|  |
| --- |
| Queen’s Formula SAE Design and Race Team |
| Point-Mass Lap Time Simulation |
| Rev B |

|  |
| --- |
| QFSAE  3-29-2022 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision Level** | **Revised By** | **Date** | **Comments** |
| A | Maurice Nayman | 28-02-2022 | Initial Release |
| B | Maurice Nayman | 29-03-2022 | Added a note in the introduction regarding MATLAB dependencies. Updated to include removal of global variables. |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Table of Contents

[1 Introduction 1](#_Toc99481867)

[1.1 Lap Simulation Assumptions 1](#_Toc99481868)

[2 QueensRacingVEHICLE 1](#_Toc99481869)

[2.1 Excel Config File 1](#_Toc99481870)

[2.1.1 Reading Excel Data into MATLAB. 2](#_Toc99481871)

[2.2 Tire Modelling 2](#_Toc99481872)

[2.2.1 lateralForce.m 3](#_Toc99481873)

[2.2.2 calculateAccelerationAndMoment.m 3](#_Toc99481874)

[2.2.3 calculateLateralAccelInterp.m 4](#_Toc99481875)

[2.2.4 extractYMD.m 5](#_Toc99481876)

[2.2.5 calculateSteadyMu.m 6](#_Toc99481877)

[2.2.6 Additional Recommendations 7](#_Toc99481878)

[2.3 Generating the Rest of the Car 7](#_Toc99481879)

[3 QueensRacingTRACK 8](#_Toc99481880)

[3.1 Generating a Track Model in Excel 9](#_Toc99481881)

[3.2 Running QueensRacingTRACK 10](#_Toc99481882)

[4 QueensRacingLAP 11](#_Toc99481883)

[4.1 How QueensRacingLAP Works 12](#_Toc99481884)

[5 Running Sensitivity Analyses 14](#_Toc99481885)

[5.1 General Sensitivity Analysis Pseudocode 15](#_Toc99481886)

[5.2 Sensitivity Analyses Using Yaw Moment Diagrams 15](#_Toc99481887)

[6 Conclusion 16](#_Toc99481888)

# Introduction

This document is mean to serve as an introduction and explanation for the underlying code and modules that comprise the QueensRacingLAP software. The code is split up into three modules, QueensRacingVEHICLE, QueensRacingTRACK, and QueensRacingLAP which are for generating vehicle models, track models, and lap simulation models, respectively. The majority of this work has been derived from the open-source lap time simulation code OpenLAP. Additional information on the OpenLAP code can be found on YouTube [here](https://www.youtube.com/watch?v=WWaouT6EhJ0&list=PLQiPsAzoaLqUtdVImOM4WjzoYrNUy7fve&ab_channel=MichaelChalkiopoulos), with the official GitHub page [here](https://github.com/mc12027/OpenLAP-Lap-Time-Simulator). These links should serve as additional explanation as well as video-based introductions into lap time simulation in MATLAB using a point mass approach.

Additionally, this document will cover suggested areas for improvement within the current code. These improvements assume that the user is already comfortable in working with the lap sim modules and understand the underlying physics. Modifications should also be made by users with an understanding of MATLAB.

As a note, QueensRacingLAP requires the Phased Array System Toolbox, which requires the Signal Processing and DSP Systems toolboxes. If these are not installed, it won’t cause the script to fail. MATLAB will create a warning and prompt you to install all needed toolboxes.

## Lap Simulation Assumptions

This lap sim assumes a steady state approach for calculating lap times. This means all transient effects are ignored, and the lap is not calculated using a time-stepping approach. Additionally, the vehicle is modelled as a point mass, which negates the effects of load transfer, finite track widths, etc. Additionally, a constant friction coefficient is assumed. This is to ensure that the simulation does not become dependent on vehicle speed to determine the friction coefficient. To improve the modelling of the tires, various efforts have been taken to better approximate an averaged friction coefficient using tire data, as well as load transfer, and aerodynamic effects. These will be detailed in later sections.

# QueensRacingVEHICLE

The most important part of the lap sim code is generating a vehicle model. In the case of our lap sim, all the quantities that are required to simulate lap times are specified within an Excel workbook. This infrastructure was ported directly from OpenVEHICLE. The main difference between QueensRacingVEHICLE and OpenVEHICLE is the way the tire model is built.

## Excel Config File

The config file for each simulation has two sheets. The first is focused on general vehicle information, and the second is a torque vs. speed curve for the engine/motor(s). Beside each parameter is the noted unit as well as recommendations when choosing certain values/sign conventions that must be maintained. New parameters can be added into the model, but the gear ratios must always be at the end of the sheet. When parameters are read into MATLAB, the last filled rows of column C are used to determine the number of gears along with their gear ratios.

### Reading Excel Data into MATLAB.

There are two functions used to read in the config file, being readInfo.m and readTorqueCurve.m. Both files have the same code structure, with the difference being the titles that are assigned to the data. The code effectively then uses the readtable function within MATLAB with the “UseExcel” option to read in the data in columns B and C for the car information and columns A and B for the torque curve.

This data is then processed in MATLAB using the function readCarFromExcel. The table2array command is used to convert the read data into usable variables. Any additional unit conversions that need to be made are performed as well. For example, the wheelbase is prescribed in millimetres within the config file, so it will be divided by 1000 to convert it to metres. Since the Excel data needs to be iterated over, an iterator variable “i” is used to loop over the data. After each variable is read in, the incrementor must be updated. As such, when adding variables to the config file, it is important that the corresponding variable is read into the MATLAB script in the correct order from the Excel file. If this is not done, variables will not be assigned correctly. A sample of how this data would be assigned to variables is shown below.

Text, letter

Description automatically generated

Figure 1 - Code used to read in parameters from Excel along with their units

If a parameter needed to be added in between the wheelbase and front track, it should first be added in the Excel file, then the corresponding data read line as shown above should be added in between the parameters L and FT in the readCarFromExcel function.

## Tire Modelling

This portion of the lap sim is where the majority of modifications have been made. This has allowed us to incorporate TTC tire data into the simulations. This section will be broken down into the various functions that are called to calculate friction coefficients, starting from the ground up. It will also explain how each function is referenced by each other. In all the tire modelling functions, the car MATLAB structure is used to prevent the need for global variables. This is to avoid issues when updating the Excel file if a sensitivity analysis for yaw moment diagrams is run in parallel. This means that vehicle parameters can be updated in the parfor loop by overwriting the vehicle parameter directly.

### lateralForce.m

This is the most basic function in vehicle modelling, which is used to calculate the friction coefficient and lateral force for a given tire, with inputs of normal force, inclination angle, tire pressure, and slip angle. This code makes use of the Pacejka Magic Formula model for modelling tires. It uses processed and normalized tire data and interpolates the various Magic Formula coefficients using three-dimensional interpolation using the inclination angle, normal force, and tire pressure. These interpolated coefficients are then used within the Magic Formula to calculate the friction coefficient for the tire, and the resultant lateral force generated by that tire. A relative grip factor is used to then scale the friction coefficient and lateral force as tire test data tends to overpredict the friction coefficient due to higher friction between the belt and the tire than is realistic. In future, this value should be correlated to various testing data, such as skidpad models to determine a better value than the current estimate of 0.75.

### calculateAccelerationAndMoment.m

This script is used to calculate the lateral acceleration, yaw moment, and average friction coefficient for all four tires for a given lateral acceleration in, tangential velocity, steer angle, and body slip angle. It should be noted that the steer angle and body slip should be prescribed using radians. Later functions that rely on calculateAccelerationAndMoment take this into account.

The first step is to calculate the yaw velocity, centripetal velocity and total velocity magnitude from the prescribed lateral acceleration, tangential velocity, and body slip angle. The static weight on the front and rear tires is calculated using the total mass, centre of gravity location, lift coefficient, and centre of pressure information. The weight transfer on each axle is then calculated using the prescribed lateral acceleration, as well as the track widths, moment arm length, front, and rear roll stiffnesses, and wheelbase. A final weight for each tire is then calculated using this load transfer. The slip angle at each tire is then calculated individually. Currently, there is no calculation for camber change. **This is something that would be beneficial to add in the future. It will impact how the lateral force is calculated for a given tire and may lead to more information regarding optimal tire setup.** Finally, the lateral force and friction coefficient are calculated using lateralForce.m for each tire to determine the lateral acceleration, yaw moment, and average friction coefficient. In the code, the front right and rear right tires’ lateral force and friction coefficient values are made negative due to sign conventions and because the tire data is given for a single tire, it must be mirrored. For the same reason, the calculated slip angles on the front left and rear left tires are made negative.

It should be noted that it most cases, the calculated lateral acceleration and the input acceleration will not be equal. As such, this function is used over a range of input accelerations, to find the output acceleration such that the output equals the input. This function will be described next.

Another note is regarding the use of elementwise operations in this function. In MATLAB, it is far more efficient to perform math operations over entire arrays than it is to use a for loop to iterate over the elements of two arrays while performing math operations on them. For example, if you have two, n element arrays where you want a third that gives you the product of each “ith” element of the array, the following code is more efficient array3 = array1 .\* array2 instead of the following:

for i = 1:length(array1)

array3(i) = array1(i) \* array2(i);

end

The dot in front of the multiplication operator means to perform the multiplication elementwise. This dot operator can be applied on the multiplication, division, and exponent operators. Addition and subtraction operators are always elementwise, so the dot operator shouldn’t be used on it.

With this information, calculateAccelerationAndMoment is set up to allow for an input array of lateral accelerations, using elementwise operations in any steps that involve arrays (when in doubt, you can add to dot multiplier just in case). This means that this function can be called for many input accelerations (currently tested to 800), and still perform faster than a for loop with only 13 iterations, performing the same calculation. The current tested speed up is about a six times improvement. As such, this structure in the function should be maintained to provide the same level of computational efficiency.

### calculateLateralAccelInterp.m

This code is used to determine a converged lateral acceleration, yaw moment, and average friction coefficient, for a given tangential velocity, steer angle, and body slip angle.

In this function, a linearly spaced vector of input accelerations is initialized from a parameter -aMax to aMax, with 400 values. Currently, aMax is set to 40 m/s2, and based on internal testing, this should be maintained. This array is then plugged into calculateAccelerationAndMoment for the prescribed input parameters, with the input acceleration for each element subtracted from it. The resulting array provides the error between the output acceleration and the input acceleration. This function is what we need to find the zero of to determine the actual lateral acceleration for a given tangential velocity, steer angle, and body slip angle.

With the error vector calculated, a quick check is performed to ensure there is a sign change in this array, which would indicate that the resultant data when interpolated would pass through zero. This then results in an if statement with two options.

The first option is if there are no sign changes. This means that an interpolated function would not cross zero, meaning there is no input acceleration that will result in the exact same output acceleration. When this case occurs, it happens such that the minimum error in acceleration out – acceleration in is on the order of less than 0.2 m/s2. As such, in this case the output acceleration resulting in the minimum absolute error to the input acceleration is used as the actual lateral acceleration for the input velocity, steer angle, and body slip angle. This would then be the converged value which is given to the user.

In the second case where the data does have a sign change, the following steps are taken. First, the point at which the absolute error function is minimized is found. Then, a function handle is set up using a shape preserving (pchip) interpolation scheme where the x values are the input acceleration, and the y values are the resultant error. The fzero function (a nonlinear root finding algorithm) is then used on this interpolated function to determine the input lateral acceleration at which the error is zero. This resulting input lateral acceleration is then used to determine the converged yaw moment and friction coefficient.

This code has been tested with a number of root finding algorithms for computational efficiency, robustness, and accuracy. Since this function uses so many input accelerations, the error in an interpolated root is very low given the benefits of a shape preserving function, and the resolution of the data. Additionally, there are very few cases where the function does not have a sign change, only occurring for very high speed, high angle maneuvers, which tend not to generate the peak lateral acceleration anyways. As such, the slightly higher error at these points is deemed more acceptable. This algorithm is also incredibly robust as it will always find a solution to within 0.2 m/s2 (from testing).

### extractYMD.m

This function is used to extract parameters useful to a yaw moment diagram analysis. As such, its use can extend past the lap sim, but its function will be defined here. Inputs to this function are a tangential velocity, steer angle, body slip angle, and a resolution for each of the steer and body slip vectors. Definition of the steer and body slip ranges can be done one of two ways. A maximum angle and resolution for each can be defined and the steer/body slip vectors will be built as a linear spaced vector from the negative max angle to the max angle, with the specified number of points. Alternatively, the steer/body slip angles can be given as pre-defined vectors, at which point the resolution parameters are ignored. This method can be useful for cases where an asymmetric angle range is necessary.

The code then will check your tangential velocity to ensure it is at least 5 m/s. This is done as values smaller than this result in yaw moment diagrams of negligible meaning. The body slip and steer angle vectors are then converted to radians to maintain units in each subsequent function calling. A nested for loop is used to call the calculateLateralAccelInterp function for each body slip and steer angle combination to create a two-dimensional matrix of lateral accelerations, yaw moments, and friction coefficients. Currently, the code is set up to return these two-dimensional matrices, as well as the maximum lateral acceleration calculated, along with the yaw moment and averaged friction coefficient at this point.

Current areas to develop this function include the calculation of stability and control derivatives based on the yaw moment diagram solutions. This improvement can be made by writing a function that accepts the lateral acceleration and yaw moment values to calculate these derivatives, or the additional code to generate the derivatives could be written into the extractYMD function (recommended).

### calculateSteadyMu.m

This function is most important to the lap sim in terms of generating a constant friction coefficient based on the vehicle configuration and tire data. It returns a constant friction coefficient, using a tangential velocity vector, weight velocity, maximum steer and body slip angles in degrees, and a step size between body slip and steer angles, in degrees. In this code, a steady-state friction coefficient is designated as the point at which the yaw moment is zero, such that the friction coefficient is maximized.

A simple overview of the algorithm is as follows. Weights are assigned from the input velocity vector and weighting velocity such that the weight for a given speed is . For this function, the weighting is maximized when the speed is at the weighting speed and minimized as the speed moves away from the weighting speed. This weighting speed should be determined based on the rule’s specification of desired averaged speeds for an event. This is typically around 50 km/h but can be referenced in the rules. Based on initial testing, a value of 16 m/s (57.6 km/h) works nicely.

Next, a yaw moment diagram is extracted for each speed. Since yaw moment diagrams are symmetric, it is only extracted for a symmetric body slip range defined from the negative max body slip to the max body slip with a resolution such that the spacing between points equals the step size, and it uses an asymmetric steer angle range from zero to the max steer angle. This helps to reduce the computational load for each iteration. Next, each column will be examined to determine if there are any sign changes in the yaw moment, signifying that there is a point in this column where if interpolated, the yaw moment would be zero. If this criterion is not met, that column is skipped and the next one is examined. If there is a sign change, a shape preserving interpolator (pchip) is applied to the column, where the x axis is the lateral acceleration and the y axis is the yaw moment. This allows the fzero function in MATLAB to find a zero. The starting point for the fzero search is then defined as the point where there is a sign change, such that the difference between the two values resulting in a sign change is minimized. This step ensures the search is done efficiently. Once the yaw moment and acceleration are interpolated to find the acceleration such that the yaw moment is zero, this acceleration is used to interpolate the friction coefficient as a function of the lateral acceleration for that interpolated acceleration. This is then the steady-state friction coefficient for this column. The maximum steady-state friction coefficient at each speed is then used to populate the vector of friction coefficients as a function of speed. Finally, the weights previously calculated are used to compute a weighted averaged of the friction coefficient, which will be the final returned value from the formula.

There are several computational speedup tricks employed to make this code run more efficiently, which will be detailed here. The first trick is the steady-state coefficient is calculated for the first velocity, before a while loop is used to calculate the remainder of the velocities. This is done because of the step taken to ensure the velocity vector has a minimum value of five. As such, there can be elements with the same velocity value, meaning there would be a redundant yaw moment diagram calculated. This allows a prior calculated value to be used to reduce the number of computations necessary. The second trick is the steer angle window used to evaluate the yaw moment diagram. The first yaw moment diagram run will determine the steer angle for which the friction coefficient is maximized. From this steer angle, the difference from the bounds of the input steer angle vector, from which the minimum difference is extracted, e.g. if the maximum steer angle is 10° and the bounds are 0° and 14°, the difference will be 4°. This extracted difference is then taken to be the maximum of this difference and two times the step size, to ensure that the steer angle vector is at least five elements long. This ensures a sufficient region of the calculation space is found. The new steer angle vector is then checked to make sure that its bounds do not exceed the maximum angle specified.

In the case where there are no sign changes calculated for all the columns of the yaw moment diagram, this speed calculation is redone, using the entire steer angle regime. This ensures that the steer angle domain is expanded such that a yaw moment of zero can be effectively interpolated.

### Additional Recommendations

Currently, the tire coefficient generation is solely for the lateral tire coefficients. Within QueensRacingLAP there is an allowance to define a longitudinal tire coefficient, which is currently equal to the lateral tire coefficient. It would be beneficial to generate a function that makes use of longitudinal tire data to calculate the tire coefficient as a function of speed based on slip ratios.

If there are any changes that are required to improve the yaw moment diagram code, it is recommended to add them to extractYMD instead of writing a new code. Since extractYMD returns a MATLAB structure, it is very easy to add elements to it that can be later retrieved. This function can then be used to generate sensitivity analyses based on yaw moment diagrams for separate vehicle dynamics developments. A model called QueensRacingYMD has been written that operates similarly to QueensRacingVEHICLE, it just only runs a yaw moment diagram. This can also be used for sensitivity analyses as outlined in Running Sensitivity Analyses section.

## Generating the Rest of the Car

With tire modelling complete in QueensRacingVEHICLE, the final step is to compute the longitudinal force the engine can generate. This is determined based on the specified torque curve and gearing/gear ratios. This also gives the code the maximum speed of the car, which will be used to limit the maximum allowable speed in the lap sim. Additionally, a matrix for the steering model will be generated. This isn’t required in the lap sim, but it’s used so the lap sim can be post processed to show steering angle over the course of a lap. All of these vehicle parameters are then saved to a .mat data file so it can be reloaded at a later time if necessary. This .mat file is then loaded into the veh structure, which is a structured dataset containing all the information the lap sim will need based on the vehicle model. QueensRacingVEHICLE will then output some plots of the car’s performance capabilities as well as some log files for further review if necessary. This is the final step in generating a vehicle model. A sample of an image summarizing a vehicle model is shown below.

Chart

Description automatically generated

Figure - Vehicle model generated based on parameters roughly characteristic of the Q22 car

To run QueensRacingVEHICLE, the function needs to be called with an input value of the excel filename that will be read to generate the car. This means every time it’s called, you can make modifications to the excel workbook using MATLAB, with those changes being propagated to the car model. This functionality is used for sensitivity analyses in the lapsim.

# QueensRacingTRACK

This function is used to generate a track model based on an Excel workbook defining a track made up of segments of straights as well as left and right turns. This code has not been extensively modified based on the OpenTRACK implementation as it is designed to integrate natively with OpenLAP/QueensRacingLAP.

## Generating a Track Model in Excel

Past track maps can typically be found over the internet, specifically in r/FSAE. These tend to be image files, and not direct data that can then be put into QueensRacingTRACK. As such, these images need to be loaded into Solidworks (or any CAD system of preference), and have the track traced on top of them using lines and arcs. Splines shouldn’t be used as they do not have a constant radius, meaning they won’t be able to be defined correctly for QueensRacingTRACK. In Solidworks, images can be loaded using the Sketch Picture command (can be found by searching for it), and in NX, using the Raster Image command will allow you to load an image into a part model. When creating the sketch of the track, it isn’t necessary to ensure the image of the track is to scale, so long as you have a reference dimension on the track. For example, some tracks will have grid lines on the image and a scale shown, some will not.

When drawing lines and curves to replicate the track, it is important to ensure all lines/arcs are tangent to each other. This is because the track should have a single, continuous path in its first derivative. This makes for a cleaner track map, that’s easier to solve. In the case of Solidworks, this can be ensured by using the tangent arc command. A sample sketch used to approximate a track map is shown below.

Chart, line chart

Description automatically generated

Figure 3 - The red line represents the track in the image, with the black path being the traced track

As a note, the track you trace will not be perfect, and that’s ok. Try to get the track as close as possible. The lap sim is meant to be a comparative tool, and not to predict our actual lap time on a lap, especially since track layouts change year after year.

Once your track path is defined, measurements should be taken of each section and placed into an Excel file, where the first column is whether it’s a straight, left, or right section. The second column is the length of that section, and the third column is the radius of curvature. In the case of straights, set the radius of curvature to 0. When setting up these points, leave the values unscaled, as a formula can be used at the end to scale the track segments from the measurements in CAD to metres based on the scaling factor in the image. These final segments should then be placed in a track template Excel file. Templates can be downloaded from the OpenLAP repository if necessary, with a current track available in the team’s VD Git Hub. The info sheet can be updated to reflect the parameters of the track. The shape sheet will be where you place all the scaled track data generated. This is the most important part to getting a track right. The elevation sheet will show any changes in elevation. For the sake of FSAE tracks, these can be left to zero, where you specify the start and end point of the track to both have zero elevation. The same can be applied to the banking sheet, unless the banking is explicitly known. These values are used to scale the normal load on the car in these sections based on the banking/inclination angles. The grip factor sheet should stay as one for the start and end points. This assumes that the track has consistent grip across the track, which is a sufficient assumption. In the sectors tab, the start of each of the three sectors is defined. This doesn’t affect FSAE tracks much, but it can be nice to break up tracks into three sectors to see where different car configurations perform better, and to understand where certain changes make the most difference. As a note, when specifying the end point for elevation, banking, etc. you must make sure the distance is not equal to the total length of the circuit. As such, it is easiest to calculate the sum of all the segments, and subtracting 0.5 from it to ensure no elevation or banking points are accidentally deleted.

## Running QueensRacingTRACK

To run QueensRacingTRACK, the function must be called, with the input variable being the excel filename that contains the desired track information, as well as the type of data being used to generate the track. This will return a MATLAB structure that contains all the information regarding the track.

QueensRacingTRACK allows you to use either a specified Excel track as described or logged data to generate the track. Only using a user-defined track has been used to generate a track so far, so using logged data remains untested. The mode parameter lets you specify either ‘logged data’ or ‘shape data’ and the log\_mode parameter specifies how the data was taken if logged data is used either as ‘speed & yaw’ or ‘speed & latacc’. At the moment, ‘shape data’ should be used for the mode parameter, with either of the log\_mode settings, as it does not get used when using shape data. Additionally, the function has a hardcoded mesh size of 0.5 m for discretizing the track. This should be sufficient for our tracks based on testing.

The code works by reading in the shape data to arrays based on each sheet of the track definition. It converts straights, left, and right turns into number values so they can be easily read by the program. The code then inserts points for long corners to ensure that they will get discretized correctly during meshing. It determines the number of points required to discretize the mesh, then it runs a shape preserving interpolator on the existing shape, elevation, banking, and grip factor data to create discretized track model based on the specified step size. The X, Y, Z scatter plot of the track is then calculated by using the discretized step sizes and the radius of curvature at each of these segments to find the position of each discretized track point in 3-D space. Apexes for the track are found by converting corner radii into curvature and using the findpeaks function on the absolute value of the curvature vector. This will be used in the lap sim as the acceleration and deceleration point for each apex. The code will then ensure that the track map is closed if specified in the track excel sheet. A sample track map and data looks as follows:

Chart, line chart

Description automatically generated

Figure 4 - QueensRacingTRACK printout for 2019 FSAE Michigan AutoX track

# QueensRacingLAP

QueensRacingLAP is the lap sim function that takes in a vehicle and track structure as generated by QueensRacingVEHICLE and QueensRacingTRACK to create a simulation structure containing lap data for a variety of vehicle parameters over the track. This data can be accessed by calling the specific data point within the structure. Typically, the lap time will get extracted for comparing to other car configurations, using the command sim.laptime.data. Other parameters can be pulled out similarly, e.g. time in corners, time spent braking, etc. A sample of the plots shown to the user after performing a lap sim are shown below:

A screenshot of a computer

Description automatically generated with medium confidence

Figure 5 - Lap sim printout, showing accelerations on the GGV diagram, speed over lap distance, etc.

## How QueensRacingLAP Works

The lap sim is a point-mass, steady state model, so it doesn’t account for dynamic vehicle behaviour, and it doesn’t solve for a lap time by iterating through the course map. Based on the engine characteristics and grip available, the lap sim accelerates forwards and decelerates backwards from each apex. It then defines the speed over the lap as the lowest combined path due to accelerating and decelerating from each apex.

The first simulation step is solving for the velocity at each point on the track assuming a purely lateral condition, for each mesh point in the track. The lateral grip model then extracts the track parameters at the mesh point, as well as the vehicle mass, weight distribution, centre of pressure, and driven wheels. If the track mesh point is a straight, it assumes the vehicle’s speed is its top speed. If in a corner, the downforce and vehicle mass are used to solve for the velocity based on the centripetal acceleration of the car. This has been modified from OpenLAP as our tire model assumes a constant friction coefficient, not a linearly varying friction coefficient. Should this feature need to be reintegrated, refer to lines 699-715 of OpenLAP. The speed calculated is then taken to be the minimum of the calculated speed and the max speed of the car. An adjust speed loop is then performed to ensure all the forces balance out on the car, with the inclusion of drag and the engine force. First, the lateral acceleration needed is defined based on the current speed and turn radius, as well as any banking, as well as the maximum lateral acceleration possible based on the vehicle parameters. The longitudinal acceleration is then calculated based on the available power/braking limit of the car, with the power limit interpolated from the engine characteristics. This is used to recalculate the lateral acceleration based on a friction ellipse that factors in the required longitudinal acceleration divided by the maximum longitudinal acceleration. This lateral acceleration based on the friction ellipse is then compared to the lateral acceleration needed. If their ratio is less than 1, meaning the car isn’t generating sufficient grip, the velocity is then scaled down to the lateral acceleration that was calculated. This adjust speed loop is iterated upon until the calculated lateral acceleration is equal to or exceeds the needed lateral acceleration. This total loop is repeated at each mesh point, providing a solution for the speed in each corner.

The apexes are then found from this initial velocity solution by using findpeaks on the negative speed vector. This is done because the straights were set to the maximum speed of the car. As such, flipping these values solves for the corners where the speed is maximized. If the track is an open track (like an autocross track), it injects an apex at the start, and ensures the starting speed is zero, replicating a standing start. If the apex speed vector is empty, an apex is added at the slowest speed calculated during the cornering calculation. The code then sorts the apexes in ascending speed to improve the solver calculation. This step improves computation efficiency as it means the solver will accelerate and decelerate from the slowest apexes first, meaning the slowest path will be calculated first, reducing extraneous calculations. Using this reordered apex table, the car speed, accelerations, throttle and braking is initialized for the length of the track, for each apex, and for an acceleration sweep, and deceleration sweep (becomes a 3D matrix).

The lap sim then runs through each apex, for each mode (acceleration and deceleration). It then checks the apex to make sure a deceleration run isn’t necessary for a standing start in the case of an open track. The speed, acceleration, throttle, and braking are then allocated from the previous apex solution. Based on if the solver is in acceleration or deceleration mode, the speed at each point is assumed to be the same as the speed at its local apex. This is then used to solve for the speed at each point using the combined vehicle model function.

The combined vehicle model starts by assuming there’s no speed overshoot, and calculates the accelerations available and accelerations needed, based on the speed at the next track point. This is combined with the lateral acceleration at that point based on the speed and corner radius, along with driver inputs for throttle and braking based off the amount of acceleration or deceleration needed. The next speed is then recalculated based on the current speed plus the resultant longitudinal acceleration using kinematic equations. If the calculated next point’s speed is greater that the maximum next speed, the overshoot flag is triggered, meaning the solver will move on to the next point. Essentially, this overshoot flag tells the solver that the maximum acceleration or deceleration has been met, while being within the grip limits for the car, and meeting the lateral acceleration for the turn.

The solver will then check if this point has been solved for a different apex’s iteration. If it finds that the speed calculated is greater than the already calculated speed, it will move on to the next point. This ensures that redundant points in the track aren’t calculated if it’s already been found to have a slower path. This flag is then updated to tell the solver that this point has been solved for when other apex solutions are generated. The solver then checks if it has solved for the entire track. For a closed circuit, this is checked by seeing if the current point is at the current apex (you’ve gone all the way around), or if you’re at the start or end of an open circuit. This marks the completion of a lap time simulation, and the calculated speeds, acceleration and driver inputs can then be post-processed to generate additional information about the lap. This includes forces, yaw rates, steering inputs, engine metrics, etc. All of these KPI’s (key performance indicators) are then written to the simulation structure for further post processing if necessary, and the main lap data plotted, as shown above.

While this explanation isn’t exhaustive regarding the functionality of the lap sim, it should be sufficient in describing the overall workings of the code itself. Any changes to the model would likely come from improvements to the tire modelling methods. For example, if non-constant tire coefficients are used, it will be important to update the lateral models within the solver to allow for a more complex apex speed calculation. That being said, how the solver operates, and its computational efficiency steps shouldn’t be modified. It will be important to troubleshoot any modifications to the vehicle model in the lap sim to ensure proper results.

# Running Sensitivity Analyses

The goal of the lap sim is to use it for analysing the effects of different changes to the car. This will mainly be different aero setups, powertrain setups, and vehicle mass. Most VD characteristics are better suited for other methods such as yaw moment diagrams, quarter vehicle models, etc. While there is no one template that works for every model, there is a general workflow for generating a sensitivity analysis.

It is recommended to sweep only two parameters at a time if the data is to be plotted visually. This ensures that the data can be understood by a reader either through line plots or surface plots. Examples of these types of visualization are shown below:

Chart

Description automatically generated

Figure 6 - Lap times as a percentage of a baseline case, sweeping CL and aerodynamic efficiency

Chart, surface chart

Description automatically generated

Figure 7 - The same data, visualized using a surface plot

In both plots, a baseline trial was used to compare lap times, so they could be represented as a percent. This can be more useful, as the lap sim will be better suited to providing performance deltas, rather than absolute lap times.

## General Sensitivity Analysis Pseudocode

The general outline for generating a sensitivity analysis is as follows. The folder where all the functions lives should be added to your MATLAB path using the addpath function. Next, you should load in a track file if it already exists or generate a new track file that will be used for all simulations. Then a filename should be set up, such that the same excel workbook will be used for all models, with specific parameters changed using the xlswrite command. This will allow you to specify which parameters to change at each loop iteration, and where in the excel sheet to change them. Then plotting routines can be used to visualize the data as chosen. Additionally, sensitivity derivatives could be calculated using this data, and with more changing variables, to get an idea of what change makes the greatest difference in lap time. A sample sensitivity analysis template is made available in the team Git Hub as a starting point, which follows this outline.

## Sensitivity Analyses Using Yaw Moment Diagrams

If you are running sensitivity analyses using just yaw moment diagrams, there is no need to overwrite parameters in the excel file, then reread these parameters. An initial vehicle model can be generated using readCarFromExcel, then within each loop iteration of a changing parameter, that specific parameter can be updated. For example, if you are varying the front roll stiffness, the vehicle model can be read in once. Then on each iteration, you can say veh.kF = newValue, where newValue is what you want the front roll stiffness to be on each loop iteration. This will help save computational time, as calling the xlswrite command becomes very cumbersome.

Additionally, running sensitivities with the yaw moment diagram can make extensive use of the parallel computing toolbox in MATLAB. Each for loop for a given variable can be redefined to be a parfor loop, where each iteration can be spawned on multiple processes. As such, you can greatly speed up the total computations. As a note, you should ensure you spawn your parallel processes before running parallel scripts, as this can be a time consuming process.

# Conclusion

Having gone through this guide, you should now have a more thorough understanding of the models that make up QueensRacingLAP, as well as how to run it and use it. With this in mind, the lap sim is not perfect, and it’s meant to be a tool like all other modelling techniques. Be wary of results that you get, as the lap sim can be misleading. Don’t use it for things such as tire selection, damping settings, anything transient, etc. Should you have any further questions regarding the model, contact the person who performed the latest revision of this document. If you don’t know how to contact them, ask your section lead for further help.