrices, including generalizations of spectral decomposition to singular value decomposition `svd()`. These lead to a generic QR decomposition and way to calculate condition number and norms. Even the idea of inverse matrix is generalized into the Moore-Penrose pseudoinverse `pinv()`. Investigate these functions as well as those for complex matrices at your own urging/pace/risk.

Dates

MATLAB has some powerful tools available for representing and manipulating time. Dates and times can be represented as numeric arrays, strings, or the Java *datetime* object. With a *datetime* object, there are various options for display, performing date math, comparing dates, and finding durations (https://www.mathworks.com/help/matlab/date-and-time-operations.html).

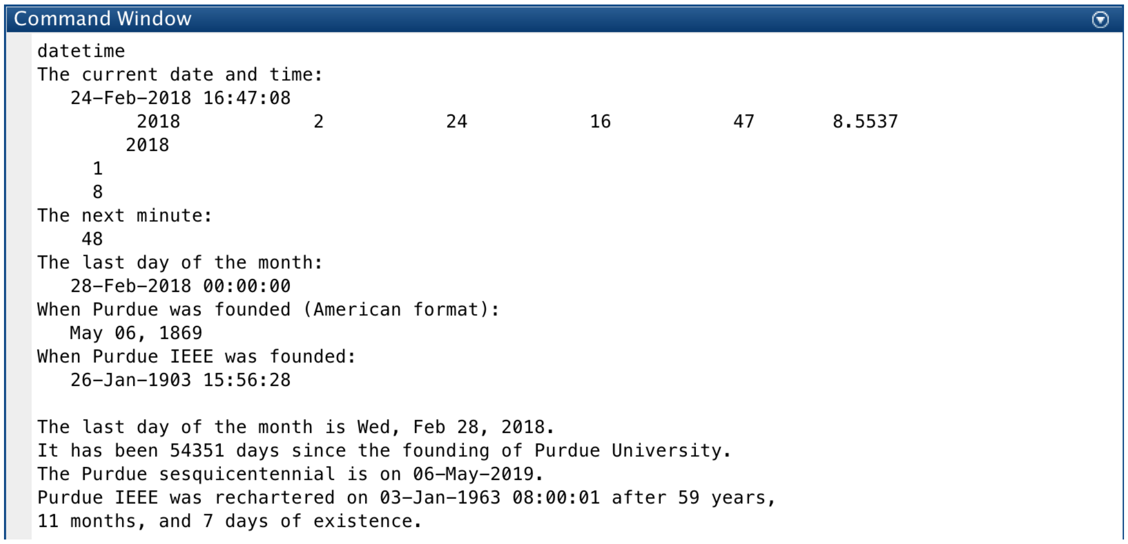
Ensure that all active lines below the "Exploration" section in "workshop2.m" are commented out, by pressing the green percent sign "Comment" (Ctrl + R in Windows) in the "Edit" section of the "Editor" ribbon or some other means. Go to a blank line below the commented out script. It is time to explore **dates and times**.

```
timestamp_now = datetime;
timestamp_update = timestamp_now.Minute + 1;
month_end = dateshift(timestamp_now, 'end', 'month');
[~, weekday_month_end] = weekday(month_end);

found_Purdue = datetime(1869, 05, 06);
found_Purdue.Format = 'MMM dd, yyyy';
time_since_found = days(timestamp_now - found_Purdue);
sesquicentennial = calyears(150);
sesqui_Purdue = found_Purdue + sesquicentennial;

found_IEEE = datetime(1903, 01, 26, 15, 56, 28);
recharter_IEEE = datetime(1963, 01, 03, 08, 00, 01);
[years_between, months_between, days_between] = ...
```

```
        split(between(found_IEEE, recharter_IEEE), ...
        {'years', 'months', 'days'});

    disp(class(timestamp_now))
    disp('The current date and time:')
    disp(timestamp_now)
    disp(datevec(timestamp_now))
    disp(year(timestamp_now))
    disp(quarter(timestamp_now))
    disp(week(timestamp_now))
    disp('The next minute:')
    disp(timestamp_update)
    disp('The last day of the month:')
    disp(month_end)
    disp('When Purdue was founded (American format):')
    disp(found_Purdue)
    disp('When Purdue IEEE was founded:')
    disp(found_IEEE)

    fprintf(1, '\nThe last day of the month is %s, %s.\n', ...
        weekday_month_end, datestr(month_end, 'mmm dd, yyyy'));
    fprintf(1, ['It has been %1.0f days since the founding of ' ...
        'Purdue University.\n'], time_since_found);
    fprintf(1, 'The Purdue sesquicentennial is on %s.\n', ...
        datestr(sesqui_Purdue));
    fprintf(1, ['Purdue IEEE was rechartered on %s after ' ...
        '%1.0f years,\n%1.0f months, and %1.0f days of ' ...
        'existence.\n\n'], datestr(recharter_IEEE), years_between, ...
        months_between, days_between);
```

MATLAB Comprehension Questions: Why does MATLAB need the `datetime` class instead of just saving dates as `double`? How much work would it be to find out things such as month or day of the week without using built-in functions? Which is harder: finding the duration between two arbitrary events or finding the later event given the starting event and a duration?

Run the script by pressing the green "Run" button (F5 in Windows). Let's make some observations on **dates, times, durations, and calendar durations**:


Command Window

```
datetime
The current date and time:
   24-Feb-2018 16:47:08
         2018            2           24           16           47       8.5537
        2018
     1
     8
The next minute:
    48
The last day of the month:
   28-Feb-2018 00:00:00
When Purdue was founded (American format):
   May 06, 1869
When Purdue IEEE was founded:
   26-Jan-1903 15:56:28

The last day of the month is Wed, Feb 28, 2018.
It has been 54351 days since the founding of Purdue University.
The Purdue sesquicentennial is on 06-May-2019.
Purdue IEEE was rechartered on 03-Jan-1963 08:00:01 after 59 years,
11 months, and 7 days of existence.
```

- There are 3 classes of variables designed to handle dates: `datetime`, `duration`, and `calendar-Duration`. A `datetime` will represent a single instance in time from the year down to seconds with three places after the decimal point. A `duration` represents the time elapsed between two instances of time from the year down to seconds with three places after the decimal point. A `calendarDuration` represents the dates elapses between two instances of time from the year down to the day as the calendar would measure it.

- The powerful `datetime()` function without any arguments returns a timestamp in the `datetime` class with the current instant of time.

- The `datetime` class acts like a *structure* (more on those later) that can be updated by calling *fields* such as `.Minute` or `.Day`.

- Addition and subtraction are allowed for `datetime` components in order to modify timestamps.

- The function `datevec()` converts a `datetime` variable into a date row vector of class `double` with each component representing year, month, day, hour (military time), minute, and seconds respectively.

- With arguments, `datetime()` creates a variable by interpreting each argument as a year, month, and day with possible extra arguments interpreted as hour, minute, and second. Alternatively, `datetime()` can take a valid date row vector to produce a `datetime` variable.

- There are stand-alone functions such as `year()` and `quarter()` that reveal the explicit and hidden parts of the `datetime` variable. Search through the MATLAB page on dates and times to find the function that is right for your needs.

- The function `dateshift()` takes in a `datetime` variable and produces a `datetime` variable for some relational instance of time that is typically easier to explain in language rather than mathematically. For example, `dateshift()` can determine the start of next week, the end of the month, the third Thursday of each month, and the day of the start of the current quarter.

- The function `weekday()` works on a `datetime` variable to produce a number for the day of the week (1 = Sunday, 7 = Saturday) and a string representing the same things. Additional arguments control the language of the string output and the use of abbreviations or the full word.

- Any `datetime` variable can be represented as a string using the `datestr()` function. A date string is the default way that any `datetime` variable is displayed, but formatted printing with `fprintf()` requires calling `datestr()` first.

- The format of a date string defaults to `` `dd-mmm-yyyy hh:MM:ss' `` where each `y` is a character for the year (2 or 4), `QQ` is for Quarter # (only works alone or with year), each `m` (capital `M` may be used for a calendar duration) is a character for the month (4 for full word, 3 for abbreviation, 2 for two-digit number, and 1 for first letter), each `d` is a charcter for the day (similar decoding as for months), `HH` or `hh` is for hour, `MM` is for minute, `SS` or `ss` is for whole seconds, `FFF` or `fff` is for milliseconds, and `AM` or `PM` use meridian labels instead of military time.

- The format of a date string may be changed with the `.Format` field of a `datetime` variable or as a second argument to `datestr()`.

- A `duration` variable is created by adding or subtracting two `datetime` variables. Functions such as `days()` or `milliseconds()` extract that information as a single number of class `double`.

- A `calendarDuration` variable is created by using functions such as `calyears()` or `caldays()` with regular numbers passed to them. Adding or subtracting two `calendarDuration` variables produces another `calendarDuration` variable with the net calendar duration. Adding or subtracting a `calendarDuration` variable from a `datetime` variable produces a `datetime` variable corresponding to the new calendar date.

- The function `between()` acts as the inverse operation to the latter case of `datetime + calendarDuration = datetime`. By passing the earlier `datetime` variable and then the later `datetime` variable, the function `between()` finds the `calendarDuration` represented.
- The function `split()` takes a `calendarDuration` variable and produces scalar `double` outputs based on the desired increments of calendar dates. Multiple output can be specified by having the second argument be a *cell array* (more on those later) of calendar date increments.

Even though the MATLAB tools for working with dates and times are very powerful, it can be quite confusing to remember which class of input `double`, `char`, `datetime`, `duration`, or `calendarDuration` each function accepts or which function is appropriate for the task. Complicating things, there are usually two or three ways to achieve the same result using entirely different functions and representations. MathWorks is actively expanding the handling and number of functions that operate on dates and times, so check online to see which options are best for your application and version of MATLAB.

Strings

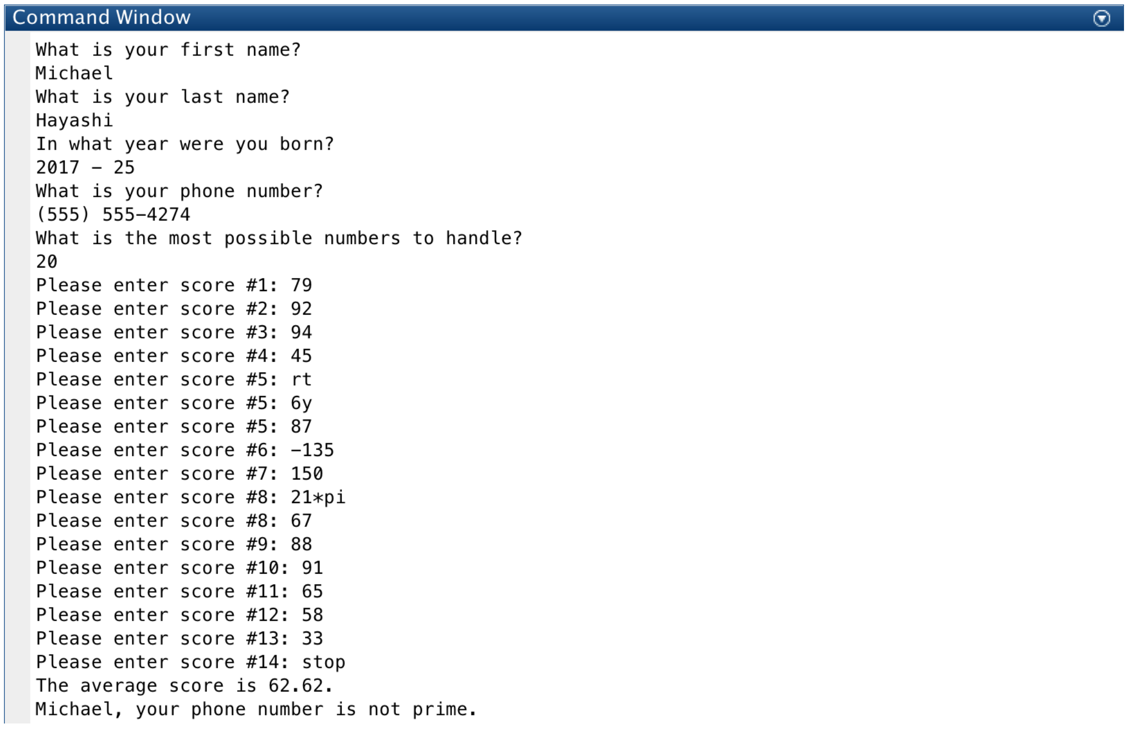
The Introduction to MATLAB Workshop covered the use of `fprintf()` and `sprintf()` for printing formatted strings to files or output strings using the *format specification* syntax (referring to https://www.mathworks.com/help/matlab/matlab_prog/formatting-strings.html is helpful). It is time to explore strings in more depth.

Ensure that all active lines below the "Exploration" section in "workshop2.m" are commented out, by pressing the green percent sign "Comment" (Ctrl + R in Windows) in the "Edit" section of the "Editor" ribbon or some other means. Go to a blank line below the commented out script. It is time to explore **string inputs** and **string functions**.

```
first_name = input('What is your first name?\n', 's');
last_name = input('What is your last name?\n', 's');
birth_year = input('In what year were you born?\n');
phone_number = input('What is your phone number?\n', 's');

N_max = input('What is the most possible numbers to handle?\n');
end_early_flag = false;
scores = nan(N_max, 1);
for indi = 1:N_max
    if end_early_flag
        break
    end
    while true
        num_input = input(sprintf('Please enter score #%d: ', ...
            indi), 's');
        if strcmp(num_input, 'stop')
            end_early_flag = true;
            break
        end
        if isfinite(str2double(num_input))
            scores(indi) = str2double(num_input);
            break
        end
    end
end

        phone_digits = double(phone_number);
        phone_digits((phone_digits < 48) || (phone_digits > 57)) = '';
        phone_digits = str2double(char(phone_digits));
        switch isprime(phone_digits)
            case 1
                prime_str = 'is';
            case 0
                prime_str = 'is not';
        end

        fprintf(1, 'The average score is $1.2f.\n', nanmean(scores));
        fprintf(1, '%s, your phone number %s prime.\n', first_name, prime_str);
```

MATLAB Comprehension Questions: When would problems arise from manually entering data using the *Variable Editor* by deliberately creating a new variable in the *Workspace*? What types of error checking should you perform? Which situation is more appropriate for MATLAB: demanding number crunching with incidental text processing or demanding text comprehension with incidental numeric processing? Do comments have a role in explaining text comprehension even more than number crunching?

Run the script by pressing the green "Run" button (F5 in Windows). Let's make some observations on **string inputs and string processing**:



Command Window

```
What is your first name?
Michael
What is your last name?
Hayashi
In what year were you born?
2017 - 25
What is your phone number?
(555) 555-4274
What is the most possible numbers to handle?
20
Please enter score #1: 79
Please enter score #2: 92
Please enter score #3: 94
Please enter score #4: 45
Please enter score #5: rt
Please enter score #5: 6y
Please enter score #5: 87
Please enter score #6: -135
Please enter score #7: 150
Please enter score #8: 21*pi
Please enter score #8: 67
Please enter score #9: 88
Please enter score #10: 91
Please enter score #11: 65
Please enter score #12: 58
Please enter score #13: 33
Please enter score #14: stop
The average score is 62.62.
Michael, your phone number is not prime.
```

- Under most circumstances, any software you write should attempt to **minimize** any potential interaction with users. Instead of requiring mid-executation commands, have the user provide execution options as extra arguments. Instead of having data be input and interpreted on-the-fly, have the user input a file instead for low-level I/O. However, there are rare circumstances where it **makes sense** to have the user interact with a script while it is executing.

- The `input()` function is the **only** recommended way to create an interactive script. The first argument is a string that appears as a prompt on the *Command Window*. Be sure to use the newline character `'\n'`, other whitespace characters, and escaped backslashes `'\\'` as needed. The second argument is an `'s'` optionally included if the user input is to be interpreted as a string (`char` array) instead of a numeric value (`double`).

- When not interpreting what the user types as a string, `input()` will return an empty `double` array if the enter/return key is pressed without typing anything else. Otherwise, MATLAB will perform any numeric simplification possible (e.g., 2017 - 25 = 1992) before storing the value to the assigned variable in memory. *This is where a host of errors can occur.*

- Error checking is a must when interacting with user input. You might think your prompt is obvious and simple, but assume that the user is dumb. For example, what will you do if the user types in `3 + tree`? Everybody will get frustrated when the software encounters these scenarios and produces an error.

- On the other hand, there are a few situations where you can circumvent bad behavior more easily using `input()` than you might without any other types of checking. For example, copying and pasting data might bring along invisible characters that `importdata()` may not like or that the *Variable Editor* will just skip over and erroneously pad with zeros.

- The score entering tool in the exploratory example has 4 types of error checking: maximum array length, early termination before maximum length reached, consecutive valid values (NaN for remainder), and corrected statistical operations.

- The user must supply a maximum number of entries. This prevents the script from running forever or a malicious user from gaining access to system memory in an attack. If the script does run forever, cancel execution with Ctrl + C. If the user does something dumb here, an error will be produced when trying to save a value to `N_max`, and the program has already been defeated.

- The `end_early_flag` is a `logical` variable that is supposed to break out of the incremental assignment of variables to the array `scores` if the value becomes `true`. The value is going to become `true` if and only if the user types in `stop`.

- Rather than using relational operators such as == or ~=, strings are compared using the function `strcmp()`. If the two strings passed to the function are identical, then `strcmp()` returns `true`. Otherwise, the function returns `false`.

- Any number can be represented as a string using `num2str()` for scalars. Vectors, matrices, and arrays can be reprsented as strings using `mat2str()`. The converse statement is not necessarily true. There are obviously some strings such as `'3 + tree'` that are not able to be represented numerically. In general, any double-precision number (real or complex) that could be produced by `sprintf()` is able to be converted back into a number using `str2double()`. The function `str2num()` is only meant to be used to produce double-precision *arrays*. It will evaluate spaces as horizontal concatenation and perform any numeric evaluation of special functions necessary, so be cautious.

- The function `isfinite()` simply checks if the variable of class `double` is not `NaN` (result if `str2double()` cannot interpret what the user typed), `Inf`, or `-Inf`. If the user did type something nonsensical, then the program will continually prompt what the score in that position *should* be.

- The function `isfinite()` is part of a big family of functions that return a `logical` based on the state of some other variable. More information on this family may be found at https://www.mathworks.com/help/matlab/ref/is.html. Some of the most used are `ischar()`, `isdiag()`, `isempty()`, `isequaln()`, `isfield()`, `isfile()`, `isnan()`, `isnumeric()`, `isreal()`, and `isvector()`. Some of my favorites are `isdst()`, `iskeyword()`, `ismember()`, `ismatrix()`, `isprime()`, `issorted()`, `isspace()`, `isstudent()`, `issymmetric()`, and `isweekend()`.

- The line `phone_digits = double(phone_number);` saves the array of characters in the phone number as an array of double-precision numbers of each character's ASCII value. (The ASCII values are http://www.asciitable.com/, though any Unicode value could be produced that depends on the region of the computer and its operating system https://unicode-table.com/en/.)

- The line `phone_digits((phone_digits < 48) |` (phone_digits > 57)) = '';| sets any elements of the array that do not correspond to the ASCII values of 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9 to be the empty character `''`. This way, only the digits of the phone number are retained for a wide variety of phone number representations.

- The line `phone_digits = str2double(char(phone_digits));` converts the ASCII `double` values back to ASCII `char` in a string. Then, the string is interpreted as a `double` based off the displayed characters. *Pause and ponder this to understand how there can be multiple number associated with a string made of 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.*

- When discussing statistics in the Introduction to MATLAB Workshop, it was claimed that not-a-number `NaN` is the enemy of data collection. In this example, we need to distinguish `0` entered by the user as a legitimate score versus the default value of `NaN` for an entry the user did not overwrite. The function `nanmean()` ignores any not-a-number values when calculating arithmetic mean. The alternative handling would be to resize `scores` based off valid entries.

As you can see, there are still ways the user can force the program into an error. Error catching is virtually unending, even if the time you can devote to programming must be finite. More advanced string processing can be done with `strfind()` to find a substring inside a larger string, `strrep()` to replace one substring in a larger string with another substring, or `regexp()` to match arbitrarily complicated patterns within a string. MATLAB has all the features of other programming languages to work with strings, but it is hardly the first choice for this purpose.

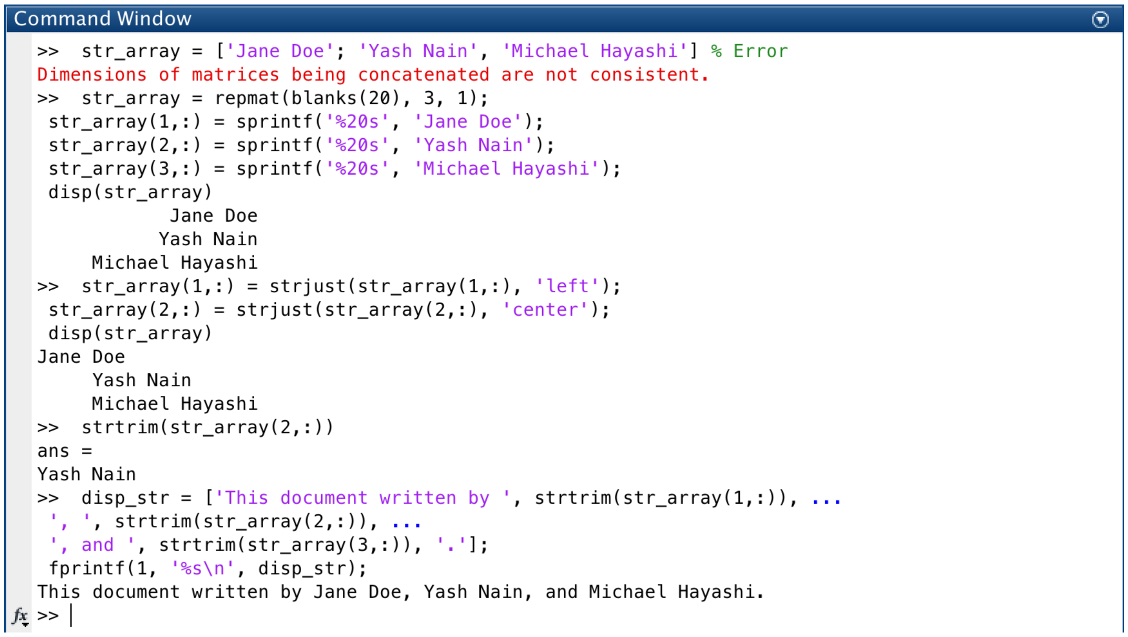
Advanced Collection Data Types: Numeric Array Limitations

By this point, virtually the entire power of arrays within MATLAB is at your command. *Collection data types* are a data type in a programming language meant to store multiple values. Arrays, including vectors and matrices, are the most intuitive way to work with multiple values of class `double`, `single`, `uint32`, `logical`, `datetime`, or similar data types with a clear numeric representation.

Here is a quick example for the MATLAB *Command Window* to show you where arrays are limited in what they can do:

```
str_array = ['Jane Doe'; 'Yash Nain', 'Michael Hayashi'] % Error
str_array = repmat(blanks(20), 3, 1);
str_array(1,:) = sprintf('%20s', 'Jane Doe');
str_array(2,:) = sprintf('%20s', 'Yash Nain');
str_array(3,:) = sprintf('%20s', 'Michael Hayashi');
disp(str_array)
str_array(1,:) = strjust(str_array(1,:), 'left');
str_array(2,:) = strjust(str_array(2,:), 'center');
disp(str_array)
strtrim(str_array(2,:))
disp_str = ['This document written by ', strtrim(str_array(1,:)), ...
', ', strtrim(str_array(2,:)), ...
', and ', strtrim(str_array(3,:)), '.'];
```

```
        fprintf(1, '%s\n', disp_str);
```



```
>>  str_array = ['Jane Doe'; 'Yash Nain', 'Michael Hayashi'] % Error
Dimensions of matrices being concatenated are not consistent.
>>  str_array = repmat(blanks(20), 3, 1);
 str_array(1,:) = sprintf('%20s', 'Jane Doe');
 str_array(2,:) = sprintf('%20s', 'Yash Nain');
 str_array(3,:) = sprintf('%20s', 'Michael Hayashi');
 disp(str_array)
            Jane Doe
           Yash Nain
     Michael Hayashi
>>  str_array(1,:) = strjust(str_array(1,:), 'left');
 str_array(2,:) = strjust(str_array(2,:), 'center');
 disp(str_array)
Jane Doe
     Yash Nain
     Michael Hayashi
>>  strtrim(str_array(2,:))
ans =
Yash Nain
>>  disp_str = ['This document written by ', strtrim(str_array(1,:)), ...
 ', ', strtrim(str_array(2,:)), ...
 ', and ', strtrim(str_array(3,:)), '.'];
 fprintf(1, '%s\n', disp_str);
This document written by Jane Doe, Yash Nain, and Michael Hayashi.
>> |
```

- Every string in MATLAB is a horizontal array (row vector) of class `char`, each character capable of being indexed. However, there are times where it is valuable to treat each string as a fundamental unit and store them collectively.

- MATLAB will throw an error if you try to vertically concatenate strings of different lengths, frustrating our attempt to store them together.

- The `blanks()` function is similar to using `zeros()` to initialize a row vector to a known length. By using `repmat()`, the two-dimensional `char` array is guaranteed to be rectangular and contain just whitespaces.

- In order to ensure that `str_array` is space-padded, each row vector in `str_array` is created with formatted printing through `sprintf()`.

- The right-justified `str_array` is somewhat ugly with a variable amount of whitespaces for each row. The justification of the text in the whitespaces can be changed with `strjust()`, but this still leaves a lot to be desired.

- Strings are saved only for some sort of comprehension or to display later. A function such as `strtrim()` or `deblank()` is needed to remove the *leading and trailing* whitespaces. The `char` variable `disp_str` horizontally concatenates string literals with rows from `str_array` for formatted printing.

MATLAB Comprehension Questions: What would happen if a name occupied more than 20 characters? What prevents us from indexing a leading or trailing whitespace character within a row? Is there a way to distinguish padded whitespace from significant whitespace?

Advanced Collection Data Types: Cell Arrays

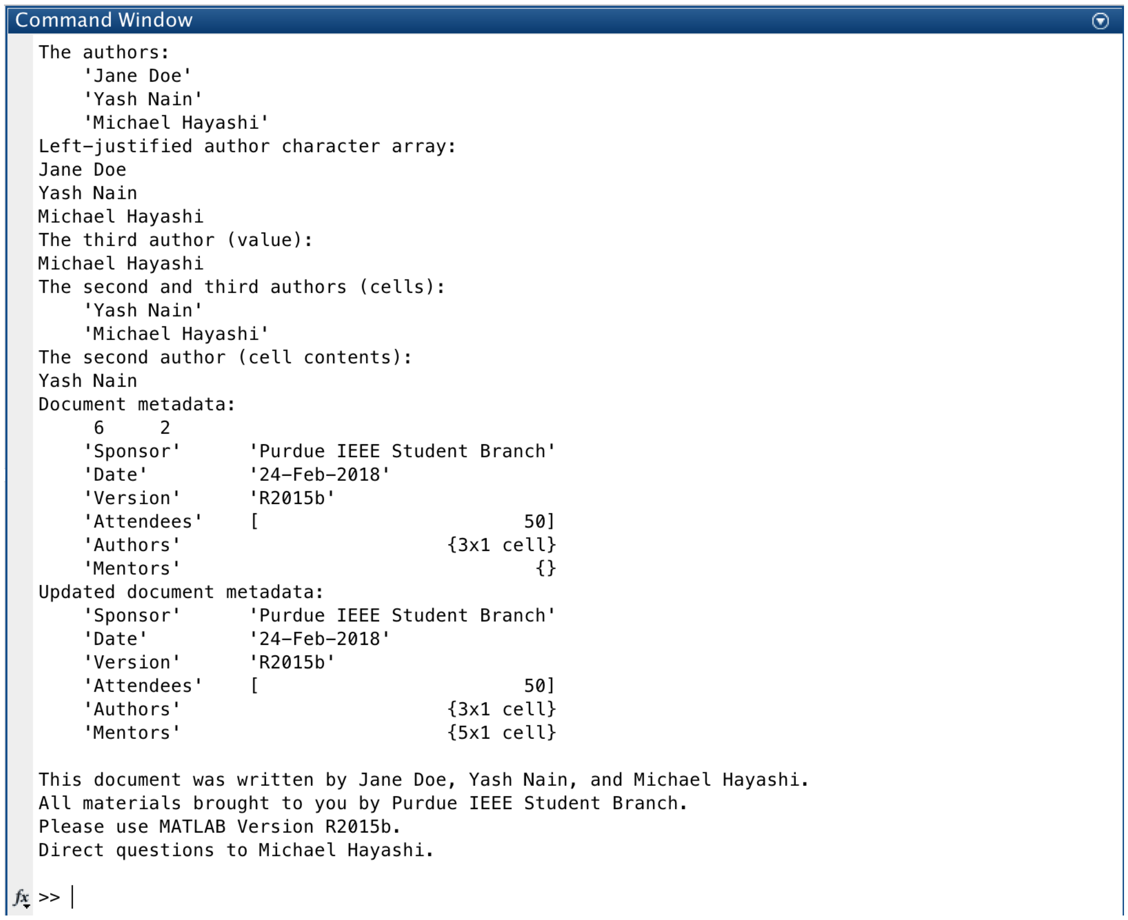
A *cell array* (https://www.mathworks.com/help/matlab/cell-arrays.html)is a collection data type of class `cell` meant to contain varying types of data and/or data of the same type but varying sizes. In essence, it can contain nearly any literal that could be typed into a cell within a Microsoft Excel spreadsheet. Not only can strings of different lengths be stored together, but data of type `char`, `function_handle`, `logical`, and `double` could all be mixed as an example.

Ensure that all active lines below the "Exploration" section in "workshop2.m" are commented out, by pressing the green percent sign "Comment" (Ctrl + R in Windows) in the "Edit" section of the "Editor" ribbon or some other means. Go to a blank line below the commented out script. It is time to explore **cell arrays**.

```
names = {'Jane Doe'; 'Yash Nain'; 'Michael Hayashi'};
str_array = char(names);
[name1, name2, name3] = deal(names{:});
document_info = {'Sponsor', 'Purdue IEEE Student Branch'; ...
    'Date', date; 'Version', 'R2015b'; 'Attendees', 50; ...
    'Authors', names; 'Mentors', cell(0, 0)};

disp('The authors:')
disp(names)
disp('Left-justified author character array:')
disp(str_array)
disp('The third author (value):')
disp(name3)
disp('The second and third authors (cells):')
disp(names(2:3))
disp('The second author (cell contents):')
disp(names{2})
disp('Document metadata:')
disp(size(document_info))
disp(document_info)

document_info{6,2} = {names{2}; names{end}; 'Albert Xu'; ...
'Nataliia Perova-Mello'; 'Ivan Ostroumov'};
disp('Updated document metadata:')
disp(document_info)

fprintf(1, ['\nThis document was written by %s, %s, and %s.\n' ...
    'All materials brought to you by %s.\n' ...
    'Please use MATLAB %s %s.\n' ...
    'Direct questions to %s.\n\n'], ...
    name1, strtrim(char(names(2))), names{3}, document_info{1,2}, ...
    document_info{3,1}, document_info{3,2}, document_info{5,2}{3});
```

MATLAB Comprehension Question: What is the difference between indexing with parentheses `()` and with curly braces `{}` for cell arrays?

Run the script by pressing the green "Run" button (F5 in Windows). Let's make some observations on **cell array creation and indexing**:



```
The authors:
    'Jane Doe'
    'Yash Nain'
    'Michael Hayashi'
Left-justified author character array:
Jane Doe
Yash Nain
Michael Hayashi
The third author (value):
Michael Hayashi
The second and third authors (cells):
    'Yash Nain'
    'Michael Hayashi'
The second author (cell contents):
Yash Nain
Document metadata:
     6     2
    'Sponsor'      'Purdue IEEE Student Branch'
    'Date'         '24-Feb-2018'
    'Version'      'R2015b'
    'Attendees'    [                        50]
    'Authors'                        {3x1 cell}
    'Mentors'                                {}
Updated document metadata:
    'Sponsor'      'Purdue IEEE Student Branch'
    'Date'         '24-Feb-2018'
    'Version'      'R2015b'
    'Attendees'    [                        50]
    'Authors'                        {3x1 cell}
    'Mentors'                        {5x1 cell}

This document was written by Jane Doe, Yash Nain, and Michael Hayashi.
All materials brought to you by Purdue IEEE Student Branch.
Please use MATLAB Version R2015b.
Direct questions to Michael Hayashi.

>>
```

- *Cell array* literals are created by placing literals of any data type or variables between curly braces `{}`.
- Comma-separated or space-separated elements places elements along a row. Semicolo-separated elements put elements on a new column.
- The three important functions that measure arrays also work for cell arrays: `length()`, `numel()`, and `size()`.
- Transposition also works for cell arrays, so using `.'` or `transpose()` is an option.
- Arithmetic operations **do not** work for cell arrays, so do not attempt to use +, \, or ^ with them. Relational operators such as <= and logical operators such as ~ or && are alos not defined.
- Cell arrays are able to handle different lengths of `char` arrays as seen with the variable `names`. Data types can also be mixed in cell arrays as seen with the variable `document_info`.
- Should the need ever arise, it is possible to use `char()` to go from a cell array of character vectors to the whitespace-padded `char` array.
- The function `deal()` takes a collection data type and assigns each output in square brackets each element of the collection data type indexed. Alternatively, is assigns all outputs the entire collection data type if the input is not indexed.
- An cell array *of empty cells* (each element within is size zero) can be initialized with `cell()`. An *empty cell array* (the cell array itself is size zero) is made with `cell(0)`.

- There are two ways to index cell arrays based off the desired output (https://www.mathworks.com/help/matlab/matlab_prog/access-data-in-a-cell-array.html). The colon operator `:` is used by itself to represent the entire range of indices or for *slicing* just as for arrays. A vector of indices is used for noncontiguous *slicing*, and the `end` keyword also works. The *striding* syntax of `start:stride_pitch:stop` and *logical indexing* also works for cell arrays.

- The typically less useful way to index a cell array is to use parentheses `()` just like with arrays to produce a **cell array subset**. Functions such as `char()` and `cell2mat()` are capable of representing the cell array output as a `char` array or `double` array, respectively.

- The more common way to index a cell array is to use curly braces `{}` to produce a number of **content** outputs. Make sure that there is a separate output to handle every cell whose contents are being indexed!

- It is easy to concatenate cell arrays by including the variable name as anv element between curly braces `{}`. The alternative is to use the function `vertcat()` to concatenate cell arrays vertically, the function `horzcat()` to concatenate cell arrays horizontally, and `cat()` to concatenate in any dimension as long as the first argument is the dimension to concatenate along.

- If the **cell array subset** with parentheses `()` for indexing is used for assignment, then *each cell* in the target will be overwritten by *each cell* of the **cell array** given.

- If the **contents** with curly braces `{}` for indexing is used for assignment, then *the contents* in the target location will be overwritten with **some type of data** given having the same size.

- Unlike with numerical arrays, cell arrays may have *arbitrarily sized* cell arrays of class `cell` within them as elements.

- Cell arrays uniquely allow for **multilevel indexing** (https://www.mathworks.com/help/matlab/matlab_prog/multilevel-indexing-to-access-parts-of-cells.html). This means that it is possible to access the contents of a *given cell* in a cell array and further access the contents *of the contents* without having to save the intermediate level of contents to its own variable. For example, `document_info{6,2}{3}` refers to the third character vector saved at the cell in the sixth row and second column. Furthermore `document_info{6,2}{3}(1)` refers to the first character in that character vector.

- The function `celldisp()` exists to recursively display the contents at each element of a cell array, but the *Command Window* becomes so cluttered even for modestly sized cell arrays that it is practically worthless.

- On the other hand, `cellplot()` is useful for understanding the data types contained in each cell array.

Because cell arrays can contain any type of array within, the possibilities are virtually limitless. It is up to you to organize your cell arrays and keep track of the data types contained within each cell. Just be careful with the indexing.

Advanced Collection Data Types: Tables

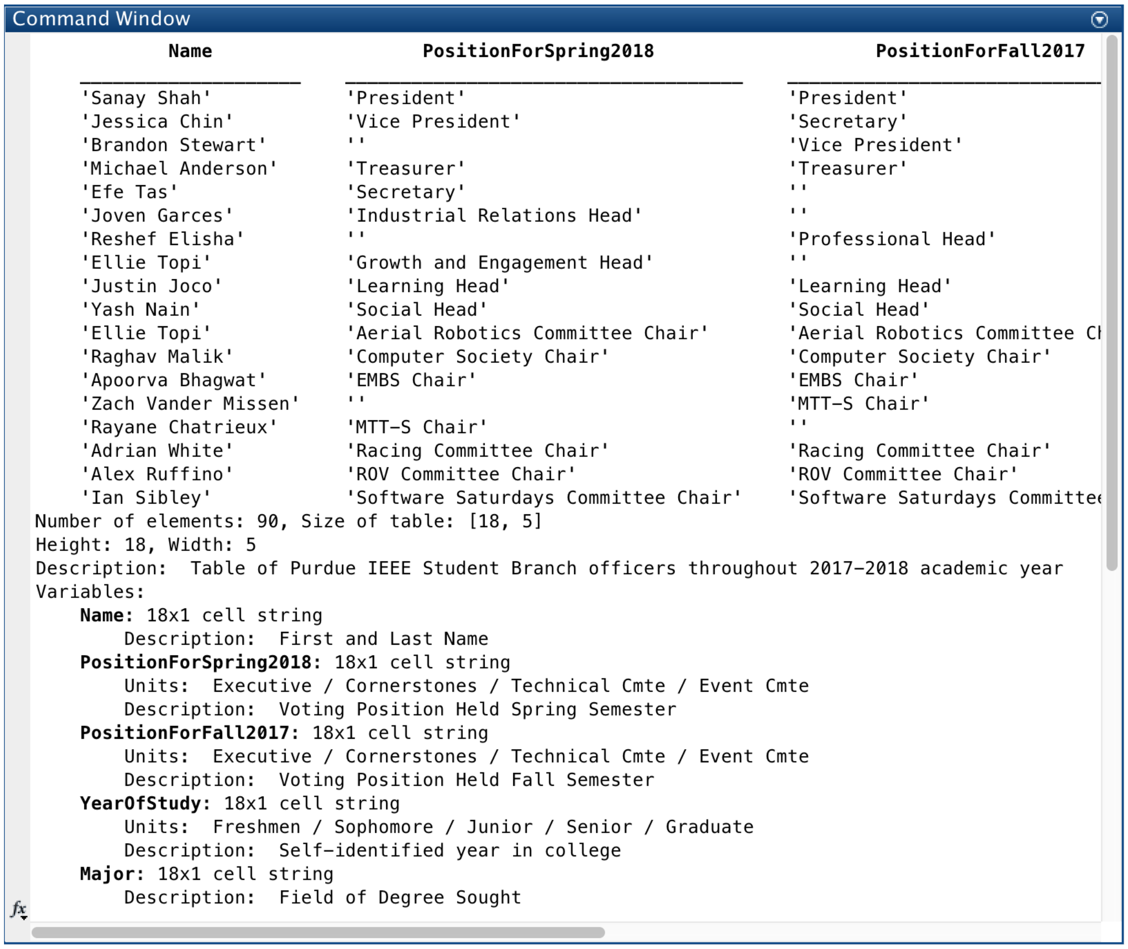
Cell arrays can handle arbitrary collections of data, so there would be no need to introduce more collection data types beyond them in theory. However, most data collections encountered in practice are not arbitrary. *Tables* (https://www.mathworks.com/help/matlab/tables.html) are a relatively recent data type (since MATLAB R2013b) that MathWorks introduced to handle data where each *column* represents a similar thing (again, MATLAB strongly prefers column-based organization). Unlike cell arrays, tables are particularly optimized to handle tens of thounsands of rows of data.

Ensure that all active lines below the "Exploration" section in "workshop2.m" are commented out, by pressing the green percent sign "Comment" (Ctrl + R in Windows) in the "Edit" section of the "Editor" ribbon or some other means. Go to a blank line below the commented out script. It is time to explore **tables**.

```
officers_IEEE_2017 = readtable('IEEE_officers.csv');
officers_cell = table2cell(officers_IEEE_2017);
officers_struct = table2struct(officers_IEEE_2017);
officers_IEEE_2017.Properties.Description = ['Table of Purdue IEEE ' ...
    'Student Branch officers throughout 2017-2018 academic year'];
officers_IEEE_2017.Properties.VariableDescriptions = ...
    {'First and Last Name', 'Voting Position Held Spring Semester', ...
    'Voting Position Held Fall Semester', ...
    'Self-identified year in college', 'Field of Degree Sought'};
officers_IEEE_2017.Properties.VariableUnits{'PositionForSpring2018'} ...
    = 'Executive / Cornerstones / Technical Cmte / Event Cmte';
officers_IEEE_2017.Properties.VariableUnits{'PositionForFall2017'} ...
    = 'Executive / Cornerstones / Technical Cmte / Event Cmte';
officers_IEEE_2017.Properties.VariableUnits{'YearOfStudy'} = ...
    'Freshmen / Sophomore / Junior / Senior / Graduate';
officers_IEEE_2017.Properties.DimensionNames{1} = 'Voting Member';

disp(officers_IEEE_2017)
fprintf(1, ['Number of elements: %1d, Size of table: [%1d, %1d]\n' ...
    'Height: %1d, Width: %1d\n'], numel(officers_IEEE_2017), ...
    size(officers_IEEE_2017), height(officers_IEEE_2017), ...
    width(officers_IEEE_2017));
summary(officers_IEEE_2017)

officers_IEEE_2017.ActiveYears = [4; 2; 3; 2; 1; 1.5; 1; 2; 2.5; 4; ...
    2; 2; 3; 4.5; 1.5; 2; 3; 2];
officers_IEEE_2017.Properties.VariableDescriptions{6} = ...
    'Number of Years in Purdue IEEE Student Branch';
officers_IEEE_2017.Properties.VariableUnits{6} = 'Years';
active_S2018 = sortrows(officers_IEEE_2017, [6, 1], ...
    {'descend', 'ascend'});
gone_Spring2018 = strcmp(active_S2018.PositionForSpring2018, '');
active_S2018(gone_Spring2018,:) = [];
active_S2018.Properties.Description = ['Table of Purdue IEEE ' ...
    'Student Branch officers for Spring 2018 by active years'];

fprintf(1, '\n\n');
summary(active_S2018)
writetable(active_S2018, 'IEEE_officer_seniority.csv')
```

MATLAB Comprehension Questions: When can a table be made into a cell array and vice versa? What sort of information is contained in the **Properties** of a table? Are the **Properties** retained when using `writetable()` and `readtable()`? When is it more convenient to reference a table variable by name versus by index number?

Run the script by pressing the green "Run" button (F5 in Windows). Let's make some observations on **table appearance and information**:



```
                 Name                    PositionForSpring2018                     PositionForFall2017
        ____________________    ____________________________________    ____________________________
        'Sanay Shah'            'President'                             'President'
        'Jessica Chin'          'Vice President'                        'Secretary'
        'Brandon Stewart'       ''                                      'Vice President'
        'Michael Anderson'      'Treasurer'                             'Treasurer'
        'Efe Tas'               'Secretary'                             ''
        'Joven Garces'          'Industrial Relations Head'             ''
        'Reshef Elisha'         ''                                      'Professional Head'
        'Ellie Topi'            'Growth and Engagement Head'            ''
        'Justin Joco'           'Learning Head'                         'Learning Head'
        'Yash Nain'             'Social Head'                           'Social Head'
        'Ellie Topi'            'Aerial Robotics Committee Chair'       'Aerial Robotics Committee Cl
        'Raghav Malik'          'Computer Society Chair'                'Computer Society Chair'
        'Apoorva Bhagwat'       'EMBS Chair'                            'EMBS Chair'
        'Zach Vander Missen'    ''                                      'MTT-S Chair'
        'Rayane Chatrieux'      'MTT-S Chair'                           ''
        'Adrian White'          'Racing Committee Chair'                'Racing Committee Chair'
        'Alex Ruffino'          'ROV Committee Chair'                   'ROV Committee Chair'
        'Ian Sibley'            'Software Saturdays Committee Chair'    'Software Saturdays Committe
    Number of elements: 90, Size of table: [18, 5]
    Height: 18, Width: 5
    Description:  Table of Purdue IEEE Student Branch officers throughout 2017-2018 academic year
    Variables:
        Name: 18x1 cell string
            Description:  First and Last Name
        PositionForSpring2018: 18x1 cell string
            Units:  Executive / Cornerstones / Technical Cmte / Event Cmte
            Description:  Voting Position Held Spring Semester
        PositionForFall2017: 18x1 cell string
            Units:  Executive / Cornerstones / Technical Cmte / Event Cmte
            Description:  Voting Position Held Fall Semester
        YearOfStudy: 18x1 cell string
            Units:  Freshmen / Sophomore / Junior / Senior / Graduate
            Description:  Self-identified year in college
        Major: 18x1 cell string
            Description:  Field of Degree Sought
```

- The function `readtable()` is a powerful data import function with similar strength as `importda-ta()`. The first line of the file (comma-separated values or .csv by default, other options available) is treated as the **VariableNames** that serve as column headings. The remainder of the lines of the file are treated as rows with data matching the column headings.

- It is possible to convert a table to and from other collection data types using functions such as `table2cell()`, `table2struct()` (structures are covered later), and `array2table()`.

- Alternatively, a table of class `table` may be created using the `table()` function using existing *column vectors*. Variable names may be specified as well through extra arguments.

- A table is **always** two-dimensional and must use two indices to reference any value.

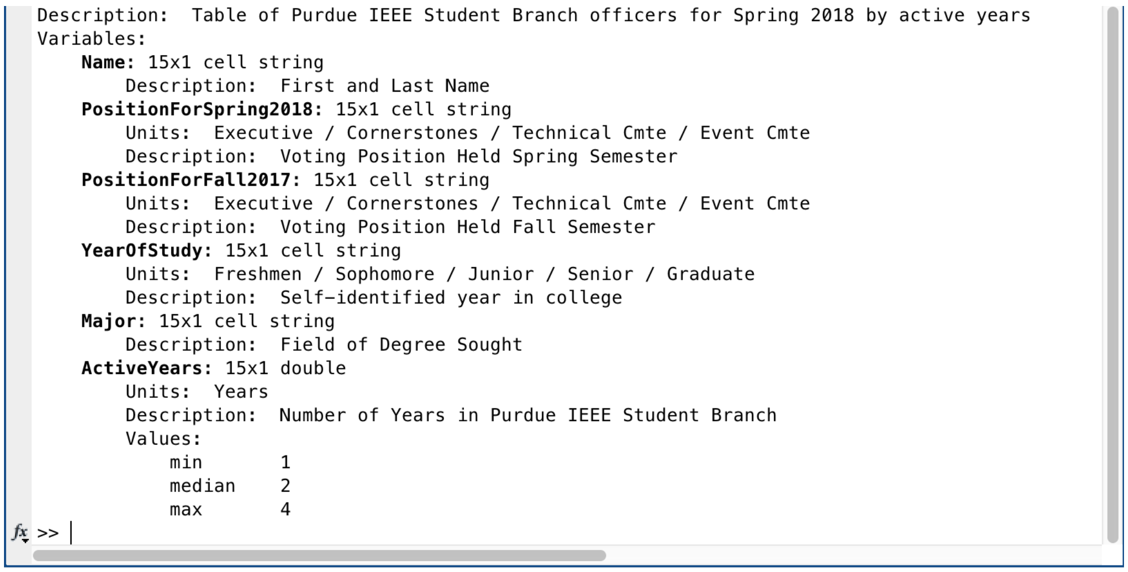
- There are four important functions that measure tables: `numel()`, `size()`, `height()`, and `width()`. The `length()` function **will not** work. They respectively give the total number of data elements in the table, the number of elements in each dimension of the table returned as an array with the answer in the corresponding dimension, the number of data rows, and the number of data columns (variables). However, the *Workspace toolbar* displays all of this information by default.

- The *table properties* in this example are stored in `officers_IEEE_2017.Properties`. Specifically, they are **Description** (string), **VariableDescriptions** (row cell array of table width), **Variable-Units** (row cell array of table width), **DimensionNames** (1x2 cell array), **UserData** (any data type), **RowNames** (row cell array of table height), and **VariableNames** (row cell array of table width).

- The ways to access information in a table are numerous (https://www.mathworks.com/help/matlab/matlab_prog/access-data-in-a-table.html). The best way to get a **subtable** is to use indexing with parentheses `()`. I recommend either indexing with curly braces `{}` or dot indexing in "table.varname" style for **content extraction** corresponding to each variable. There is an advanced data access technique called **subscripting by variable type** that is beyond the scope of this workshop.

- The traditional `disp()` function will display the contents of the talbe with each column headed by the **VariableNames** with a thick, ruled line underneath. However, the `summary()` function is better for displaying information *about* the table, not the contents. A table summary is printed to *Command Window* that has all the *table properties*. The size and data type of each variable is displayed. Furthermore, numeric data has statistics such as the minimum value, median value, and maximum value included. Logical data has a count of `true` and `false` given.

Some observations can be made on **table manipulation**:



```
Description:  Table of Purdue IEEE Student Branch officers for Spring 2018 by active years
Variables:
    Name: 15x1 cell string
        Description:  First and Last Name
    PositionForSpring2018: 15x1 cell string
        Units:  Executive / Cornerstones / Technical Cmte / Event Cmte
        Description:  Voting Position Held Spring Semester
    PositionForFall2017: 15x1 cell string
        Units:  Executive / Cornerstones / Technical Cmte / Event Cmte
        Description:  Voting Position Held Fall Semester
    YearOfStudy: 15x1 cell string
        Units:  Freshmen / Sophomore / Junior / Senior / Graduate
        Description:  Self-identified year in college
    Major: 15x1 cell string
        Description:  Field of Degree Sought
    ActiveYears: 15x1 double
        Units:  Years
        Description:  Number of Years in Purdue IEEE Student Branch
        Values:
            min       1
            median    2
            max       4
fx >> 
```

- The same techniques used to access information in table such as **subtable** indexing or **content extraction** work for assignment of new variables. Logical indexing and other indexing tricks used with cell arrays work with tables.

- Table properties can be updated with any valid **content extraction** indexing technique.

- The function `sortrows()` is one of the most powerful sorting algorithms in MATLAB. The first argument is a table of unsorted information. The second argument is an array of indices such that the first is the primary sorting metric, the second is used for ties in the first index, and so on. The third argument is a cell array of `'descend'` or `'ascend'` of the same size as the second argument that gives the direction for the sorting. The result is a table sorted fast.

- Rows or variables may be deleted from a table by assigning the empty array `[]` to their respective **subtable** of class `table`.

- The function `writetable()` writes a CSV file by default with the table as the first argument and the file name as the second argument. Other than the **VariableNames**, all the other *table properties* will be lost.

Tables are a more organized way of handling large sets of data than cell arrays or plain numeric arrays. A table with good utilization of the *table properties* should need few explanatory comments or none at all. They remain an active area of development for MathWorks, so check which features are best for your application and MATLAB version.

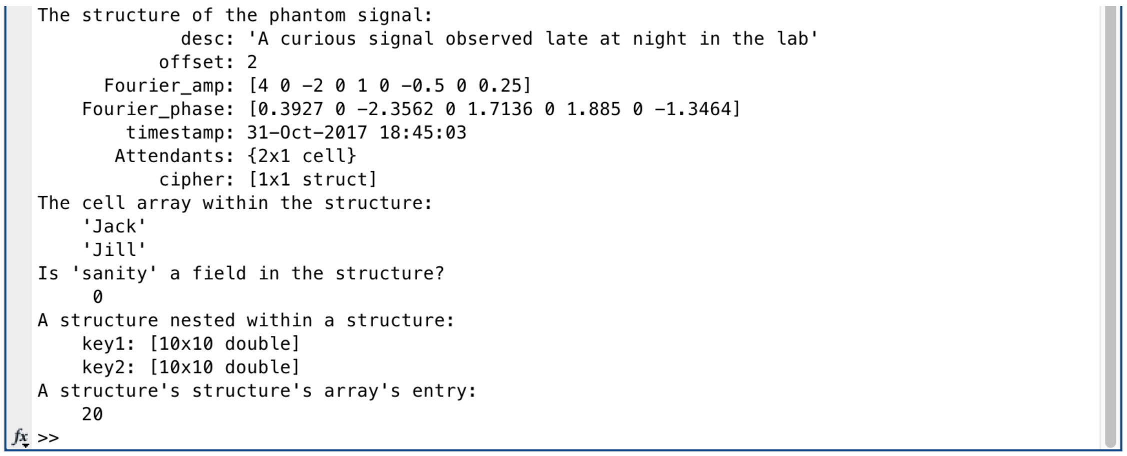
Advanced Collection Data Types: Structures

Structures (https://www.mathworks.com/help/matlab/structures.html) are formally known as *structure arrays* (we will use the shorter name) and serve as a way to achieve something akin to *object-oriented programming* in MATLAB.

Here is a quick example for the MATLAB *Command Window* to show you how useful structures are.

```
new_field_name = 'Attendants';
on_site = {'Jack'; 'Jill'};
phantom_sig.desc = 'A curious signal observed late at night in the lab';
phantom_sig.(new_field_name) = on_site;
phantom_sig.offset = 2;
phantom_sig.Fourier_amp = [4, 0, -2, 0, 1, 0, -0.5, 0, 0.25];
phantom_sig.Fourier_phase = [pi/8, 0, -3*pi/4, 0, 6*pi/11, ...
    0, 3*pi/5, 0, -3*pi/7];
phantom_sig.cipher.key1 = magic(10);
phantom_sig.cipher.key2 = randi(8, 10, 10);

disp('The structure of the phantom signal:')
disp(phantom_sig)
disp('The cell array within the structure:')
disp(phantom_sig.Attendants)
disp('Is ''sanity'' a field in the structure?')
disp(isfield(phantom_sig, 'sanity'))
disp('A structure nested within a structure:')
disp(phantom_sig.cipher)
disp('A structure''s structure''s array''s entry:')
disp(phantom_sig.cipher.key1(3,4))
```

% *MATLAB Comprehension Questions:* When should structures be used? Is there a way to number the elements of a structure or do only the names exist? Is there a common data type that cannot be embedded as a **field** in a structure?



```
The structure of the phantom signal:
             desc: 'A curious signal observed late at night in the lab'
           offset: 2
      Fourier_amp: [4 0 -2 0 1 0 -0.5 0 0.25]
    Fourier_phase: [0.3927 0 -2.3562 0 1.7136 0 1.885 0 -1.3464]
        timestamp: 31-Oct-2017 18:45:03
       Attendants: {2x1 cell}
           cipher: [1x1 struct]
The cell array within the structure:
    'Jack'
    'Jill'
Is 'sanity' a field in the structure?
     0
A structure nested within a structure:
    key1: [10x10 double]
    key2: [10x10 double]
A structure's structure's array's entry:
    20
>>
```

- Paradoxically, one must understand how to index elements in a **structure** before learning how to create a structure. The elements of a structure are stored in **fields**, a named data container that is accessed in the form "struct.field". The parts of a structure can *only* be accessed with the dot notation of the field; there is no ordering of the fields that gives way to a numbered index scheme.

- A structure of class `struct` is created as simply as assigning a value to "struct_name.first_field_name". A structure can accept nearly any data type for a field, and different fields may contain different data types. In fact, it is possible to nest other structures, tables, cell arrays, or arrays within structures repetitively.
- If an inner structure is also a field of an outer structure, then the field "infield4" may be referenced as "out_struct.in_struct.infield4".
- There is a practical limit to how deeply structures can be nested before MATLAB gives an `OutOfMemory` error.
- The two important state testing functions for structures are `isstruct()` and `isfield()`. The function `isstruct()` returns a logical indicating whether the variable passed to it is a structure, which might be needed because some classes such as `table` and `datetime` have certain dot notations that produce structures and some functions stealthily return structures. The function `isfield()` takes a variable for a `struct` as the first argument and a cell array of strings as the second argument and returns a `logical` array indicating if each string corresponds to a field in the structure.
- Sometimes the desired name of a field is not known in general and must be assigned programmatically. The syntax "struct.(var_of_name)" will make the field have the name contained in "var_of_name".

Structures are a practical way of aggregating data in a purposeful way. They are particularly handy for passing a many pieces of related data as arguments to custom functions or returning non-exhaustive amounts of data from a function based on what is available. For the most part, fields of a structure accessed with the **dot notation** behave little differently than the data type contained. The only major downside to structures is that variable names can become ridiculously long.

Plotting

Plotting is explored in real-time during the workshop.

Publishing

Press the "Publish" button in the "Publish" section of the "Publish" tab of the ribbon in the script editor and see what happens (https://www.mathworks.com/help/matlab/matlab_prog/publishing-matlab-code.html). With a little bit of extra effort to make comments in natural language, MATLAB can remove the need to pass data to word procesing software to make a report for a class. You should see the value in using *cell mode* to organize your software into sections. The MATLAB publishing markup may be found at https://www.mathworks.com/help/matlab/matlab_prog/marking-up-matlab-comments-for-publishing.html. scripts.

Published with MATLAB® R2015b