Lab Assignment #3

Due 2/12/2016 at 4pm on bCourses

Some guidelines for successfully completing an E7 assignment:

- First convert the engineering problem into a function that can be effectively calculated.
  - The function should be written down using pen and paper in the language of mathematics.
- Program the function.
  - Translate the mathematical language into a programming language, i.e., the language understood by your computer.
  - Then run the program and test it to see if its outputs match those specified by the function. If they do, compute the function on the input data and check that your answers make sense and are reasonable. If they do not...go back to the beginning and look for a mistake in your math or in your coding.
- Write comments throughout your code.
  - Clear, informative comments in your code are perhaps just as important as the code itself. The purpose of the comments is to help anybody who is reading the code later to understand what is happening. Useful comments include: describing (in English) what the important variables are and what information they hold, and describing what each chunk of code does. While the functions you will write in this assignment are relatively short, it is important to develop the habit of thoroughly commenting your code.

The assignment will be partially graded by an autograder which will check the outputs of your functions. For your functions to be scored properly, it is important that the names of your functions **exactly** match the names specified in the problem statements, and input and output variables to each function are in the correct order (i.e. use the given function header exactly). Instructions for submitting your assignment are included at the end of this document.

1. Array vs. Matrix Multiplication

   Write a "smart" function that multiplies two arrays together in the manner that makes the most sense for their given dimensions. Use the function header:

   `function [result] = mySmartMultiply(m1,m2)`

   This function should return the result of multiplying the two arrays together using matrix multiplication if their inner dimensions are the same (i.e. `m1` is an M x N matrix and `m2` is a N x P matrix). Alternatively, it should return the result of multiplying the two arrays together using element-wise multiplication if the arrays have the same dimensions in both directions (i.e. m1 is an M x N matrix and m2 is a M x N matrix). In the situation that either multiplication can be used, it should return the string `multiplication ambiguous`, and if neither multiplication can be used, it should return the string `no valid multiplication`. The function should consider multiplying `m1` by `m2`, and not consider the possibility of multiplying `m2` by `m1`. The function should also be able to handle the case when one or both inputs are scalar, and return the appropriate product. This Matlab help page may be useful.

2. Subcritical vs. Supercritical Open Channel Flow

   In the field of open channel hydraulics, flow in a channel or a river can be classified using a dimensionless number called the Froude number. The Froude number represents the speed of the flow relative to the speed that a wave travels across the water's surface, and can be calculated as:

$$Fr = \frac{u}{\sqrt{gh}}$$

   where u is the fluid velocity, g is the gravitational constant, and h is the depth of the flow. The Froude number then determines when a flow is supercritical, critical, or subcritical using the following criteria:

$$\begin{aligned} Fr &> 1, && \text{supercritical} \\ Fr &= 1, && \text{critical} \\ Fr &< 1, && \text{subcritical} \end{aligned}$$

   Write a function that classifies a given flow as supercritical, critical, or subcritical.

```
function [classification] =  classifyFlow(u, h, unitsys)
```

   `classification` should be a string that is either `supercritical`, `critical`, or `subcritical`.

   `unitsys` is an input that describes the unit system that is being used for the calculation, either `metric` or `imperial`. Note that you should use a gravitational constant of

$$\text{g} = 9.81 \text{ m/s}^2 \text{ for metric units}$$
$$\text{g} = 32.2 \text{ ft/s}^2 \text{ for imperial units}$$

   Computer calculations often generate small errors when making calculations, called floating point errors. These errors occur because many fractions cannot be exactly represented in the computer's native binary format. (To see an example of this, try writing the code `0.1 + 0.2 == 0.3` in Matlab. We will discuss this more later in the course.) To account for this, your function should only consider the first 3 decimal places when comparing the Froude number values. This can easily be accomplished by using the Matlab `round` function. The Matlab command `strcmpi` may be useful in assessing the input argument `unitsys`.

3. Sprite Collisions

   You have seen how Snap! can be used to make animations and computer games. Here you will consider the programming decisions made for collisions between sprites using Matlab. Pretend you are programming an arcade style video game and have already written code that checks for collisions between sprites during every step of the game. Now, all that remains is for you to program a function that tells your script how to handle collisions between different types of sprite during the game. Use the function header:

   `function [result] = collision(sprite1, sprite2)`

   where both `sprite1` and `sprite2` could be any of `’laser’`, `’rocket’`, `’player’`, `’fighter’`, or `’mothership’`. The function should return a 1x3 array with elements that represent the points tallied, whether `sprite1` should be destroyed, and whether `sprite2` should be destroyed, in that order. Note that the points tallied are a result of the type of collision, and not assigned to either sprite. Whether or not a sprite should be destroyed should be represented as a double with one representing that the sprite should be destroyed and zero representing that a sprite should not be destroyed.

   Your function should represent the following behavior:

   (a) Lasers destroy rockets, players, and fighters but are destroyed when colliding with the mothership.
   (b) Rockets destroy both sprites when colliding with other rockets, players, fighters, or the mothership.
   (c) The player destroys both sprites when colliding with fighters or the mothership.
   (d) No other collisions have any effect.
   (e) Each fighter destroyed is worth 1 point.
   (f) Destroying the mothership is worth 20 points.

   Note that your function should be able to handle collisions no matter what order the sprites are entered. i.e. Both `collision(’player’,’fighter’)` and `collision(’fighter’,’player’)` are valid functions calls that should be handled appropriately.

   Test Cases:

```
EDU >> collision('rocket','player')
ans =
    [0  1  1]

EDU >> collision('fighter','laser')
ans =
    [1  1  0]

EDU >> collision('mothership','fighter')
ans =
    [0  0  0]
```

4. To EV or not to EV.

   Part I: Consumer Vehicle Recommendations

   The transportation industry accounts for nearly one third of greenhouse gas (GHG) emissions in the United States. As the United States aims to curb its emissions, it will be crucial to wean the transportation industry off of oil. Electric vehicles will play an ever important role in the electrification of transportation. Consumers now need to make smart economic decisions on whether or not investing in the higher capital costs of an electric vehicle (EV) is financially viable as compared to fuel efficient hybrids or standard internal combustion engine (ICE) cars.

   Your task is to develop a function that will recommend both a low-emitting and a cost-conscious vehicle to a consumer given their location (state), annual kilometers traveled, and annual budget (USD 2015). The function will take the form: `function [consumerStruct] = vehicleRecommendation(consumerName, state, annualkmTraveled, annualBudget)`
   Note that `consumerName` and `state` are of type `char`, while `annualkmTraveled` and `annualBudget` are of type `double`.

   This function will return a structure of the following form:

```
EDU >> consumer =
    Name: 'Janet'
    State: 'CA'
    GHG_Recommendation: [1x1 struct]
    Cost_Recommendation: [1x1 struct]

EDU >> consumer.GHG_Recommendation
ans =
    Vehicle: 'BMW i3'
    Cost: 2560
    GHG: 1370000

EDU >> consumer.Cost_Recommendation
ans =
    Vehicle: 'Chevrolet Spark'
    Cost: 1410
    GHG: 2661000
```

   The function should be able to:

   (a) Manipulate the vehicle data (in the file `EV_Comparison.mat`) and perform calculations using the given inputs. Values for California (CA), Kansas (KS), and Florida (FL) are provided for each vehicle. Notice that due to the different gas prices, electricity prices, and electricity generation emissions in each state, the carbon footprint and normalized cost values vary significantly. For example, Kansas has more coal generation than the other states, which means that the electricity used to power the EVs generates more GHG emissions. For this reason, GHG emissions for EVs are significantly higher in Kansas than the other states. Emissions attributed to the manufacturing of the vehicle (life-cycle carbon footprint) are captured in the carbon footprint values. Also note that the normalized cost represents the net present value of purchasing, maintaining, running, and salvaging the vehicle spread over a lifetime of 12 years. The societal cost of carbon, an externality in economic terms, is also captured in the normalized cost. For a given consumer, use the following equations to calculate the annual GHG emissions and annual Cost of each vehicle based on the consumer's state.

- Annual GHG Emissions $(\frac{gCO_{2,e}}{year})$ =
  Total Life-Cycle Carbon Footprint $(\frac{gCO_{2,e}}{km})\times$ Annual km Traveled $(\frac{km}{year})$
- Annual Cost $(\frac{\$}{year})$ =
  Normalized Cost $(\frac{\$}{km})\times$ Annual km Traveled $(\frac{km}{year})$

(b) Use branching statements (*if*-statements) to determine the GHG recommendation and the Cost recommendation for the consumer. Consider using the `strcmpi` and `find` functions for finding the index of the vehicle recommendation based on the calculations you conducted in part (a).

- The GHG recommendation should be the vehicle with the lowest annual GHG emissions for the consumer based on the consumer's state and annual km traveled.
- The Cost recommendation should be the vehicle whose annual cost is closest to the consumer's budget. NOTE: this is not necessarily the cheapest car for the consumer, but rather the car that is closest to the consumer's budget without exceeding it. For example, consider a consumer with an annual budget of \$5,000 per year wanting to purchase a higher-end car within his or her budget. If two cars have an annual cost of \$3,000 and \$4,500 respectively, your function should recommend the \$4,500 car since it is closest to the consumer's budget without exceeding it.

(c) Output a structure including the consumer's name, the GHG recommendation (including the vehicles make and model (see `strcat`), the annual cost of the vehicle, and the annual GHG emissions), and the Cost recommendation (including the vehicle's make and model, the annual cost of the vehicle, and the annual GHG emissions). Make sure your structure output matches exactly the format of the example given above.

NOTE: This function needs the information in the file `EV_Comparison.mat` in order to work properly. You can use the `load` command inside the function to access this data. When your function is being graded, this file will be available in the same directory that your function is in, so do not include a path in the call to `load`. In other words, `load('EV_Comparison.mat')` is okay, but
`load('C:/Users/Brad/Documents/EV_Comparison.mat')` is NOT okay.

Part II: Consumer Vehicle Comparisons

You've written a function that can make both low-emitting and cost-conscious vehicle recommendations; however, consumers may want to have a single recommendation made for them. Write a function of the form:
`function [comparison] = vehicleComparison(consumerStruct)`
This function will take in `consumerStruct` (same structure format as in Part I) and return a string that states the cheaper of the two recommendations and the difference in GHG emissions (consider using `sprintf`). The output string has the following format where italics indicate words or values that change depending on the recommendation:

'The *Vehicle1* costs $*xx.xx* per year less but emits *xx* g CO2e per year more than the *Vehicle2*.'

If a vehicle is cheaper **and** has lower GHG emissions, return the string:

'The *Vehicle1* is the best option for *consumerName* because it costs $*xx.xx* per year less and emits *xx* g CO2e per year less than the *Vehicle2*.'

For example:

'The Toyota Sienna costs $1000.00 per year less but emits 100 g CO2e per year more than the Toyota Sequoia.'

or

'The Toyota Prius is the best option for Joe Schmoe because it costs $100.00 per year less and emits 100 g CO2e per year less than the Toyota Sienna.'

Test Case:

```
EDU >> out = vehicleRecommendation('Brad', 'CA', 10000, 2000)
out =
    Name: 'Brad'
    State: 'CA'
    GHG_Recommendation: [1x1 struct]
    Cost_Recommendation: [1x1 struct]

EDU >> out.GHG_Recommendation
ans =
    Vehicle: 'BMW i3'
    Cost: 2560
    GHG: 1370000

EDU >> out.Cost_Recommendation
ans =
    Vehicle: 'Toyota Tacoma'
    Cost: 1990
    GHG: 3765000

EDU >> vehicleComparison(out)
ans =
The Toyota Tacoma costs $570 per year less but emits 2395000g CO2e per
year more than the BMW i3.
```

What to hand-in

You will submit a .zip file containing the following `.m` files for this assignment:

```
1) mySmartMultiply.m
2) classifyFlow.m
3) collision.m
4) vehicleRecommendation.m
5) vehicleComparison.m
```

Don't forget that the function headers you use must appear *exactly* as they do in this problem set. So, make sure the variables have the right capitalization, the functions have the correct name, and the inputs and outputs are in the correct order.