MATLAB Workshop!!!
Learn how to work with vectors and plot!!!
By Matthew Gallagher, Ryan Hessler, Brandon Mitchell

Introduction:

The purpose of this lab is to show to you that MATLAB is not only a great language, but possibly the best language ever created. In this lab, you will read in data from an Excel spreadsheet, clean the data, and then perform calculations on the data. After that, you will plot it and lose yourself in the endless plot customizations that MATLAB has. MATLAB is good at data processing and its plotting capabilities are quite useful and powerful. Some might say it's too powerful.

Please don't start until we are done presenting, but feel free to read through the paper and start thinking about what features of the language will be useful. In addition please try it yourselves before using the code snippets. Once the workshop begins, ask for guidance if you need it, though also consult the reference guide attached below. There is also a solution.m file on the website for when you are done.

First Graph:

Start by downloading the template.m and associated Excel files from the Programming Languages website. It includes the code for reading in the Excel files and cleaning the data for you. MATLAB script files end with a .m extension. MATLAB has several other extensions associated with it, but this is the only one you need to know. Look at it and understand it, but don't edit existing code. Also, don't forget to be good and put your name and a description of the program in the header! The lines

```
clear
close all
```

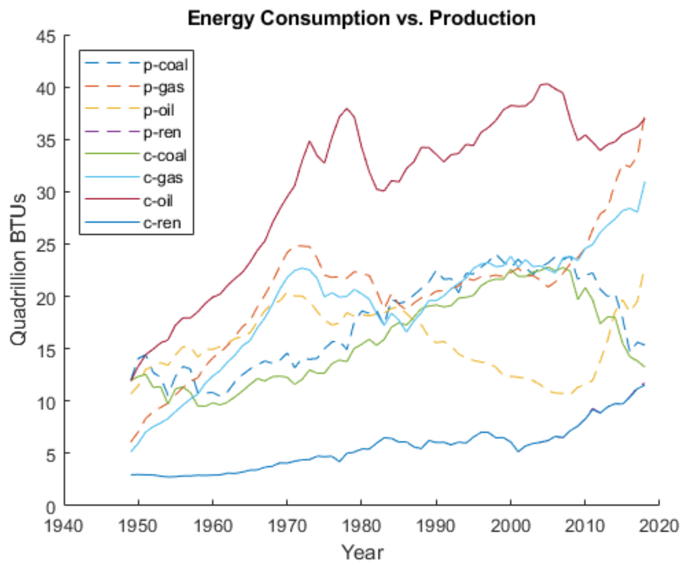
erase all variables from the workspace and closes all open figure windows. We want to clear out all variables as that helps with debugging. For example, if I am using the variable `x` in my code, but I have never defined it in the script, it may not complain as it may find it in the workspace if I had used it previously. Closing all figure windows helps prevent problems with plotting. If a figure window was already open, then we may accidentally plot in it when we don't want to.

The next few lines read in the spreadsheet and use logical indexing to remove invalid data. Please note that the function `xlsread` is not available in GNU Octave. These lines

```
pdata(isnan(pdata)) = 0;
cdata(isnan(cdata)) = 0;
```

find all data that is not a number (or a NaN value) and sets it equal to zero so as to not interfere with the plotting and other stuff.

The first plot is going to look a little like this bad boy right here.



Isn't that a beaut? We will be graphing the coal, gas, oil, and renewable energy consumption and production from 1949 to 2018 in this one graph. As always, there are a couple of ways to go about this, but I will be walking you through what I believe to be the cleanest way.

Start by typing

```
hold on
```

This tells MATLAB that you will be making multiple plot calls and that it should put them all on the same figure, not overwrite them. The years make up the first column of the matrix and will also be the x axis of the graph. We want all the rows in column one, we can use the colon in our indexing. The colon means "take everything". So, to get the years, we can do

```
pdata(:, 1)
```

This gives us a 70 x 1 column vector. The two matrices you give plot must have matching lengths. The second parameter we give it will be a 70 x 4 matrix. They have the same length, or longest dimension, so plot is able to figure it out and we get four lines. For the y axis, we want all the rows in column 2, 3, 4, and 12 as these are the columns for coal, natural gas, oil, and renewable, respectively. Try to do the indexing before reading my method. You should index all columns at once by giving it a vector of indices. If you are stumped, try

```
pdata(:, [2:4, 12])
```

Now, you have both of the matrices you need! Either plop both pieces of code into a plot call or assign them to separate variables. I will plop them into one plot call.

```
plot(pdata(:, 1), pdata(:, [2:4, 12]))
```

Now, do the same thing, but with the consumption data. Since the years, coal, natural gas, oil, and renewables are in the same columns, the indexes don't need to change.

```
plot(cdata(:, 1), cdata(:, [2:4, 12]))
```

You should see eight lines in the figure window now. Eight separate plot calls could be used, but then it can get a bit messy looking. Unfortunately, it is difficult to tell the lines apart. Thankfully, plot is a very powerful and customizable function. With a simple addition to either of the plot calls, we can turn four of the lines dashed. Add `'--'` as an argument to one of the plots. We also want to call `hold off` now that we are done plotting into this specific window. Your code should look like this.

```
hold on
plot(pdata(:, 1), pdata(:, [2:4, 12]), "--")
plot(cdata(:, 1), cdata(:, [2:4, 12]))
hold off
```

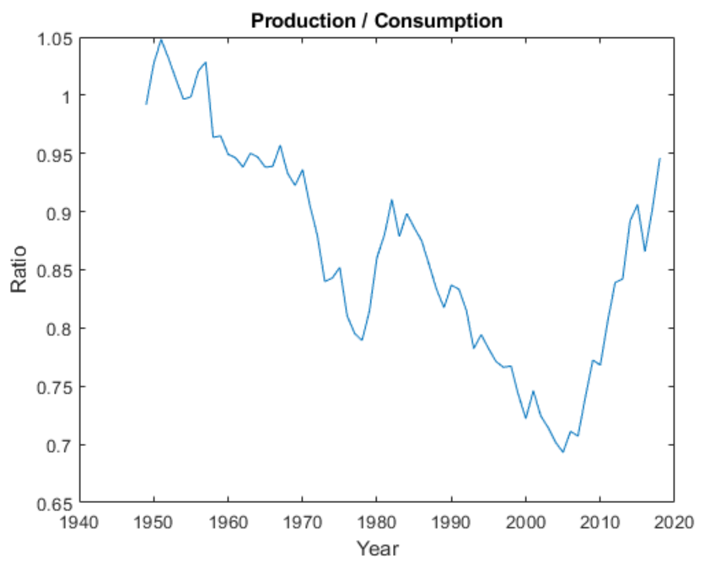
Right now, the graph is quite bland and boring and I don't really have any idea what it is actually showing. So, we need to add labels to the x and y axis, a title, and a legend. You can use the function `xlabel`, `ylabel`, and `title` for this. All three take a char array as input, and they are quite simple, so I won't show how to use them. The legend is a bit more complicated. The function is just called `legend` and it takes as many char arrays as there are lines on the plot. In this case, we have eight lines, so we should give it eight char arrays. Several MATLAB functions are like this and can take a variable number of inputs. The first input into legend will go with the first line, the second with the second line, and so on. Since I plotted the coal production first (column two in pdata), I want to give my label for it first. In the end, I have a legend call that looks like this. You will want to change yours if you did not plot in the same order. The ellipses tell the MATLAB interpreter to continue reading on the next line. You use them to break a single line statement into multiple lines. It is similar to the backslash line-continuation in Python.

```
legend("p-coal", "p-gas", "p-oil", "p-ren", ...
       "c-coal", "c-gas", "c-oil", "c-ren", ...
       'Location', 'northwest')
```

You might be wondering what the last part is about. After the eight labels we need, we can give `legend` special values to get it to do different things. In this case, we want to position the graph in the northwest corner so as to not cover up the important details. As far as I know, these options are not case sensitive, but they must be char arrays. You should avoid using any of these special values as the names of labels. Instead of `‘northwest’`, you can give it `‘best’` and it will figure out the best location for you. How helpful, MATLAB!

Second Graph:

This one is far simpler, but you will need to perform some calculations on the data before plotting it. I want you to graph the total energy production over the total energy consumption, which is column thirteen in both matrices. Also, both matrices have the same years, so it doesn’t matter which matrix you get the year data from. Please don’t use a dirty for loop! Use MATLAB’s vectorized operators for this. Specifically, you want an element-wise operator. Start after the “figure(2)” call in the template. This opens a new figure window so the second plot doesn’t overwrite the first. Here is the example



Stumped? Try

```
plot(pdata(:, 1), (pdata(:, 13) ./ cdata(:, 13)))
```

We want total energy production data over, or divided by, total energy consumption data. We get the data using a similar indexing technique as before. Since we want to divide the first element in `pdata(:, 13)` by the first element in `cdata(:, 13)` and so on, we must use the element-wise division. Please also give this bad boy some titles and labels. In case you don't know what this graph means, it is showing us the ratio of energy production over energy consumption. If it is one, then we are producing and consuming the same amount. If it is lower than one, then we are consuming more than we produce, so we must be importing the energy from someone else. Higher, we are producing more than we need and must be exporting the excess. Interesting, huh?

Conclusion:

So, do you now understand why MATLAB is the superior language? Like, imagine doing the ratio in some other language. In Python, you can do it in one line, but it will be in a list comprehension, which aren't the easiest things to comprehend. In C, you would need a for loop. In most languages, you would need to import a special library or module for the plotting capabilities, but you don't here. MATLAB's vectorized functions and operators actually are faster than for loops in most cases, so be sure to use them.

Anyhow, you can clearly see that MATLAB is very useful for data processing and graphing. Its capabilities are far greater than what we just went over here. Hopefully you are inspired to write that lexical analyzer in MATLAB first and then translate it to the other languages. MATLAB does have a very powerful regular expression engine for parsing text data, so you could do it if you really wanted to.

View the solution.m file on the website to check your work. It is more important the graphs match the images I showed you rather than the code being one to one. As long as you didn't use a dirty for loop for the second part, you passed!

Reference Sheet!!!

Documentation:

If you are ever unsure how to use a function, type in the command window

```
doc nameOfFunction
```

Basic Assignment:

Assignment is done with the = operator. Multiple assignments and expressions can be chained together if they are separated with commas or semicolons. A semicolon suppresses output of any returned value to the command window.

```
x = 5
x = 5, y = ‘hat’; z = 2^8
```

Vector Creation:

Vectors can be created with the brackets in a way similar to Python. Values can be variables, literals, or expressions but must all be of the same type. Values can be delimited with spaces or commas.

```
arr = [1 2 3 4 5]
x = 7; arr = [x, 7 / 3, sin(3)];
```

A semicolon can be used to create two dimensional matrices. The semicolon marks the beginning of the next row. Jagged arrays are not supported.

```
[1 2 3; 4 5 6] ⇒ [1 2 3
                   4 5 6]
```

The colon operator can be used to quickly create vectors. Its syntax is `start:increment:stop`. If the increment is left out, it defaults to one.

```
1:1:5 ⇒ [1 2 3 4 5]
3:6 ⇒ [3 4 5 6]
2.5:0.5:4.5 ⇒ [2.5 3.0 3.5 4.0 4.5]
10:-2:1 ⇒ [10 8 6 4 2]
```

The colon operator may be used within brackets to create vectors.

```
[1 2 3:5 6] ⇒ [1 2 3 4 5 6]
```

Vector indexing:

Indexing in MATLAB is done with parentheses instead of brackets. Indexing is also one-based.

```
arr = [3 6 9 12]; arr(1) ⇒ 3
```

Indexing can be done with a vector of indices. This will return a subvector or submatrix.

```
arr = [3 6 9 12]; arr([2 4]) ⇒ [6 12]
```

When indexing into two dimensional matrices, the row comes first, then the column. `ones` creates a matrix of ones.

```
mat = ones(5); mat(2, 2) ⇒ 1
```

Like before, vectors can be used for indexing. First three values in second row would be

```
mat = ones(5); mat(2, 1:3) ⇒ [1 1 1]
```

A colon means to take everything. If I wanted to take all the columns in the third row

```
mat = ones(5); mat(3, :) ⇒ [1 1 1 1 1]
```

Plotting:

Plotting a line is done with plot. At its simplest, it takes two vectors of equal length (length being the longest dimension) and constructs a line from them. The first vector is the x values and the second is the y values. Multiple lines can be plotted at once.

```
plot(x, y) ⇒ one line
plot(x1, y1, x2, y2) ⇒ two lines with separate x values
plot(x, [y1 y2]) ⇒ two lines with same x values
```

Some simple customization features are `xlabel`, `ylabel`, `title`, and `legend`. The first three take a char array. `legend` takes a variable number of char arrays equal to the number of lines plotted. Two lines, two char arrays.

```
xlabel('This is the x axis!')
legend('First Line', 'Second Line')
```