MATLAB Workshop A24

This workshop is intended to be useful to students in either RBE 2001 or RBE 3001. There are several sections, some of which cover more basic skills, while others focus on more advanced skills specific to RBE 3001. Feel free to go through the sections that interest you and skip the ones that don't.

Section 1: Plots

1.1 Basic Plots

1.1.1 Get Data

Data for plots should be stored in arrays. When collecting experimental data, you can create a variable with an array of zeros assigned to it, then update the values within the array with your data. For the purposes of this workshop, we will focus on using data that has already been collected and saved to a file.

A `.mat` file stores the variables of a MATLAB workspace and is proprietary to MATLAB. To save your data, you would use the `save` command like so: `save example.mat variableToSave otherVariableToSave`. This command takes the filename first, then the names of the variables you'd like to save. To load the variables from a `.mat` file, use the `load` command: `load example.mat`.

Another way to save and load data is via `.csv` files. CSV stands for comma separated values, and a `.csv` file stores data uniformly by separating columns of data with commas and separating rows with newlines. MATLAB uses the function `writematrix` to save data to `.csv` and the function `readmatrix` to load data from a `.csv`. The following code example shows saving and loading data to and from a `.csv`.

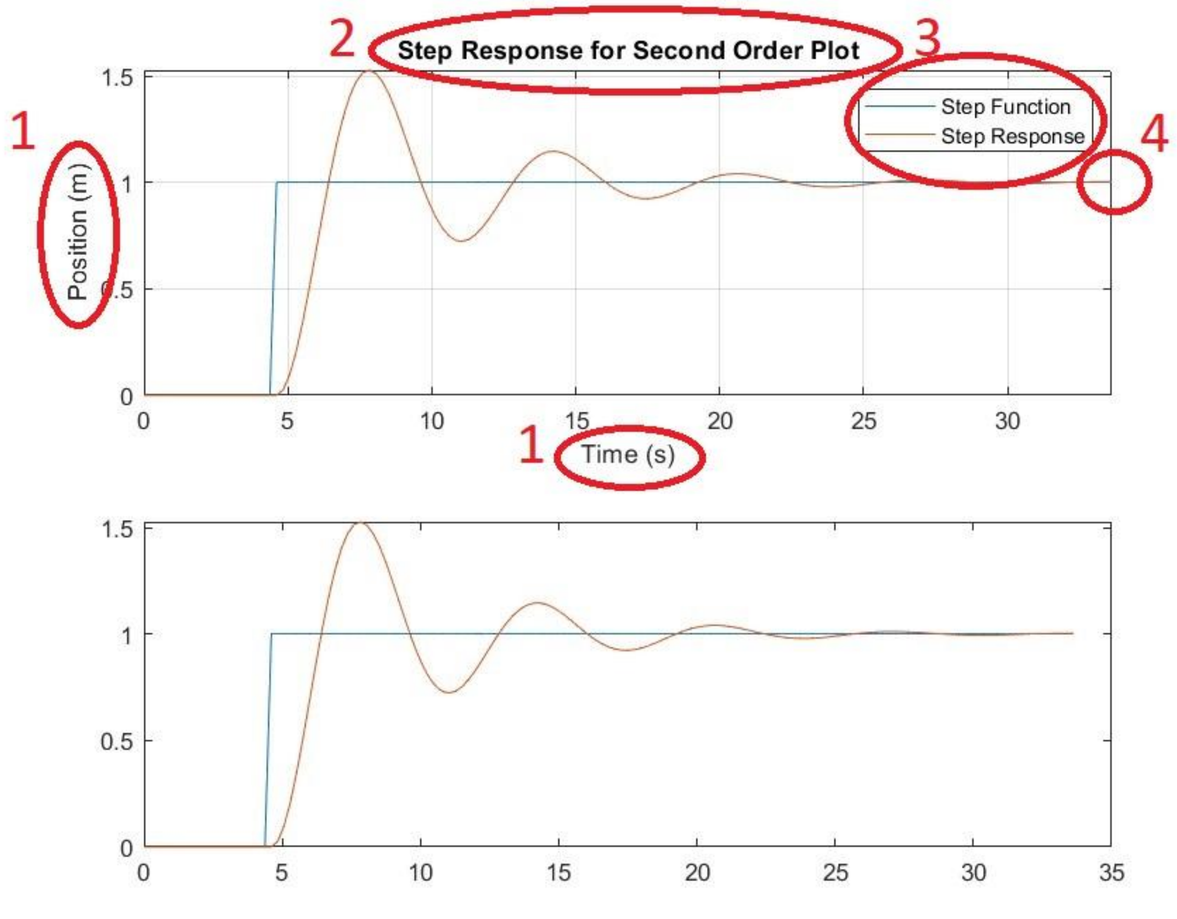
```
positionData = [0, 0, 0, 0;
                0, 1, 1, 0;
                1, 0, 0, 1];

writematrix(positionData,'posData.csv','Delimiter',',');
loadedData = readmatrix('posData.csv');
```

1.1.2 Plots and Labels

The fastest way to make a plot is to use the `plot` function and pass it the values for the x and y axes: `plot(xVals, yVals)`, where `xVals` and `yVals` are arrays of numbers. For a histogram, use the `histogram` function with the values to put into bins and the steps for each bin. Below is an example of how to make a histogram.

```
timeVals = [1 1 1 2 3 4 4 5 3 2 4 2 1 1 3 5 3];
timeSteps = 1: 1: 5; % Array of values from 1 to 5 with spacing of 1
→ 1 2 3 4 5
histogram(timeVals, timeSteps);
```

When plotting in MATLAB, there are a few characteristics that are fundamental to every plot that is created. Shown in the image below is a side-by-side comparison of the same data with and without the proper attributes.



1. **Axis titles:** The specific units of the plot are critical to fully understanding what the figure is trying to represent. These are appended to a plot through the xlabel and ylabel labeling functions.
2. **Main title:** All plots should have a descriptive title to explain the general purpose of the data it is representing. These ca\n be added through the MATLAB title function.
3. **Legend:** When plots have multiple lines, a legend is incredibly important to assure that the reader is able to distinguish between the data. MATLAB's legend function makes adding a legend very straightforward.
4. **Axis limits on the x and y axis:** In general, MATLAB autosizes data, but that constraint does not always look the cleanest. These plots can be trimmed with the xlim function and the ylim function.
5. **Grid line:** They can make the plot clearer and easier match a line to a value. Grid lines can be added with the grid function in MATLAB.

1.1.3 Multiple Lines on One Plot

To plot multiple lines of data on one plot, you must signal to MATLAB to keep plotting data on the same figure rather than making a new figure for each plot. To do this, use the `hold` command. `hold on` tells MATLAB to keep plotting data on the current figure, while `hold off` tells MATLAB to move on. Below is an example using the same x values but different y values.

```
hold on
plot(timeSteps, posData1);
plot(timeSteps, posData2);
plot(timeSteps, posData3);
hold off

legend({"Position 1", "Position 2", "Position 3"}, 'Location',
'southeast');
```

1.1.4 Subplots

Say we want to have two sets of data graphed together but not on the same graph. This is called creating subplots. You can think of a figure (the window that MATLAB pops up whenever you tell it to make a graph) as a blank canvas. The `subplot` function simply tells MATLAB how to size and place your graph on this blank canvas. The most common form of the subplot function is `subplot(m,n,p)`, where `m` is the number of rows of graphs in your plot, `n` is the number of columns of graphs in your plot, and `p` is the index of the plot. Indices increase from top to bottom and then left to right. So if you have a 2x2 subplot the first index will be the top left, the second index will be the bottom left, the third index will be the top right, and the fourth index will be the bottom right.

For example, to create a subplot with a single column and two rows, use `subplot(2,1,1)` above your first plot instruction and then `subplot (2,1,2)` above your second plot instruction. This tells MATLAB where the following graph is going in the figure.

1.2 Live Plots

Say we need to update our plot live as we receive data. We still use the same functions to plot the data, but we use a loop for iterating through data and updating the plot repeatedly.

As a note, for loops compute very quickly in MATLAB, so we also need to slow the plot down a little bit. We will be using the `pause` function, which takes a number of seconds as an argument. This lets us view the plot updating live. You can play with the pause length to see what refresh rate looks good for the live update. If you want to see an example of what this plot could look like, click here.

We will use the same `plot` function from earlier, we now just need to store data, for which we have a few methods. The first method is that we make a matrix of zeros that is the size of the data and the timestamps included in the data. The second one, is that we initialize an empty matrix `plotData = []`, and with each iteration we update our data, by saying `plotData = [plotData, data(:, timestamp)]`. The second method may be more computationally expensive to run, but it makes the plotting a bit easier, as then your animation won't have a line from the end of the current plot to (0, 0). When you are working with real robots, you want to initialize your matrices with zeros, ones, or whatever value makes sense. Memory allocation is costly and makes our code slower, and that can be detrimental to some robotics applications.

1.3 Plot Derivatives

There are a few ways to calculate the discrete derivative of datasets in MATLAB. One of the most common ways is to use MATLAB's built in gradient function. This function takes in a single array of data and calculates the difference between each element and the next. Recall that a derivative is $\frac{\Delta y}{\Delta x}$. You'll need to calculate the gradient of both the y vector and the x vector, and then divide them appropriately. If you don't want to use a built-in function, you can also create this code on your own by taking the change and x and y for each element of your array and dividing them. Below is an example of getting velocity from position data using the `gradient` function. Note that the change in time is uniform for this example.

```
timeSteps = [0: .5: 5];
position = [0 1 1 1 2 2 5 5 1 3 3];
velocity = gradient(position);

hold on
plot(timeSteps, position);
plot(timeSteps, velocity);
hold off

legend({"Position", "Velocity"});
```

Section 2: Solve Systems of Equations

There are two main ways to solve a system of linear equations in MATLAB. The first uses `linsolve` to solve the equation Ax = B, where A is a matrix of and B is a column vector. For solving linear systems, A is a matrix of variable coefficients where each **column** is a variable and each **row** represents an equation in the system. B is a column of what each equation is equal to. For example, take the following system:

$3x + 5y - 6z = 10$

$4x - 9y + 10z = 0$

$-2x + y - z = 20$

The following shows the matrices A and B that represent this system, plus the use of `linsolve` to solve the system.

```
A = [ 3 5 -6;
      4 -9 10;
      -2 1 -1];
B = [10; 0; 20];
solution = linsolve(A, B) % [x; y; z]
```

We can also use the `solve` function to solve the same system. The `solve` function takes in the equations themselves as opposed to the matrices of coefficients. Here we use the `syms` command to declare our variables x, y, and z as symbolic variables as opposed to MATLAB variables. **Note: You must have the Symbolic Math Toolbox installed in order to use this method.**

```
syms x y z;
eqn1 = 3*x +5* y - 6*z == 10;
eqn2 = 4*x - 9*y + 10*z == 0;
eqn3 = -2*x + y - z == 20;

solution = solve([eqn1, eqn2, eqn3], [x, y, z]);
xVal = solution.x
yVal = solution.y
zVal = solution.z
```

Section 3: Class Fundamentals

Like other object-oriented programming languages, you can create classes within MATLAB (if you aren't familiar with Object Oriented Programming, refresh yourself here). To learn the basics on how to create a class in MATLAB read through this documentation from Mathworks. The easiest way to create a class in MATLAB is to:

1. Click the dropdown under *New*
2. Click class
3. Add `< handle` to the `classdef` line so that the class is usable in your scripts

```
classdef BasicClass < handle
   % Example code for RBE matlab workshop

   properties
   % Set properties for this class
       Value = 420;
       Name = 'Rho Beta'
   end

   methods
       %Constructor
       function self = BasicClass(name)
           self.Name = name;
       end
       %Getter Function
       function r = getValue(self)
           r = self.Value;
       end
       % Setter Function
       function setValue(self, newVal)
           self.Value = newVal;
       end
   end
end
```

Class variables, or properties of the class, are defined in the `properties` section of the class definition. Any variables you may want to access in your functions or script should be declared here. You don’t have to define the variable values in the properties section: this can also be done in the constructor or other methods.

Class functions are referred to as methods in MATLAB, and they are declared and defined in the `methods` section of the class definition. An individual method is declared with the following template: `function [name of return variable, if there is one] = [name of method]([function arguments])`. At the end of the function goes the `end` keyword.

Each class must have a constructor, even if it is empty. The constructor is called each time an instance of the class is created. The constructor follows a more specific method template: `function self = [class name]([constructor arguments])`. Like Python, MATLAB uses the variable `self` to refer to the class object itself. When accessing a class property within a class method, you must include `self` as an argument and use the `self` keyword in front of the property name in order to get the class property (see the functions in the above example).

Helpful Debugging Tip: MATLAB class files cannot be run like a normal script. In order to create an instance of your class, run your constructor in the terminal or in a separate MATLAB script. For example, `bc = BasicClass(‘Rob Test’)` will create an object called `bc` of the `BasicClass`.

Section 4: Symbolic Math

Note: You must have the Symbolic Math Toolbox installed.

Say you want to obtain the symbolic solution to a problem in order to have a template for future operations so that you don't have to solve for the same equation each calculation. MATLAB allows you to use symbolic variables to perform operations on equations and matrices without substituting in values for your variables. To create individual symbolic variables, use the `syms` command: `syms x y z`. If you want to create multiple versions of one variable (e.g. theta1, theta2, theta3), you can create an array of symbolic variables using the `sym` function and then access the individual ones you need. The below shows how to make any number of a symbolic variable from 1 to i. The `sym` function takes the name of the symbolic variable (which includes notation here for a changing aspect of the name) and the number of iterations of that variable represented as a range (here it is 1 to i).

```
% Assume q to be an array of any length
theta = sym('theta%d', [1 size(q, 1)]); % theta1, theta2, etc.
y = theta(1) + 2*theta(2); % would fail if size(q) < 2
```

Helpful Debugging Tip: Some variables, such as `pi` cannot be used as symbolic variables because they already have a pre-assigned value in MATLAB. Other examples include `alpha` and `inf`.

To substitute values in later, use the `subs` function, passing the symbolic equation, the names of the variables to substitute, and values for those variables as arguments. Below is an example where we assume variables theta1, theta2, and theta3 are symbolic variables created above. Note that in practice, the symbolic variables must be within the same scope (i.e. function) as the `subs` function call so that they are properly defined.

```
eqn = theta1/2 + (theta2 - 90) + theta3;
subs(eqn, [theta1 theta2], [45, -90]) % leaves theta3
```