ECE 275 - Project 3

D2L: Project 3 (alpha)
Due: 10 March 2016, 11:59 PM
D2L: Project 3
Due: 24 March 2016, 11:59 PM

Please review the Project Companion for general instructions regarding submission instructions and formatting of submission materials. The requirements for the alpha submission are detailed in the final section of the project description.

Notes and Revisions:

Usage

```
Usage: catcar controlInputs stateOutputs
```

Requirements Summary

Create a C++ program that reads control inputs for a self-driving car from an input file. The program will simulate the car's state over the time horizon given by the control inputs, and write the results to an output file. The major problem is: the input file was written in such a way that the control inputs are stored out of order!

Assignment Name

The assignment name for this assignment is: `catcar`

---

1. Kinematic Model Simulation

In this project, you will create a program that controls an autonomous vehicle from an input control file. The only problem is that the input controls are jumbled, since a function that operated on the linked list that stored them was badly designed: they were stored in order in a queue, but the queue was designed by a Visual Basic programmer who doesn't understand memory. They printed the nodes out by order of memory address of the list nodes, instead of by the node order prescribed by the next pointers.

So you need to put them back in the right order before proceeding. This program will read in time value at which this command should be executed, then the commanded speed, and tire angle. The duration of the command is the difference between this command's time value, and the next time value (when the nodes are in order), at a maximum of 201 ms. If there are any durations longer than 201 ms, the input file should be declared invalid, and no simulation should take place.

With this sequence of commands, you should save the vehicle's state at each time step. During the execution of control commands, this program will record the vehicle's state as it moves (or stands still), and save the output states to a file.

2. File Format

2.1 controlInputs

For this assignment, the input text file will consist of control information, each line contains sample time (in seconds), commanded vehicle velocity (m/s), and tire angle rate (radians/s), with elements separated by whitespace (even the first number could be padded by a whitespace in the front). All values should be stored as doubles.

```
0.06 5 0.523
0.02 20 -0.523
0.07 8 0
0.00 0 0
```

In this example, at time 0.06 the vehicle should take in as its input command a velocity of 5, with a tire-angle rate of 0.523, for 0.01 seconds. The duration of 0.01 is determined by looking at the next time step (after the vector is confirmed to be sorted), and taking the difference (0.07 - 0.06).

Clearly the above example is an extreme case of driving aggressively. Your program should run regardless of the unlikely control inputs, as long as the below conditions are met:

- The rst element of the sorted list must be at exactly time 0.
- Commanded tire angle rate must be between [□-0.5236, 0.5236] radians/sec (i.e., $\pm\pi / 6$ radians/sec).
- Commanded velocity must be between [0, 30] m/s;
- Time values must be non-negative; and
- Duration between sorted input objects must be between [5, 201] ms.

If any of these are violated for any input value in the file, then the `controlInputs` file is deemed invalid, and an empty output file is written. If at any time a line fails to parse, then the file should be declared invalid. This policy is for safety: you wouldn't want to start controlling an autonomous car if you accidentally passed in the wrong file, and it started somehow parsing lines!

2.2 outputFile

This assignment should output a file with information on the state values of the vehicle throughout its journey. The format for this file is one called csv (comma separated values), and is common for inputting data into MATLAB. Each output entry is as follows:

```
t,x1,x2,x3,x4\n
```

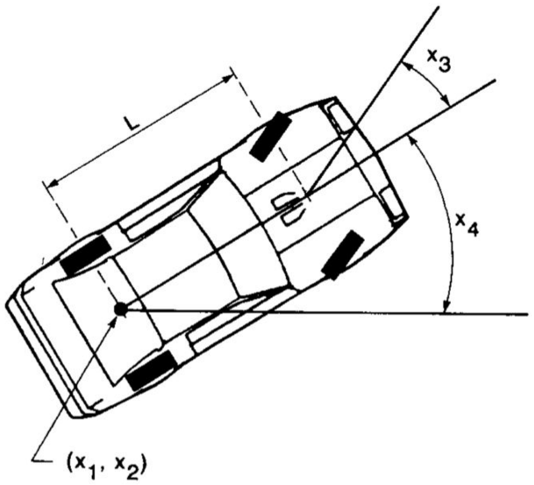
Where t is the time at which this state value was measured, and the values in x are given by the kinematic equation (1). The precision for each value should be whatever is the default when using C++ iostream methods.

If the control inputs are determined to be invalid, you should create an empty file with the name provided from the command line.

3. Vehicle Model

You are developing a vehicle simulator, which describes the kinematic motion of a front-steered, two-wheel drive vehicle. The vehicle you are simulating is visually depicted below. The equations of motion for this vehicle are provided in (1).

$$
\begin{aligned}
x_1 &= u_1 \cos(x_3) \cos(x_4) \\
x_2 &= u_1 \cos(x_3) \sin(x_4) \\
x_3 &= u_2 \\
x_4 &= u_1 (1/L) \sin(x_3)
\end{aligned}
\tag{1}
$$



Where x_1 is translational forward motion, x_2 is translational left/right motion, x_3 is the tire angle, and x_4 is the heading of the vehicle. For control inputs, u_1 represents vehicle velocity, and u_2 represents angular rate of change for the tire angle. The wheelbase L is defined in the Vehicle.h header file.

Using a simple discretization of these differential equations with a duration *t*, we can use the following equations for motion:

$$
\begin{aligned}
x_1(t+\Delta_t) &= x_1(t) + \Delta_t\, u_1(t) \cos(x_3(t)) \cos(x_4(t)) \\
x_2(t+\Delta_t) &= x_2(t) + \Delta_t\, u_1(t) \cos(x_3(t)) \sin(x_4(t)) \\
x_3(t+\Delta_t) &= x_3(t) + \Delta_t\, u_2(t) \\
x_4(t+\Delta_t) &= x_4(t) + \Delta_t\, u_1(t)\, (1/L) \sin(x_3(t))
\end{aligned}
\tag{2}
$$

The value for x_3 (tire angle) must always be between [□-0.5236, 0.5236] radians/sec (i.e., $\pm\pi / 6$ radians/sec). If a value is commanded outside this range, then x_3 should saturate using the above range. Ex: If the tire angle rate is commanded to be the value 0.7156, the tire angle rate should equal the maximum value of 0.5236.

The heading should always be between $[0, 2\pi)$. If the heading is a negative value, the heading should be converted into the range $[0, 2\pi)$ by repeatedly adding 2π . Ex: If the heading is $-.5\pi$, the heading can be converted to $-0.5\pi+2\pi = 1.5\pi$.

Defined values are present for these ranges inside of `State.h`

4. Class designs

The following class definitions must be used for the indicated classes. You may (if you wish) create your own classes to do other tasks. The below classes are prescribed in whole (or in part):

> `Input`, `State`, `Vehicle` (must use exact prescribed class definitions)
> `DataSource`, `DataSink`, `Director` (are at your discretion, except for the sort method prescribed).

4.1 Classes that must use prescribed definitions

Please use the exact interfaces for Vehicle, State, and Input, or your alpha release may not compile. These are the only classes for which the design is fixed.

4.1.1 Input

The `Input` class holds the values for the u variables used in the kinematic model. Its interface is included on Piazza as a resource for this Project.

4.1.2 State

The `State` class is similar to Input, and its interface is included on Piazza as a resource for this Project. However, it does ensure that tire angle values and heading values stay within the designated ranges if the setters for the class are called.

4.2 Vehicle

The `Vehicle` class executes a control input for the designated duration. The interface is included on Piazza as a resource for this Project.

The `Vehicle` keeps its own state, receives a control input, and updates its state. The initial state value for the `Vehicle` is $x_1 = 0$, $x_2 = 0$, $x_3 = 0$, and $x_4 = 0$, e.g., (0, 0) position, tire angle of 0, and heading of 0. The `Vehicle` class assumes that any invalid input values have been removed, so it does not do any error checking. The class does not permit anyone to update its state, except by providing an `Input` object through the `stateUpdate` method prescribed in the header file.

4.3 Classes left largely to your design discretion

Anything you want to add to the interface for `DataSource`, `DataSink`, or `Director` is up to you.

The `DataSource` class keeps a vector of `Input` objects (control inputs), in order. Likewise, the `DataSink` class keeps a vector of `State` objects (state outputs), in order.

The `Director` is used to pass `Input` values along to the `Vehicle`.

Hint: The `bool` data type is defined in C++, so it won't work if you try to include the headers from previous projects. Use, instead, the builtin type from C++ for bool functionality.

5. Recommended Functional Decomposition

In C++, global functions are a sign of a bad design. Rather, you provide methods inside the scope of a class. If you use global functions for this assignment, you will receive significant design deductions. Just like in C programs, your main function should be small. If yours includes lots of logic, rather than depending on the methods of classes, you will receive significant design deductions.

5.1 Reading Control Inputs

Define a class method (not a global function) that reads all the control inputs from a provided filename. As you read them in, don't worry about how they're sorted. The inputs will be stored in a vector in the class. I think you can figure out which class this should be.

5.2 Sorting Control Inputs

Puts the control inputs in order of their timestamp. The interface for the sort function should be:

```
void sort ();        // performs qsort
```

If your algorithm does not perform quicksort, you will receive a deduction. Insertion and bubble sort are each unacceptable. You should write your own sort routine, not use the sort routines in STL or any other library. You are, of course, welcome to use other implementations to confirm that your qsort implementation is working, as part of your own tests. The inputs will be already be stored in a vector in the class. I think you can figure out which class this should be.

5.2.1 Validating Control Inputs

This method should be part of the `DataSource` class. The interface for this method should be:

```
// should be called only after the vector is sorted
// returns true if the vector in Input objects is valid
bool validate ();
```

The vector of `Input` objects is invalid if any of the criteria from Section 2.1 are discovered.

5.3 Writing State Outputs

Define a class method (not a global function) that writes a vector of `Output` objects to a file of the provided filename. I think you can figure out which class this should be.

5.4 Driving the Car

Once you have an (ordered) set of control inputs, you need a `Director` to coordinate the consumption of inputs, and save each output. The `Director` will take the next Input (stored and sorted by the `DataSource`) to the vehicle, fetch that `Output`, and store it in the output vector in the `DataSink`. When the last `Input` is consumed, a duration of 200 ms should be used. After simulating this `Input`, the simulation is ended. In order to do this, the `Director` will "contain" a `Vehicle` object, and decide how to interact with it.

Part of the assignment is coming up with and writing the pseudocode for this algorithm, in order to stage your design.

5.5 What goes in main, then?

In your main function, you should instantiate your classes, check your arguments and return the usage statement. Put all the logic and error handling for file I/O, initialization of the `Vehicle`, etc., in your methods as defined in your other files and in the order that makes sense. You want your main to be very simple, so that if you want to reuse your code, you can do so without a main function.

Generally, part of assignments going forward include coming up with and writing the pseudocode for this algorithm, in order that your design can be performed in stages. To bootstrap this process, we provide as an example how to sketch an algorithm is pseudocode below:

```
check usage
open input file, and output file
execute director
  while inputs
    execute input(s)
    save output(s)
write and close output file, input file
```

6. Alpha Submission

Your alpha assignment is to implement the Vehicle class methods, and turn in your `Vehicle.cpp` file to the dropbox called Project 3 (alpha) on D2L. We will use your file with our own main function and test files, to check the behavior of each of the `Vehicle` methods we prescribe. Use the same `vehicle.h` file given above and provided from the website, or be in danger of your alpha not compiling.

As a note, you will have to complete your implementation of `Input` and `State` in order to test your alpha on your own. Submit only your `Vehicle.cpp` -- *do not submit implementations for Input and Output*, we will use our own. We have already included `Input.h` and `State.h` in our `vehicle.h` interface definition, but you may need to include other headers for standard C++ functionality in your cpp file. We do not recommend including headers for other custom classes in your cpp file, since you will not be submitting those headers or their implementation.