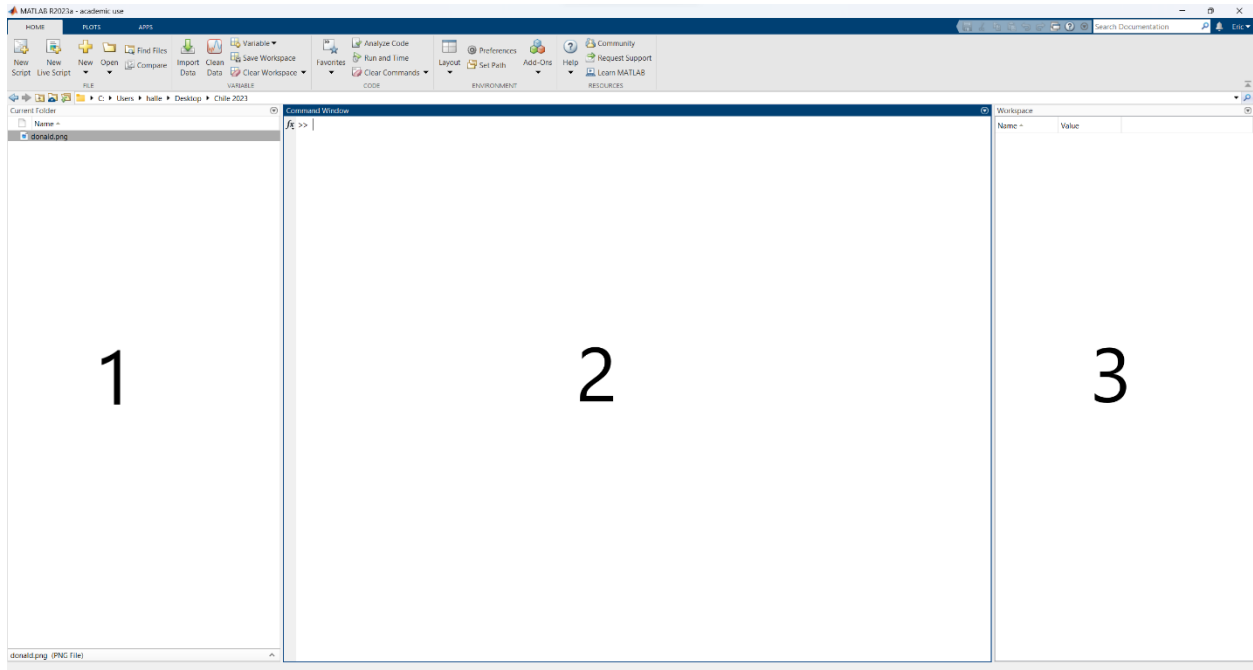


# MATLAB Tutorial/Cheat Sheet

*For ABRD-514*

## *The Basics*

When you first start MATLAB, it should look something like this:



There are three main windows to get familiar with.

### **1. Current folders**

- When running MATLAB, you must choose a folder to work in. In this example, I am in a folder in my Desktop called Chile 2023.
- The window will show all the files and folders in your current folder. This becomes very useful when trying to remember things you have saved or not saved.

### **2. The Command Window**

- This is where you perform tasks and run scripts in MATLAB. More on this is just a second.

### **3. The Workspace**

- a. This is where all your current variables that you have saved can be seen.

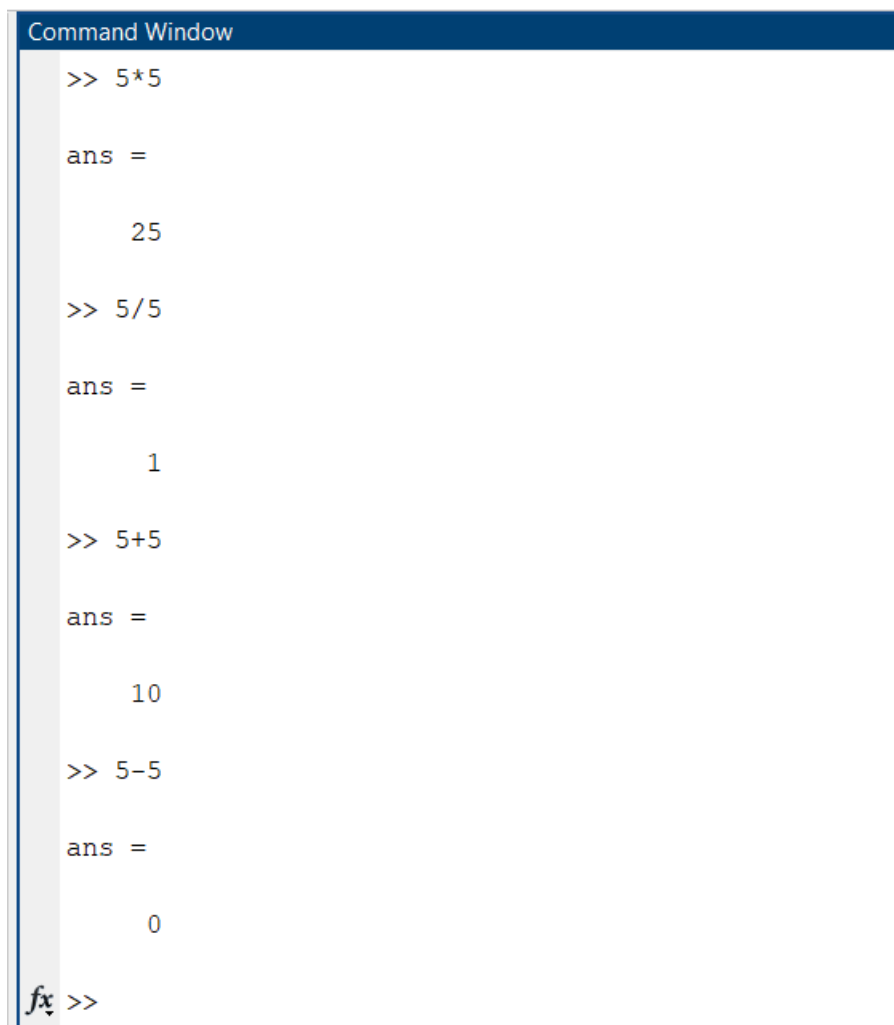
Most of the time, you'll want to keep these windows open as you will be using them quite a bit.

## Getting Started With The Command Window

### *Math*

Let's try using the command window! Try the following math statements in the command line:

1.  $5*5$
2.  $5/5$
3.  $5 + 5$
4.  $5 - 5$



```
Command Window
>> 5*5
ans =
    25
>> 5/5
ans =
     1
>> 5+5
ans =
    10
>> 5-5
ans =
     0
fx >>
```

Great! However, does MATLAB follow order of operations?? Let's see:

```
>> 5 + 5*5  
  
ans =  
  
    30  
  
fx >>
```

Yes, it does! However, for visual clarity, it is best practice to use parentheses. Let's say instead we wanted to add the first two numbers, and then multiply.

```
>> (5+5) *5  
  
ans =  
  
    50  
  
fx >> |
```

### *Variables*

Let's say now, we want to store our number to use later. This is where variables come in.

```
>> my_num = (5+5) *5  
  
my_num =  
  
    50  
  
fx >>
```

Great! Our number is stored in a variable called "my\_num". Let's take a look at the workspace as well.

Name	Value
ans	50
my_num	50

While we see our variable `my_num`, there is also a special variable called “ans”. MATLAB always saves the last *unassigned* call in the Command Window to “ans”. This means that if we called `5*5`, “ans” would be assigned as 25. However, if called `my_num = 5*6`, “ans” **would not** be assigned as 30.

### Using the up-arrow

Whoops! Let’s say you called `5*10`, but really wanted to assign it to a variable instead. You can use the up-arrow key to automatically retype any of the previous commands you have input!

```

>> 5*10

ans =

    50

fx >> |

```

```

5/5
5+5
5-5
5 + 5*5
(5+5)*5
my_num = (5+5)*5
my_num = 60
clc
★ 5*10
fx >> 5*10|

```

Nice. Let's now assign it a variable, "my\_num2".

```
>> my_num2 = 5*10;
fx >>
```

\*\*\***Notice the semicolon!!**\*\*\* If you would like to not have the output show in the command window, simply add a semi-colon at the end of it. This becomes especially helpful when writing scripts, so that every line is not printed. More on that soon.

### Matrices

One of MATLAB core strengths is importing, creating, and working with matrices. Let create one, using the following notation!

```
Command Window
>> my_list = [1 2 3 4 5]

my_list =

     1     2     3     4     5
fx >>
```

Excellent. Now, what if we are really lazy, and don't want to type all those numbers? Let's try this instead.

```

>> list_2 = 1:5

list_2 =

     1     2     3     4     5

fx >>

```

Great! This is extremely helpful when arrays are incredibly long. However, an important question still remains. What if we don't want to count up by 1? To change the step size between each number, add a third number in between your range which specifies that size.

```

>> list_3 = 1:.5:5

list_3 =

 1.0000  1.5000  2.0000  2.5000  3.0000  3.5000  4.0000  4.5000  5.0000

fx >>

```

Like we mentioned above, matrices are a great way of storing data. Lastly, how do we get that data? Let's say, we want the 5<sup>th</sup> value.

```

>> list_3 = 1:.5:5

list_3 =

 1.0000  1.5000  2.0000  2.5000  3.0000  3.5000  4.0000  4.5000  5.0000

>> list_3(5)

ans =

     3

fx >> |

```

Or, what if we want a *range* of values?

```
Command Window
>> list_3(1:3)

ans =

    1.0000    1.5000    2.0000

fx >> |
```

### 2D matrices

Let's say I wanted to make a 2D matrix instead of just a 1D vector. How would I do that?

```
>> my_2d_array = [1 2 3 4 5;6 7 8 9 10]

my_2d_array =

     1     2     3     4     5
     6     7     8     9    10

fx >>
```

Notice the semi-colon, which is MATLABs way of establishing a new row when creating a matrix.

### Intro to MATLAB Functions and Useful Commands

Let's say we want to create a really long array, for example 100. However, what if want it to be a 10 by 10 instead of 1 by 100? Typing this out would take forever, so let's try a function called "reshape". But, how do I use this function?? Try typing "help reshape" in the command window.

```

>> help reshape
reshape Reshape array.
reshape(X,M,N) or reshape(X,[M,N]) returns the M-by-N matrix
whose elements are taken columnwise from X. An error results
if X does not have M*N elements.

reshape(X,M,N,P,...) or reshape(X,[M,N,P,...]) returns an
N-D array with the same elements as X but reshaped to have
the size M-by-N-by-P-by-.... The product of the specified
dimensions, M*N*P*..., must be the same as NUMEL(X).

reshape(X,...,[],...) calculates the length of the dimension
represented by [], such that the product of the dimensions
equals NUMEL(X). The value of NUMEL(X) must be evenly divisible
by the product of the specified dimensions. You can use only one
occurrence of [].

See also squeeze, shiftdim, colon.

Documentation for reshape
Other uses of reshape

fx >>

```

The `help` command is one of the most powerful resources in MATLAB. It is your best friend when trying to learn how to use built-in functions. In this case, to use `reshape` we would first input our array, followed by the dimensions we want to reshape.

```

>> reshape(my_list, 10,10)

ans =

     1     11     21     31     41     51     61     71     81     91
     2     12     22     32     42     52     62     72     82     92
     3     13     23     33     43     53     63     73     83     93
     4     14     24     34     44     54     64     74     84     94
     5     15     25     35     45     55     65     75     85     95
     6     16     26     36     46     56     66     76     86     96
     7     17     27     37     47     57     67     77     87     97
     8     18     28     38     48     58     68     78     88     98
     9     19     29     39     49     59     69     79     89     99
    10     20     30     40     50     60     70     80     90    100

fx >>

```

Great! Another useful tip: if you would rather organize your array counting left-right rather than up-down, you can transpose the array by simply adding an apostrophe to the end of the command.



```

>> reshape(my_list, 10,10) '
ans =
     1     2     3     4     5     6     7     8     9    10
    11    12    13    14    15    16    17    18    19    20
    21    22    23    24    25    26    27    28    29    30
    31    32    33    34    35    36    37    38    39    40
    41    42    43    44    45    46    47    48    49    50
    51    52    53    54    55    56    57    58    59    60
    61    62    63    64    65    66    67    68    69    70
    71    72    73    74    75    76    77    78    79    80
    81    82    83    84    85    86    87    88    89    90
    91    92    93    94    95    96    97    98    99   100
fx >>

```

MATLAB has a dizzying number of built-in functions, which is one of the great reasons it is a good starting language. In addition to “help”, here are a select few that are particularly helpful.

1. “whos”
  - a. This prints your current variables and some information about them in the command window. If you care about a particular variable, you can write “whos your\_variable”.

```

>> whos
Name              Size              Bytes  Class  Attributes

ans               10x10              800   double

list_2            1x5                 40   double

list_3            1x9                 72   double

my_2d_array       2x5                 80   double

my_list           1x100              800   double

my_num            1x1                  8   double

my_num2           1x1                  8   double

>> whos my_num
Name              Size              Bytes  Class  Attributes

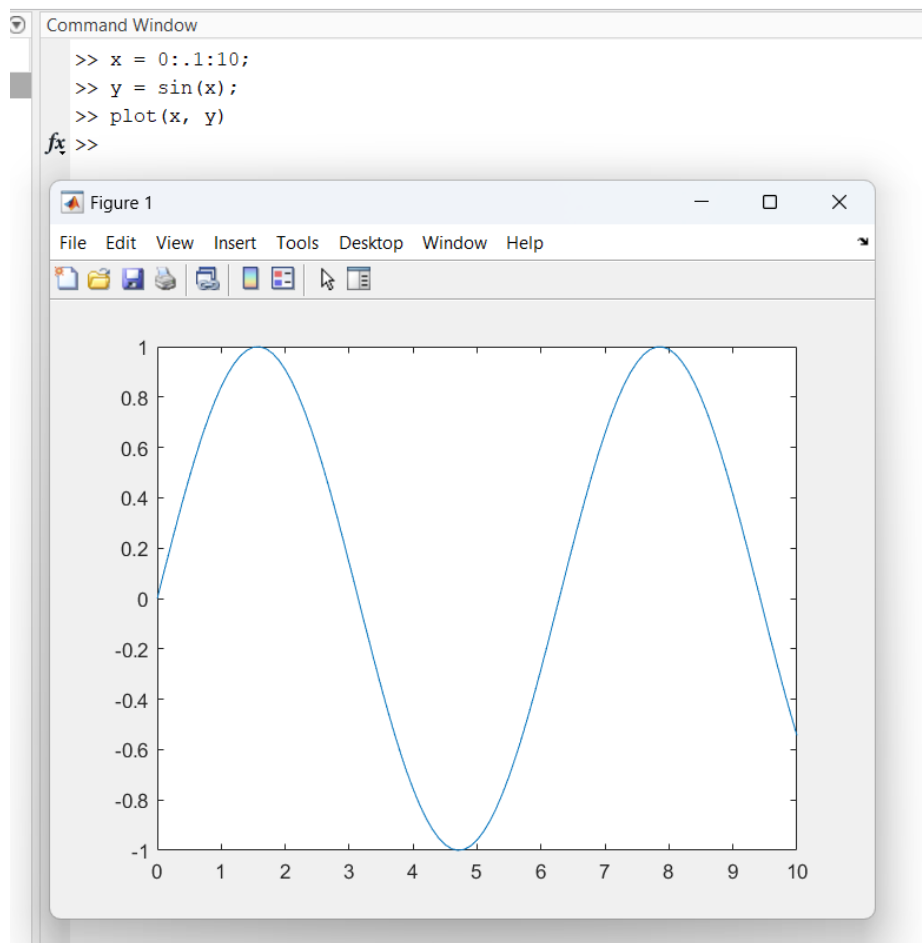
my_num            1x1                  8   double
fx >> |

```

2. “lookfor”
  - a. This is a good way to search for functions you may not know the name of. For example, if you wanted a function to average some numbers, you could search “lookfor average”.
  - b. However, if you cannot seem to find what you are looking for, Google is still your best friend 😊. This will usually lead you straight to MATLABs documentation.
3. “zeros” and “ones”
  - a. These created m by n matrices of zeros or ones, respectively. I use these all the time!!

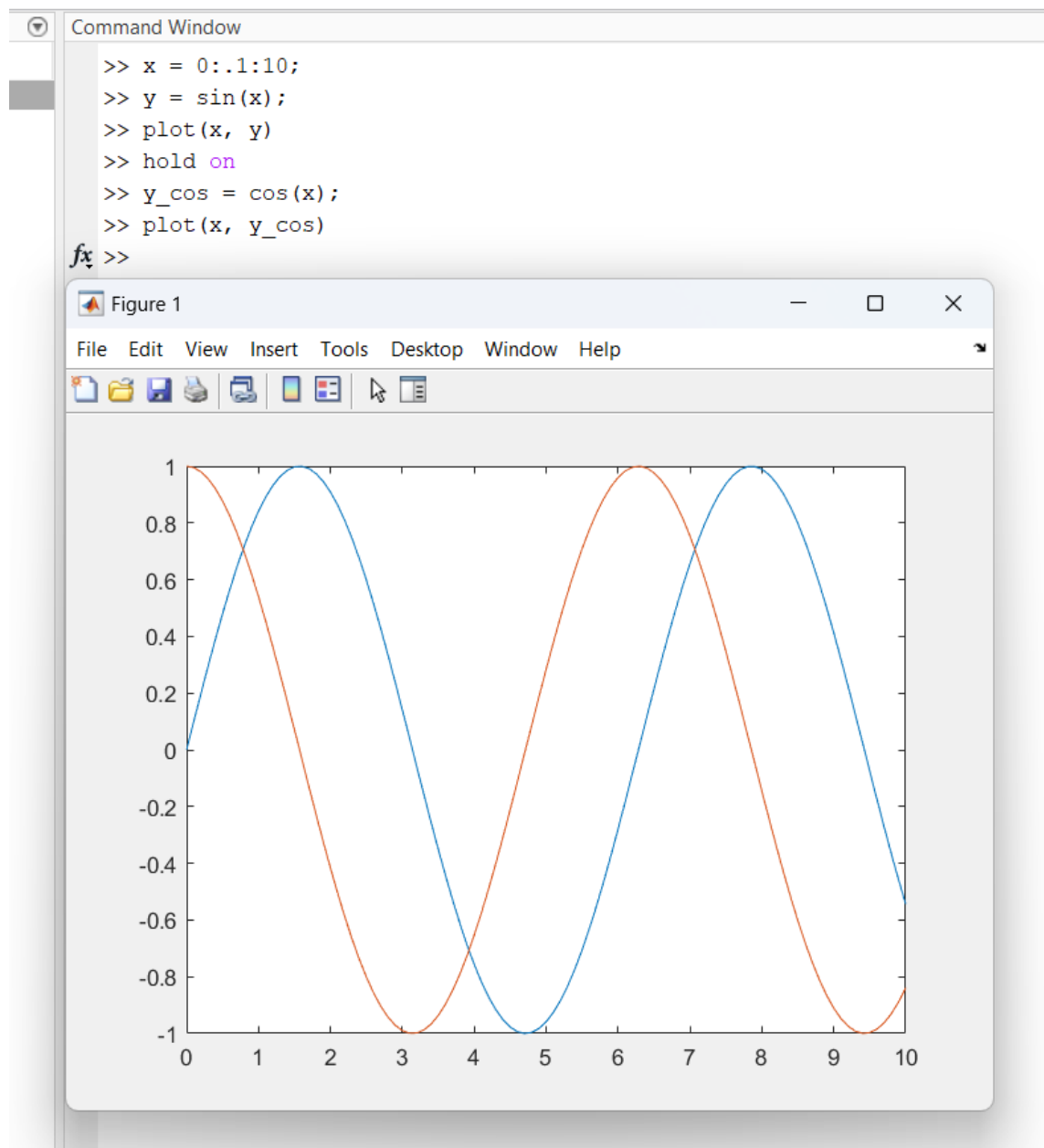
## Plotting and Visualization

MATLAB also has built-in plotting functions that help visualize signals and images. Let’s try creating a sin wave, and then plotting it!



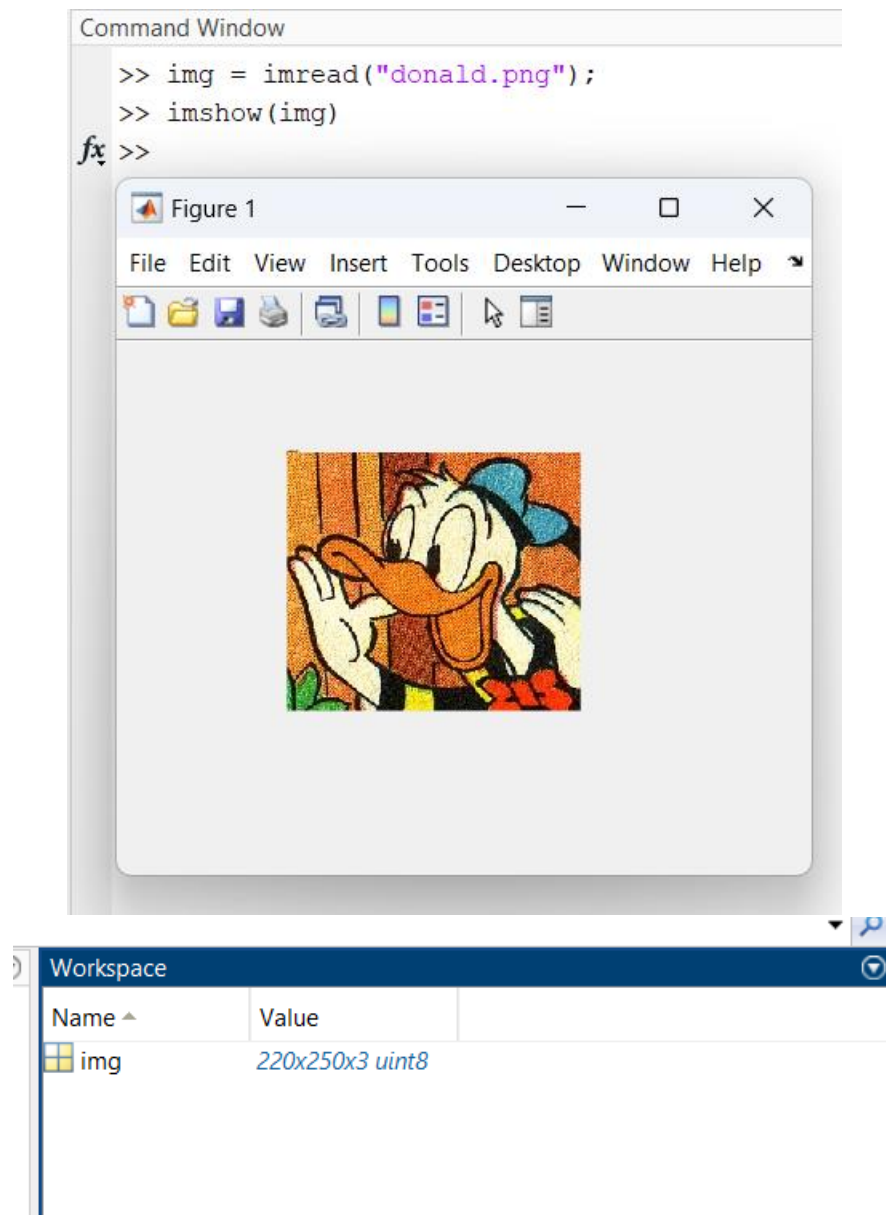
Nice! Here are some more tips on plotting:

1. You can add labels to an axis, as well as a title. To do this:
  - a. `xlabel("Your label string")`
  - b. `ylabel("Your label string")`
  - c. `title("Your title")`
2. You can plot several plots on top of each other. To do this, first create your initial plot. Then type "hold on". The next time you call "plot", it will instead add it to your already existing plot!

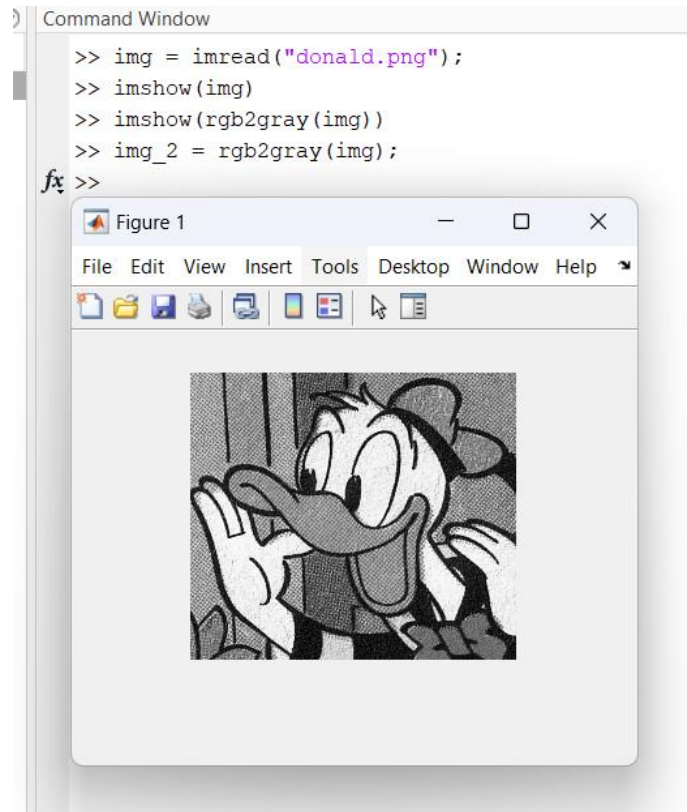


## Images

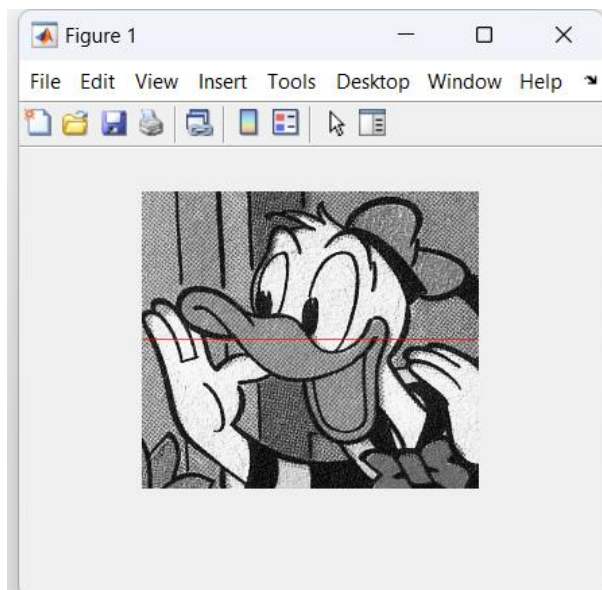
Let's say instead of a signal, we want to display an image. First, we need to find one! If you look at my "Current Folder" window, you'll notice I have one file called "donald.png". Let's try showing it.



You'll notice, the array is not 2D but rather 3D! That is because it is a color image, and contains three arrays of red, green, and blue (RGB) versions of the image. To make it gray, we can call "rgb2gray".



One thing you may do a lot of is analyze sections and lines of an image. Let's say we wanted to grab a line of data on the red line below, which is halfway down the image.

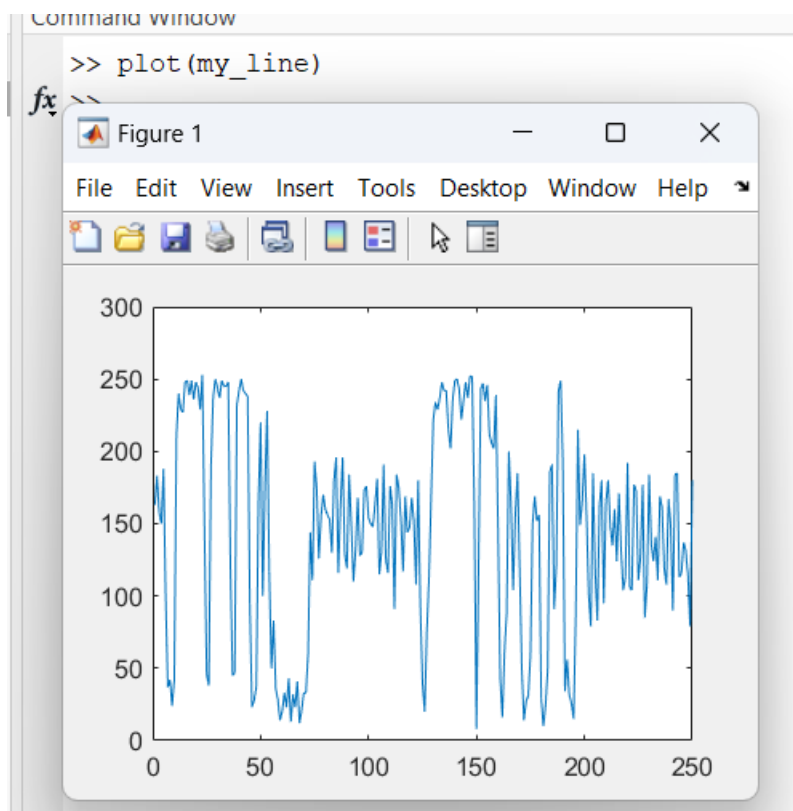


Remember: images are really just m by n arrays, and we know how to get values from those already! Using a similar technique as we did before, we can grab a specific rows (or columns) with the following command.

```
>> my_line = img_2(220/2, :);  
fx >>
```

Workspace	
Name ^	Value
img	220x250x3 uint8
img_2	220x250 uint8
my_line	1x250 uint8

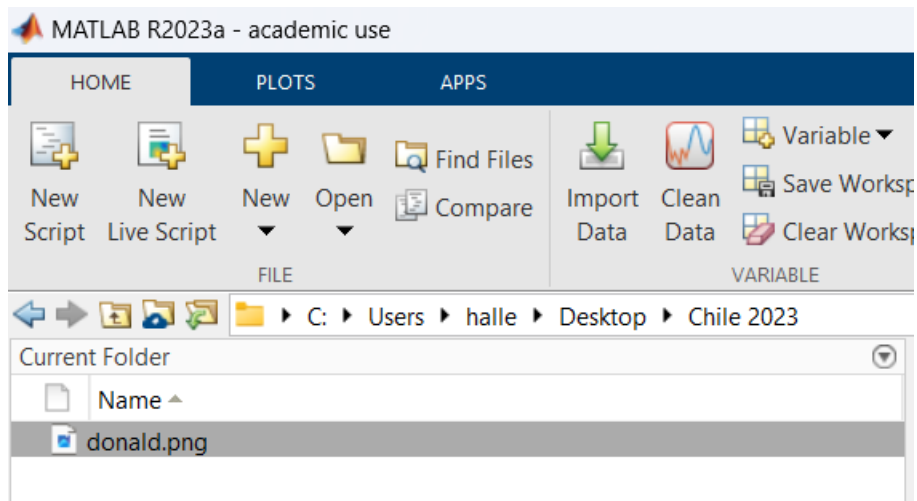
Great! Let's plot it.



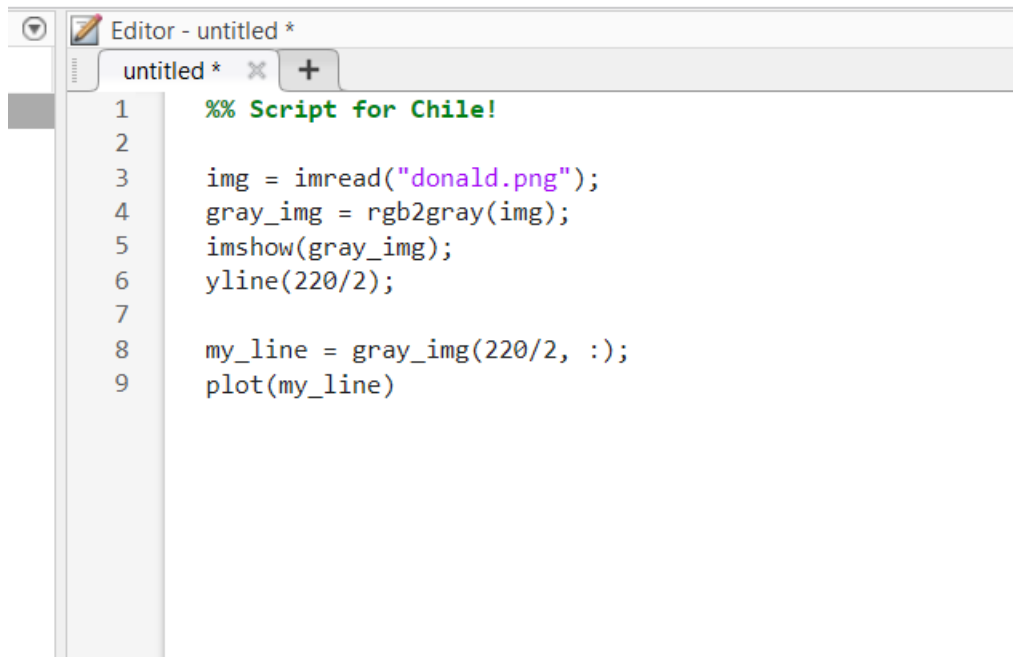
## Scripts and Functions

So we've worked very hard to import our image, change it to gray, and create our plot for the data we want. Now, let's say we need to do that *every time we open MATLAB*. Do we want to type out all those commands every time? Nope! This is where scripts come in.

In the top left window, click "New Script".

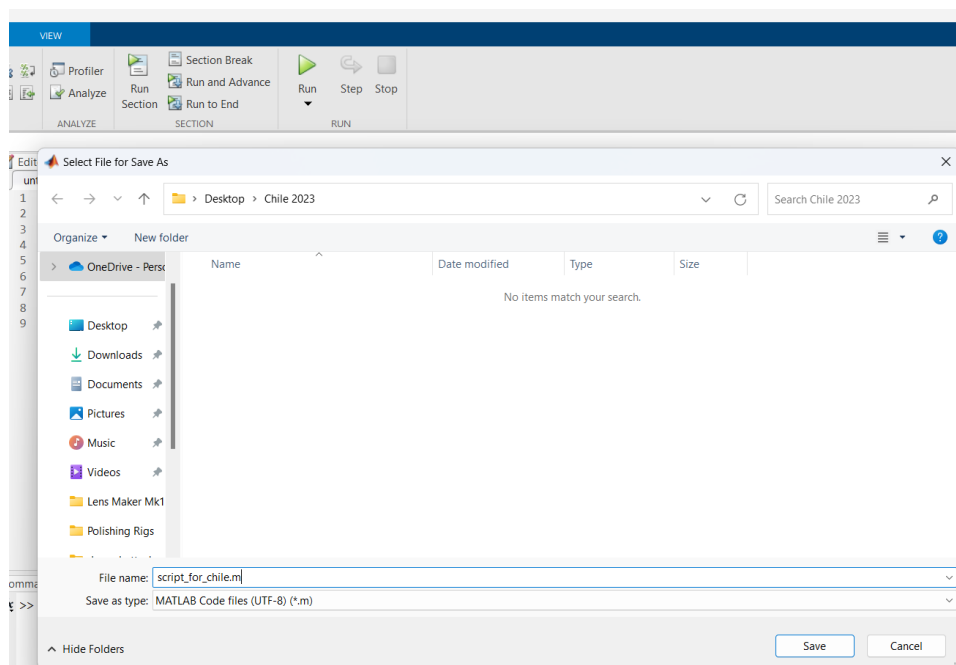


A new window called the “Editor” will open. Here, you can write lines of code that can be executed all at once! Let’s write a quick script that replicates what we did for the image.



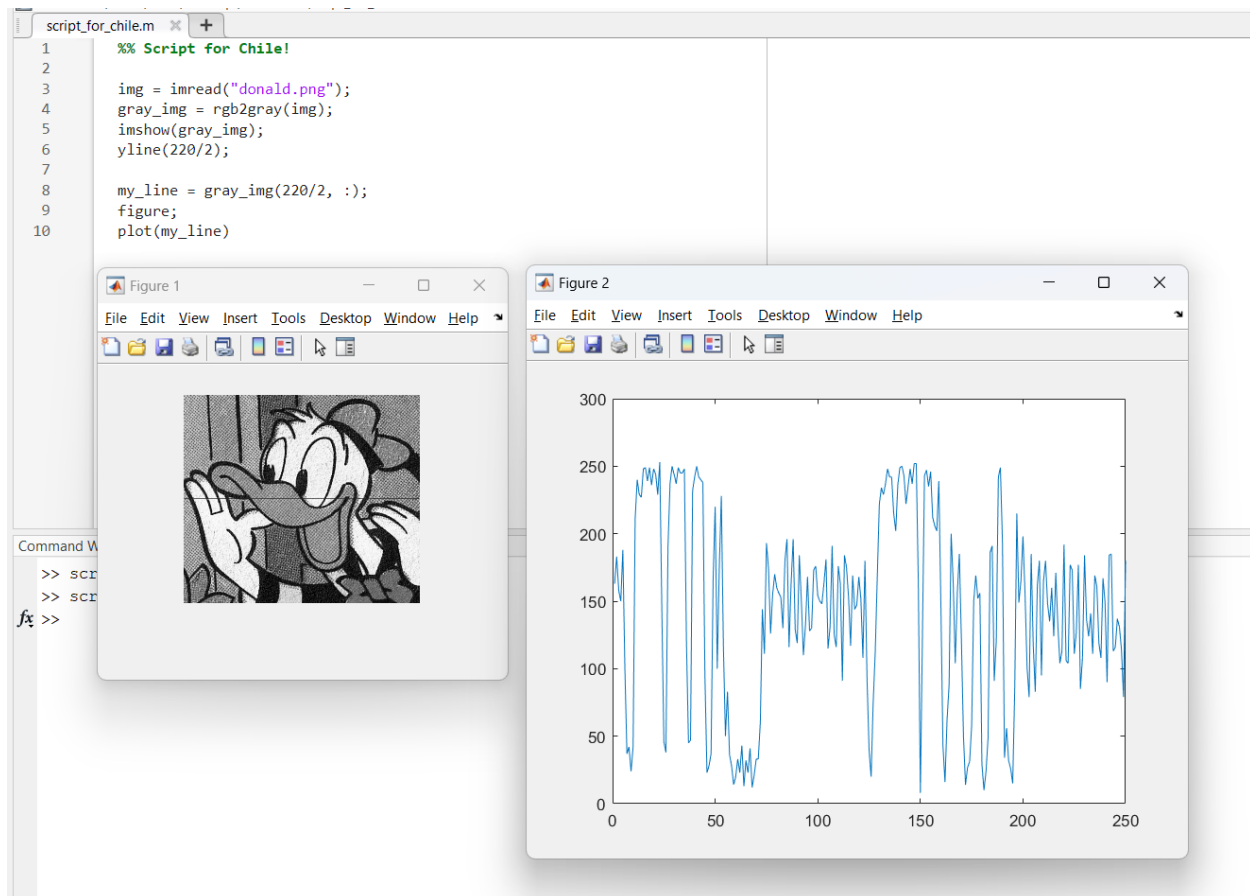
```
1 %% Script for Chile!  
2  
3 img = imread("donald.png");  
4 gray_img = rgb2gray(img);  
5 imshow(gray_img);  
6 yline(220/2);  
7  
8 my_line = gray_img(220/2, :);  
9 plot(my_line)
```

When you are happy with your script, click “Run”. Save your file first!





Two figures should pop up!



### *Functions*

Here is another oddly specific but incredibly useful situation. Let's say you are writing a script, and you want to use the code we just wrote inside of another script. Should we rewrite/copy paste it? You could, but that would be time-consuming to do so. What if instead, we created our own function! Here is how you could do that.

The screenshot shows the MATLAB environment. The Editor window displays a function named `dispImg` defined as follows:

```
1 %% Function for Chile!  
2  
3 function gray_img = dispImg(img)  
4     gray_img = rgb2gray(img);  
5     imshow(gray_img);  
6     ylabel(220/2);  
7  
8     my_line = gray_img(220/2, :);  
9     figure;  
10    plot(my_line)  
11 end  
12
```

The Command Window shows the execution of the function:

```
>> img = imread("donald.png");  
>> dispImg(img);  
fx >>
```

Two figure windows are open. Figure 1 displays a grayscale image of Donald Duck. Figure 2 displays a line plot of the data extracted from the image, showing a highly oscillatory signal with values ranging from approximately 0 to 300 over an x-axis from 0 to 250.

Functions are a great way to reuse code you know will be called a lot. Try to use them as much as you can!