Important Things

- ;
    - Blocks the code from displaying the input in the command window
    - Good for decluttering the command window
- \n
    - Forces the next line of code (or user input) to be on the next line
    - Can be used to separate different outputs and make user understand information better
- clc
    - Clears all the text in the command window
- clear
    - Clears all the variables from variable window
- Documentation
    - Can click on the “Help” button on the top right corner of the screen and search for functions you don’t know
    - Can also type ‘doc ‘query’ ’ into command window to access documentation

Random Numbers

- **rand**
    - Will generate random numbers between 0 and 1 (not inclusive)
    - Ex:
      rand(5,10)
        - This will generate a matrix with 5 rows and 10 columns with numbers between 0 and 1 (floats)
- **randi**
    - Will generate random integers, can specify range
    - Ex:
      randi([1 10],5,10)

        - This will generate a matrix with 5 rows and 10 columns with numbers between 1 and 10

Importing/Outputting Data

- **.mat files**
    - Can store many variables in one file
        - Data does not have to be same type
    - Use load function to pull data from a .mat file
    - Ex:
      load('my_file.mat')
        - This function will take all the data from "my_file" and put them into the workspace
    - Can also load specific variables from the .mat file
    - Ex:
      load('my_file.mat','var_1','var_2')
        - This function will specific load the variables named var_1 and var_2 from the "my_file" file
    - Can also use save function to save new data to a file
    - Ex:
      save('new_file.mat)
      save('new_file.mat','var_1','var_2')
        - The top one will save all variables to the file, the second one will save only those named variables
- **.csv files**
    - Use writematrix command to export matrices
    - Ex:
      writematrix(a,'my_file.csv')
        - This command will export variable a to the file "my_file"
        - This command only works with matrices, need to use other commands for other data formats
    - Use readmatrix command to import matrices

    - Ex:
      vec=readmatrix('my_file.csv')
        - This command will take the data from "my_file" and store it in the variable "vec"
        - This command only works with matrices, need to use other commands for other data formats
- **fprintf**
    - nickname = fopen('file_name') → used to open file and get file ID
    - fprintf(nickname, '_____') → used to write things in the file, in between fopen and fclose
    - fclose(nickname) → used to close the file
    - fscanf(nickname, '%__',#_of_points_read) → used to read the file, in between fopen and fclose
        - Files are read from the top left and goes to the right
        - If the items being scanned are strings, have curly brackets around the index of the variable if you have index
        - Ex:
          for idx = 1:10
          strings{idx} = fscanf(data,'%s',1)
          end
        - H

<u>Debugging</u>

- **What is a bug?**
    - An error that prevents the code from running
- **How to minimize the amount of bugs**
    - Create an algorithm before coding
    - Add comments in the code
    - Write readable code
    - Reese's pieces approach

        - Build small portions of code and then stitch them together

Inputs

- variable_name = ____
    - Simplest way, just type into the script window and the program will store the value under the variable name
    - Ex: x = 4
    - Ex: First_name = ‘Karthik’
- variable_name=Input(‘prompt \n’, ‘s’);
    - Lets user put in a value for a variable name
    - Add the highlighted portion if the input is a character/string
    - Ex: x=input(‘What is 2+2? \n’);
    - Ex: x=input(‘What is your first name? \n’, ‘s’)

Outputs

- disp(variable_name)
    - Simplest way, just type in the script and the variable will be displayed in the command window
    - Works for both number and string/character variables
- fprintf(‘text with %_(s)’, variable_name(s))
    - Allows you to fill in the blank for sentences based on earlier inputs
    - This highlighted portion are placeholders for the variable within the sentence you are outputting, the placeholder changes based on what kind of variable it is

Symbol	What kind of variable?	How do you put it into the script window?	What comes in the command window?
%s	Strings	first_name=’Karthik’; fprintf(‘Your first name is %s’, first_name)	Your first name is Karthik

%f	Numbers (floating numbers)	x=9.800; fprintf('Acceleration due to gravity is %.1f m/s^2', x)	Acceleration due to gravity is 9.8 m/s^2
%i / %d	Numbers	x=4; fprintf('The answer you put in was %i', x)	The answer you put in was 4
%c	Character	middle_intial='D'; fprintf("My name is John %c. Doe", middle_initial)	My name is John D. Doe
%e	Numbers	x=48458723485276948569236489; fprintf('This is the number %e', x)	This is the number 4.845872e+25

- This highlighted portion contains the variable names in the order that they are placed into the sentence
  - Ex:
  - First_name='Karthik';
  - Age=17;
  - fprintf('You are %i years old and your first name is %s', Age, First_name)

Vectors

- Variable_name = [number1 number2 number3 ….]
  - This is how you can manually create vectors, just put numbers within the brackets with spaces between them
  - Can put as many numbers as you want
- Variable_name = [number1;number2;number3 ….]
  - This does the same thing but the semi-colons make the vector vertical
- Note: Can also have functions for each cell of a vector and compute the value of each cell within the command
- Variable_name = [initial_number:final_number]
  - This creates a vector between the initial number and the final number with a step size of 1
  - Ex: [1:10] => 1 2 3 4 5 6 7 8 9 10
- Variable_name = [initial number:step size:final number]

    - This creates a vector with numbers with a specific step size that you want
    - Ex: [0:2:10] => 0 2 4 6 8 10
- linspace(initial number,final number,amount of numbers)
    - This creates a vector with a specific size you indicated between the initial and final numbers
    - Ex: linspace(0,1,5) => 0 0.250 0.500 0.750 1.000
- Note: Vectors created automatically are row vectors, to make a row vector into a column vector do variable_name = vector_variable_name’
    - Doing disp(variable_name) will output a column vector
- **Indexing**
    - Each cell in a vector has a number, the leftmost cell is 1 and increases as you go to the right
        - If it is a vertical vector, the top cell is 1 and increases going down
    - Can call that value in the cell by doing vector_name(position#)
    - Can also change the value of a cell by doing vector_name(position#)=new value
    - Can change range of values through variable_name(initial position#:final position#)=new value
        - Note: Can use “end” as the final position to make the range go to the last cell of the vector
- **Operations**
    - To add, subtract, multiply, or divide between a vector and scalar value, just do vector_name(+-*/)scalar value
    - To add or subtract two vectors, do vector1_name(+-)vector2_name
    - To multiply two vectors, do vector1_name .* vector2_name
    - Can do max(vector_name) to find maximum value of vector

        - Can find have 2 outputs by doing [output1_name, output2_name]=max(vector_name)
        - Output 1 is the max value itself
        - Output 2 is the position of the max value on the vector
    - Can do size(vector_name) to find how many values are in vector
- **Plotting**
    - Can do plot(vector1,vector2)
        - Vector1 will be on the x axis and vector2 on the y axis
    - After defining the axis, you can also include color, line type, and point type
        - Ex: plot(vector1,vector2,“r--*”)
            - This creates a line that is red, with dashed lines, and points marked by a star
    - Can use “hold on” to get the next graph you plot to plot onto the previously created plot
        - If you want to stop having new plots that you make be put onto the same graph, run the “hold off” command
    - If you plot only one vector, then the y-axis will be the vector values and the x-axis will the position numbers of the vector
    - Can add a labels to a graph
        - Graph title: title(“graph title”)
        - x-axis: xlabel(“x axis title”)
        - Y-axis: ylabel(“y axis title”)
    - Can add legend by doing legend(“label 1”, “label 2”...)
        - Label 1 is the first graph that you put onto the plot, the graphs you added later will be label 2, label 3, etc.

<u>Matrices</u>

- Variable_name = [number1 number2 number3; number 4 number 5 number 6]

    - This is the manual way to create a matrix
    - The amount of numbers on each side of the semi-colon has to be the same
- Variable_name = [vector1;vector2]
    - If you have two vectors of the same length, then you can combine them like this
- **Indexing**
    - Rows and columns start with position 1 at the top right corner
    - Can call that value in the cell by doing matrix_name(row#,column#)
        - Note: Can substitute “end” for either the row or column number if your want the last row or column respectively
    - Doing matrix_name(row#,:) or matrix_name(:,column#) means you want just that row or column
    - Can isolate range of rows or columns by having the initial row/column number:final row/column number on their respective side of the comma
- **Operations**
    - Can do size(matrix_name) to find dimensions (rows columns) of matrix
        - Can find have 2 outputs by doing [output1_name, output2_name]=size(matrix_name)
        - Output 1 is the number of rows
        - Output 2 is the number of columns

Repetition

- **Definite “For” Loops**
    - Set a beginning and final parameter for the amount of loops you want
        - Can also set a step size for the loop number

- Set “count” or some variable as the loop number in the “for” command
- Structure example:

```
for count = 1:1:4
    fprintf('The count is %i.\n', count);
end
```

  - This loop will run four times and based on the loop count (either 1, 2, 3, or 4) it will print out a sentence with the corresponding loop number
- Another example:

```
for count = 1:5
    fprintf('I will not talk in class.\n');
end
```

  - This loop wll print out the sentence 5 times
- Making vectors:

```
for idx = 1:4
    v(idx) = idx*2;
end
disp(v)
```

  - This loop will go run 4 times and add more cells to the vector “v”

- **Nested “For” Loops**
  - Example:

```
for row = 1:4
    for col = 1:3
        eagle(row,col) = row^col;
    end
end
disp(eagle)
```

    - This loop will print out the matrix:

        -

1	1	1
2	4	8
3	9	27
4	16	64

        - The loop will start with row 1, then do column 1, then column 2, then column 3, then it will move on the row 2 and start again
- **Loops and Vector Output**
    - If v=[5, 8, 6, 3, 4, 7]

```
for k = 2:2:6
    fprintf('%i\n', v(k));
end
```

        - This loop will only print the positions that that k will equal to
- **Indefinite "While" Loops**
    - Checks if the condition is true or false
    - If the conditions comes as "true", then the loop will continue to run
    - If the condition comes as "false", then the loop will stop
    - Example:

```
number = 0;
while number < 5
    fprintf('%i is an acceptable number.\n',number);
    number = input('Enter a number.\n');
end
fprintf('The end.\n')
```

        - This loop will continue to ask the user to input a number until the user inputs a number greater or equal to 5
- **Break**
    - Can use "break" function to break loop (including nested loop)

Comparisons and Selection Structures

- Computers denote 1 as “true” and 0 as “false”, these are called Boolean numbers
- **Relational Operators**
  - “<” : Less than
  - “>” : Greater than
  - “<=” : Less than or equal to
  - “>=” : Greater than or equal to
  - “==” : Equal to
  - “~=” : Not equal to
- **Logical Operators**
  - “&” : Both relations in a condition have to be true in order for the computer to return “true”
  - “|” : At least one relation has to be true in order for the computer to return “true”
- **If Statements**
  - Structure:
    ```
    if (condition)
        action(s)
    end
    ```
  - Can also have “elseif (condition)” before the “end” command to keep everything within one structure
  - Doing “else” does not require condition, can only use once
    - Used for inputs that do not fall into any specified conditions
  - Conditions can also be character comparisons
  - Example:
    ```
    if (initial == 'A')
        disp('Go to the front of the line.\n');
    end
    ```

- **Error Function**
    - One of the action(s) within the if statement can be an "error"
    - If the condition for an error is satisfied, then the computer will print an error message
    - Syntax:

```
if (condition)
    error('message')
End
```

Graphing Data

- **Plot Command**
    - plot(y_data) where y_data is a vector
        - If plotting a vector, the x-axis will be the index values and the y-axis will be the numbers in the vector
    - If plotting with two axis: plot(x_data, y_data)
        - x_data and y_data are corresponding vectors with equal length
    - Add 'LineSpec' within the parentheses to change the plot marker, color, and line type
        - Specification list:

Color

b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

Symbol

.	point
o	circle
x	x-mark
+	plus
*	star
s	square
d	diamond
v	triangle (down)
^	triangle (up)
<	triangle (left)
>	triangle (right)
p	pentagram
h	hexagram

Line Style

-	solid
:	dotted
-.	dashdot
--	dashed
(none)	no line

        - Example of line specification:
        - plot(x_data, y_data, ‘ro’)
            - This plot will have points that look like “o” and be red, since no line style was indicated no line will be drawn
        - Plots are spoken as y vs. x, but MATLAB plots (x,y)
            - Plot wind vs. time should be coded as plot(time, wind)
- **Labeling Plots**
    - xlabel(‘Label for x-axis’)
    - ylabel(‘Label for y-axis’)
    - title(‘Title of my plot’)
        - Need descriptive title, not just y vs. x
- **Multiple Figure Windows**
    - Usually, creating a new graph will wipe out any graph you previously made on the figure
    - Need to use figure() command to fix this
    - Example:

```
t=[0:0.1:10]
y1=sin(t)
```

```
y2=2*sin(t)
figure(1)
plot(t,y1)
figure(2)
plot(t,y2)
```

- This will cause two figure windows to pop up at the same time showing each graph

- **Multiple Plots on One Figure**
  - Option 1:

    ```
    t=[0:0.1:10]
    y1=sin(t)
    y2=cos(t)
    plot(t,y1,t,y2)
    ```

    - The central comma differentiates between the two plots within one plot command
  - Option 2:

    ```
    t=[0:0.1:10]
    y1=sin(t)
    y2=0.5*sin(t)
    y3=2*sin(t)

    hold on
    plot(t,y1)
    plot(t,y2)
    plot(t,y3)
    hold off
    ```

    - Allows you to cleanly input all of your plots
- **Legend**
  - Need a legend when having multiple plots on one window
  - Syntax:

    ```
    legend('label 1','label 2')
    ```

        - Label 1 corresponds to the first plot inputted and so on

Polyfit and Polyval

- **polyfit**
    - Polyfit will create an equation for the data you input into the function
    - polyfit(x_data,y_data,order)
        - x_data and y_data are vectors that are the x and y values of the plot
        - Order is the type of graph (linear, quadratic, cubed)
        - The result from this function is a vector with the coefficients for each
    - Ex:
      polyfit(x_data,y_data,2)

      Output: [a,b,c]

        - The equation is ax^2+bx+c
    - Can plot the given equation onto the graph, but this could cause error
        - Use polyval instead
- **polyval**
    - This function uses the coefficients from polyfit and the x values to calculate the y values for the equation
    - p=polyval(coefficients,x_values)
        - Now you can plot the equation onto the graph
        - plot(x,p)

Logical Vectors + Indexing

- **Logical vectors**
  - These are vectors that will show when a condition you put onto a vector are true
  - Ex:
    Vector = [1,2,3,4,5]
    Big = vector>3

    Output: Big = [0,0,0,1,1]
    - Logical vectors use Boolean operators (0 is false and 1 is true)
  - Logical vectors are good at telling where the condition is true, but it will not give what the values are at the positions
    - Need to use logical indexing
  - If you make a vector going from 1 to the length of the vector, then insert the logical vector you will get the index positions
  - Ex:
    vec=[1:length(Vector)];
    vec(Big)

    Output: vec(Big)= [4,5] (indexes of Vector)
- **Logical Indexing**
  - These are vectors that will give the values in the original vector that match the condition
  - Ex:
    Vector = [1,2,3,4,5]
    Big=vector(vector>3)

    Output: Big=[4,5]
  - Logical indexing works very well when you want to have functions (like mean and max) to manipulate the data

Strings

- **Character Vectors**
  - These are words/sentences with single quotes around them
  - The length() of these are the amount of characters (including spaces)
- **Strings**
  - These are words/sentences with double quotes
  - The length() of these is always 1 since it counts as one item
  - Can use strlength() to find the amount of characters in a string without needing to convert to character vector
- **Strings to Character Vectors**
  - Can use curly brackets to convert a string to a character vector
  - Can then index the character vector
  - Ex:
    my_string=”Hello there!”
    my_string(1) → ”Hello there!”
    my_string{1} → ’Hello there!’
    my_string{1}(1:5) → ’Hello’
- **strcat**
  - Can join together both character vectors and strings
  - Ex:
    strcat(“Harry ”,”Potter”) → “Harry Potter”
      - For strings, spaces will be kept (space is after Harry)

    strcat(‘Ron ’, ‘Weasly’) → ‘RonWeasly)
      - For character vectors, all spaces will be removed (space was after Ron)

- **strtrim**
  - Will remove leading and trailing blanks for both strings and character vectors
  - Ex:
    strtrim("   Goodbye blank characters!   ") → “Goodbye blank characters)
      - This will also work for character vectors
- **strcomp**
  - Will check if two strings/character vectors are the same will return a Boolean value (1=same, 0=different)
  - Ex:
    word1 = 'wand'
    word2 = 'WAND'
    word3 = "wand"
    comp1 = strcmp(word1, word2) → 0
      - Case matters, thus it returns a 0 since they are different

    comp2 = strcmp(word1, word3) → 1
      - Can compare between character vectors and string, if they are the same it will return a 1
- **split**
  - Will split a string into a string vector with each word becoming a cell in the vector
  - Splits along spaces, tab, or a new line within the original string
  - Ex:
    phrase = "My name is Karthik"
    words = split(phrase) →
    “My”
    “name”
    “is”
    “Karthik”

        - Will make it into a vertical vector, to make a horizontal do: split(phrase)’

    disp(words(3)) → is

        - disp will not output with quotation marks

    word = words{1}(1) → ‘M’

- **strfind**
    - Will find the indexes of substrings within a string or character vector
    - Function will work between string and character vectors, does not matter at all
    - Ex:

    a = strfind('abcde', 'd') → 4

    b = strfind("abcdefghi", 'ef') → 5

        - Will give the index of where the first matching letter is

    strfind('abcdabceabcdabcdddd', "ab") → [1 5 9 13]

    strfind("abcd", "e") → [ ]

        - Will create empty vector if substring can not be found

- **strrep**
    - Will find indicated substring and replace it with a new designated substring
    - Ex:

    strrep('abcdefg', 'e', "x") → “abcdxfg”

        - Again, it does not matter at all if there are character vectors and strings in the same function
        - If any of the three inputs are strings, then the output will be a string no matter what

- **count**
    - Will count the amount of times a substring shows up
    - Ex:

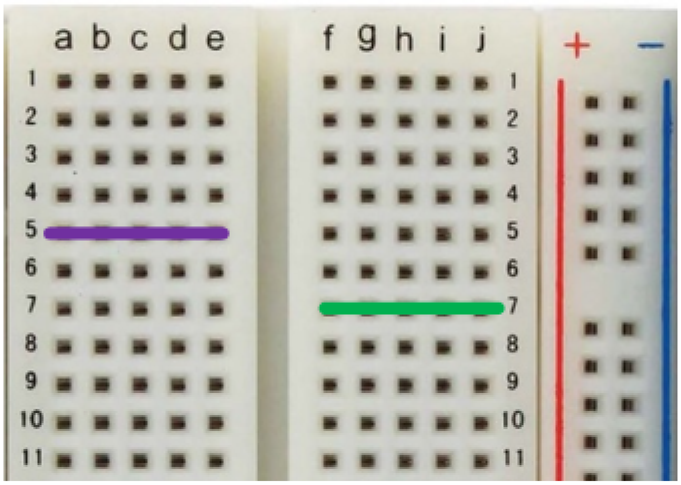
    count("abcdabceabcdabcdddd", 'd') → 6

- Again, it does not matter at all if there are character vectors and strings in the same function

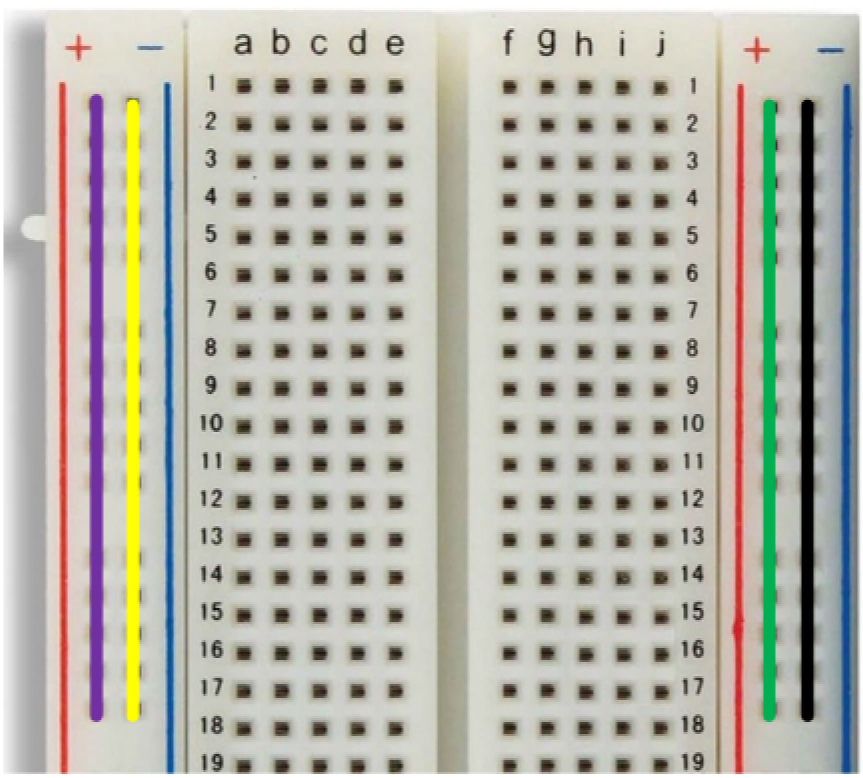
Arduino + Breadboard

- **Pins**
    - Pins labeled A0 to A5 are analog pins for analog sensors
        - For analog waves
    - Pines labeled D0 to D14 are digital pins for digital sensors
        - For digital waves
        - Pins with “~” can work with analog sensors
- **Coding**
    - First line of code should be:
      a = arduino();
    - To turn pin on, do:
      writeDigitalPin(a,’pin’,state)
        - a is the variable assigned to the arduino
        - pin is the designation of the pin you are turning on
            - Use D13 for the LED light
        - state is either 1 (on) or 0 (off)
    - Can use pause code to keep pin on or off for a certain amount of seconds
        - Ex:
          writeDigitalPin(a,’D13’,1)
          pause(5)
          writeDigitalPin(a,’D13’,0)
            - This code will keep the LED on for 5 seconds
    - Can use loops to get the pin to turn on and off a certain amount of times
- **Arduino Commands**
    - Use “readVoltage(a,’pin’)” command read the voltage coming from an analog sensor

- **Breadboard**
  - Holes are connected in groups of five across the board
  - Holes NOT connected row to row
  - NOT connected across the middle gap



- Holes are connected up and down a column as shown.
- Each column is NOT connected to the others



User Inputted Functions

- Functions that the user can create, not the functions that are already built into the code
- Format for creating functions:
  function [output variables] = Function_Name(input variables)
  Operations;
  More operations;
  end
  - Operations is where you show how the input variable(s) will become the output variable(s)
  - Good idea to suppress the operations to avoid clutter
- If you are going to use a function you made in the code, have the calculation using the function be before the code that defines the function
- Ex:
  n=4;
  n_squared=square_it(n);
  function square_val=square_it(num)
  square_val=num*num;
  end
  - The function square_it is "used" before it gets defined later in the code
- Can call a function from another script file by doing this:
  - Create a script file and put your function in there (do not include housekeeping)
  - Name that script file with the function as the same name of the function
  - Create a new script file and call the function
  - Function will be used in calculation

Control Theory

- Too high level for this class, but oh well
- System: A system is a process or device with inputs and outputs, and its behavior is described by mathematical equations.
- Controller: A controller is a mechanism that adjusts the inputs to a system to achieve desired outputs or regulate the system's behavior.
- Feedback: Feedback is a crucial concept in control theory. It involves measuring the system's output and comparing it to the desired output. The difference, known as the error, is used to adjust the system.
- Open-loop: The output is not used to influence the system.
- Closed-loop (feedback): The output is fed back to the system to regulate its behavior.
- There are several ways to measure the effectiveness of a controller and feedback systems.
- Rise Time: The time it takes for the system's response to reach a certain percentage (commonly 90%) of its final value after a step change in the input.
- Delay time: Time it takes to reach 50% of steady state value
- Settling Time: The time it takes for the system's response to settle within a specified range around its final value after a step change in the input.
- Overshoot: The extent by which the system's response exceeds its final steady-state value during a transient period.