UCI FSAE Lap Sim Documentation

By Matt McMurry

September 5, 2017

# Table of Contents

Page | Item

3 [Introduction](#_Introduction)

3 [General Description](#_General_Description)

4 [The Car Class](#_The_Car_Class)

4 [Using the Car Class](#_Using_the_Car)

6 [Methods](#_Methods)

6 [\_\_init\_\_](#_Car(name,_m,_P,)

7 [getDownforce](#_Car.getDownforce(velocity))

7 [getDrag](#_Car.getDrag(velocity))

7 [findCorneringSpeed](#_Car.findCorneringSpeed(radius))

7 [findCorneringTime](#_Car.findCorneringTime(radius,_theta)

7 [findStraightTime](#_Car.findStraightTime(v_i,_v_f,_leng)

7 [findBrakingDistance](#_Car.findBrakingDistance(v,_v_f))

7 [findLapTime](#_Car.findLapTime(track,_ax_bool=None)

8 [findDynamicTimes](#_Car.findDynamicTimes(p_bool))

8 [Other Methods](#_Other_Methods:)

8 [quadraticFormula](#_quadraticFormula(a,_b,_c))

8 [getVertex](#_getVertex(a,_b,_c))

8 [readTrackFile](#_Car.readTrackFile(path))

9 [Methodology/Derivations](#_Methodology)

9 [Cornering Speeds](#_Cornering_Speeds)

9 [Straight Time](#_Find_Straight_Time)

9 [Braking Distance](#_Braking_Distance)

# Introduction

One of the most important things for an FSAE team to have is a good lap simulator. This tool allows the team to use real data to inform their decisions in the design phase of the project. With a decent lap sim, a team can go from guessing “we need to lose weight because it will make us faster” to “we need to lose fifteen pounds because it is the most cost-effective way for us to gain thirty points in competition”.

The program can estimate lap times, acceleration times, and cornering speeds given some basic statistics about a car. These statistics can be easily tweaked to see what changes to the car will give the biggest performance gains.

Since UCI has never had any kind of lap sim, this program is the start of what will hopefully be a powerful tool in our design pipeline. Over time, it will grow from the simple point-mass model it is now into a much more accurate and useful simulation, and this documentation should help future teams develop and refine it.

# General Description

This simulation uses a very basic model of the car to estimate its performance. The car is essentially modeled as a point mass with constant power and constant traction that is power limited in straight lines (it just accelerates without concern for wheel-slip, only longitudinal forces) and traction limited in turns (it goes at the maximum limit of the tires through a turn). The program also assumes that all turns are of constant radius. These assumptions were made to make the program easy to develop and easy to add on to. Use of the program is generally just importing tracks, creating a Car object, and using Car.findDynamicTimes() to estimate dynamic event times.

# The Car Class

The Car class is used to create Car objects that can be used to estimate lap times, cornering times and speeds, and acceleration times. It has a variety of built in methods for accomplishing this. It uses metric units for all calculations (m/s, kg, W, etc).

## Using the Car Class

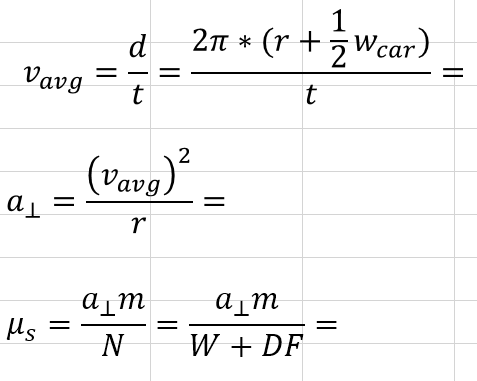
There are two ways to use the car class as a lap sim. You can edit the example file/make your own Python file where you import the Car class and use the built-in methods to do some testing, or you can you the Jupyter notebook version. They both work exactly the same, the Jupyter notebook version is just has a nicer user interface. To use the plain Python files, download all the .py files in the Drive folder, as well as the track map .txt files. In “Example lap sim testing.py” change the paths to the track map documents to match what they are on your computer. When you have set up the tests you want to run, go to the command line and run the “Example lap sim testing” file. For the Jupyter version, download the .ipynb file, the .txt track map files, and then go to <try.jupyter.org>. Upload the lap sim file, then open it. Change the track map paths so that they point to wherever you saved the track maps. Use Shift+Enter to make the selected code block execute.

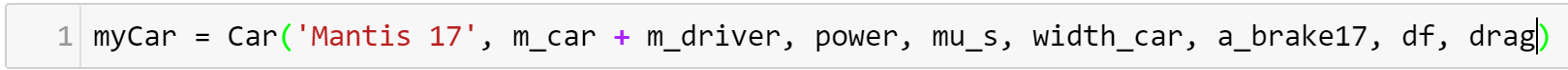
Setting up a Car object requires some trial and error to get the parameters of the car to values that returns accurate results. The mass of the car (in kilograms) and width (in meters) should be readily available. Include the mass of the driver in the car mass. The braking acceleration is just an average and can be found from track data, or possibly by timing how long it takes to stop from a specific speed. The rest of the variables require some more work to find.

After mass, the next parameter to find is the drag. First, you need data about how much drag your car has (in Newtons) at various speeds (in m/s). Then, a graph of drag vs. speed can be produced in Excel, and a quadratic regression can be used to find the quadratic equation that describes the graph. Now you have the constants needed to input into the model [A, B, C]. The program will find the vertex for you, or you can put in a different vertex (y,x) at the end of the list if you want the linear part of the model to behave differently. Repeat this process for downforce.

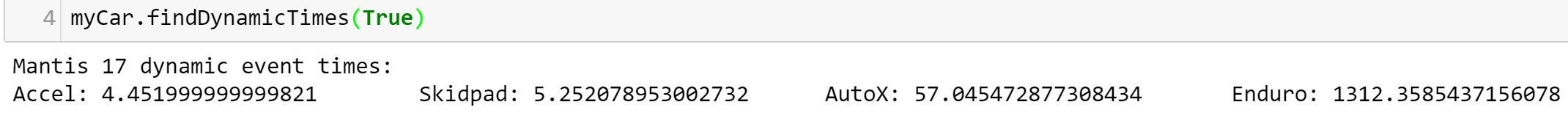
For power, do NOT use the manufacturer given power for the engine, as this is usually the maximum power output at a specific RPM. Instead, find an average power (in Watts) that makes myCar.findStraightTime(0, -1, 75) return your actual competition acceleration time.

To find the coefficient of friction, plug numbers into the following equations:



Now that you have all of the necessary information gathered create a Car object like so: 

Then, you can test how it performs in dynamic events:



Now you can make other Car objects with different characteristics and compare how they perform in the different events. In the Google Drive folder containing all of the lap sim files, there is one that already has two Car objects set up, one for the 2017 car and one for the 2017 car with wings and undertray.

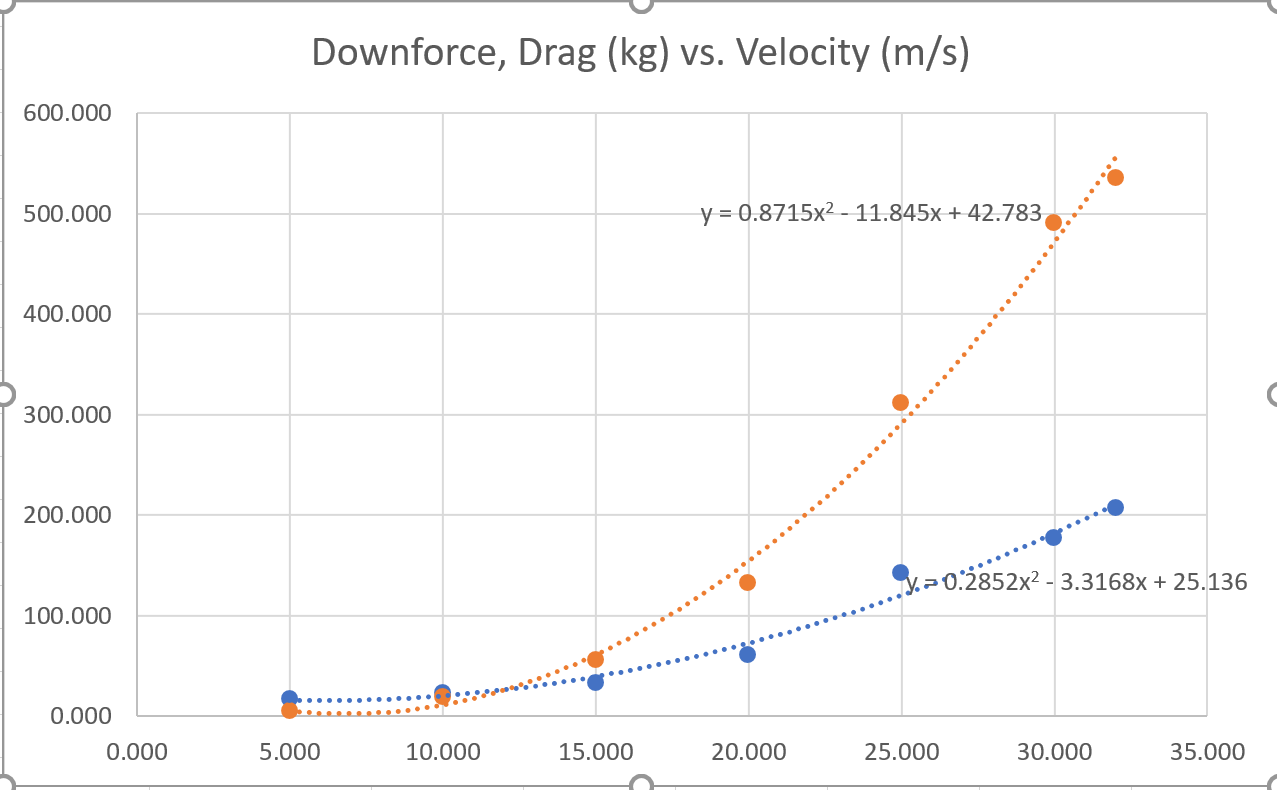
## Methods

### Car(name, m, P, mu, width, a\_brake, df, drag)

When instantiating a Car object, you must pass in a string name, float mass, power, coefficient of friction, and braking acceleration; and a list for the downforce constants and another list for the drag constants.

Mass should be in kilograms, power in Watts, and coefficient of friction is dimensionless, and braking acceleration in m/s2.

Because downforce and drag are related to speed by its square, downforce and drag are modeled using a quadratic, and a linear function to correct for errors at low speeds. The lists passed in for df and drag should be of the form [A, B, C, y, x] where A, B, and C come from Ax2 + Bx + C, and x and y are the coordinates of the vertex of the parabola. The linear portion is useful when the vertex of the parabola is not at zero speed. These constants can be easily found in Excel with wind tunnel data.



### Car.getDownforce(velocity)

Returns positive downforce in Newtons using the quadratic model of downforce if velocity >= vertex of the parabola, and linear otherwise.

### Car.getDrag(velocity)

Returns positive drag in Newtons using the quadratic model of drag if velocity >= vertex of the parabola, and linear otherwise.

### Car.findCorneringSpeed(radius)

Returns maximum cornering speed in meters per second for a car object around a corner of given radius.

### Car.findCorneringTime(radius, theta)

Returns time in seconds for Car object to go around a corner of given radius and angle.

### Car.findStraightTime(v\_i, v\_f, length)

Returns time in seconds for Car object to go down a straight (including brake zone) given initial velocity, velocity the car needs to be going at the end of the straight, and the length of the straight.

Uses Euler’s method for numerical integration to find the time it takes to go down the straight, including time spend braking. Velocity is calculated from

### Car.findBrakingDistance(v, v\_f)

Returns distance in meters that the Car object needs to slow down from v to v\_f. If v\_f is -1, returns 0.

### Car.findLapTime(track, ax\_bool=None)

Returns time in seconds for Car object to go around a track. If ax\_bool=True then the lap time is calculated as if it is an autocross track, i.e. you accelerate through the finish line.

### Car.findDynamicTimes(p\_bool, autox\_track, endurance\_track)

Returns a length 4 list of times to complete the four dynamic events in seconds. The list is ordered [acceleration, skidpad, autocross, endurance]. The variables autox\_track and endurance\_track must be set to their respective track lists. If p\_bool=True, the times are printed like this:

## Other Methods:

### quadraticFormula(a, b, c)

Returns a length 2 list containing the x-coordinates of the zeroes of the quadratic equation ax2 + bx + c = 0.

### getVertex(a, b, c)

Returns a length 2 list containing the (x,y) coordinates of the vertex of ax2 + bx + c = 0

### readTrackFile(path)

Returns a track list given the path to a file containing track data. The file must have the components of each turn tab separated, and each turn on a new line.

# Methodology/Derivations

## Cornering Speeds

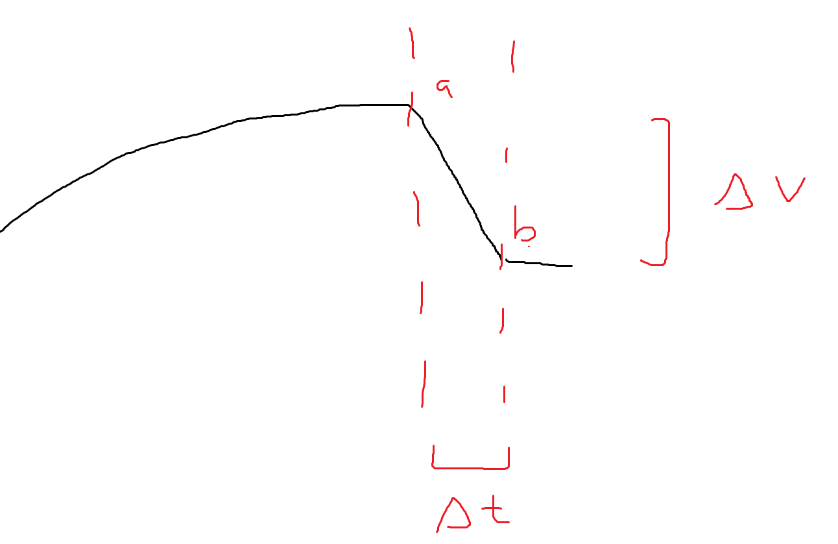
Cornering speeds are found by solving for the speed where cornering force is equal to centripetal force. This equation can be reduced to a quadratic, which is solved using the quadratic formula. Because of the way downforce is modeled, cornering force is a piecewise equation, so there are two quadratics to use depending on the speed of the car. The maximum of the two zeroes returned from the quadratic formula is selected as the cornering speed of the car.

## Straight Time

The time it takes to accelerate down a straight and brake for the next corner is found using Car.findStraightTime(). This method uses Euler’s Method for numerical integration to find the time it takes to go a certain distance. Each timestep, the engine increases the amount of kinetic energy the car has by Power \* Δt, and the drag decreases the energy by Fdrag\*Δdistance (the work done by drag). The velocity is then computed using the equation . The car continues to accelerate until the braking distance is greater than or equal to the distance remaining, at which point it decelerates at a constant rate until it has gone the length of the straight.

## Braking Distance

An equation for finding braking distance was derived by geometrically integrating the car’s velocity curve under braking.

First, assume that the car decelerates at a constant rate *abrake* while braking, and let the velocity curve be called *v*, which is a function of time. This means that while braking, *v* is a straight line with constant downward slope of *abrake* from one velocity to another. So, *v* can be integrated between a and b to find the displacement, s.

The integral of *v* from a to b is just the area of the triangle it forms plus the rectangle from a to b and up to *v(b)*.

Because Δt is not easy to calculate, we will replace it with Δv÷abrake

Which can be reduced to:

speed where cornering force =