Table of Contents

Introduction to MATLAB Workshop

This document describes the steps covered to teach MATLAB Fundamentals and prepare attendees for the MathWorks Certified MATLAB Associate Exam.

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

What is MATLAB?

MATLAB (matrix laboratory) is a numerical computing engine and very high-level programming language created by MathWorks. MATLAB is:

- written in C, C++, and Java

- multiplatform (Windows, macOS, and Linux operating systems)

- an evolution of the original release in 1984

- proprietary software updated twice a year

- multi-paradigm, with a focus on linear algebra

- interpreted (similar vein as Python)

- fond of of the .m file extension to the point of overloading

- influencer of Julia, Octave, and other numerical computing packages

- equally beloved and hated by engineers and scientists the world over.

MATLAB has a few key strengths. MATLAB takes an "everything and the kitchen sink" approach to the language. There are existing functions that handle the most common applications imaginable. The built in help makes it simple to figure out what a given named function does. There are about 62 toolboxes that MathWorks offers (for a considerable sum) that handle even more specific applications. Purdue University has access to nearly all of them.

MATLAB was built to perform linear algebra operations. There is no additional penalty to computation for a matrix multiplication operation as compared to a scalar multiplication operation because MATLAB stores numbers as arrays by default. There is a certain sense of "compactness" that comes from being able to store data as arrays and visualize the results in the built-in Workspace.

MATLAB can do anything that a power graphing calculator or spreadsheet like Microsoft Excel® can do. In fact, there is little reason to choose to start storing data in an Excel® spreadsheet if you have MATLAB available. The only thing that MATLAB truly lacks is the collaborative potential and centralized cloud storage of Google Sheets or Excel for OneDrive.

Many of MATLAB's biggest flaws come from the fact that large sections are written in Java. The program is gigantic, bloated, and difficult to navigate at times. Its size makes it difficult to multitask with other programs when memory is limited, and multiple instances are often out of the question. `for` and `while` loops are really slow compared to most programming languages, a terrible flaw to have.

MATLAB is used across many industries as a crucial piece of software. Most big enterprises will have some numerical computing tool that can interpret the MATLAB syntax and perform the high-level calculations involved. Over the course of your studies, many of you will learn how to implement many of the tools MATLAB offers yourself in lower-level languages. However, there are many times that you probably cannot be bothered to do so, and that is where MATLAB scripts step in. There will likely be many times that you need to process some data for a science lab or some research either once or infrequently enough that MATLAB's lethargy is not an issue.

User Interface: Ribbons, Toolbars, and the Script Editor
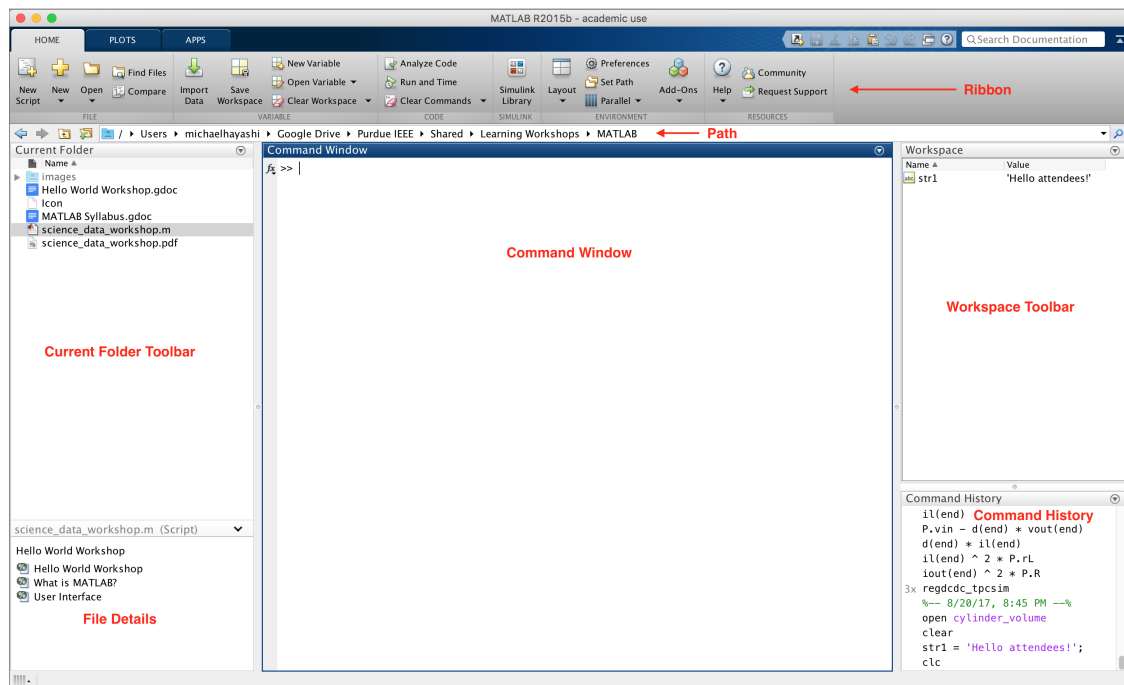

MATLAB has a similar ribbon and menu structure as the Microsoft Office suite. When starting MATLAB, you should see the "Home", "Plots", and "Apps" ribbons in the *toolstrip* at the top of the window. The "Home" ribbon will be used the majority of the time. The "Plots" ribbon contains many toolstrip buttons for instantly producing many types of graphs; however, we will nearly always be plotting from a script instead to exercise a higher degree of control over the plot appearance. The "Apps" ribbon contains toolstrip buttons that launch custom graphical user interfaces (GUI's) for manipulating certain kinds of input or controlling the output in situations such as parallel computing. These features can be very useful, but are generally only sought out in very specific circumstances.

The "Home" ribbon has the standard commands such as "New", "Open", "Preferences", and "Help" that we have come to expect out of all modern applications. The "File" and "Variable" sections of the "Home" ribbon have basic actions that can be easily done as commands or functions. The "Code" section is used after scripts have been run or commands have been issued. "Simulink" is a graphical programming tool similar to LabVIEW that is worthy of its own workshop. "Environment" is about controlling the display on screen and how the code is run. "Resources" is the built-in assistance that MATLAB provides.
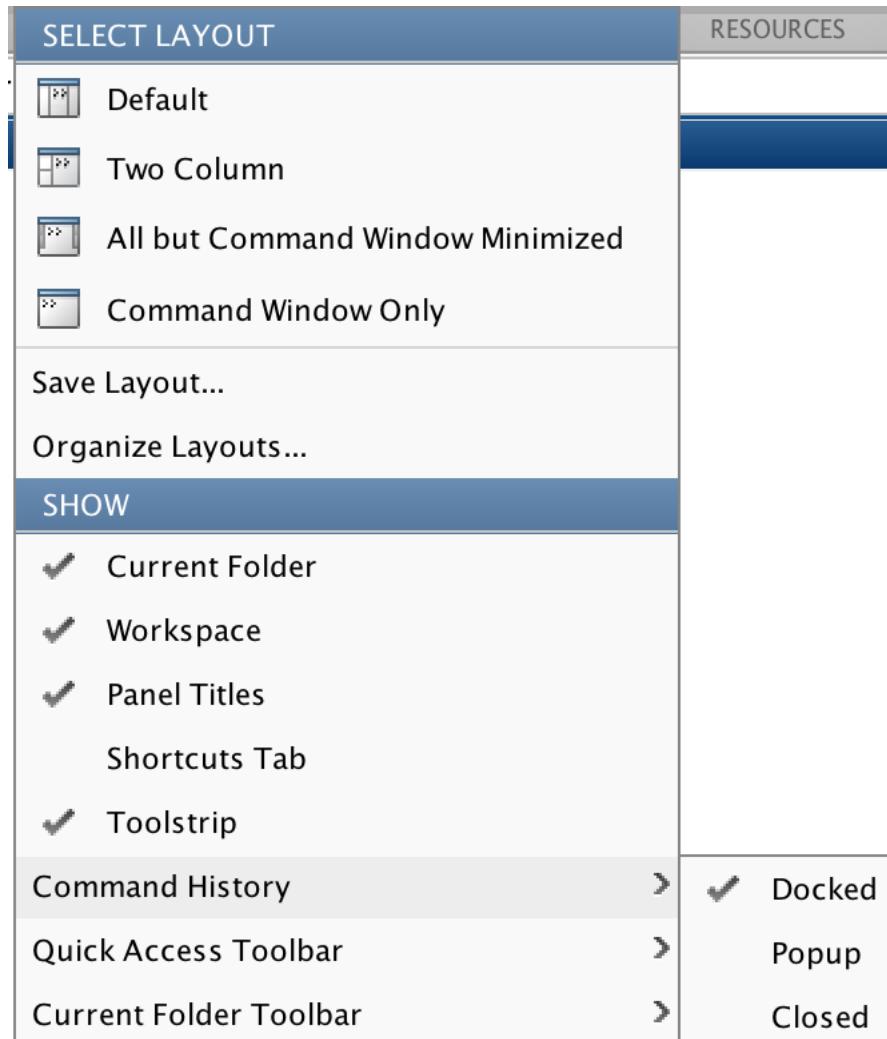
It is important to make sure that the important parts of the MATLAB window are visible. Click the "Layout" button in the "Environment" section of the "Home" ribbon. There are some preset layouts available here. Choose whichever style suits your tastes. Ensure that the *Current Folder* toolbar, *Workspace* toolbar, *Panel Titles* (important when learning the parts of the GUI), and *Toolstrip* are set to show. Also, I recommend having the *Command History* docked. This way, all of your file information, variables in the workspace, and past commands can be easily found by looking at the MATLAB window. Arrange the toolbars however you please.
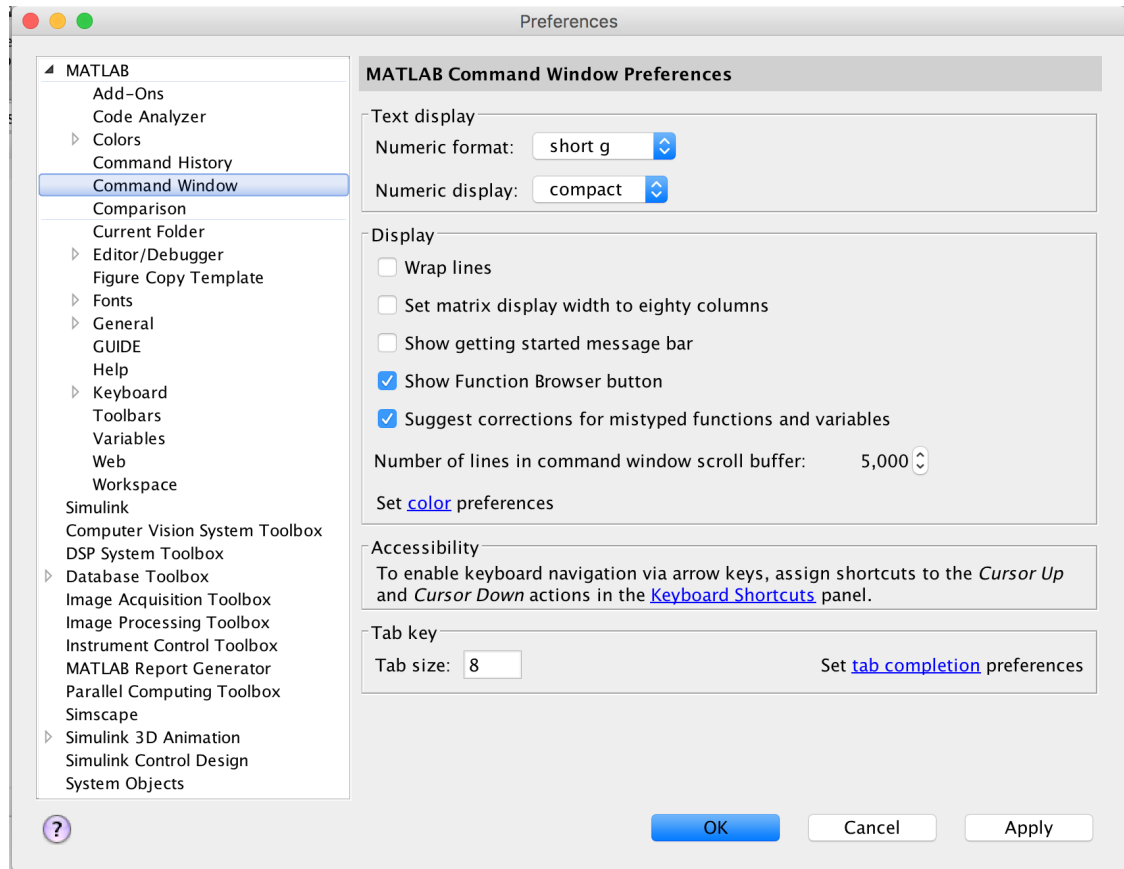
Click on the "Preferences" button in the "Environment" section of the "Home" ribbon. There are hundreds of settings to adjust, but most of the defaults work nicely when first getting started. I recommend going to "Command Window" under "MATLAB" in the sidebar and changing the numeric format to `short` and the numeric display to `compact` in the *Command Window* in order to keep most of the information in the window when using a laptop. Click "OK" when done.

It is time to perform some simple math to get used to the *Command Window*:

```
6 + 7
3.5 + 8.2
12345 - 6.8
```


This should be what you see on screen. We can see some simple evidence of typing going on. Adding two integers produces an integer result. Adding two floating-point numbers produces a floating-point result. The third option may *appear* to be an integer written in scientific notation, but this is an artifact of the

`short g` display style. Closer investigation of the value reveals that this is indeed a floating-point number and that MATLAB took care of casting the integer to a float when performing the subtraction.
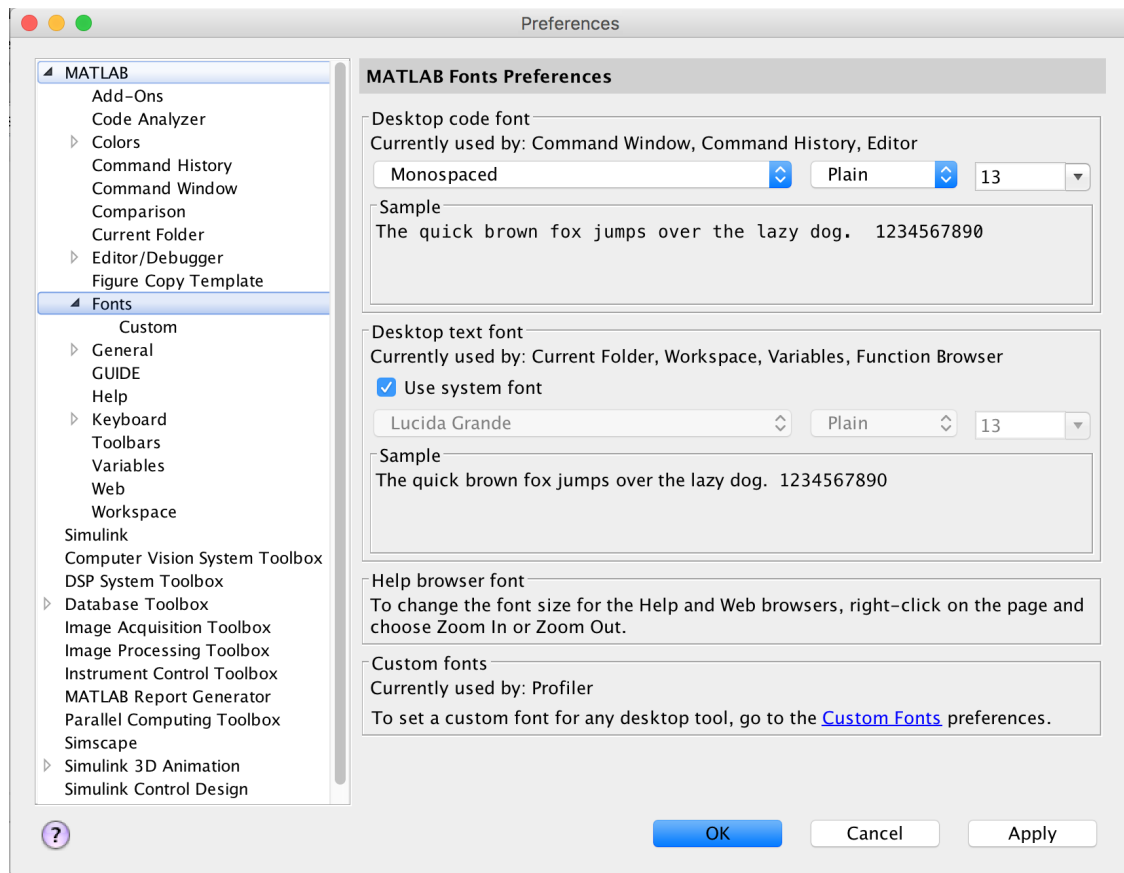
Return to the "Preferences" dialog box and go to "Fonts" under "MATLAB" in the sidebar. If you are having trouble seeing the text in the Command Window, go ahead and adjust the way the fonts display. There are many other preferences to change. For example, seasoned programmers might want to use their own editor instead of the built-in MATLAB editor. For now, just ignore the other options and click "OK".

The *path* controls where MATLAB stores new files and which functions are in the scope. The default is a directory names "MATLAB" in your documents folder. This will do for this workshop, but it is advisable to work in a project- or lab-specific folder separated out by course for your future uses of MATLAB. The "Browse for folder" button will allow you to change the *Current Folder* in the path. If you need to access the path you used before when starting up MATLAB again, the black arrow at the right remembers the previous paths you visited.
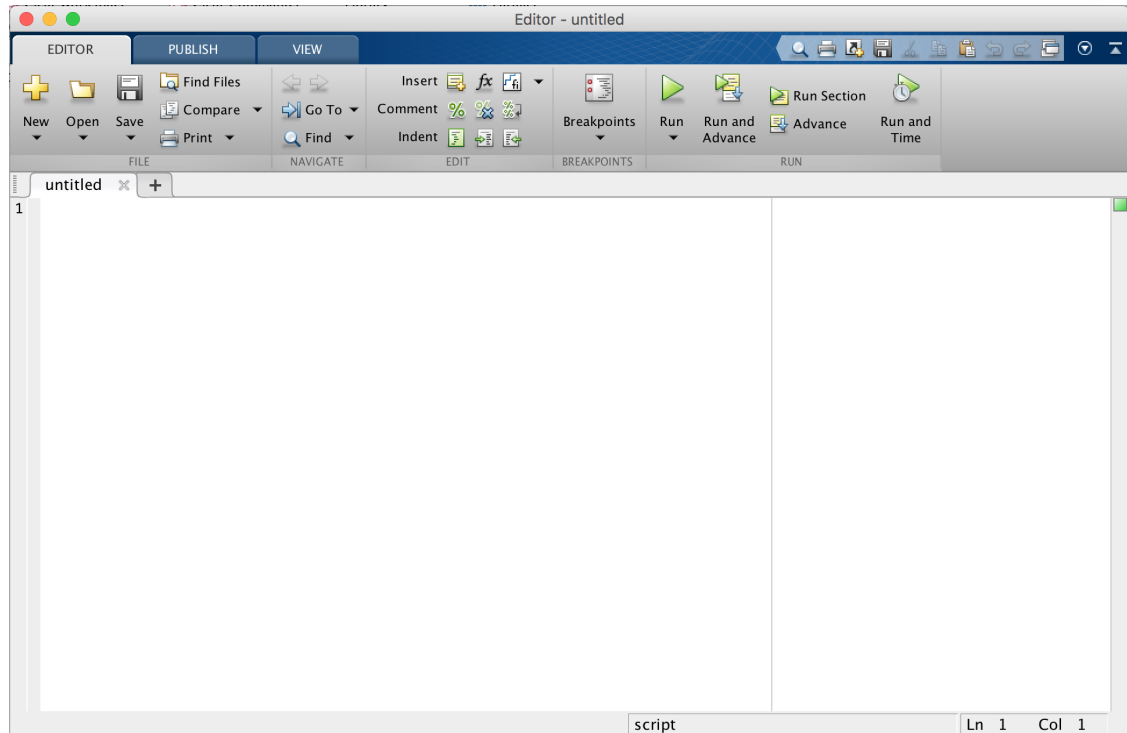
Click on the "New Script" button (Ctrl + N on Windows) in the "File" section of the "Home" ribbon. A text editor should pop up in a separate window. The editor has its own ribbon and tabs: the "Editor" ribbon, "Publish" ribbon, "View" ribbon, and tabs for each file that is currently open. Just like the "Home" ribbon in the MATLAB window, the "Editor" ribbon is where a programmer will spend the majority of the time. The "Publish" ribbon has toolstrip buttons that are only used after the script is finished. The "View" ribbon has some toolstrip buttons that change the appearance of the editor. Go ahead and change the appearance around as you wish (I recommend the default view), and return to the "Editor" tab before closing out of the script editor.

User Interface: Help and Documentation

MATLAB has amazing documentation on the MathWorks website that explain its many, many functions. It is extremely convenient that most of the documentation is accessible from within MATLAB. Enter the following into the Command Window:

```
help linspace
doc find
```

```
Command Window                                                              ⊙
>> help linspace
 linspace Linearly spaced vector.
    linspace(X1, X2) generates a row vector of 100 linearly
    equally spaced points between X1 and X2.

    linspace(X1, X2, N) generates N points between X1 and X2.
    For N = 1, linspace returns X2.

    Class support for inputs X1,X2:
       float: double, single

    See also logspace, colon.

    Reference page for linspace
    Other functions named linspace

fx >> |
```
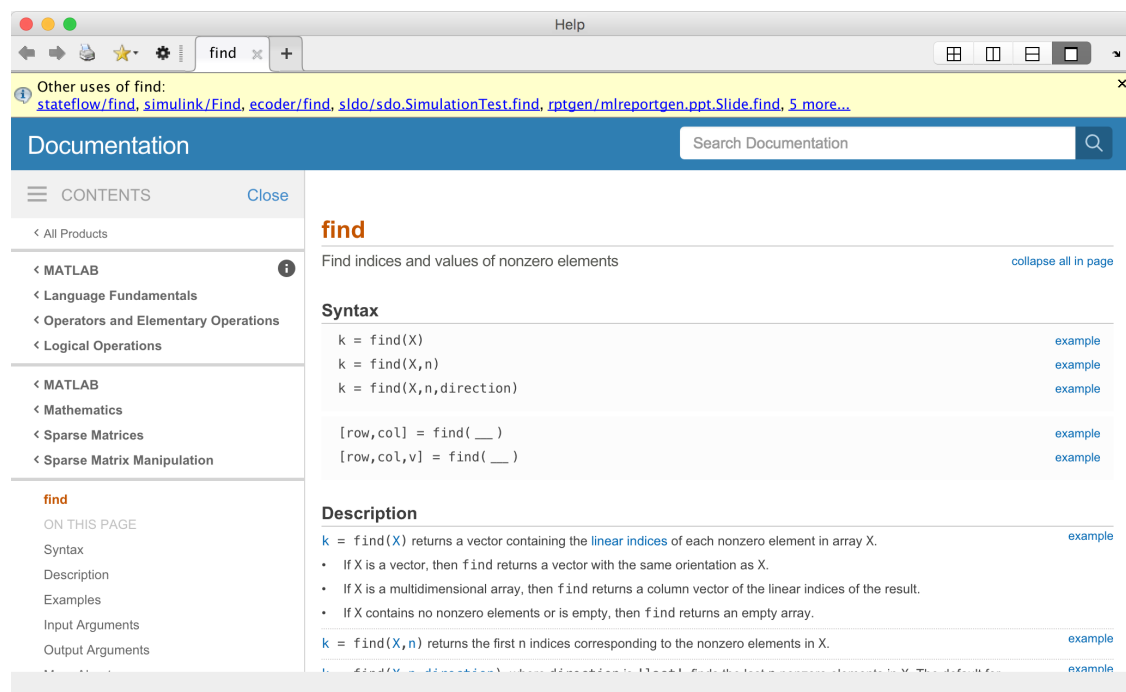
You can get a one-line description of what `linspace()` does, prototypical uses based on the input style with descriptions of the resulting output, input data types, related functions, and hyperlinks to other resources in the description. Using `help` followed by the name of any MATLAB function will produce similar text for that function.

Using `doc` followed by the name of a function will open the help browser after loading for a bit. There will be an HTML page that gives the full documentation of that function's behavior with a tree describing the location of the documentation page in the entire contents of MATLAB. In fact, the documentation should be the same as that on MathWorks' website: https://www.mathworks.com/help/matlab/ref/find.html. This is an extremely powerful tool for developers and consequentially the reason why MATLAB exams are paper-and-pencil rather than online.

At this time, you will notice that every command typed into the *Command Window* has appeared in the *Command History*. Commands can be reissued by clicking on them in *Command History*. Similarly, there is a single variable stored in the *Workspace*, a scalar called `ans` that shows the result of the last operation, in this case, 1.2338e+04. Any variable we create will be listed in the *Workspace* going forward.

Math Operations: Variables, Data Types, and Scalar/Logical Operators

From now on, we will do all our work in scripts. The *Command Window* is useful for testing small snippets of code, especially as you are learning new functions, but the ephemeral nature of what it typed there makes it difficult to revisit code.

Type the following into a script and save it as "workshop.m":

```
var1 = 55.1;
var2 = 2;
var3 = 46372819;
var4 = 'c';
var5 = 'Hello World!';
var6 = false;
var7 = @help;

result1 = var1 + var2;
result2 = var3 - var2;
result3 = var1 * var2;
result4 = var3 / var1;
result5 = var1 \ var3;
result6 = var1 ^ var2;
result7 = (var1 < var2);
result8 = ~var6;
result9 = var1 + 1j*var2;

disp(var1)
disp(class(var1))
disp(var2)
disp(class(var2))
disp(var3)
disp(class(var3))
disp(var4)
disp(class(var4))
disp(var5)
disp(class(var5))
disp(var6)
disp(class(var6))
disp(var7)
disp(class(var7))

disp('This is a string literal followed by nothing.')
disp('')
disp(result1)
disp(class(result1))
disp(result2)
disp(class(result2))
disp(result3)
disp(class(result3))
disp(result4)
disp(class(result4))
disp(result5)
```
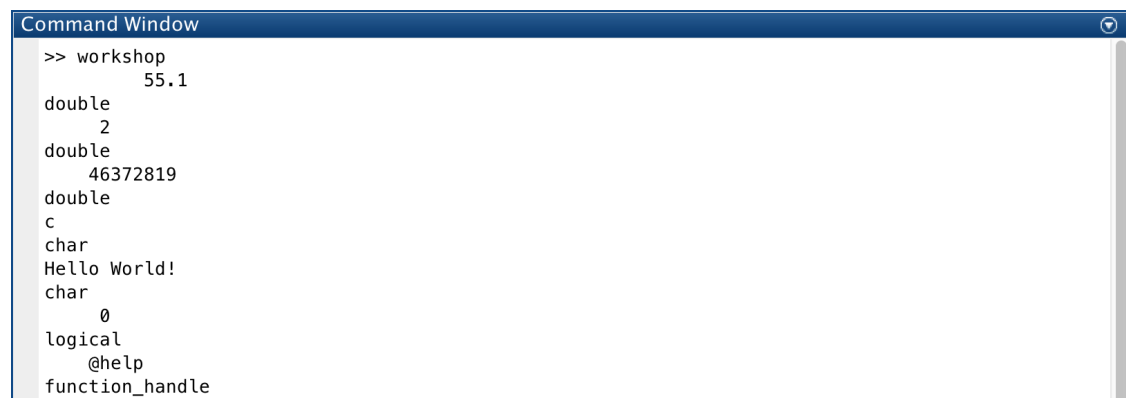
```
disp(class(result5))
disp(result6)
disp(class(result6))
disp(result7)
disp(class(result7))
disp(result8)
disp(class(result8))
disp(result9)
disp(class(result9))

disp('Let''s demonstrate escape characters and type casting.')
disp(int32(result1))
disp(class(int32(result1)))
disp(logical(result1))
disp(class(logical(result1)))
disp(double(result7))
disp(class(double(result7)))
```

```
Command Window
>> workshop
        55.1
double
        2
double
    46372819
double
c
char
Hello World!
char
        0
logical
    @help
function_handle
```

Run the script by pressing the green "Run" button (F5 in Windows) in the "Run" section of the "Editor" ribbon in the editor window. Let's make some observations about the **variables and their data types**:

- The assignment operator for variables is = as it is in most languages.

- The semicolon ; has a unique function as the "output suppression character" that is very different from its use in the C language. The semicolon prevents the result of an operation from displaying on the *Command Window*. It does **not** signify the end of a command like in C and is not needed for functions that produce no output.

- The function name workshop appears in the *Command Window* when we execute the script.

- The disp() command is smart enough to show either the contents of a variable, result of an operation, or string in plain text when used.

- Nested functions are evaluated inside-to-outside as they are in most languages.

- Even though the *Workspace* shows var2 = 2; as containing an integer, the class function, which displays the data type of whatever is passed to it, as a double. This means that MATLAB stores scalars as double-precision numbers by default, making it no wonder why it is a memory hog.

- MATLAB does not differentiate between characters and character arrays (i.e., strings), so var4 and var5 are both strings, just of different lengths (i.e., length 1 and length 12). The editor has syntax highlighting that puts strings in purple.

- Boolean values (i.e., `true` and `false`) are displayed as integers, but the *Workspace* reveals that they are indeed stored as a different data type called `logical`.

- There is another data type called `function_handle` that is created by putting the `@` character in front of the name of a valid MATLAB function. Its uses are fairly advanced, but it is worth mentioning now.

```
This is a string literal followed by nothing.
        57.1
double
    46372817
double
       110.2
double
   8.4161e+05
double
   8.4161e+05
double
        3036
double
      0
logical
      1
logical
        55.1 +          2i
double
```

Some observations can be made about the **elementary scalar operators** (https://www.mathworks.com/help/matlab/matlab_prog/matlab-operators-and-special-characters.html):

- The function `disp()` works for literals equally well as variables, as expected, and ignores empty strings.

- Addition, subtraction, multiplication, and division use the expected keys. Modulus does not have its own operator.

- MATLAB has right division (`/` for dividend followed by divisor) and left division (`\` for divisor followed by dividend). The result is the scalar quotient in both cases, but the need for two different operators becomes clear when doing matrix operations.

- Exponentiation is built into the `^` operator.

- Parentheses `()` serve as grouping symbols that define order of operations and can make raw code appear cleaner.

- The relational operators equal to `==`, **not equal to ~=**, greater than `>`, greater than or equal to `>=`, less than `<`, and less than or equal to `<=` produce a result of type `logical` when used.

- The default logical operators are logical AND `&`, logical OR (the vertical pipe), **logical AND with short-circuiting &&**, **logical OR with short-circuiting (two vertical pipes)**, and **logical NOT ~**. Pay attention to the character used for NOT. Short-circuiting is a time-saving measure where the result is calculated immediately if one variable can determine the output. You should use the doubled short-circuiting logical operators for most of your Boolean expressions.

- Complex numbers can be formed by including an imaginary part that arises when multiplying any scalar by the value of `1j`, which is the imaginary unit $j^2 = -1$ as used in electrical engineering. (`1i` may be used alternatively.) The default display style is $\Re\{z\} + i\Im\{z\}$. Complex numbers are stored with the real part and imaginary part as double-precision numbers, making the variable `complex double` data type.

```
     Let's demonstrate escape characters and type casting.
             57
     int32
         1
     logical
         0
     double
fx >>
```

Type casting can be accomplished by using functions such as `int32()` (32-bit integer), `uint64()` (64-bit unsigned integer), `single()` (single-precision, floating-point number), `double()` (double-precision, floating-point number), and `logical()` (Boolean). The single quote `'` can be escaped in a string by substituting in `''`.

Math Operations: Vector Operators

Highlight the text in "workshop.m", and press the green percent sign "Comment" (Ctrl + R in Windows) in the "Edit" section of the "Editor" ribbon. Go to a blank line below the commented out script that is syntax highlighted in green. It is time to move on to **vector operations**.

```
vec1 = [1, 2, 3, 4]
disp('The length, number of elements, and size of vec1:')
disp(length(vec1))
disp(numel(vec1))
disp(size(vec1))
vec2 = [1; 2; 3; 4]
disp('The length, number of elements, and size of vec2:')
disp(length(vec2))
disp(numel(vec2))
disp(size(vec2))
vec3 = [1; 2; 3; 4; 5; 6]
vec4 = 25:30
vec5 = (10:2:30).'
vec6 = (100:-10:50).'
vec7 = linspace(0, 2*pi, 60)
vec8 = logspace(2, 6.8, 40)

res1 = 3 * vec1
res2 = vec6 - vec3
res3 = vec2 + 1j*vec2
res4 = res3.'
res5 = res3'
res6 = [vec1, 10, 11]
res7 = vec4 ./ res6
```

```
Command Window                                                              ⊙
>> workshop
vec1 =
     1     2     3     4
The length, number of elements, and size of vec1:
     4
     4
     1     4
vec2 =
     1
     2
     3
     4
The length, number of elements, and size of vec2:
     4
     4
     4     1
vec3 =
     1
     2
     3
     4
     5
     6
vec4 =
    25    26    27    28    29    30
vec5 =
    10
    12
    14
    16
    18
    20
    22
    24
    26
    28
    30
```

```
vec6 =
   100
    90
    80
    70
    60
    50
vec7 =
  Columns 1 through 7
        0      0.10649      0.21299      0.31948      0.42598      0.53247      0.63897
  Columns 8 through 14
    0.74546      0.85196      0.95845       1.0649       1.1714       1.2779       1.3844
  Columns 15 through 21
     1.4909       1.5974       1.7039       1.8104       1.9169       2.0234       2.1299
  Columns 22 through 28
     2.2364       2.3429       2.4494       2.5559       2.6624       2.7689       2.8754
  Columns 29 through 35
     2.9819       3.0883       3.1948       3.3013       3.4078       3.5143       3.6208
  Columns 36 through 42
     3.7273       3.8338       3.9403       4.0468       4.1533       4.2598       4.3663
  Columns 43 through 49
     4.4728       4.5793       4.6858       4.7923       4.8988       5.0052       5.1117
  Columns 50 through 56
     5.2182       5.3247       5.4312       5.5377       5.6442       5.7507       5.8572
  Columns 57 through 60
     5.9637       6.0702       6.1767       6.2832
vec8 =
  Columns 1 through 7
       100       132.76       176.26       234.01       310.68       412.46        547.6
  Columns 8 through 14
    727.01        965.2       1281.4       1701.3       2258.6       2998.6       3981.1
  Columns 15 through 21
    5285.4         7017         9316        12368        16420        21800        28943
  Columns 22 through 28
     38425        51014        67728        89918    1.1938e+05   1.5849e+05   2.1042e+05
  Columns 29 through 35
 2.7935e+05   3.7088e+05   4.9239e+05   6.5371e+05   8.6788e+05   1.1522e+06   1.5297e+06
  Columns 36 through 40
 2.0309e+06   2.6963e+06   3.5797e+06   4.7525e+06   6.3096e+06
```

Run the script by pressing the green "Run" button (F5 in Windows) in the "Run" section of the "Editor" ribbon in the editor window. Let's make some observations about the **creation of vectors**:

- Without the semicolon `;`, each assigned variable as well as the results of operations will be displayed on the *Command Window*. This quickly becomes untidy as the number of operations eventually necessitates scrolling through the clutter.

- Vector literals are created by placing integers between square braces `[ ]`.

- Comma-separated or space-separated values create row vectors. MATLAB displays row vectors as an actual row on the *Command Window*.

- There are three important functions giving the property of vectors: `length()`, `numel()`, and `size()`. They respectively give the number of elements along the longest dimension of an array, the total number of elements in the array, and the number of elements in each dimension of the array returned as an array with the answer in the corresponding dimension. It can be inferred that the first dimension is vertical (down the column) and that the second dimension is horizontal (right along the row), consistent with common mathematics parlance. However, the *Workspace toolbar* displays all of this information by default.

- Semicolon-separated values create column vectors. MATLAB displays column vectors in a column on the *Command Window*. While `vec1` and `vec2` have the same length and same elements, the sizes are difference because of the distinction between row vectors and column vectors.

- Some vectors are too long to be typed out as literals. The *colon operator* `:` or `colon()`, roughly translated as "everything in the range in between", is really helpful. A single colon operator in `vec4` creates a **row vector** from 25 **up to and including** 30 by incrementing the value.

- The colon operator can be used twice to specify what the difference between elements should be. In `vec5`, the start value is specified as 10, the spacing is 2, and the end value is 30. The result is a row vector that contains all the even numbers between 10 and 30 inclusive.

- Negative values for spacing when the colon is used twice are allowable. Now it is possible to create vectors with decreasing values as in `vec6`.

- There is something else special about `vec5` and `vec6`. By enclosing the statement in parentheses `()` for grouping and appending a period and apostrophe `.'` to the end (or by using the function `transpose()`), the vector is *transposed*. Now there is an easy way to convert between row and column vectors.

- Often when constructing inputs, the start and end values of your trials will be known. Usually, there is a certain number of inputs desired to be put to test rather than a known spacing between them. For this reason, `linspace()` is a frequently used function. The first argument is the start point, the second argument is the end point to include, and the third argument is the total number of elements in the resulting row vector. MATLAB knows the number `pi` is $\pi \approx 3.14159265$, so `vec6` is a vector of $6°$ increments around a circle expressed in radians. Note: MATLAB provides helpful functions such as `rad2deg()` and `deg2rad()`.

- Sometimes the input needs to span multiple orders of magnitude. The function `logspace()` comes in handy for this, working to produce logarithmically spaced values in base 10. The first argument is the **common logarithm of the start point**, the second argument is the **common logarithm of the end point**, and the third argument is the number of points to include. Note: there is non-standard behavior when the second argument is π.

```
Now the Results
res1 =
     3     6     9    12
res2 =
    99
    88
    77
    66
    55
    44
res3 =
           1 +         1i
           2 +         2i
           3 +         3i
           4 +         4i
res4 =
  Columns 1 through 3
           1 +         1i           2 +         2i           3 +         3i
  Column 4
           4 +         4i
res5 =
  Columns 1 through 3
           1 -         1i           2 -         2i           3 -         3i
  Column 4
           4 -         4i
res6 =
     1     2     3     4    10    11
res7 =
          25          13           9           7         2.9      2.7273
fx >>
```

Some observations can be made about **vector math operations**:

- Scalar multiplication works as expected by using the asterisk `*` between a scalar and a vector.

- Vectors of the same *size* can be added. Not only do vectors have to be the same length, but make sure both are row vectors OR both are column vectors (standard for linear algebra). Complex vectors can be created through scalar multiplication of the imaginary part by $j = \sqrt{-1}$ and addition to the real part. Note: the "Matrix dimensions must agree." error appears on screen in red if this rule is violated.

- For any real vector, the effects of transposition (period and apostrophe `.'` or `transpose()`) and conjugate transposition/Hermitian transposition (just the apostrophe `'` or `ctranspose()`) are the same. However, `res4` and `res5` are complex conjugates of each other ($z = a + bi$ and $\overline{z} = a - bi$) and illustrate the difference for complex numbers.

- It is easy to concatenate vectors by including the variable name as anv entry between square braces `[ ]` as `res6` shows. The alternative is to use the function `vertcat()` to concatenate vectors vertically, the function `horzcat()` to concatenate vectors horizontally, and `cat()` to concatenate in any dimension as long as the first argument is the dimension to concatenate along.

From a strict mathematical standpoint, vector multiplication can only occur when a row vector multiplies a column vector of the same length to produce a scalar (the **dot product** accomplishable through `dot()`, also known as the *inner product* or *scalar product*), a column vector multiplies a row vector of the same length to produce a matrix which has dimensions equal to length-by-length (the **outer product** or matrix product), a 3-vector multiplies another 3-vector to produce a 3-vector perpendicular to them both (the **cross product** accomplishable through `cross()`, also known as the *vector product*), or some combination of the previous products. Vector division is out of the question. Most of the time when processing data, each entry in a vector represents an independent trial. It would be nice to perform arithmetic operations on each entry of identically sized vectors without having to write code that unpacks them every time after all the effort to organize it.

The dot operator `.` allows vectors, matrices, and arrays to perform operations element-wise rather than by linear algebra rules. The options are element-wise multiplication `.*`, element-wise right division `./` as evidenced by `res7`, element-wise left division `.\`, and element-wise power `.^`. Note that `.*` is mathematically known as the **Hadamard product** or the *Schur product* or the *entrywise product* when justifying the mathematical legality of this operation to purist faculty.

Math Operations: Matrix Operators

Highlight the active text in "workshop.m", and press the green percent sign "Comment" (Ctrl + R in Windows) in the "Edit" section of the "Editor" ribbon. Go to a blank line below the commented out script. It is time to move on to **matrix operations**.

```
mat1 = [1, 2; 3, 4; 5, 6]
disp('The length, number of elements, and size of mat1')
disp(length(mat1))
disp(numel(mat1))
disp(size(mat1))
mat2 = [linspace(10, 15, 3).', linspace(22, 11, 3).']
mat3 = 1/2 * mat1 + 1j * mat2
mat4 = mat3.'
mat5 = mat3'
mat6 = randi([0, 20], size(mat1))
mat7 = zeros(3, 4)
mat8 = ones(2)
mat9 = eye(3)
mat10 = mat6 * mat5
```

```
mat11 = mat3 .^ 2
vecx = [3, 1; -4, -2] \ [-1; 0]
```

Run the script by pressing the green "Run" button (F5 in Windows). Let's make some observations about the **creation of matrices and matrix operations**:

- Matrix creation is similar to vector creation. Matrix literals are also made using square brackets `[ ]`. Comma-separated (or space-separated) values build up rows by adding columns along dimension 2, and semicolons mark the start of a new row along dimension 1.

- There are three important functions giving the property of matrices: `length()`, `numel()`, and `size()`. They respectively give the number of elements along the longest dimension of an array, the total number of elements in the array, and the number of elements in each dimension of the array returned as an array with the answer in the corresponding dimension. With matrices, it will be evident that

the length is not equal to the number of elements in general. Also, the size usually contains numbers greater than 1. Again, the *Workspace toolbar* displays all of this information by default.

- The `linspace()` and `logspace()` functions can help build matrices. Keep in mind the output is a row vector by default, so transposition or conjugate transposition may be needed before concatenating.

- Scalar multiplication with `*` and addition of matrices of the same size with `+` is straightforward. Complex matrices may be formed in ways similar to complex vectors.

- The matrices `mat4` and `mat5` illustrate the differences between transposition (`.'` or `transpose()`) and conjugate transposition/Hermitian transposition (`'` or `ctranspose()`).

- MATLAB has several function to create pseudorandom matrices. The function `randi()` will create a matrix with integer entries uniformly distributed. The first argument is either the largest integer allowed in the output matrix or a length-2 row vector with the smallest integer allowed followed by the largest integer allowed. The second argument to `randi()` is a row vector with the number of elements to contain in each dimension of the output matrix. (Your value for `mat6` will differ from mine.)

- Mathematical operations naturally have *identity elements*. When a given binary mathematical operation is performed on an identity element and another element, then the result is the other element. For matrix addition with `+`, the identity element is the zero matrix $\mathbf{0}_{m,n}$ formed by the function `zeros()` with every entry `0`. For matrix multiplication of square matrices with `*`, the identity element is the identity matrix $\mathbf{I}_n$ with `1` on each main diagonal entry and `0` everywhere else formed by the function `eye()`. For element-wise multiplication with `.*`, the identity element is the matrix of ones $\mathbf{J}_{m,n}$ formed by the function `ones()` with every entry `1`.

- Matrix multiplication is valid when the number of columns of the first factor is equal to the number of rows of the second factor. The output will have the same number of rows as the first factor and the same number of columns as the second factor. Only a square matrix can be raised to a power.

- Element-wise multiplication `.*`, element-wise right division `./`, element-wise left division `.\`, and element-wise power `.^` all work for matrices.

- Suppose that the common problem of $\mathbf{A}\vec{x} = \vec{b}$ needs to be solved where $\mathbf{A}$ is the matrix of a linear transformation, $\vec{b}$ is the resulting column vector, and $\vec{x}$ is the column vector of inputs to find. Unfortunately, $\vec{x} = \mathbf{A}^{-1}\vec{b}$ is computationally expensive to find because matrix inversion is slow. Luckily, MATLAB is good at computational Gaussian elimination when called using left division `\`. The syntax used is `vecx = A \ b;`. (This can be remembered by mentally equating `A \ b` to `A ^ -1 * b`.)

Many types of research deal with **sparse matrices**. These are matrices where the majority of entries are `0` as opposed to dense matrices with many nonzero entries. MATLAB is well-equipped with functions such as `sparse()` to create the `sparse double` class to make storage easier. I will not spend any additional time on this matter.

Math Operations: Statistics

Just like any good spreadsheet, MATLAB has accessible functions to perform common **statistical operations**. We make some modifications to the active portion of "workshop.m".
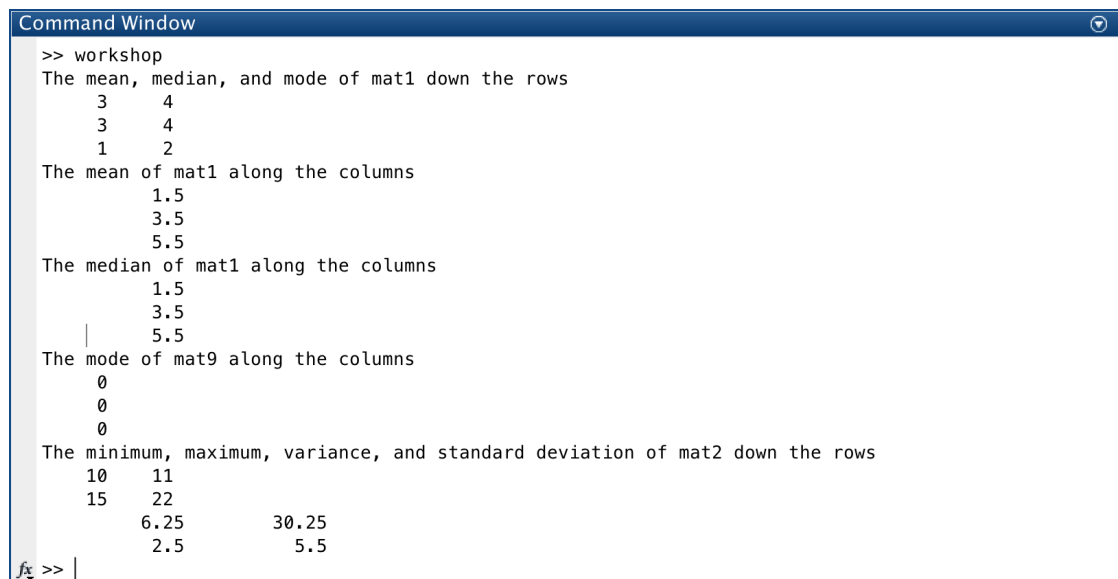
```
mat1 = [1, 2; 3, 4; 5, 6];
% disp('The length, number of elements, and size of mat1')
% disp(length(mat1))
% disp(numel(mat1))
```

```matlab
% disp(size(mat1))
mat2 = [linspace(10, 15, 3).', linspace(22, 11, 3).'];
mat3 = 1/2 * mat1 + 1j * mat2;
mat4 = mat3.';
mat5 = mat3';
mat6 = randi([0, 20], size(mat1));
mat7 = zeros(3, 4);
mat8 = ones(2);
mat9 = eye(3);
mat10 = mat6 * mat5;
mat11 = mat3 .^ 2;
vecx = [3, 1; -4, -2] \ [-1; 0];

disp('The mean, median, and mode of mat1 down the rows')
disp(mean(mat1))
disp(median(mat1))
disp(mode(mat1))
disp('The mean of mat1 along the columns')
disp(mean(mat1, 2))
disp('The median of mat1 along the columns')
disp(median(mat1, 2))
disp('The mode of mat9 along the columns')
disp(mode(mat9, 2))
disp(['The minimum, maximum, variance, and standard deviation of ' ...
'mat2 down the rows'])
disp(min(mat2))
disp(max(mat2))
disp(var(mat2))
disp(std(mat2))
```

```
Command Window                                                    ⊙
  >> workshop
  The mean, median, and mode of mat1 down the rows
        3     4
        3     4
        1     2
  The mean of mat1 along the columns
          1.5
          3.5
          5.5
  The median of mat1 along the columns
          1.5
          3.5
       |  5.5
  The mode of mat9 along the columns
        0
        0
        0
  The minimum, maximum, variance, and standard deviation of mat2 down the rows
       10    11
       15    22
            6.25        30.25
            2.5           5.5
fx >> |
```

Run the script by pressing the green "Run" button (F5 in Windows). Let's make some observations about the **statistical operations**:

- MATLAB has all the *measures of central tendency*: (arithmetic) mean, median, and mode. Arithmetic mean or average (mean()) is the sum of all values divided by the number of values. Median (medi-

an()) is the center value of the ordered list of values, or the arithmetic mean of the two centermost values for even lengths of lists. Mode mode() is the most frequently occurring value in a set, though multiple modes may exist.

- The default MATLAB behavior is to show the results of each statistical operator taken along the first dimension of length greater than 1. For matrices and arrays, this means that the statistical result for each row must require a second argument in the operator, the dimension to operate along.

- The preference for taking results along each column means that it is typical to store data in MATLAB as arrays with entries in a given column all representing measurements of the same quantity. Each row represents a collection of simultaneous measurements typically.

- Should other types of means be required, MATLAB has geomean() for the geometric mean and harmmean() for the harmonic mean. The root-mean-square value of a set rms() used often in engineering is also available and works similarly.

- The ellipsis formed from three periods ... serves as the line continuation character in MATLAB, allowing a single command to span multiple lines in the editor to maintain readability. Continuing strings across multiple lines is somewhat tricky. Essentially, each line contains a standalone string demarcated with single quotation marks. The presence of square brackets gives away what MATLAB is doing: the resulting string is merely a horizontal concatenation of smaller string literals.

- MATLAB also has functions for measuring *dispersion*: min(), max(), range(), iqr(), var(), and std(). The minimum is the smallest values in a set. The maximum is the largest value in a set. The range (no example given) is the difference of the maximum and the minimum. The interquartile range, *IQR* (no example given), is the difference of the seventy-fifth percentile and twenty-fifth percentile. Variance var() is the mean of the squares of each value minus the mean. Standard deviation std() is the square root of the variance.

- Note that var() and std() use the concept of *sample standard deviation*, which uses a divisor of $N - 1$ in the calculation, instead of the *population standard deviation*, which uses a divisor of N.

- MATLAB also has the ability to calculate *moments* such as skewness skewness() and kurtosis(). These are higher-order generalizations of mean and variance that are normalized by the standard deviation. These are seldom used outside of statistics, so not much attention will be paid here.

- MATLAB can also determine the covariance and correlation coefficients between samples of two distributions with cov() and corrcoef(). These functions can be handy when investigating trends and calculating the coefficient of determination R^2, but these intricacies will not be explored now.

- Users who rely on these statistical measures often should consider having them displayed for each variable in the *Workspace toolbar*.

Not-a-number NaN is the enemy of data collection. Each of the statistical operators has optional arguments for how NaN is treated. There are also function such as nanmean() that make it more obvious how NaN is treated; in this case, it is ignored. Most of the time, it is far better to check for the presence of NaN using functions such as isnan() or constructs such as if sum(sum(isnan(A))) ~= 0 and purge the data ahead of time rather than work around and try to interpret how NaN affects your analysis.

Strings

We naturally have worked with strings with disp() when making output to the *Command Window* appear more clean. We have indirectly seen how MATLAB automatically concatenates string literals into

a row character array (i.e., a larger string) when using the line continuation character. There are many other string functions available that are similar to those in the C language: `strcmp()`, `strfind()`, and `strrep()`. Like any good programming language, MATLAB supports regular expressions with `regexp()` that manage to baffle students around the globe.

Since there are too many basic topics to cover today, we will mostly leave strings alone. The majority of MATLAB users are using the software for numeric data anyway, so it is possible to go years without needing any string functions.

However, it would be nice to find a better way of formatting data in the *Command Window*. I advocate the use of `fprintf()` (https://www.mathworks.com/help/matlab/ref/fprintf.html) to replace `disp()` in all cases where scalars are used. The first argument of `fprintf()` is an integer called the file identifier. The file identifier 1 is reserved for standard *Command Window* output, and the file identifier 2 is reserved for *Command Window* errors. The second argument of `fprintf()` is the format string. The subsequent arguments are all data inputs to the format string taken columnwise.
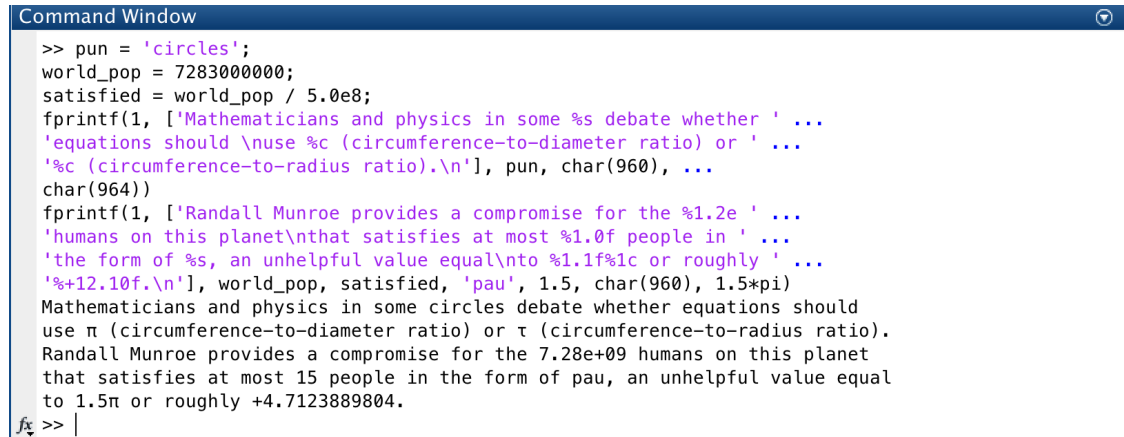
MATLAB has its own *format specification* syntax on which a respectable instructor should share a reference (https://www.mathworks.com/help/matlab/matlab_prog/formatting-strings.html). Here are the highlights:

- Each format variable begins with with the percent sign `%`.

- `fprintf()` will use each data input in order for each corresponding format variable. I do not recommend changing the order, but it can be done through the use of *value identifiers*.

- *Flags* determine what shows up before numeric data: spaces, padded zeros (leading zeros that come before nonzero digits left of the decimal point), the plus or minus signs, or left-justified omitting signs where possible.

- The *field width* designates the minimum number of characters that the formatted output should occupy. Unused characters are padded with spaces on the left. The field width does nothing if the output **must** take up more spaces than allotted since the formatted output will expand accordingly.

- The *precision* is either the number of characters to follow the decimal point (or decimal point equivalent for other numeric bases) for floating-point representations **or** the number of places after the decimal for scientific notation representations (i.e., one less than the number of significant figures).

- A *conversion character* denotes how the information should be displayed. `c` is used for single character, and `s` is used for string. `d` is used for signed decimals, and `u` is used for unsigned decimals. For base 8 (octal), `o` is used, while `x` is used for base 16 (hexadecimal). When it comes to fixed-point decimals, `f` is the best. Exponential notation uses `e`, and `g` will automatically choose between exponential notation or fixed-point notation depending on which is more compact. `E`, `G`, and `X` behave like their lowercase counterparts except that they are displayed with capital letters whenever they are needed to represent numbers.

Here is a quick example for the MATLAB *Command Window*:

```
pun = 'circles';
world_pop = 7283000000;
satisfied = world_pop / 5.0e8;
fprintf(1, ['Mathematicians and physics in some %s debate whether ' ...
 'equations should \nuse %c (circumference-to-diameter ratio) or ' ...
'%c (circumference-to-radius ratio).\n'], pun, char(960), ...
char(964))
```

```matlab
fprintf(1, ['Randall Munroe provides a compromise for the %1.2e ' ...
'humans on this planet\nthat satisfies at most %1.0f people in ' ...
'the form of %s, an unhelpful value equal\nto %1.1f%1c or roughly ...
 ' ...
'%+12.10f.\n'], world_pop, satisfied, 'pau', 1.5, char(960), 1.5*pi)
```

```
Command Window                                                          ⊙
  >> pun = 'circles';
  world_pop = 7283000000;
  satisfied = world_pop / 5.0e8;
  fprintf(1, ['Mathematicians and physics in some %s debate whether ' ...
  'equations should \nuse %c (circumference-to-diameter ratio) or ' ...
  '%c (circumference-to-radius ratio).\n'], pun, char(960), ...
  char(964))
  fprintf(1, ['Randall Munroe provides a compromise for the %1.2e ' ...
  'humans on this planet\nthat satisfies at most %1.0f people in ' ...
  'the form of %s, an unhelpful value equal\nto %1.1f%1c or roughly ' ...
  '%+12.10f.\n'], world_pop, satisfied, 'pau', 1.5, char(960), 1.5*pi)
  Mathematicians and physics in some circles debate whether equations should
  use π (circumference-to-diameter ratio) or τ (circumference-to-radius ratio).
  Randall Munroe provides a compromise for the 7.28e+09 humans on this planet
  that satisfies at most 15 people in the form of pau, an unhelpful value equal
  to 1.5π or roughly +4.7123889804.
fx >> |
```

Note that the character that results from passing an integer to `char()` can vary based on operating system and the character encoding used. Results can differ between devices. For that reason, be careful when relying on output to the *Command Window* when displaying non-ASCII characters.

The function `fprintf()` will not produce an output unless forced. When forced to produce an output, it returns the number of bytes written. A similar function called `sprintf()` exists that does not take a file identifier as the first argument and **always** returns a string. This function may seem less useful until we discuss plotting.

Dates

At this time, I would like to acknowledge that dates exist in MATLAB. Having access to real time is handy when logging data. The function `date` without arguments returns today's date as "DD-MMM-YYYY". Similarly, `clock` returns a vector describing the time down to fractions of a second. Stopwatches can be started with `tic` and ended with `toc`, though this hardly the preferred way to time events in Java-heavy MATLAB. There are a lot of intricacies regarding string representation and numeric representation of dates, but all you need to know for now is that time-related functions exist to help you.

Saving Data: MATLAB Data Files (.mat)

It is useful to save large sets of numeric data in non-volatile memory when MATLAB is not being used. Conveniently, MATLAB has a save function named `save()`.

There is probably a lot of clutter that has accumulated in your *Command Window* and *Workspace toolbar*. Three commands can help you. Even though we have not covered plotting and figures yet, `close()` can close out of figure windows that you have made. To remove variables from the *Workspace*, use the function `clear()` or `clearvars()`. To wipe the *Command Window* clean, use the command `clc` without any parentheses. When you write scripts, it is good practice to put the commands as follows in your header to avoid contamination of data between projects and prove that no cheating occurred beforehand.

```matlab
close('all')
```

```
clearvars
clc
```

Append the following to the end of "workshop.m" before running the script:

```
save('workshop_data.mat', 'mat1', 'mat2', 'mat6', 'vecx')
clear('all')
```
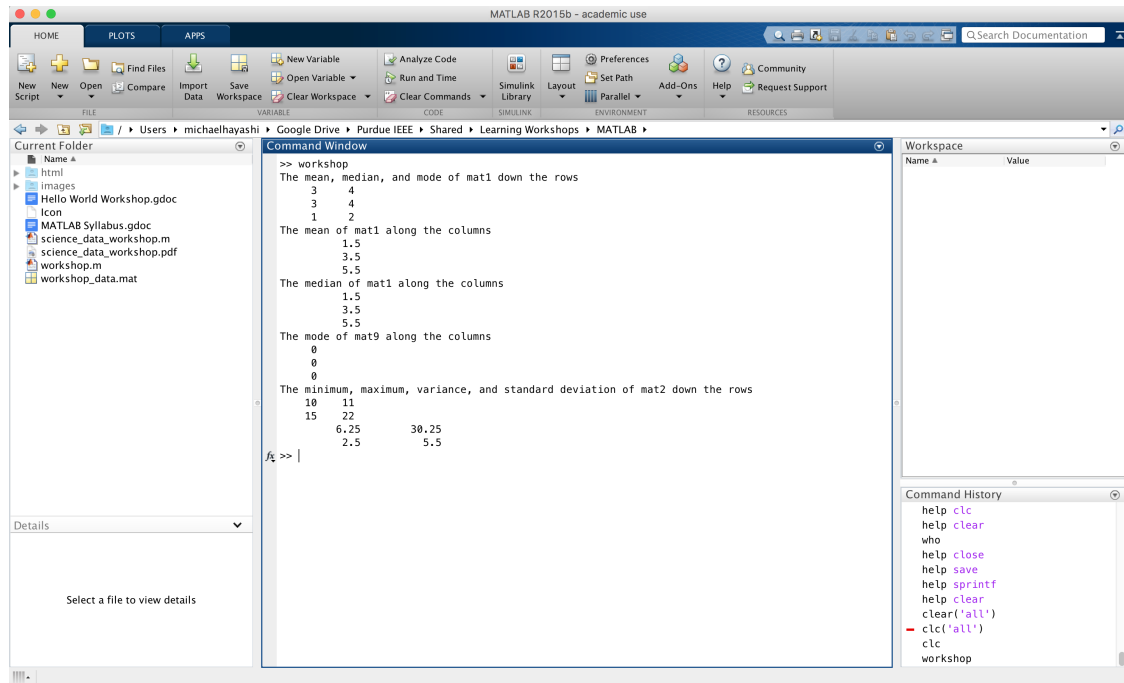

Two major changes have occurred since we last ran "workshop.m":

1. The *Workspace toolbar* is now blank, signifying that no variables are currently stored in memory.

2. A file named "workshop_data.mat" has appeared in the *Current Folder toolbar*.

The preferred MATLAB file type is .mat for storing data in a ready form for MATLAB to load. The *Details* toolbar under the *Current Folder toolbar* is even able to preview the contents of .mat files when selected. There are two ways to load .mat files specifically into the *Workspace*:

1. Drag and drop "workshop_data.mat" using the cursor from the *Current Folder toolbar* to the *Command Window*.

2. Append the following line to "workshop.m" before running:

```
load('workshop_data.mat')
```
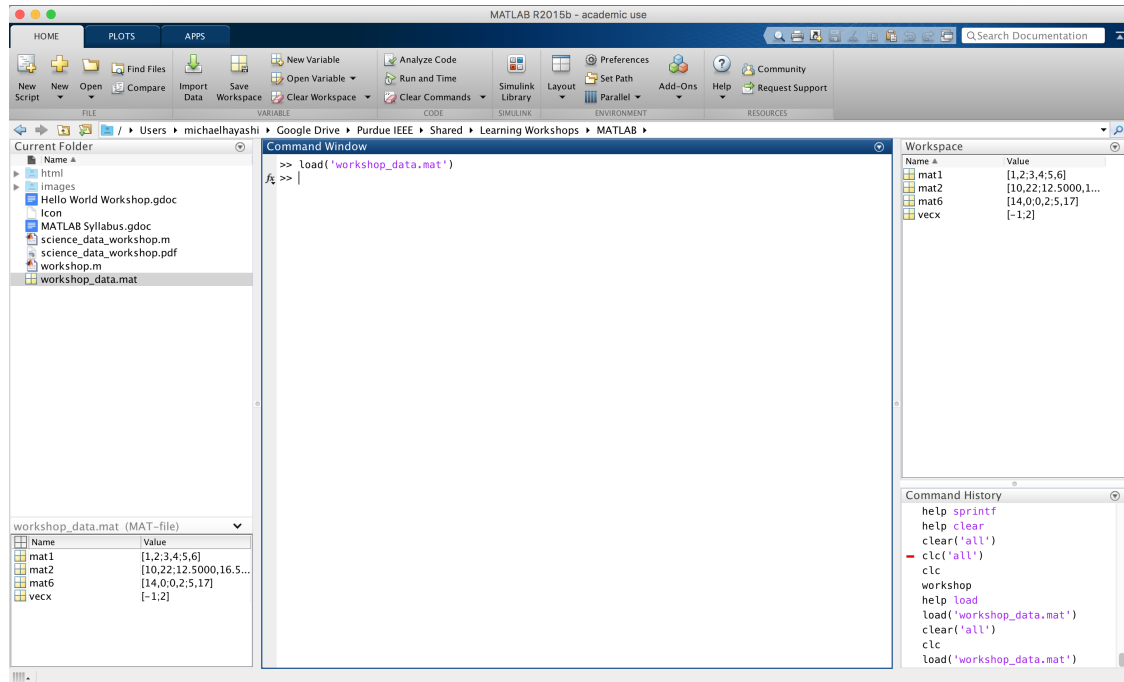
Once variables are in the *Workspace*, they may be manipulated by commands and scripts just like any other variable. While the temptation might be to store all intermediate steps when handling data, the real-world constraints of file sizes means that only aggregations of raw data are typically stored. Sometimes, final calculations are stored alongside figures and tables.

Saving Data: Loading Other File Types

Suppose that you want to load some non-native file into MATLAB for processing, such as a .csv file (Comma-separated value), .xls file (Microsoft Excel workbook), .au file (Unix audio), .jpg file (Joint Photographic Experts Group), or .mov file (Apple QuickTime video). The `importdata()` function is a really powerful function that attempts to predict how you want the data stored in a variable by analyzing the file extension in the string passed to it. The `importdata()` function will handle most of the setup for functions such as `sscanf()`, `textscan()`, and `xlsread()`. While it is sometimes useful to use functions such as `csvread()` (to load CSV files) or `dlmread()` (to load other ASCII delimited files), most of the time you can let `importdata()` take care of the specifics for you.

Here is a simple, built-in example that you can do in the *Command Window*:

```
nebula_im = importdata('ngc6543a.jpg');
image(nebula_im)
```

NGC 6543

PR95-01a · ST ScI OPO · January 1995 · P. Harrington (U.MD), NASA

HST · WFPC2

12/13/94 zgl
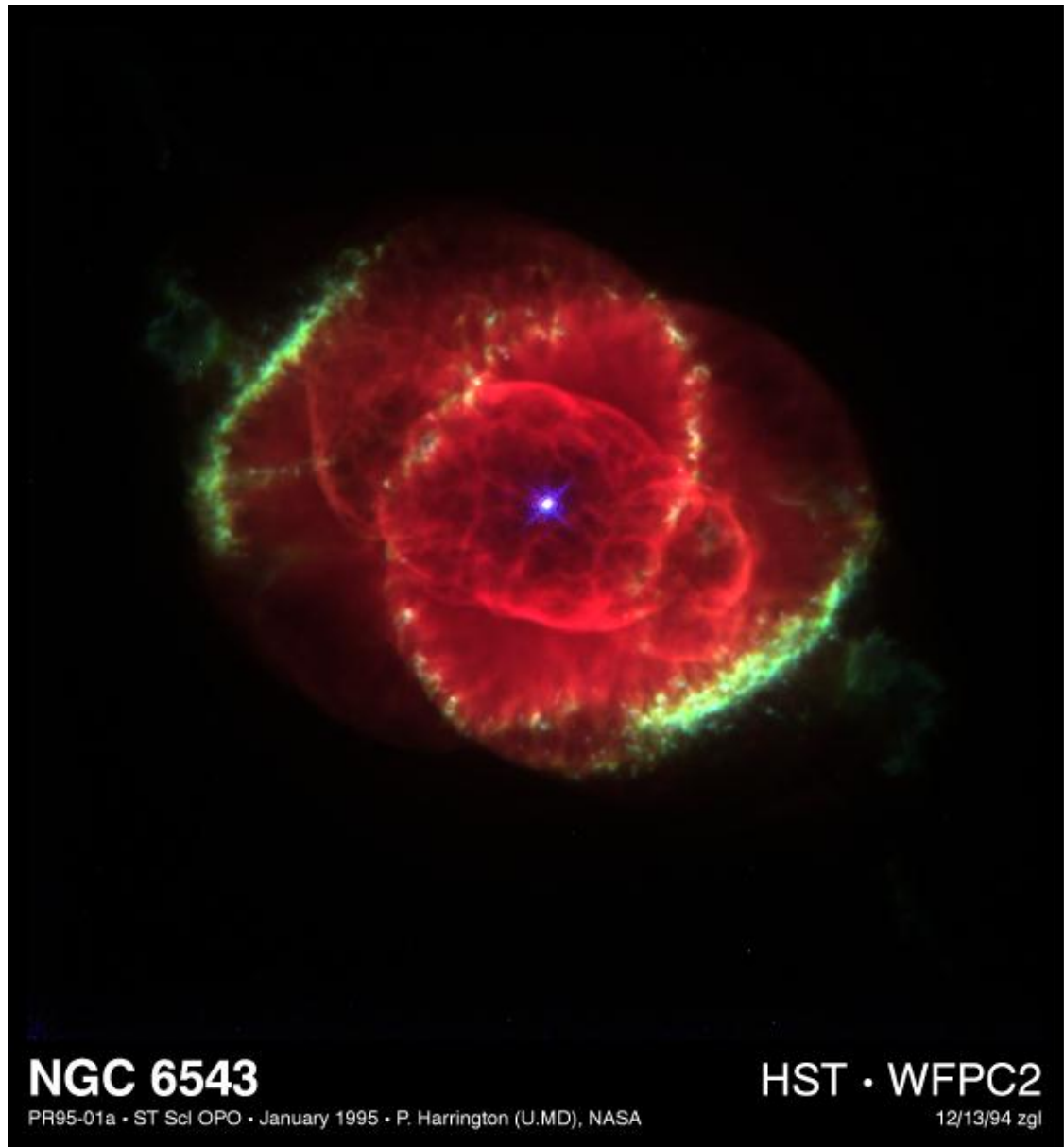
When you are done with the image, you can close out of the figure window with the cursor or type the following:

```
close(1)
```

`importdata()` is a good function of first resort. The second resort is usually to use `uiimport()` and follow the graphical import wizard steps. The next resort is usually to learn how the more specific functions work if the data is not loading properly.

# Saving Data: Text File I/O

At this point, it is useful to cover low-level file input/output (I/O). There are several functions in MATLAB reminiscent of the C language to handle file I/O (https://www.mathworks.com/help/matlab/low-level-file-i-o.html). Simple text files can store basic information about what was performed in MATLAB as well as

transport information between MATLAB and other software you or your colleagues might use. Highlight the active text in "workshop.m", and press the green percent sign "Comment" (Ctrl + R in Windows) in the "Edit" section of the "Editor" ribbon. Go to a blank line below the commented out script. It is time to move on to **writing simple text files**.

```matlab
x = linspace(0, 2, 25);
quad_func = 2 * (x - 1) .^ 2 + 1;

fileid = fopen('workshop_quad.txt', 'w');
fprintf(fileid, 'Quadratic Function Example\n');
fprintf(fileid, '%1.4f %1.4f\n', [x; quad_func]);
fclose(fileid);
```

When you run "workshop.m", you should see a file appear in the *Current Folder toolbar* named "workshop_quad.txt". Double click that file to open it up in the editor.
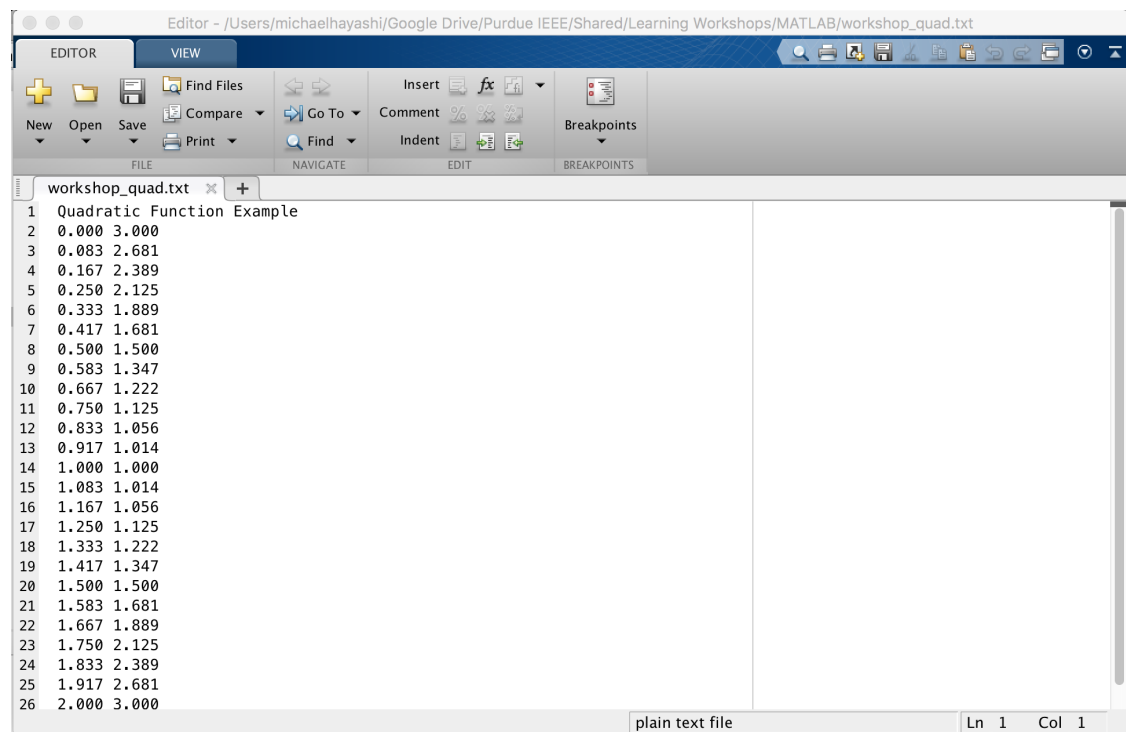
Here are some basic observations about writing to text files:

- Anytime we interact with an external file, we need to use `fopen()` in the first place. The first argument is the name of the file to read or modify or it can be the name of the file to create in the *Current Folder*.

- There are *permissions* that must be written out when using `fopen()`. Permissions prevent MATLAB from doing undesirable things on our behalf. The three main types of permissions are read `'r'`, write new/overwrite existing contents `'w'`, and append to end of file `'a'`. There are also enhanced permissions, namely read and overwrite an existing file `'r+'`, write a new file and read it/overwrite an existing file and read it `'w+'`, and append to the end of a file and read it `'a+'`. Attempting to append to a file that does not exist yet will create a new file.

- The output of `fopen()` is a file identifier that must be used whenever file I/O functions are used. Checking the file identifier in the *Workspace toolbar*, it takes on an integer-valued double-precision

number that is usually incrementally assigned. (Recall from the discussion of `fprintf()` that `1` is reserved for *Command Window* output and that `2` is reserved for *Command Window* errors.) The value is unimportant since file identifiers for external files should always be referenced through a variable.

- We see the true power of `fprintf()` here: being able to write formatted text not just to the *Command Window* but to any ASCII file. Remember to include newlines `'\n'`.

- In a move that baffles me, MATLAB accepts matrices as data inputs to `fprintf()` as long as it can somehow interpret the format string to them. This is much easier than relying on a `for` loop, but the row/column behavior is not immediately obvious. MATLAB is seen trying to take each *column* of our matrix and transform it into a *row* of the text file. The number of rows created by the `fprintf()` call is equal to the number of columns in the matrix passed to it. From a visual standpoint, `fprintf()` has transposed the matrix data input.

- It is proper form to close any file opened by calling `fclose()` on the file identifier. This prevents any unauthorized access by the program once you are done with all input and output. The tricky thing is being able to close the file **if an error occurs as your script is interpreted**.

Highlight the active text in "workshop.m", and press the green percent sign "Comment" (Ctrl + R in Windows) in the "Edit" section of the "Editor" ribbon. Go to a blank line below the commented out script. It is time to move on to **reading simple text files**.

```
clearvars
fileid = fopen('workshop_quad.txt', 'r');
title = fgetl(fileid);
data = fscanf(fileid, '%f %f\n', [2, Inf]);
fclose(fileid);
```

```
Workspace                        ⊙
Name ▲              | Value
⊞ ans               | 0
⊞ data              | 2x25 double
⊞ fileid            | 3
abc title           | 'Quadratic Functi...
```

Some observations can be made about reading simple text files:

- Text files can be difficult to read if there is intervening text at the top. Despite the setbacks, the powerful tools `importdata()` and `uiimport()` do a decent job of figuring out what to do.

- The read *permission* `'r'` is the only one needed with `fopen()`.

- The function `fgetl()` will "get" the next line in the file and return it. This is handy when you need a fine degree of manual control over file I/O. The function `fgets()` does the same thing except that newline characters are not stripped out.

- The function `fscanf()` will attempt to read formatted data in a file and return an array with the data. If the format specifier reads multiple data per line, then the output array is filled going *down the columns* and then *right through the rows*. Be aware of this behavior. If the number of lines to read is unknown, you may allow `fscanf()` to output up to an infinite number of columns.

- When MATLAB was reading the file, it "knew" that it had already processed the title. There is a concept called a *file pointer* or *file position indicator* that keeps track of the number of bytes into the file a read

or write operation is. Functions such as `ftell()`, `fseek()`, and `frewind()` give low-level control over the file position indicator that should seldom be needed if the file was sensibly and methodically written. Note: these functions go back to the days of magnetic tape when there was a recording head and a reel to control and C functions actually meant that mechanical parts were moving.

- There is an invisible, whitespace character at the end of every plain text file called "EOF" (*end-of-file character*, usually represented as ASCII 004, the end-of-transmission control character). Attempting to read past the end of a file will cause functions such as `fgetl()` to return `-1`. The function `feof()` detects the "EOF" and returns a logical so that the user can properly fix the situation.

- Again, it is proper to close any open file, even if there were errors that interrupted the script as it was being interpreted.

Flow of Control: `if` Statements and `for` Loops

Rounding out the basics of any programming language is *flow of control* or *control flow*, the ability to change the order in which code is evaluated based on decisions or loops. Rather than teach the concepts in isolation, we will do a power example by implementing a variant of the popular programming test, Fizz Buzz. Highlight the active text in "workshop.m", and press the green percent sign "Comment" (Ctrl + R in Windows) in the "Edit" section of the "Editor" ribbon. Go to a blank line below the commented out script to play **Fizz Buzz**.

```matlab
game_start = 1;
game_end = 200;
for indi = game_start:game_end
    str_out = '';
    if mod(indi, 3) == 0
        str_out = [str_out, 'Fizz'];
    end
    if mod(indi, 5) == 0
        str_out = [str_out, 'Buzz'];
    end
    if mod(indi, 7) == 0
        str_out = [str_out, 'Bang'];
    end
    if mod(indi, 3) * mod(indi, 5) * mod(indi,7) ~= 0
        str_out = num2str(indi);
    else
        if length(str_out) == 4
            str_out = ['(', str_out, ')'];
        elseif length(str_out) == 8
            str_out = ['[', str_out, ']'];
        else
            str_out = ['{', str_out, '}'];
        end
    end
    fprintf(1, '%s\n', str_out);
end
```

```
Command Window                                                              ⊙
>> workshop
1
2
(Fizz)
4
(Buzz)
(Fizz)
(Bang)
8
(Fizz)
(Buzz)
11
(Fizz)
13
(Bang)
[FizzBuzz]
16
17
(Fizz)
19
(Buzz)
[FizzBang]
22
23
(Fizz)
(Buzz)
26
(Fizz)
(Bang)
29
[FizzBuzz]
31
32
(Fizz)
34
[BuzzBang]
(Fizz)
fx  37
```
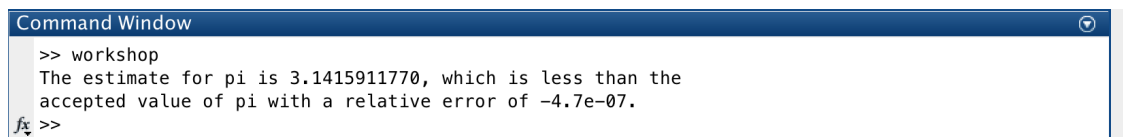
Analyzing this script that we wrote:

- In this variant of Fizz Buzz, we are working with the numbers 1 through 200. If the number is divisible by three, then the word "Fizz" is displayed in place of the number. If the number is divisible by five, then the word "Buzz" is displayed in place of the number. If the number is divisible by seven, then the word "Bang" is displayed in place of the number. Numbers that are divisible by some combination of three, five, and seven have the corresponding words displayed in order in CamelCase. If the number is not divisible by three, five, or seven, then the numerals are displayed. If a single word replaces a number, then the word is put in parentheses. If two words replace a number, then the word is put in square brackets. If three words replace a number, then the word is put in curly braces.

- `for` loops have a unique syntax in MATLAB. They behave more like "for each" in Python rather than a pure "for" loop as in C language. The iterator follows the word `for`. An equal sign = goes in between the iterator and the vector whose entries are being interated over, even though this is not an assignment in the strictest sense.

- All control of flow structures in MATLAB use the keyword `end` to mark the extent of the block. The default editor should automatically provide an indent of four spaces.

- My solution relies on modulus, the remainder after division, accessible through the two-argument function `mod()`. If a dividend is perfectly divisible by a given divisor, then the answer will be zero. Note that the condition for printing the number unmodified requires that modulus result of the checks against three, five, and seven all be nonzero.

- `if` statements are pretty normal in MATLAB. Some result of class `logical` must follow the word `if`, so logical operators acting on variables are a natural choice.

- Functions such as `num2str()`, `mat2str()`, and `str2num()` make it handy to convert information that can be stored as multiple data types between those data types.

- `if` statements can progressively check criteria by using `elseif` within the same structure. The universal alternative decision is given under `else` if needed.

Flow of Control: `switch` Statements and `while` Loops

There are two more flow of control structures to analyze, so this power example will combine them with a numerical estimation of π. Highlight the active text in "workshop.m", and press the green percent sign "Comment" (Ctrl + R in Windows) in the "Edit" section of the "Editor" ribbon. Go to a blank line below the commented out script to start the numerical estimation of π.

```
targ_err = 4.7e-7;
indk = 0;
pi_est = 4.0;
rel_err = (pi_est - pi) / pi;
while abs(rel_err) > targ_err
    indk = indk + 1;
    pi_est = pi_est + (-1) ^ indk * 4 / (2 * indk + 1);
    rel_err = (pi_est - pi) / pi;
end
switch sign(rel_err)
    case +1
        word = 'greater than';
    case -1
        word = 'less than';
    otherwise
        word = '(within the precision of MATLAB) equal to';
end
fprintf(1, ['The estimate for pi is %1.10f, which is %s the\n' ...
    'accepted value of pi with a relative error of %1.3g.\n'], ...
    pi_est, word, rel_err);
```

```
Command Window                                                    ⊙
  >> workshop
  The estimate for pi is 3.1415911770, which is less than the
  accepted value of pi with a relative error of -4.7e-07.
fx >>
```

Analyzing this script that we wrote:

- The algorithm that we are using to estimate π is the *Leibniz formula*. It is based on the Maclaurin series of $\arctan(x)$ and the fact that $\arctan\left(\frac{\pi}{4}\right) = 1$. The Leibniz formula $\pi = \sum_{k=0}^{\infty} \frac{(-1)^k 4}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots$ converges sublinearly and forms an alternating series, which makes it awful to use as is to estimate π without modifications but is suitable for a demonstration of flow of control with an unclear number of iterations suggesting the use of `while`.

- Since the relative error `rel_err` can be positive or negative, make sure to use `abs()` to find the absolute value before comparing to the target error `targ_err`.

- Make sure to increment the index and recalculate the relative error used in the *loop condition* to avoid the *infinite loop*. Like many other programming languages, you can stop script execution by the Ctrl + C combination in Windows, macOS (**not** command), and Linux.

- The `switch`, `case`, and `otherwise` syntax is an alternative to the `if`, `else if`, `else` syntax for decisions. This syntax is fairly common in languages that support it, even though it is rather situational. Here the `signum()` function is used in a contrived fashion returns +1 for positive quantities, -1 for negative quantities, and 0 for zero quantities. The code under each `case` block is executed if the *switch statement* evaluates equal to the *case statement*. Certain relational operators are banned. Using a `switch` statement over an `if` statement is sensible when it might take the computer a lot of time to evaluate a complicated logical expression that governs the decision.

- Notice that it took 677,255 iterations to converge to the desired relative error. A `while` loop was certainly justified over trying to allocate a vector of that length in memory for a `for` loop instead.

Matrix Indexing

Unlike the majority of programming languages in common use, MATLAB is *one-indexed* instead of *zero-indexed*. This means that the first element of an array or any collection data type is stored at element number 1 instead of element number 0. The defense of this choice is largely based on maintaining backwards compatibility with previous MATLAB versions the way that Cleve Moler, the creator of MATLAB, originally wrote the language. Much to the consternation of traditional computer programmers the world over, the 1-based index of MATLAB works out well for most users. Unlike other languages where subscripting and indices correspond to offsets in memory, usually MATLAB arrays are conceptualized as collections of countable things. This way of enumerating entries in a matrix is consistent with how linear algebra is taught.

Highlight the active text in "workshop.m", and press the green percent sign "Comment" (Ctrl + R in Windows) in the "Edit" section of the "Editor" ribbon. Go to a blank line below the commented out script to visualize **matrix indexing**. It is straightforward to modify elements of an array with the assignment operator =.

```
clc
mat1 = [(1:6).', (7:12).', (13:18).', (19:24).', (25:30).', ...
 (31:36).'];
disp('Original matrix:')
disp(mat1)
mat1(2,1) = -7;
disp('Matrix with entry reassigned:')
disp(mat1)
mat2 = mat1(2:end,1:3);
disp('Matrix sliced in two dimensions:')
disp(mat2)
mat3 = mat1([1,3,6],:);
disp('Selected rows from matrix:')
disp(mat3)
mat4 = mat1(:,[floor(size(mat1, 2)/2) + 1:end, ...
    1:floor(size(mat1, 2)/2)]);
disp('Matrix with left and right swapped:')
disp(mat4)
```

```matlab
mat5 = mat1(end-1:-1:1, 2:2:end);
disp('Matrix strided backwards and forwards:')
disp(mat5)

disp('Logical Indexing')
ind_div_by_5 = (mod(mat1, 5) == 0);
disp('Logical matrix of locations of multiples-of-5:')
disp(ind_div_by_5)
disp('Vector with multiples-of-5 extracted:')
disp(mat1(ind_div_by_5).')
disp('Vector of values between 5 and 31:')
disp(mat1((mat1 >= 5) & (mat1 <=31)).')
ind_near_pi_mult = find(abs(mod(mat1, pi) - pi/2) >= pi/2 - 0.5);
disp('Vector of values within one-half of pi:')
disp(mat1(ind_near_pi_mult).')
disp('Vector result using same indices on different matrix:')
disp(mat3(ind_near_pi_mult).')
```

```
Command Window                                                          ⊙
  Original matrix:
       1     7    13    19    25    31
       2     8    14    20    26    32
       3     9    15    21    27    33
       4    10    16    22    28    34
       5    11    17    23    29    35
       6    12    18    24    30    36
  Matrix with entry reassigned:
       1     7    13    19    25    31
      -7     8    14    20    26    32
       3     9    15    21    27    33
       4    10    16    22    28    34
       5    11    17    23    29    35
       6    12    18    24    30    36
  Matrix sliced in two dimensions:
      -7     8    14
       3     9    15
       4    10    16
       5    11    17
       6    12    18
  Selected rows from matrix:
       1     7    13    19    25    31
       3     9    15    21    27    33
       6    12    18    24    30    36
  Matrix with left and right swapped:
      19    25    31     1     7    13
      20    26    32    -7     8    14
      21    27    33     3     9    15
      22    28    34     4    10    16
      23    29    35     5    11    17
      24    30    36     6    12    18
  Matrix strided backwards and forwards:
      11    23    35
      10    22    34
       9    21    33
       8    20    32
       7    19    31
```

```
  Logical Indexing
  Logical matrix of locations of multiples-of-5:
      0     0     0     0     1     0
      0     0     0     1     0     0
      0     0     1     0     0     0
      0     1     0     0     0     0
      1     0     0     0     0     1
      0     0     0     0     1     0
  Vector with multiples-of-5 extracted:
      5    10    15    20    25    30    35
  Vector of values between 5 and 31:
    Columns 1 through 16
      5     6     7     8     9    10    11    12    13    14    15    16    17    18    19    20
    Columns 17 through 27
     21    22    23    24    25    26    27    28    29    30    31
  Vector of values within one-half of pi:
      3     6     9    13    16    19    22    25    28    31    35
  Vector result using same indices on different matrix:
     21    24    27    31    34     1     4     7    10    13    17
fx >> |
```

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

- MATLAB indexes matrices by *row first* and *column second*. So `mat(indm, indn)` refers to the mth entry down and the nth entry to the right. By referencing a specific entry or group of entries as a target for assignment, the values of that entry or group of entries is overwritten in memory while the rest of the matrix remains unchanged.

- The colon `:` is used for *slicing* the matrix to obtain contiguous parts. The keyword `end` gives easy access to the value representing the last row or last column. By itself, the colon `:` represents the entire range of indices available.

- A vector of indices may be given for the row or column to *slice* the matrix into noncontiguous parts.

- *Striding* refers to modifying the interval between array elements in a slice. The syntax for indices looks like `start:stride_pitch:stop`, where the center number controls the interval between elements. A positive value for the interval greater than `1` will result in skipped numbers in the forward direction. A negative value for the interval will read the elements backwards.

- *Logical indexing* is a powerful MATLAB technique to extract only the entries of a vector or matrix that meet some condition. This was touched upon when a vector was given for the indices of a matrix. Notice that a matrix of logicals results when performing the logical operators such as AND, OR, and NOT along with the relational operators such as equal to or less than or equal to. When a matrix of logicals is passed as indices to a matrix, the output is a vector where the matrix of logicals' entries are `true`.

- The `find()` function works alongside *logical indexing* very well. The first argument is a matrix of logicals, usually the result of relational operators and logical operators performed on the matrix under examination. The second argument is the maximum number of indices to return, and the third argument is `'last'` if the condition should start being tested from the end of the vector or matrix. `find()` returns a column vector with the indices where the matrix of logicals evaluated `true`. Passing the resulting column vector to the matrix or vector under examination will produce a vector with entries meeting the desired condition.

Scripts and Functions

To finish today's workshop, it is worthwhile to impart some knowledge that will create cleaner scripts. Working in scripts as we have done so far makes it easier to revisit code and make edits rather than copy and paste in the *Command Window*.

In any programming language, commenting is a vital part of software development. Leaving descriptive comments of code sections, nonintuitive groupings of operations, and variable representing obtuse ideas is necessary for the understanding of reviewers, future users, and yourself years after you have left development of any given piece of software. A lot of grief can be prevented by periodically reviewing what your have written and adding comments. Depending on the code standards for your supervisory body, you may end writing scripts with more lines of comments than executable code the majority of the time.

Create a new file called "water_pot_cooling_plot.m". I recommend descriptive names for scripts based on the application. Put the following comments in your new script, populating placeholder fields with your own information. I recommend putting *header* information at the start of all your scripts for course or research purposes.

```
% Author: <First> <Last>
% <Affiliation>
% <Optional: Author Contact Information>
% <Course/Research Group/Event>
% <Assignment/Research Project/Exercise>
% <Day> <Month> <Year>
% <Supplemental Information>
close('all')
clearvars
clc
```

In particular, I used the following information. Notice that nothing is executed when the script is run.

```
% Author: Michael Hayashi
% Purdue IEEE Student Branch
% Email: mhayashi@purdue.edu
% Introduction to MATLAB Workshop
% Analysis of Cooling Pot of Water
% 16 November 2017
% Accessible: https://purdueieee.org/learning/matlab
% Version: MATLAB(R) R2015b
% Runtime: 4.3 s
close('all')
clearvars
clc
```

Long scripts can be tiring to scroll through. MATLAB has *cell mode* to divide scripts into sections to help developers (https://www.mathworks.com/help/matlab/matlab_prog/run-sections-of-programs.html). A section starts with a special comment with two percent signs and a space |%% |. The rest of the comment on that line is bolded to serve as the *section heading*. The rest of the code is part of that section until the next section heading. Active sections (the one containing the cursor) have a yellow background, and inactive sections have a white background. Sections show up in the *Details* toolbar when a file is highlighted in the *Current Folder* toolbar. Comments immediately below the section heading with no intervening whitespace are treated as *descriptive text* for the heading, more or less a paragraph explaining what the section does. Our first section will be called "Header" and serve as the first line, and the subsequent sections go below the code we have written.

```
%% Header
% Author: <First> <Last> ...

%% Import Water Data from Excel Workbook

%% Convert Units
```

```
%% Plot Data

%% Save Figures to File
```

There are special buttons in the "Run" section of the "Editor" ribbon in the editor window. "Run Section" only executes the active section with the yellow background, helpful for finding errors and fixing small mistakes that arise after running the whole script. "Run and Advance" executes the active section and advances the cursor into the next section. "Advance" skips the current section and advances the cursor into the next section.

Sometimes, there are chunks of code that need to be repeated often enough to be annoying. To that end, it is useful to call custom *functions* or *subroutines* while executing a MATLAB script. A new function in MATLAB can be created by going to "New" in the "File" section of the "Editor" ribbon in the editor window or by including the `function` keyword at the beginning of a script and following the appropriate syntax.
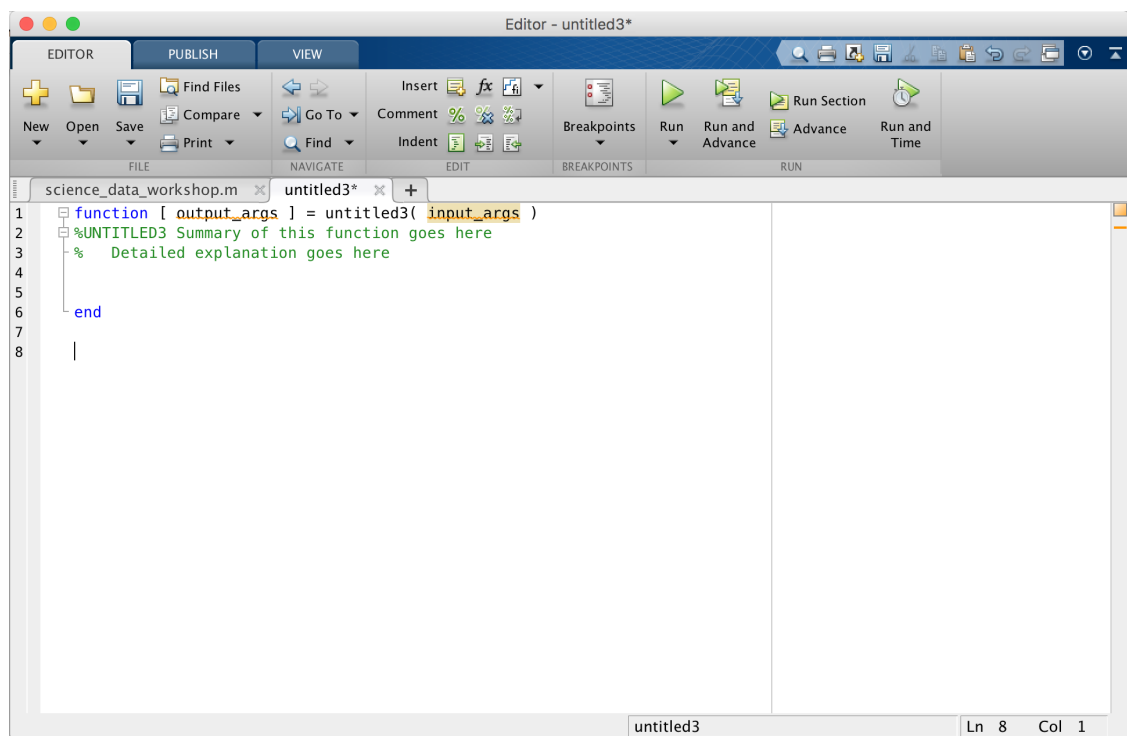
The function block begins with the keyword `function` and ends with the keyword `end` just like the control of flow blocks. Following the keyword `function` is the group of outputs (enclosed in square brackets `[ ]` and separated by commas `,`). After the outputs comes a single equal sign `=`. The name of the custom function follows with comma-separated inputs as arguments in parentheses `( )`. By carefully structuring the comments beneath the first line of the function code, it is possible to produce a function preview in the *Details* toolbar and useful information to display with `help()` that blends in with the native MATLAB functions.
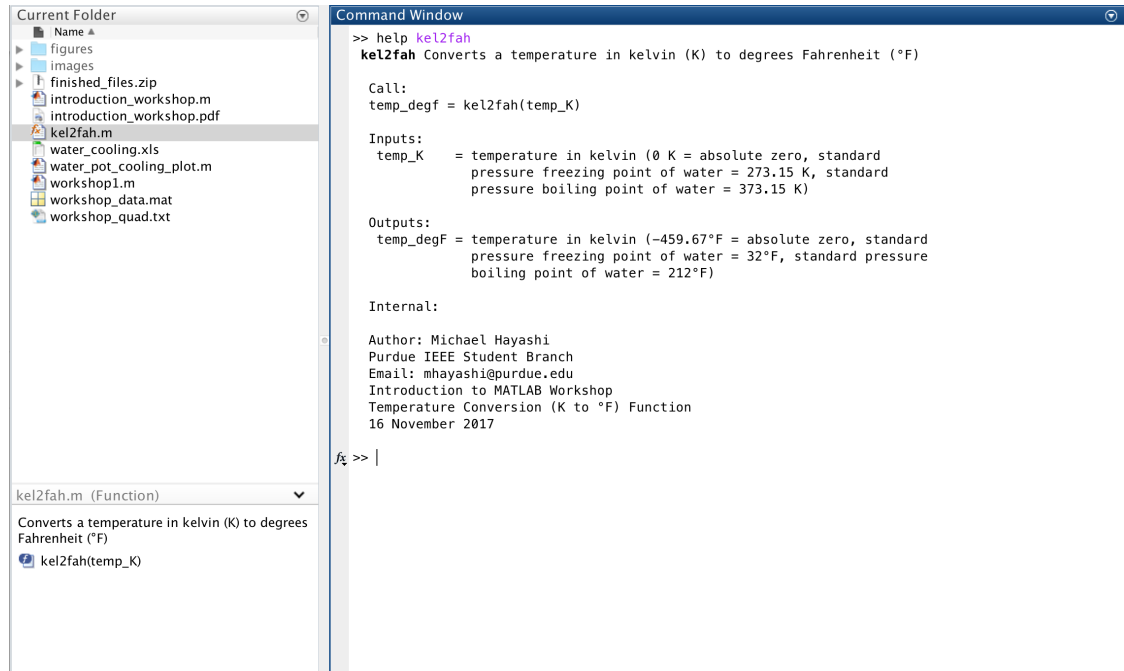
Some helpful tips when writing comments for custom functions:

- The file name **must** be the same as the function name with a ".m" at the end.

- The function name in ALL CAPITAL LETTERS on the first line of comments causes the help documentation to display the function name in **bold**.

- The comments up until the first line with no characters to display show up as the "short description" of the function.

- Subsequent lines after the first comment line with no characters to display only show up with `help()`.

- It is good practice to explain what the inputs and outputs should look like and what they represent. If there are internal variables that only exist in the frame of the function, it is best to label them and possibly explain the mathematics of the operations forming them.

- If there are multiple ways to call the function, such as a variable number of inputs with `varargin` or variable number of outputs with `varargout`, then suggested usage of the function call should be documented. It is optional to do so for fixed-input, fixed-output functions.

- Header information for functions can follow a similar style for scripts except that it must appear after the short description of the function.

Here is a function that converts temperature in kelvin to degrees Fahrenheit that you should write as well:

```matlab
function [temp_degF] = kel2fah(temp_K)
%KEL2FAH Converts a temperature in kelvin (K) to degrees Fahrenheit
 (°F)
%
% Call:
% temp_degf = kel2fah(temp_K)
%
% Inputs:
```

```matlab
%   temp_K    = temperature in kelvin (0 K = absolute zero, standard
%               pressure freezing point of water = 273.15 K, standard
%               pressure boiling point of water = 373.15 K)
%
% Outputs:
%   temp_degF = temperature in kelvin (-459.67°F = absolute zero,
% standard
%               pressure freezing point of water = 32°F, standard
% pressure
%               boiling point of water = 212°F)
%
% Internal:
%
% Author: Michael Hayashi
% Purdue IEEE Student Branch
% Email: mhayashi@purdue.edu
% Introduction to MATLAB Workshop
% Temperature Conversion (K to °F) Function
% 16 November 2017

temp_degF = 1.8 * temp_K - 459.67;
end
```

Remember that only functions part of the current *path* can be called. Functions outside the scope of the path will not be recognized by MATLAB when a script is interpreted. While the path can be manually edited in the "Environment" section of the "Home" ribbon in the main MATLAB window, it is good long-term practice only to do this for well-tested, broadly applicable functions that will be used across a multitude of projects or when using custom libraries. In most cases, it is **recommended** to keep functions in the same folder as the scripts that need them to prevent confusion. On a related note, secondary functions written in the same file as a named, custom function are only in the scope of the primary function. The secondary functions lie outside the path of the Current Folder and have their own isolated scopes.

It is time to finish the script "water_pot_cooling_plot.m". Place the file "water_cooling.xls" into the current folder. The import data section should look like the following:

```matlab
%% Import Water Data from Excel Workbook

% Perform Import from Excel
water_data = importdata('water_cooling.xls');

% Break Structure into Strings and Arrays
data_title = water_data.textdata{1,1};
time_label = water_data.colheaders{1};
temp_label = water_data.colheaders{2};
mass_label = water_data.colheaders{3};
time = water_data.data(:,1); % Time in experiment (min)
temp = water_data.data(:,2); % Temperature of water (K)
mass = water_data.data(:,3); % Mass of water present (kg)
```

Notice that `importdata()`, our generic import function of first resort, does most of the heavy lifting for us. Unfortunately, `water_data` is returned as a structure, a *collection data type* that must be left for a future MATLAB workshop. Extracting fields out of a structure is left for another day, so just copy the code above verbatim. Notice how comments keep track of what each variable represents and the associated units.

The unit conversion section should look like the following:

```
%% Convert Units

% Make the Data Ostentatiously Patriotic (i.e. Convert SI to Imperial)
temp_imperial = kel2fah(temp); % Temperature of water (°F)
mass_imperial = mass / 14.593903 * 32.174049; % Mass of water present
 (lbm)
```

The function `kel2fah()` that we wrote is used alongside a conversion from kilograms to slugs to pounds-mass to make the plots USA-friendly. Comments keep track of the what the new variables represent.

Debugging

The MATLAB script editor has a sophisticated debugging tool built in (https://www.mathworks.com/help/matlab/matlab_prog/debugging-process-and-features.html). While scripting languages usually make it easy to tell on which line an error occurs, it can be difficult to pinpoint where the programmer made the logical mistake or unhelpful shortcut that led to the error. For this reason, MATLAB can set *breakpoints* to stop a script after pushing "Run". To set a breakpoint, click on the hyphen (-) that appears next to the line number of the executable line in the script editor. Alternatively, put the cursor on that line and hit F12 on Windows. A red dot will take the place of the hyphen.
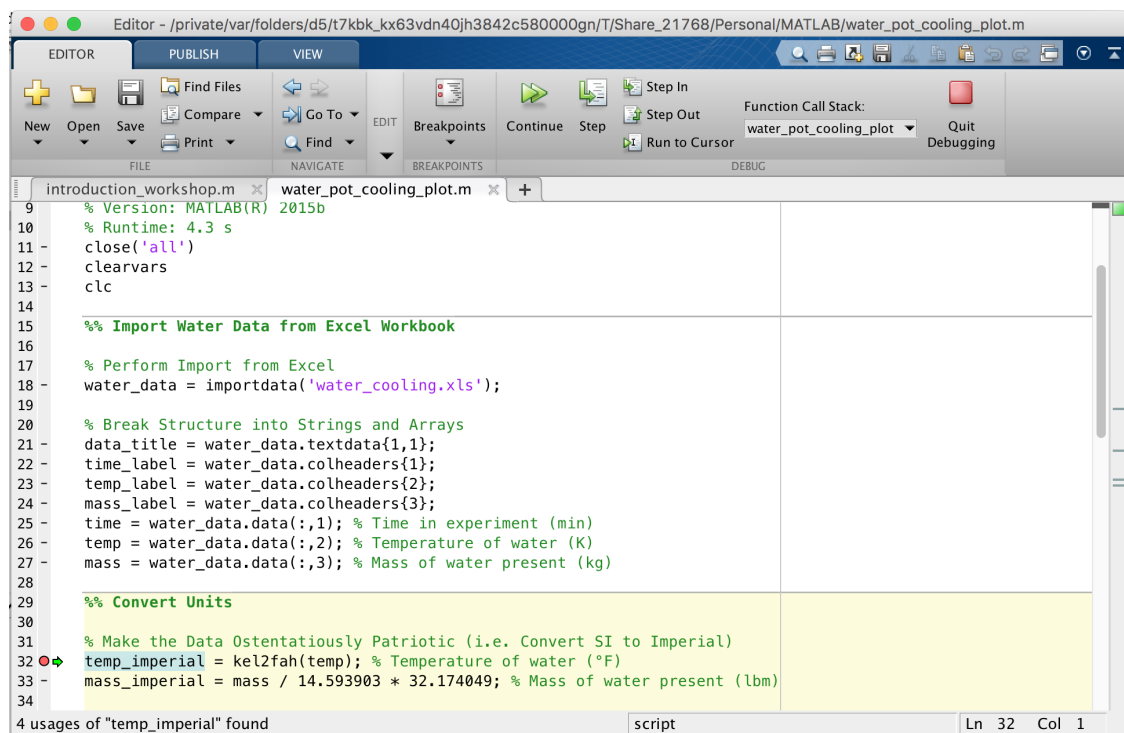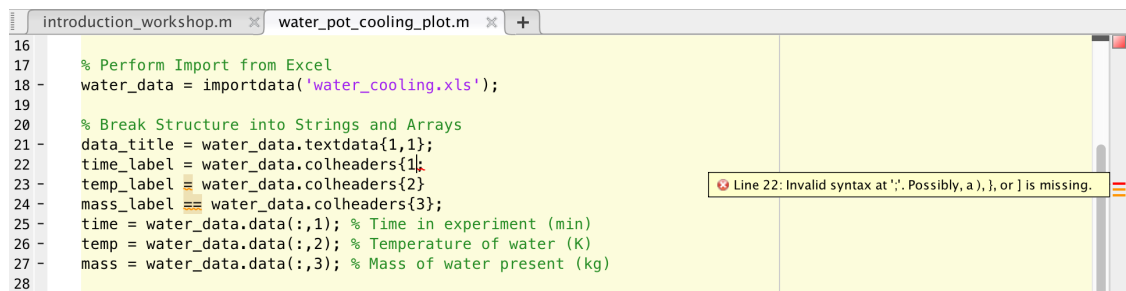

Be sure to save your script/function. Click on the "Run" button (F5 on Windows) in the "Run" section of the "Editor" ribbon in the script editor. Unless an error happens before the breakpoint, the script/function being called will execute up to but not including the breakpoint, and the "Run" button becomes the "Pause" button or the "Continue" button in the new "Debug" section of the "Editor" ribbon. The command window prompt changes from `>>` to `K>>` letting the keyboard take over in the scope of the script/function with the active breakpoint.

The programmer now has precision control over the script. The "Continue" button (F5 in Windows) in the "Debug" section will run the rest of the script/function in autopilot until another breakpoint, an error,

or the end of the script is reached. The "Step" button (F10 on Windows) allows the breakpoint line and all subsequent lines to be executed one-by-one. The "Function Call Stack" is keeping track of how many functions within functions the execution is. There are other controls available as well based on the position within the function call stack.

If you want to stop debugging before the script has finished executing, press "Quit Debugging" in the "Debug" section. Breakpoints can be removed by clicking on the red dot for the executable line, by pressing F12 on Windows, or by other options in the "Breakpoints" section of the "Editor" ribbon in the script editor. Hopefully, you can judiciously choose breakpoints that allow you to fix coding errors, remove breakpoints, and get the script or function behaving properly.

MATLAB is acutely aware of its own syntax and best practices. The *MATLAB Code Analyzer* (https://www.mathworks.com/help/matlab/matlab_prog/check-code-for-errors-and-warnings.html) tries to understand what you want to accomplish and help you write better scripts and functions. In the upper right of the script editor is a colored square by the vertical scroll bar that indicates errors (red), warnings (orange), or all clear (green) within the script or function being written. Moving the cursor over the square will give a line-by-line breakdown of any issues or coding inefficiencies. The lines with warnings will have an orange, squiggly line under the questionable part, and the lines with syntax errors will have a red, squiggly line under the incorrect portion. I **highly** recommend that you strive for a green square with every script. Most of the time, warnings can be avoided by restructuring your solution to the problem.

```matlab
introduction_workshop.m    water_pot_cooling_plot.m    +
16
17      % Perform Import from Excel
18 -    water_data = importdata('water_cooling.xls');
19
20      % Break Structure into Strings and Arrays
21 -    data_title = water_data.textdata{1,1};
22      time_label = water_data.colheaders{1};
23 -    temp_label = water_data.colheaders{2}      Line 22: Invalid syntax at ';'. Possibly a ), }, or ] is missing.
24 -    mass_label == water_data.colheaders{3};
25 -    time = water_data.data(:,1); % Time in experiment (min)
26 -    temp = water_data.data(:,2); % Temperature of water (K)
27 -    mass = water_data.data(:,3); % Mass of water present (kg)
28
```

Plotting

The plotting section of the script "water_pot_cooling_plot.m" should look like the following:

%% Plot Data

```matlab
% Figure 1: Temperature of Water During Experiment
figure(1)
plot(time, temp_imperial)
grid('on')
title('Temperature of Water after Removed from Heat')
xlabel('Time (min)')
ylabel('Temperature (°F)')
text(0.7 * time(end), 1.2 * temp_imperial(end), 0, ...
    sprintf('Final Temperature:\nT_{end} = %1.1f°F', ...
 temp_imperial(end)))

% Figure 2: Mass of Water During Experiment
figure(2)
% subplot(1, 1, 1)
plot(time, mass_imperial)
% axis([0, 120, 7.42, 7.44])
```
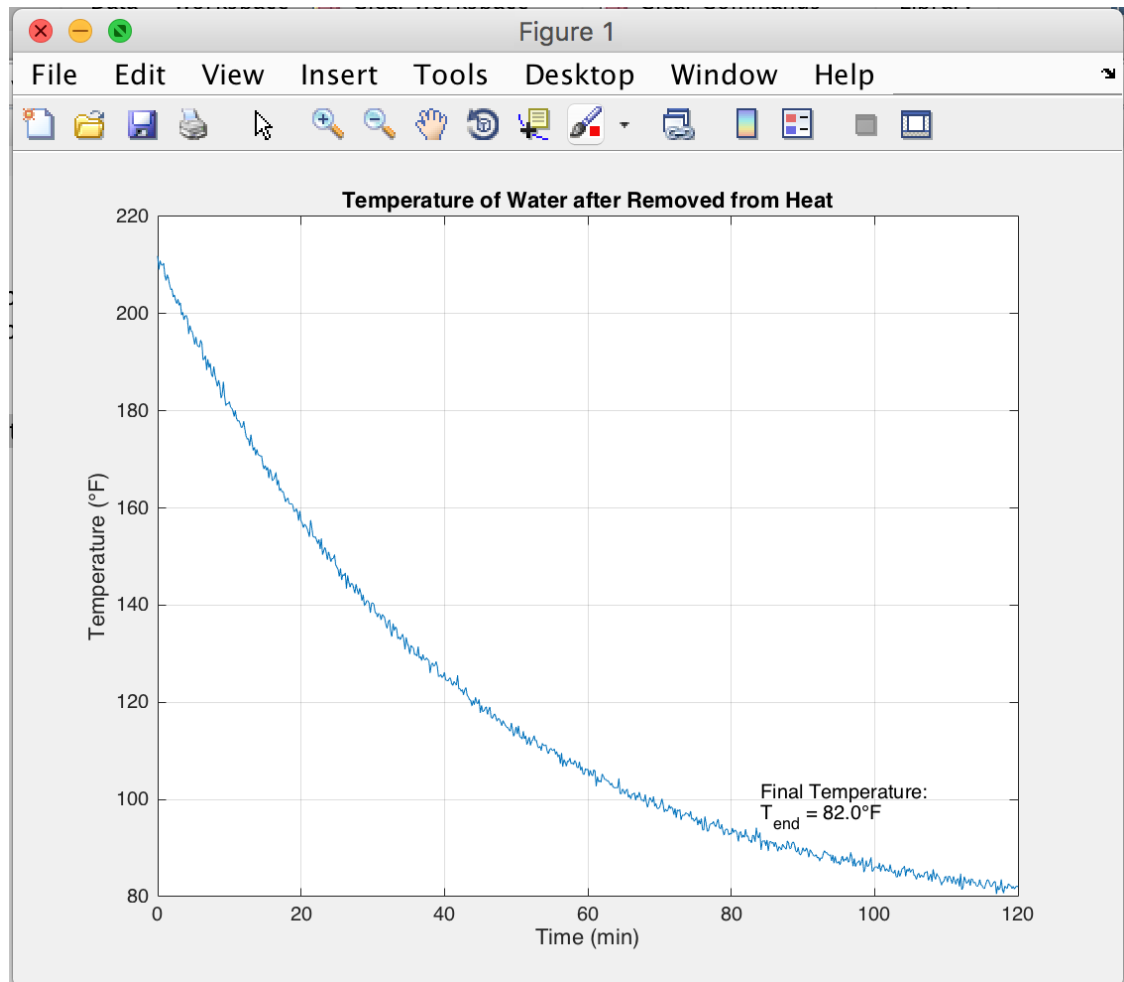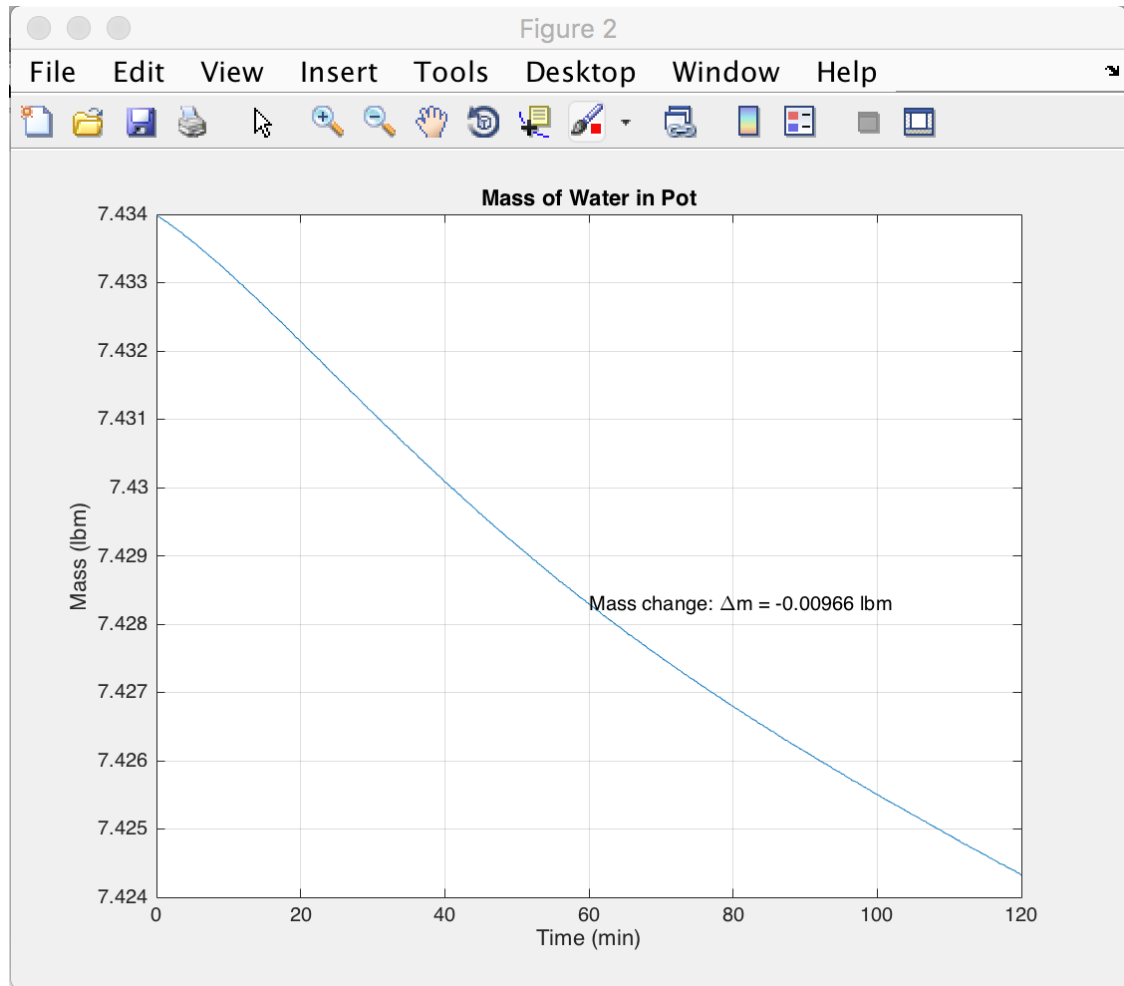
```
grid('on')
title('Mass of Water in Pot')
xlabel('Time (min)')
ylabel('Mass (lbm)')
text(0.5 * time(end), median(mass_imperial), 0, ...
    sprintf('Mass change: %sm = %1.3g lbm', '\Delta', ...
    mass_imperial(end) - mass_imperial(1)))
% legend('Mass', 'Location', 'Best')
```

It is time to understand more about one of MATLAB's most common and most powerful features: **figures and plotting**.

- A *figure* is a pop-up window MATLAB makes that displays data graphically rather than textually as in the "Command Window".

- Figures can be split so that separate plots appear on each of them by using the `subplot()` command. The first argument to `subplot()` is the number of separate plots to include vertically, the second argument is the number of separate plots to include in a rectangle horizontally, and the third argument is the number of the current set of subplot axes in the figure (numbering is left-to-right, then top-to-bottom).

- While there are ways to make figures from the MATLAB GUI and figures will be made in sequence as needed, I highly recommend making every figure using the command `figure()` with the argument being the number of the figure in question. This way, a programmer can always be certain of which data is in which figure window, especially if some figures are only generated optionally or can be done as part of a broader scope which may alter the position in figure numbering from the default.

- The most commonly used function to plot by far is `plot()`. It accepts arguments in sets of two or sets of three. The first argument is the vector of x-axis data. The second argument is the vector of the same length containing the y-axis data. The option third argument is a *line specification* (https://www.mathworks.com/help/matlab/ref/linespec.html) that controls the way data points are connected,

the markers for each data point, and the color of the curve. Each set of arguments to `plot()` corresponds to a curve on the current set of axes.

- Other commonly used functions for 2D plotting are `semilogx()` (logarithmic x-axis and Cartesian y-axis), `semilogy()` (Cartesian x-axis and logarithmic y-axis), and `loglog()` (both axes logarithmic).

- The best time to modify the appearance of the graph such as the axes limits and tick marks is after plotting. `grid()` can turn on grid lines, and `axis()` along with specific counterparts `xlim()` and `ylim()` can adjust the limits to plot in each dimension.

- All figures should include a title at the top, easily accomplished by passing a string to `title()` for the current figure. The x-axis and y-axis should be labeled, including units, using `xlabel()` and `ylabel()` respectively. The axis label functions each take a single string as an argument.

- The `text()` command can be used to write strings onto the axes. I often do this to call out starting values, stopping values, discontinuous events, extrema, limits, outliers, out-of-specification points, and slopes. The first three arguments to `text()` are the x-value, y-value, and z-value about which to anchor the text on the plot. The fourth argument is a string. See the example given to place text on figures programmatically and use `sprintf()` to display certain values even if the vectors of data change between script runs.

- The `legend()` command allows each curve on a set of axes to be labeled with a string. By including `..., 'Location', 'Best')` at the end of the function arguments, MATLAB will attempt to put the legend box in a place that covers up the least amount of data points.

MATLAB also has a host of functions for 3D plotting on figures. The ordering of arguments typically goes x-axis data matrix, y-axis data matrix, and z-axis data matrix. The way to go from x-axis and y-axis data vectors to data matrices is with a handy function called `meshgrid()`. The primary functions for 3D plotting are `mesh()` and `surf()`. The specifics are left for a future workshop.

The figure saving section should look like the following: %% Save Figures to File

```
% Create Folder for Figures if Needed
if ~exist('figures', 'dir')
    mkdir('figures');
end
cd('./figures')

% Save Figures to Dedicated Output Folder
print(1, '-dpng', sprintf('Water_Pot_Temperature_%s.jpg', date))
print(2, '-djpeg90', sprintf('Water_Pot_Mass_%s.png', date), '-r200')

% Return to Previous Folder
cd('..')
```

There are some nuances when it comes to organizing output files:

- Functions such as `exist()` (checking if something is in the "Current Folder"), `mkdir()` (create a new, empty directory), and `cd()` (change directory) all borrow from basic UNIX commands to navigate a file tree on a computer. The script is checking to see if a folder called "figures" exists and will make one if it does not yet.

- The script briefly leaves the path or current folder where the script is executed to travel down to the "figures" folder. All output from the script is placed in here.

- The `print()` function is the one stop for outputting graphical data either to a physical printer or to a file. The first argument is the figure number to print. The second argument is a driver (check the help

documentation for details) that allows virtually any file type you could want as an output. The final argument is a string with the file name.

- The script should return the original view of the *Current Folder* toolbar with a new folder called "figures" within it.

- Note that it might be helpful to use a date string in the file name when sorting between runs made on different days. This way, old data from previous days is saved and not overwritten by `print()` on successive runs of the script.

Conclusion

I hope that this workshop was helpful to you, and that you feel confident choosing MATLAB whenever it is convenient to do so in your academic, professional, or personal life. I encourage you to keep exploring. For example, try clicking the "Publish" button in the "Publish" section of the "Publish" ribbon of your script once you are finished. MATLAB is supposed to be a relatively painless way to sort through numeric data, not a scare tool of faculty. Please feel free to email me at mhayashi@purdue.edu with MATLAB questions, and stay tuned for repeat workshops and continuation workshops in the MATLAB series.

Published with MATLAB® R2015b