MONASH MOTORSPORT

FINAL YEAR THESIS COLLECTION

Aero Map for Formula Student

Paul Hendy - 2019

The Final Year Thesis is a technical engineering assignment undertaken by students of Monash University. Monash Motorsport team members often choose to conduct this assignment in conjunction with the team.

The theses shared in the Monash Motorsport Final Year Thesis Collection are just some examples of those completed.

These theses have been the cornerstone for much of the team's success. We would like to thank those students that were not only part of the team while at university but also contributed to the team through their Final Year Thesis.

The purpose of the team releasing the Monash Motorsport Final Year Thesis Collection is to share knowledge and foster progress in the Formula Student and Formula-SAE community.

We ask that you please do not contact the authors or supervisors directly, instead for any related questions please email info@monashmotorsport.com

AERO MAP FOR FORMULA STUDENT

SUMMARY

Formula Student and Formula SAE are two design and build engineering competitions, based on an automobile racing environment (https://www.formulastudent.de/fsg/). Formula student teams in the early 2000s became aware that aerodynamics even at the low average speeds of a formula student circuit can have a positive effect on performance. In the beginning, teams focused on a front and rear wing concepts, although now many teams also use underbody aerodynamics.

This project aims to expand on Monash Motorsports (MMS) tools for analyzing aerodynamics effects on performance by examining how changes to the vehicles attitude effects the aerodynamic load and moments on the vehicle. These tools were designed analyze and predict the flow about the car as it moves around the track and changes attitude. The main tool developed was the aero map which aims to predict aerodynamic force and moments as a function of front and rear ride height, roll, steer and crosswind angle (yaw). Analysis tools in MATLAB and MoTeC were also discussed as they help draw the maximum value from the aero map.

The simulated flow was also validated with the Monash Wind Tunnel to determine the strengths and weakness of the CFD predictions. A differential pressure measurement system (DPMS) was developed as a tool to analyze aerodynamic pressures on-track to better understand how the vehicles attitude affects flow structures on the car. This helps increase confidence and understanding from the simulated values to further increase the aero maps effectiveness.

TABLE OF CONTENTS

1. INTRODUCTION

Formula Student and Formula SAE are two design and build engineering competitions based on an automobile racing environment (https://www.formulastudent.de/fsg/). Formula student teams in the early 2000s became aware that aerodynamics even at the low average speeds of a formula student circuit can have a positive effect on performance. In the beginning teams focused on a front and rear wing concepts although now many teams also use underbody aerodynamics.

The complexity of the aerodynamic designs by the MMS have progressively become more aggressive, complex and reliant on 'ground effect'. Ground effect is sensitive to ground clearance which changes as the race car moves around the track. MMS has yet to address the problems with predicting and understanding the effects of vehicle attitudes on aerodynamic performance and vice versa. The attitude of the vehicle in this report will not use the standard vehicle dynamic system of using heave, pitch, roll (angles between the chassis reference and ground reference) and steer instead the vehicle attitude will be characterized by FRH, RRH, roll, steer and yaw.

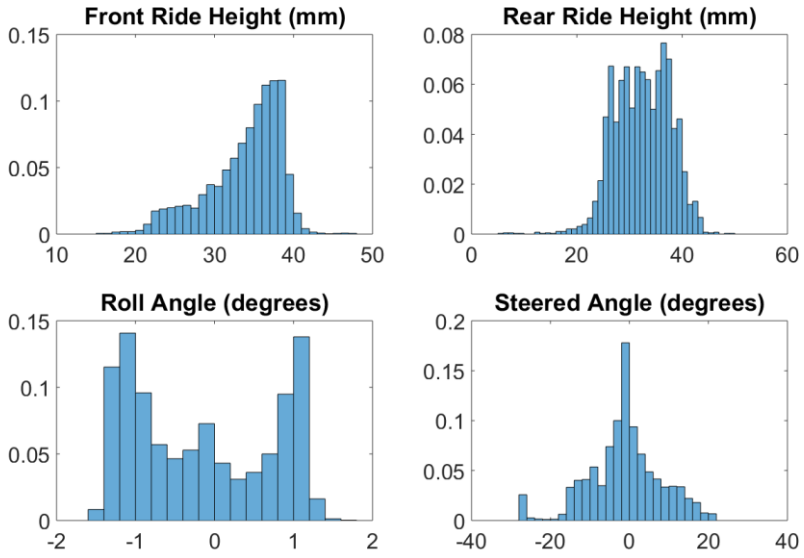


Figure 1 shows histograms of the vehicle attitudes from the fastest endurance lap of the 2018 Formula Student Germany Competition.

This yaw angle can be attributed to two primary causes, cross winds and vehicle slip which occurs when the vehicle is driven to the limit of grip and all 4 tires experience slip. While vehicle speed can influence the Reynolds number and hence the amount of turbulence in the flow. This report uses a standard of 16.67m/s vehicle speed as wind tunnel tests by MMS have shown that the vehicle is Reynolds independent.

An aerodynamic force map (aero map) describes the aerodynamic forces and moments of the vehicle across its feasible attitude parameters (FRH, RRH, roll, steer and yaw). It would be ideal to create a 5-dimensional matrix for each of the six components of the forces and moments however this would create a variety of challenges including the size of the matrix which in this case could have more than 6000 elements and hence require 6000 simulations. With each simulation taking 6 hours the total time for the simulations would be 36000 hours or approximately 4 years!

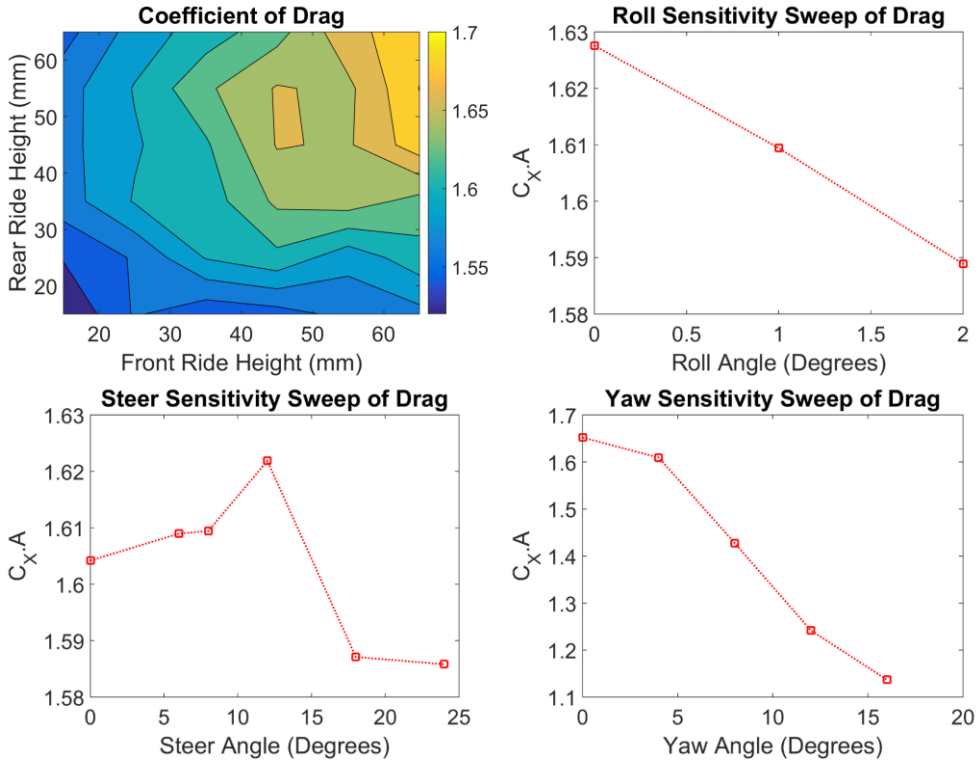


Figure 2 is a demonstration of the aero map for the X force component (ie drag) across the mapped parameters.

Hence this paper simplifies an aero map to a 6 by 6 ride height map (FRH v. RRH), 3 roll angles, 5 steering angles and 5 yaw angles. Each attitude in the map is a variation on a baseline which means that only 47 simulations are required to create the map which takes 3 days to simulate on the MonARCH (Monash Advanced Research Computing Hybrid) Cluster and whose results for the C_x values can be seen in Figure 2.

For the aero map not to become just an aesthetic plot but a design tool a variety of analysis tools must be created to allow engineers to use the data contained in the map to predict the effect of setup changes and determine where to focus their design efforts. By automating the map process this allows designers to focus on analysis and on developing tools for validating the simulations with on-track and wind tunnel data and use the on-track data for tuning and vehicle development.

2. SIMULATION SETUP

A complete rework of the simulation methods used by MMS aerodynamics team between 2008 and 2017 was needed to make it practical to complete simulations at various attitudes and batch process across a map of attitudes. The old method involved the use of ANSYS work bench and ANSYS CFX which posed a variety of issues for automation namely that the process could not be automated. The first issue is that the geometry – including the fluid domain – is produced in the team's CAD software (Siemens NX 12.0) which means that either the geometries needed to be created by the CAD software – which didn't have an elegant solution – or the simulation script needed to create the domain which was very difficult if not impossible with ANSYS workbench.

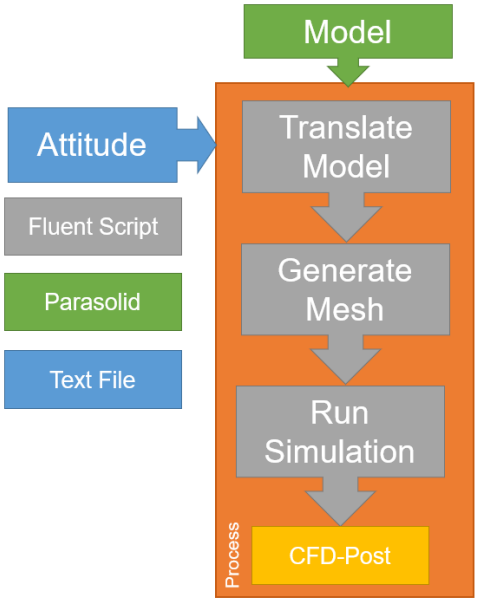


Figure 3 is a representation of the key steps to complete a scripted Fluent simulation.

This was the first factor which pushed for a change in simulation package the next factors were that CFX didn't allow for scripting and that ANSYS had ceased to support CFX preferring to support the industry favorite of Fluent which they also produced. This meant that the team members using the new simulation setup would have skills better aligned to the software that they could use in industry.

Fluent's popularity in industry comes from two primary factors; it has a simple and usable scripting language and that both the mesh generator and solver are operated within the same GUI which would allow for one script – actually three – to complete an entire simulation.

2.1 CAD Layout and Geometry Export

To make the simulation process independent of which CAD package the team chooses to use and because the cluster is unable to recognize NX CAD parts each of the components has to be exported as its own individual Parasolid – a generic CAD part type – which not only helps separate the wheels from the rest of the components but also helps separate faces into their own name selection so that they can be identified in post processing. It is expected that some faces will intersect with each other and that some components might be almost entirely contained by other components however some care should be taken to avoid; faces which are perfectly overlapping, angles more acute that 20 degrees and components with clearance below 10mm – in which case the components should just be made to overlap. Further the suspension should be made to overlap so much with the wheels so that none of the above issues would appear at extreme attitudes as the suspension moves while the wheels stay stationary or turn.

In Figure 4 while it is not clear from the images each wheel is not only is own separate part but also the wheels bends are separate parts with nearly identical names. This is done because down the line the wheels will have a different boundary condition to the blends however in all other situations they should be treated as a single unit.

Each components Parasolid file name must follow a predefined pattern because the simulation setup relies heavily on wildcards to maximize the flexibility of the process for the number of components

present. Fluent will only import Parasolid files with the prefix “cfd-“ which means that every components file name needs to start with “cfd-“ for it to be included in the simulation hence why all the names in Figure 5 have the same prefix.

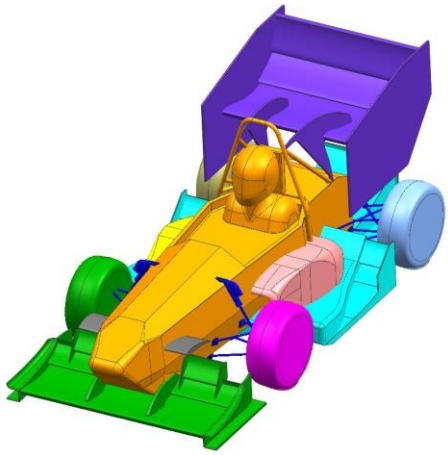


Figure 4 shows the individual Parasolids imported into Fluent which will describe the boundary names in the force and moment summaries.

The next part of the name can take one of two paths, either it can have a required name such as “default”, “wheel” or “suspension” or it can be identified as “wing”. All elements which do not follow this pattern will be lumped in with “default” and meshed with a low-resolution inflation layer and large element sizes. The “suspension” group is used because the default face resolution is not adequate to capture the suspension tube sizes and hence the extra name only means that the suspension gets a slightly finer resolution mesh to capture the tubes. The “wheel” group is used to separate all the wheels with their appropriate blends so that they can be excluded from the sprung elements when they have their attitude applied and have the steering applied to the fronts separately.

Figure 5 shows the files and their file name formatting when completing a simulation

The rest of the elements – radiators are currently treated as default – are classified as “wing” even if they are a bodywork element. The grouping “wing” just identifies components which need fine surface mesh resolution and a large number of inflation layers. All the elements will keep their name, except for the prefix “cfd-”, throughout the scripted simulation so that parts can be separated into individual faces if the force on that face can be a separate output.

2.2 Geometry Importing and Attitudes

Once the parts have been imported their name is transferred to being their label and all the components which are not the wheels and their blends are grouped as an object called “sprung” – highlighted in the maroon color in Figure 6 – this is the object which the ride heights and roll is applied to.

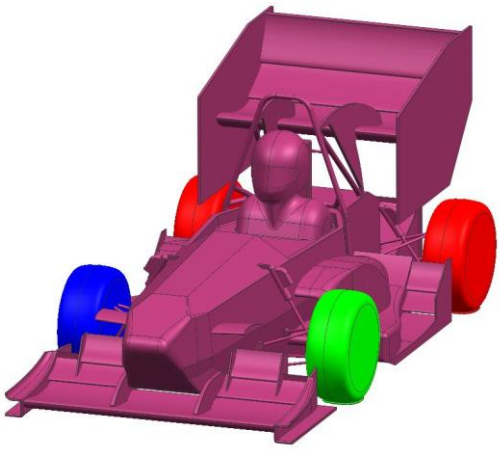


Figure 6 shows the vehicle model with highlighting to show the separation of components when the attitudes are applied

While the rear wheels can be grouped together because they do not steer the front wheels are grouped separately so that they can be steered about their individual steering axis. Once the parts have been separated into their groups that attitude definition file can be imported.

Table 1 breaks down the parameters in the "attitude.jou" file

Variable Name	**Description**
frh	Front ride height is an attitude parameter
rrh	Rear ride height is an attitude parameter
roll	Roll angle is an attitude parameter
steer	Steer angle is an attitude parameter
yaw	Yaw angle is an attitude parameter
wb	Wheel base is a vehicle parameter so should be left constant except for during concept changes

tw	Track width is a vehicle parameter so should be left constant except for during concept changes
rh	The static ride height is the ride height the vehicle is at in the CAD model
rch	The roll center height is the Z coordinate of the origin of the roll axis

Once the attitude parameters have been imported the attitude can be applied to the imported geometry as described by the meshing overview in Figure 7 which highlights that while all the other parameters are easy enough to apply yaw presents an issue which is tied to the mesh generation. The volume within the simulation is filled with cubes (equilateral hexahedron mesh called hexcore) which are aligned with the coordinate system within Fluent. It is best to have the hexcore align with the domain because the fluid flows along the domain and hence along the cubes. However, this makes post processing more difficult because the car will be yawed at different angles for different simulations so after the mesh is generated the yaw is removed which results in the car being aligned with the coordinate axis and the domain and the hexcore ends up at the yaw angle which will also facility applying the rotating wheels in the solver as they remain in the same location as CAD.

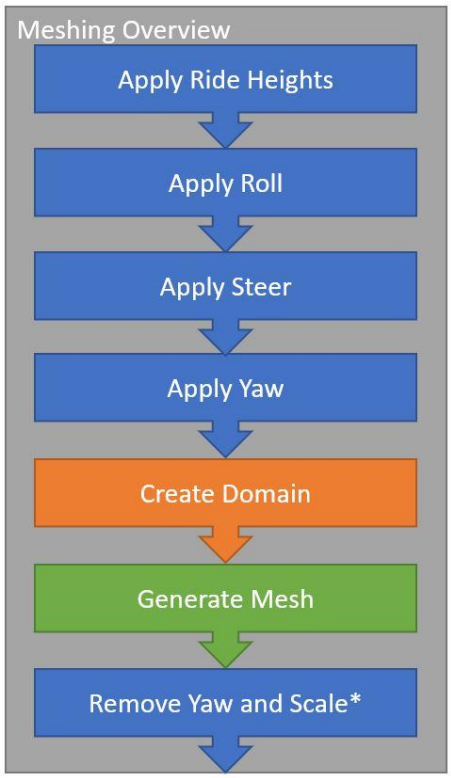


Figure 7 shows the how the attitude is applied to the model in the context of the entire meshing process (*Fluent meshing works in mm while Fluent solver works in m)

Applying the roll, steer and yaw is the relatively simple process of applying the rotation transformation about a single axis compared to applying the ride heights which requires a 3-step process outlined in Figure 8. To simplify the process the vehicles static ride height is remove so that the vehicle technically has zero front and rear ride height this means that pitch can be applied about the primary axis. The

pitch is applied about the front ride height point so the pitch angle which produces the correct delta between the front and rear ride heights is calculated. This then means that when the front ride height is added the rear will be at the specified height.

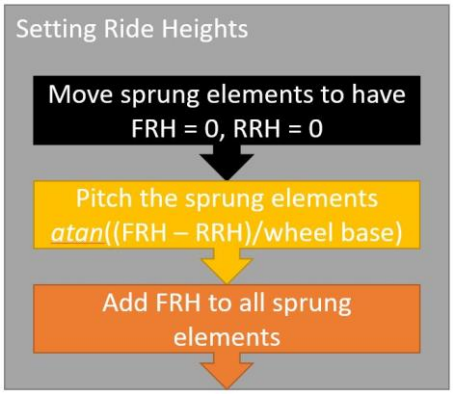


Figure 8 shows the process of applying the ride heights to the vehicle

Once the vehicle is at the correct attitude the fluid domain is created and the faces are labelled as inlet, outlet, ground and walls. This results in the domain being aligned with the coordinate axis and means that after the domain, sprung and wheel objects are united the size field can be created with all the refinement zones which are be deleted as soon as the size field – which defines the element sizes – is created. This leads on to the most import part of this simulation process: surface wrapping. This is what allows the whole attitude process to work because without it the simulation could not be completed at a parameterized attitude.

Once the surface mesh is wrapped the old mesh is discarded and the fluid volume for the volume mesh is calculated but only after the surface mesh quality is improved. The first step of improvement tries to move nodes to improve the surface quality below 0.6 skewness then any elements which could not be corrected are removed. The process of removing the bad elements by collapsing them, which does reduce accuracy of the surface but by a tiny margin, has little to no effect on the results.

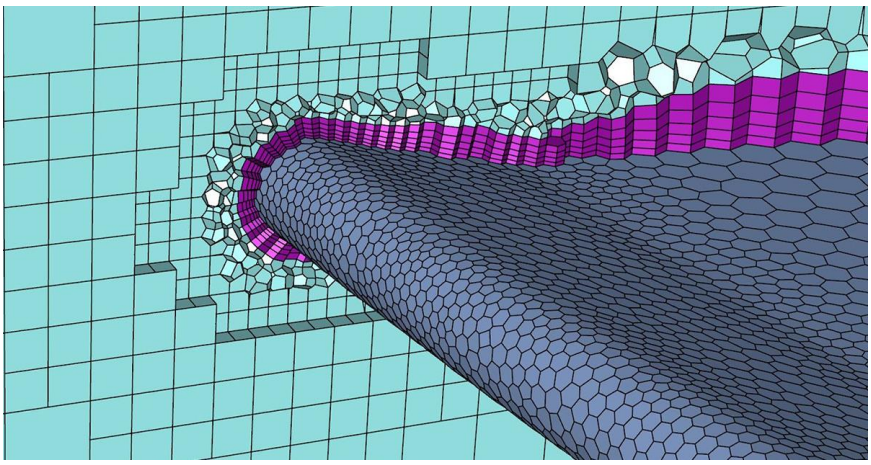


Figure 9 shows an example of the poly hexcore mesh produced by Fluent (https://www.ansys.com/-/media/ansys/corporate/social/fluids/meshing_1200x600.jpg)

The Poly Hexcore mesh is then produced automatically by the Fluent Mesher using the sizing field generated just before the wrapping process. This meshing style was chosen because it provided the

best memory efficiency for any given resolution (Hashan, 2018) and produces elements with better quality which helps the solver produce more accurate and consistent results (ANSYS help) this is thanks to the majority of the volume having a consistent structure of elements and the surfaces consisting of polyhedral prisms which produce inflation layers with the same number of layers but with 20% of the elements. Even though the polyhedral prisms utilize twice the RAM (Hashan, 2018) this still results in a reduction of memory usage of more than 50%.

Once the mesh has been generated a procedure called "auto node move" attempts to improve the mesh quality further. This results in the worst element having a skewness of 0.8 compared to before the improvement where the worst element can be as high as 0.99.

Once the geometry is imported, the attituded is applied, the mesh is produced, the domain is scaled from mm to m and the domain is yawed so that the vehicle is not, the script process for Fluent Meshing is completed and the mesh is saved before switching to the Fluent Solver. Where the boundary conditions, turbulence model and solver parameters are set before the model is solved.

2.3 Boundary Conditions

The boundary conditions are very simple for this simulation because it involves only straight flow through the tunnel and with a potentially yawed vehicle. The yawing and steering presented the biggest challenge for modelling the rotating wall of the tires because transformations had to be done to ensure that each tire rotated about the correct axis. However, even this challenge can be overcome with some simple trigonometry and vector calculations based on only the steer angles thanks to the fact the car is kept straight while the wind tunnel is yawed.

Table 2 shows the boundary conditions for the CFD simulation

Boundary	**Condition**	**Description**
Walls (specifically top and sides)	Free Slip Boundary Condition	This boundary conditions means that while the air cannot pass through the wall the wall will not generate a boundary layer.
Inlet	Even 16.67m/s (60km/h) flow normal to the wall	
Outlet	Even 0 Pa gauge at the wall	
Ground	16.67m/s velocity with the flow of air	This boundary condition models the ground moving under the vehicle.
Wheels	Rotating about their individual centers at 74.55 rad/s	This boundary condition models the wheels turning
Wheel blends	Free Slip Boundary Condition	With rotating tires the region between the tire and the

		ground becomes challenging for the model so blends with free slip are added to reduce the modeling difficulties.
Vehicle Boundaries	No Slip wall	The no slip condition causes boundary layers to form on these surfaces as they would on any normal wall.

2.4 Solver

Once the model has been defined Fluent has to be told how the problem should be solved which can have a large effect on the accuracy of the solution, the memory required to solve and the time it takes to find a solution. Two white papers from ANSYS helped select settings which could find a solution quickly and consistently. To align with the team's need to iterate through a large number of models and to ensure that the aero map could be produced in less than a week the Pressure Coupled Pseudo Transient Solver (PCPTS) from Fluent was selected. Swapping from a segregated solver to the pressure-coupled solver can result in simulations taking a fifth of the time but does require up to 50% more computer memory (Kelecy, 2008). In 2010 ANSYS introduced a pseudo transient solver as a subset of the pressure-coupled solver which could reduce the number of iterations required by between half and a tenth of the time (Keating, 2011).

Keating (2011) also showed that the hybrid initializer in Fluent could help start from a point closer to the solution which can reduce the number of iterations by a further 15%. This was the primary reason why the hybrid initializer was used although it proved to also help avoid instabilities during the early phases of solving. Some time was spent setting up the interpolation of previous results for initialization except this proved to be unstable especially when the vehicle was at different attitudes.

Using these setting meant that a solution where the forces had converged could be reaching within 120 iterations compared to the previous solver which required 600 or more to reach an acceptable solution. This meant that the RAM requirements were 50% higher (48GB to 75GB) but the solution could be reached within a quarter of the time. While the RAM requirements could cause an issue for solving the MMS team gained access to the MonARCH Cluster which meant that RAM was no longer a limitation for solving.

3. AUTOMATED RUN SUBMISSION AND RESULT PROCESSING

A key part alongside fully scripted simulations is being able to specify the attitudes required in an easy and flexible manner. This section covers all the automated processes created within this process to make submitting runs to create a map easy so that it can be used as part of a normal design period and within the team were member time is at a premium.

3.1 Generating Flexible Code

This subsection defines how the Python class was defined and how to use it while the next subsection will describe the exact script for creating the map shown in this paper.

Python was selected because it allows for readable and intuitive scripting with object orientated programming (OOP) which was used here to create a class (object) called “sweep”. An object contains variables and functions – which either reads the objects variables or adjusts them – which means that all the steps to setting up a batch of simulations can be contained within the object – from defining the attitudes with no repeated attitudes, creating the attitude journals, creating folders for all the runs, submitting the runs to the cluster then after the simulations reading the outputs and organizing the results into a .csv file.

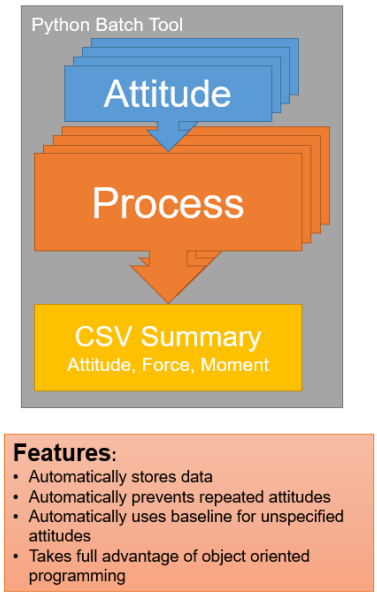


Figure 10 is a representation of the batch process for generating the aero map

The sweep class relies heavily on dicts (long named dictionaries) which are a variable type in Python which store values based on keywords. They can be very useful because in our case every number is useless unless it comes with a parameter, so our parameters are keywords. Strings can also be formatted based on a dict for example:

```
myDict = {‘name’ : ‘John Doe’, ‘age’ : 25, ‘height’ : 190}

myString = ‘Hi my name is {name} I am {age} years old and {height} cm tall’

print(myString.format(**mydict))

>>> Hi my name is John Doe I am 25 years old and 190 cm tall
```

This method is used to create the contents of the attitude.jou file used to differentiate each simulation. So, each case within the object is just a dict of each parameter with its value plus if you have “collected” after simulating the force and moment values (named ‘force.x’, ’force.y’, ‘force.z’, ‘moment.x’, ect.)

Table 3 is a table which describes the methods within the sweep object

Function (Method)	**inputs**	**Description**

__init__	base: should be a dict which contains the values for the parameters at the baseline attitude. attitude: is a string which contains named formatted strings (using {NAME}) in the place of the parameters. baselineval: is the position of the baseline (usually unused) setval: is just a counter starter (usually unused)	No inputs are required, however it is suggested that the base dict is provided if you are setting up a sweep. This is the function called when the object is created. Example: `myAeroMap = sweep(base = {'frh' : 35, 'rrh' : 35, ...}`
add	keyword arguments: these are flexible arguments used within python . (see description)	The formatting comes in as "parameter = value" which is then converted in to a dict. When you don't provide a value for all the parameters the remaining parameters are assumed to be the same as the baseline. The function will also check that the case isn't identical to any of the other cases to avoid repetition. Say you wanted to add a case with FRH = 50 and RRH = 20 to your sweep the code would look like this: `myAeroMap.add(frh = 50, rrh = 20)` This function returns the dict created for this case which was added to the list of cases.
get	n: says what the reference number of the case is.	Returns the dict of case number n in cases variable.
get_attitude	n: says what the reference number of the case is.	Returns the string for the contents of the attitude journal file.
save	filename: the name of the saved file (defaults to 'sweep.data')	This saves all the case data to file so that it can be reread after the simulations have been completed or if a sweep is being repeated.
load	filename: the name of the saved file (defaults to 'sweep.data')	This reads out the file that contains the cases and saves it to the cases list.

setup	save: is a Boolean which tells the setup function if it should save the cases to file by default (defaults to True) start: is a Boolean which tells the setup function if it should start the simulations (defaults to True)	This function will create the sub-directories and then copy the 'sweep.sh' in the same directory as the Python script into the subdirectory then create the attitude file then submit the case to the cluster (note: the script must be run on the cluster!). It will do this for each case.
collect	save: is a Boolean which tells the collect function if it should save the cases to file by default (defaults to True) load: is a Boolean which tells the setup function if it should load the cases from file at the start (defaults to True)	This function will look through the results files and collect the 3 components of the force and moment coefficient then save them into the object.
export	filename: the name of the saved file (defaults to 'map.csv')	Saves all the cases data to a csv file note that this will not have forces if collect hasn't been run.

The code relies on the 'sweep.sh' file copying the required files from a pre-set master folder except for the attitude file and that the resulting forces are summarized in the force.txt and moment.txt files otherwise the map.csv cannot be collected.

3.2 Submitting Batch Jobs

Appendix 9.2 contains the code to be run on the cluster in Python 3.6 to step up the aero map sweep. The code is very simply structured to define the baseline parameters, define the swept values of the parameters, create the sweep then add the attitudes to the sweep. Because the ride height map involves two parameters it requires nested for loops with two parameters being changed in the "sweep.add()" module. All the other attitudes involve single for loops through each of the parameters possible without skipping what would be the baseline value because the object checks whether the simulated attitude is a repeated version of any other attitude. Once all the attitudes have been added to the object all the user needs to do is call the "sweep.setup()" method which automatically saves the attitudes and submits all the simulations to the cluster.

Once the simulations have been completed a new script reads in the saved attitudes and collects all the force and moment data into a CSV file which also contains the attitude information. This file can then be imported into MATLAB, opened in Excel and with some processing added to a MoTeC i2 Pro work book.

4. USING THE AERO MAP

This section will demonstrate how to utilize the data from the aero map to assist with the operation and development of a formula student race car. The areas of application will cover an aerodynamics model for vehicle modeling and simulation which will become an essential part of MMS with its new driverless car. When implemented properly the simulation tool can also be used to determine the

sensitivity of each point on the aero map to help the aerodynamics design team focus on the most important attitudes.

There will also be some focus on visualizing the data with MATLAB and importing the map into MoTeC i2 Pro (the data logging and visualization tool used by MMS) to analyze the potential effects of a setup change on the vehicle attitudes and hence the aerodynamic loads.

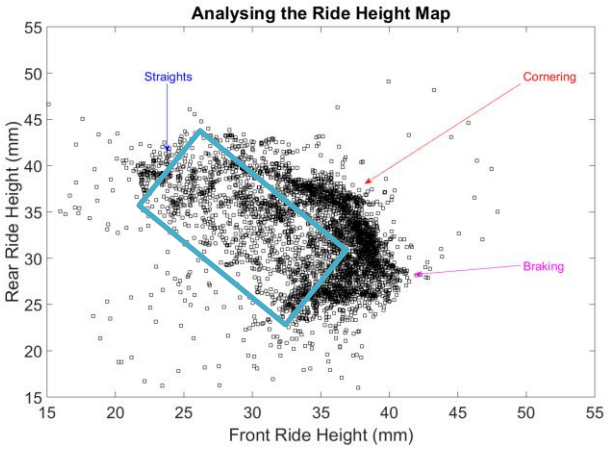


Figure 11 is a plot of front and rear ride heights over a lap with annotations of what each cluster represents in terms of vehicle conditions.

In Figure 11 above there is a scatter plot of the vehicles ride heights around a lap which is important to understand because it can help with deciding on a design point as well as attempting to select a more optimum ride height setup for the vehicle. The essential parts of the map are the clusters for braking and cornering when looking at vertical load while reducing drag down the straights. The region directly between braking and straights (highlighted in aqua) is the region where the vehicle transitions into braking and is separate from the cornering cluster.

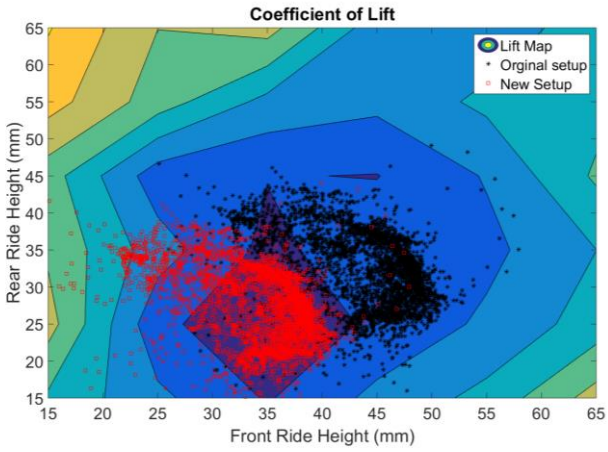


Figure 12 shows the scatter of ride heights between two distinct setups where the cornering cluster is placed closer to the peak load region.

By decreasing the front ride height by 10mm and the rear ride height by 5mm the cornering cluster can be better placed on the region of peak load which can be seen in Figure 12. This means that the vertical aero load can be increased during cornering when it is needed while sacrificing performance

on the straights where it is not needed. This can be seen in Figure 13 which shows a map of the German competition track which is highlighted by vertical load delta between the original setup and the new setup. This method of adjusting vehicle setup to position vehicle attitude could be just as easily done to influence how balance moves between cornering, braking and the straights.

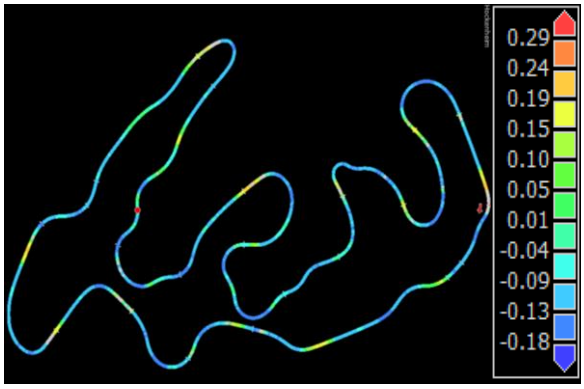


Figure 13 shows a track map highlighted based on the setup change above.

4.1 MATLAB Aerodynamic Function

The Aerodynamic function just returns 2 3-dimensional vectors for the aerodynamic force and moment of the vehicle based on FRH, RRH, roll, steer, yaw and speed. The biggest challenge is with converting the data produced into a map given that there is no combination of that data that works out of the box. The structure of the map which is baselined to a fixed attitude means that each parameter can be evaluated as a delta to the baseline attitude.

The baseline attitude is combined with the ride height map for simplicity but for the rest of the rest of the parameters the baseline values are subtracted to get a delta to the baseline for all the parameters.

Table 4 shows the values for each parameter at the baseline attitude.

Front Ride Height (mm)	Rear Ride Height (mm)	Roll (degrees)	Steer (degrees)	Yaw (degrees)
35	35	1	8	4

The predicted value – could be any or every component of either the aerodynamic forces or moments – is the sum of each map – a 2D map for ride heights, and 1D maps for roll, steer and yaw – then the baseline values have to be subtracted 3 times –subtract the baseline values once for the 4 maps then add it back on once to get a net of 3 subtracts. This is shown symbolically in Equation 1.

Equation 1 shows the generic approximation of forces and moments from the aerodynamics of the vehicle

$$Value = RideHeight_{Map}(FRH, RRH) + Roll_{Map}(Roll) + Steer_{Map}(Steer) + Yaw_{Map}(Yaw) - 3 \times Baseline_{Value}$$

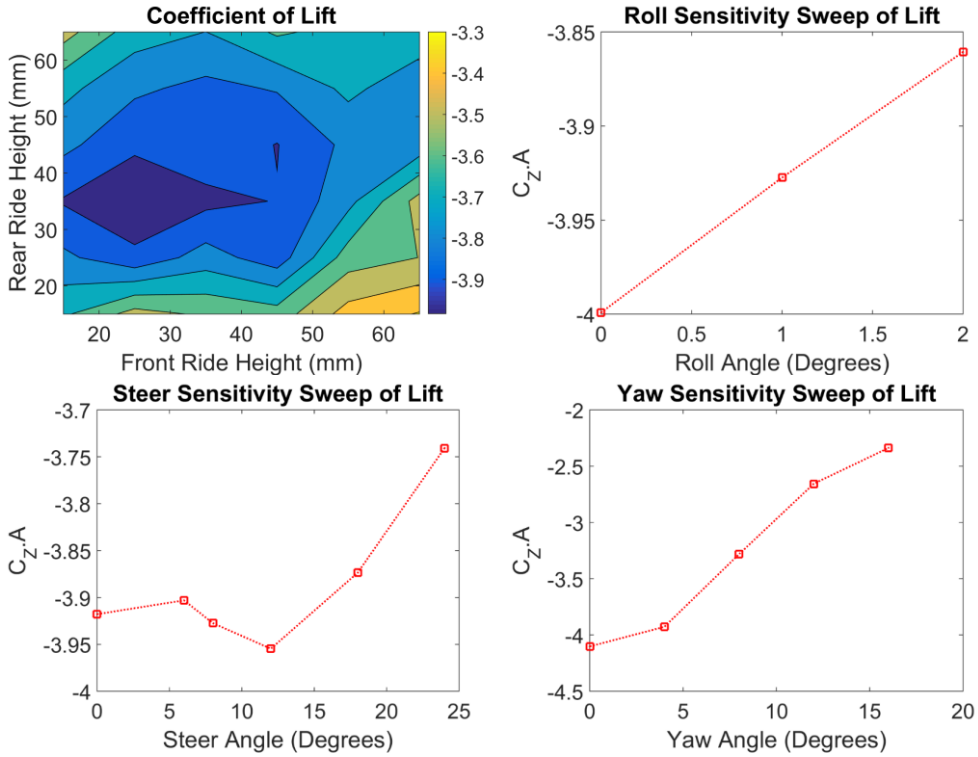


Figure 14 shows the maps for the Z components (lift) of force across all the 5 parameters that would be used in Equation 1.

Appendix 9.5 shows the MATLAB code for this process although it should be taken as pseudo code for any programming language that it is needed in for vehicle modeling in the future. This function would be able to replace the aerodynamic model in whatever vehicle simulation suite being used by MMS.

4.2 MATLAB Visualizing Data

Visualizing the data produced is one of the most important parts of utilizing the data produced by completing an aero map and MATLAB is the best way to produce plots to understand the results and evaluate changes if any are made. An essential part of having the map data within MATLAB is to structure the data in such a way which makes it easiest to access the desired data with the least number of transformations. The script "map_generate.m" within the "map" project folder that accompanies this report structures the data for just this purpose using MATLABS "struct".

- `map`
  - `rhs (Ride Heights)`
  - `roll`
  - `steer`
  - `yaw`

Under each value are 3 arrays called "values", "force", "moment" and a struct called "name" which contains a short and long name as well as the units for the parameter. "values" Contains all the unique values which that parameter can hold while "force" and "moment" contains the 3 components of the force and moment for each of the values. This means plotting the Z component of force vs roll angle is as simple as the code in Figure 15.

```
plot(map.roll.values, map.roll.force(:, 3))
```

Figure 15 code demonstrating structure of map struct (note: roll could be swapped with steer or yaw with no other changes required)

Using the strings contained under “name”, plots can also be generated in batch because in the case of Figure 15 the X axis label can be created using the long name variable and units which can be seen in the code in Appendix 9.6 where the plots for roll, steer and yaw are produced in a single for loop rather than requiring the repetition of code.

MATLAB can also, with ease, create animations from images so with the images generated in Section 9.5 and data exported directly from MoTeC i2 Pro containing the front and rear ride heights for a given amount of driving to generate an animation of the pressures under the car as can be seen in Figure 16. This allows for the visualization of loads during driving which can help with visualizing the aerodynamic loads and how they transition around a lap.

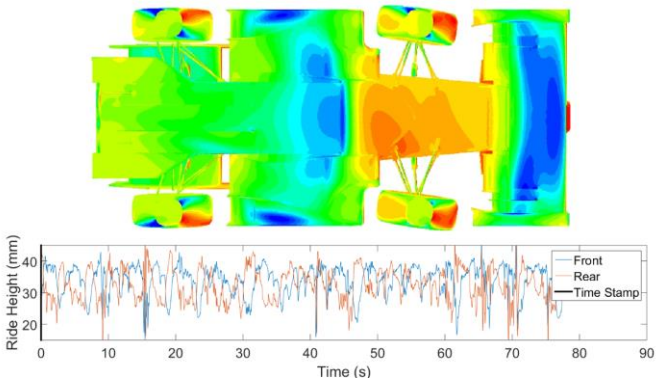
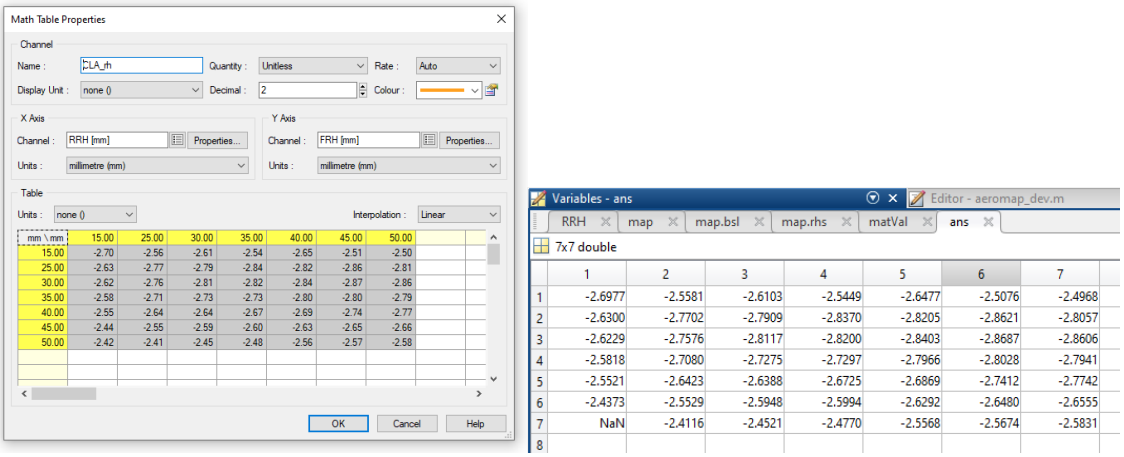


Figure 16 shows the first frame of the pressure animation generated by a MATLAB script.

4.3 MoTeC i2 Pro

MoTeC i2 Pro is one of the best racing data analysis tools available and while it is free, it requires MoTeC hardware to produce log files which it can read. The Monash Motorsports (MMS) team has been using MoTeC software and hardware since before 2008 so it has become a key part of testing, tuning and development on the team but has been lacking any aerodynamic data until now. This section discusses how to add the aero map data to MoTeC so that the performance data of the aero package can be plotted with the rest of the vehicle data to better understand the effect aerodynamics are having on the vehicle around the track.



	1	2	3	4	5	6	7
1	-2.6977	-2.5581	-2.6103	-2.5449	-2.6477	-2.5076	-2.4968
2	-2.6300	-2.7702	-2.7909	-2.8370	-2.8205	-2.8621	-2.8057
3	-2.6229	-2.7576	-2.8117	-2.8200	-2.8403	-2.8687	-2.8606
4	-2.5818	-2.7080	-2.7275	-2.7297	-2.7966	-2.8028	-2.7941
5	-2.5521	-2.6423	-2.6388	-2.6725	-2.6869	-2.7412	-2.7742
6	-2.4373	-2.5529	-2.5948	-2.5994	-2.6292	-2.6480	-2.6555
7	NaN	-2.4116	-2.4521	-2.4770	-2.5568	-2.5674	-2.5831
8							

Figure 17 shows how a variable table in MATLAB can be copied straight into a 2D math table in MoTeC i2 Pro to create a CLA prediction based on ride height

For the ride height map to be transferred to i2 Pro it needs to be converted into a 2D array which can either be done with MATLAB or with an Excel Pivot Table then as shown in Figure 17 the entire table can be copied in one go from either MATLAB or Excel straight into i2 Pro which make the entire process very easy. Then a 1D map for roll and steer – yaw angles are not logged and cannot be calculated – can be added as well this is the same process as the 2D map except the data should be in a column. All three data channels can then be added to each other and 2 time the baseline value subtracted to get the predicted value based on ride height, roll and steer.

5. VALIDATION

While simulation techniques have improved the predictions of time averaged turbulent flow still hasn't reached a point where it can be trusted out of the box especially when the model is simplified such as in this situation. While using a wind tunnel allows for a controlled environment it limits the number of parameters to 1 – yaw angle – which means that it cannot validate the most important part of this project. Hence, on-track validation is required to validate the aero map which also opens the door for on-track design and development.

5.1 Wind Tunnel Validation

While the wind tunnel does not allow for data to be collected at different vehicle attitudes it does allow for results to be collected which would be infeasible on-track such as a sweep for 10 4-hole probes on a rake which swept through various heights to generated the data in Figure 18 which was only a subset of the highly detailed data produced by the rake sweep. The strengths and weakness of the computer simulation can be seen within the data in Figure 18 and Figure 20.

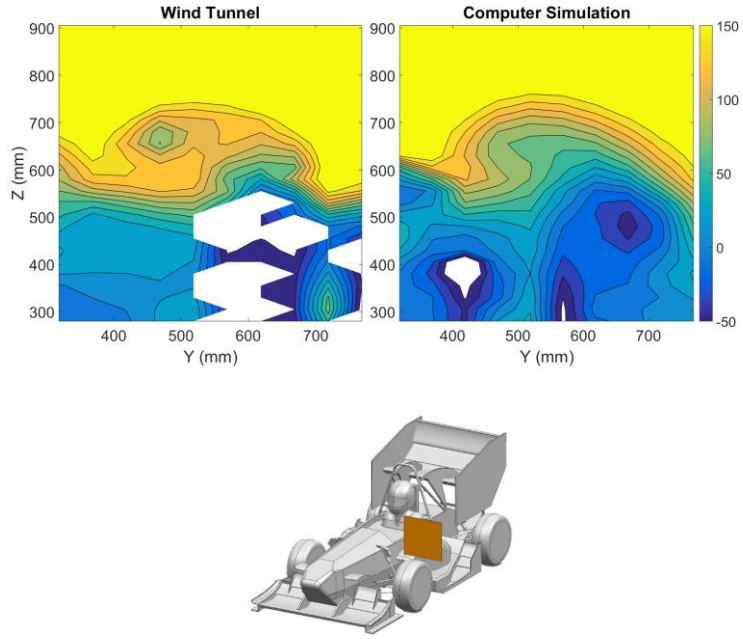


Figure 18 shows the total pressure in Pascals for both the wind tunnel and computer simulation at the plane shown in the image below.

The computer simulations are completed as Reynolds-Averaged versions of the incompressible Navier-Stokes equations which means that the transient nature of the flow is simplified to a steady state solution. As the flow becomes more turbulent and hence time dependent the time averaged solution begins to lose its accuracy. In Figure 18 the computer simulation data appears far smoother

especially around the nose wing vortex at Y=420mm, Z=680mm which in the wind tunnel data consists of a tighter vortex with higher even pressure around it. While there is data lacking in the wheel wake region at around Y = 600m for the wind tunnel the region below Z=500mm and inside Y=500mm shows a very different pressure profile between the CFD which predict very low energy compared to the wind tunnel which showed total pressures between 0 and 50Pa compared to CFD with values below -50Pa.

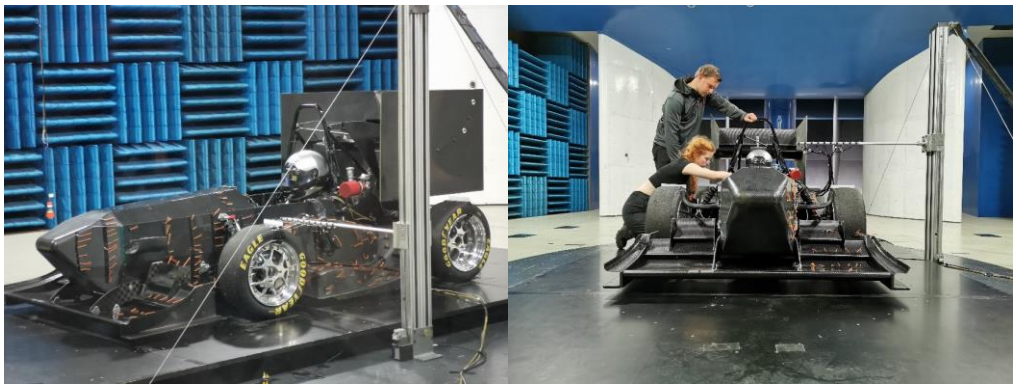


Figure 19 Shows the setup of the rake in the wind tunnel when the data was collected.

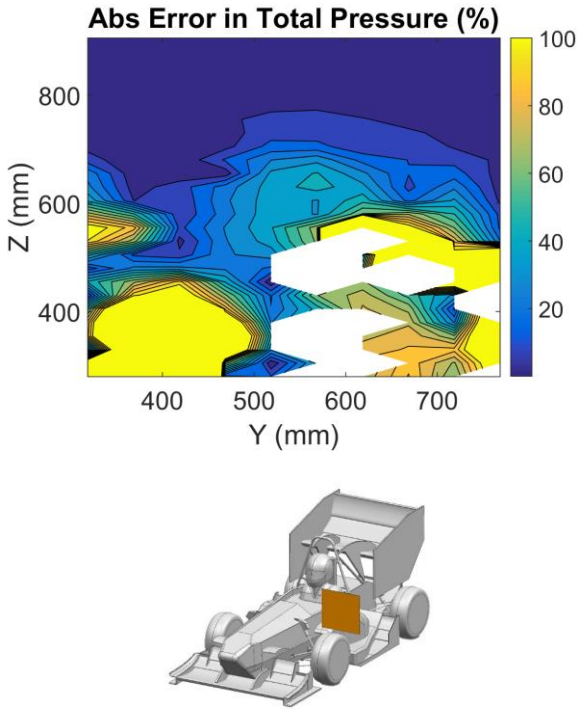


Figure 20 shows the percentage error in total pressure between the wind tunnel and computer simulations.

Figure 20 highlights the error regions which occur primarily in the front wheel wake and front wing wake which are both below Z=600mm while the region with the nose wing vortex and free stream flow show little to no error. This confirms the expectation that the CFD is very capable of predicting the flow in regions with low to no turbulence but is unable to predict the flow in turbulent regions behind the vehicle.

The simulation data seems to smooth out the turbulent regions and exaggerate the low pressures in the wing wake. This would indicate that the CFD can predict the structures and their locations well but is unable to predict the effect the structures have on the rest of the flow.

5.2 On-Track Methods for Validation

While the team has produced differential pressure measurement systems in the past, they have never been able to produce valuable results because while they met the primary goal of measuring pressures the readings were never produced in the context of the rest of the car or in a time efficient manner. To ensure that the system developed within this section of the report can become an effective tool the objective and requirements of this system had to be well described.

The best place to start is in some analysis of the strengths and weaknesses of previous designs which each got used a maximum of twice due to the complexity of the systems and the requirement of the designer to operate it. A system which can be operated by any team member and be used with little to no understanding of the system would allow it to have a longer effective life and add more value from each use.

The previous systems also logged the pressure data onto an SD memory card as a comma separated values (CSV) file which while it did provide easily readable and analyzed data there was available context for the data which is essential to analyze an aero map. The context of the data would include information such as suspension position – which is used to calculate ride heights and roll angles – steered angle, speed, acceleration and track position all of which helps the analyst understand what the vehicle was doing at each point the pressure was measured.

Table 5 shows the design requirements and specifications of the differential pressure measurement system

Requirements/Specification	**Description**
Measure aerodynamic pressures on the vehicle	-3 to 1 CP at 120 km/h = (-3 * 680Pa or -2kPa to 680Pa)
Measure with accuracy and resolution	Accuracy +/- 5Pa, Resolution +/- 1Pa
Sample fast enough to average turbulence	1kHz
Work on vehicles CAN-BUS with minimal disruption	No more that 1% of 1Mb/s CAN-BUS so 10kb/s. Each tap is 16 bits with 24 taps that makes 384 bits except a CAN message contains only 50% data so that means 800 bits per sample. So ~10Hz data rate.
Be logged with vehicle data by MoTeC data logger	This can be done either with a CAN expander or by logging directly to the CAN-BUS
Can be mounted easily anywhere on the car	The more compact the system the better because that means it can fit anywhere on the vehicle which means that a wider range of components can be tested.

Log a maximum number of taps with one unit.	Between 16 and 32 pressure taps would be required to make the system useful.

What would be ideal is an “off the shelf” solution from the motorsports industry however given that it is the motorsports industry the prices for these kinds of units can be nearly $4000 AUD (https://www.motorsportselectronics.com/products/texense-16xpdif-16-channel-differential-pressure-sensor?variant=21334340740). Techmor offers a solution which can be seen in Figure 21 that offers an excellent product at a more reasonable price however with a student team budget and no guarantee that the solution would be used more than once the first attempt was not going to be off the shelf.

Figure 21 Techmor’s $2000 solution which would be perfect (https://www.techmor.com/aero-pressure-sensor-array-ap-1/)

The first step to designing the system was selecting a pressure transducer which could either output a digital or analogue signal. While a digital signal could be advantageous for low noise it introduces complexity especially with the last DPMS whose sensors all had the same digital ID which removed the advantage of having digital sensors by requiring a multiplexor which also meant that the sample rates were around 10Hz. But if an analogue sensor was to be selected a device with 16 or more analogue in channels would be required. The transducer which best met the requirements of +/-2kPa, analogue and 3-5V operating range was the MPXV7002DP which met all the requirements and cost $14 each compared to the $125 per transducer price tag from Techmor. While the two holed version of the transducer was purchased in hind sight if a duplicate system were to be made it is suggested that only the single port version of the transducer be purchased because the reference adds more complexity that it is worth given that one transducer could be used as the reference.

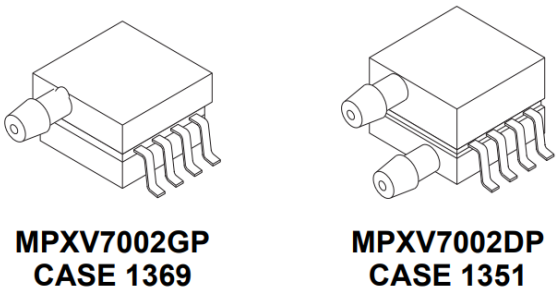


Figure 22 the MPXV7002 in both the differential (right) and gauge pressure (left) layouts (source: NXP data sheet).

Given that the transducers were analogue the microcontroller for this system needed to have more than 16 analogue pins and had to meet the other requirements of CAN capabilities, 12-bit read resolution and to be compact in size. The Teensy 3.5 was met all these requirements with the advantage of being programmable in the user-friendly Arduino environment which was very attractive. All other Arduino based products required CAN shields that were as large if not larger than the Teensy and the microcontrollers where usually larger than the Teensy. The Teensy 3.5 has 26 analogue pins and only requires a tiny adapter to convert a digital signal to the CAN high low standard.

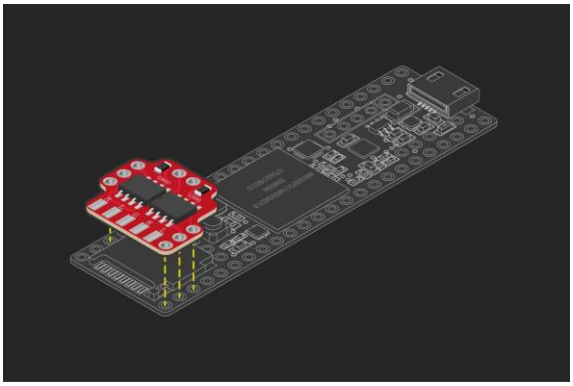


Figure 23 shows the Teensy 3.5 with the CAN adapter overlaid over it. Teensy 3.5 size (62.3mm x 18mm) (https://www.tindie.com/products/Fusion/dual-can-bus-adapter-for-teensy-35-36/)

The better the transducers could be packaged the smaller the pressure array could be and with 24 taps in a 6 x 4 array reducing the spacing between units by 5mm would reduce the volume by 40% which is very significant. This meant that mounting all the transducers onto one printer circuit board (PCB) while simpler electronically would result in a volume 5 times larger than the layout selected which can be seen in Figure 24.

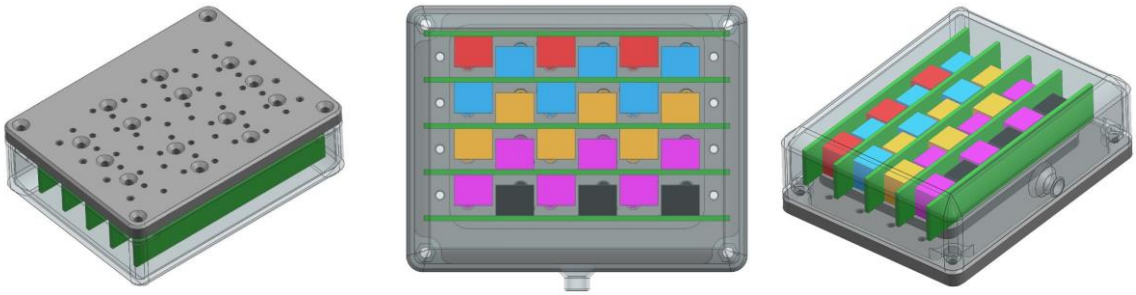


Figure 24 shows the CAD assembly of the system with each PCBs transducers highlighted differently to highlight the tight packaging within the box. Left: isometric of front face, middle: back view and right: isometric of back face

Each PCB could take 6 pressure transducers (3 on either side) such that they would tesselate as seen in Figure 24 which meant that each PCB had 8 pins – one for each channel plus power and ground – the power and ground could be spliced together from each PCB. Although it would be logical to think that only 4 PCBs were required to achieve optimal tessellation the end units only held 3 taps so 5 PCBs had to be used. Each PCB had the 3 capacitors defined in the MPXV7000 spec sheet for each transducer to condition the supply voltage and signal. Otherwise the supply voltage was connected to the 5V input which also powered the Teensy via the supply pin. The standard 4-Pin DTM connector layout for CAN devices on the car comes with CAN-High, CAN-Low, 5V Power and Ground which means

that all the devices (bar the CAN adapter which runs off a regulated 3.3V output on the Teensy) can be powered directly from the connector pins.

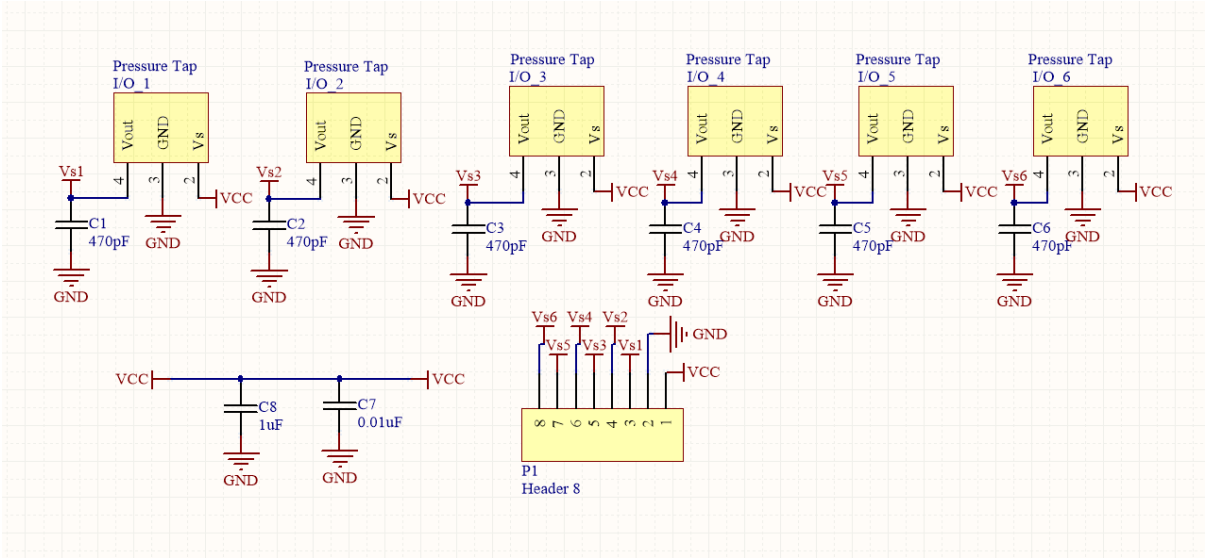


Figure 25 shows the electrical diagram for the PCB which holds the transducers.

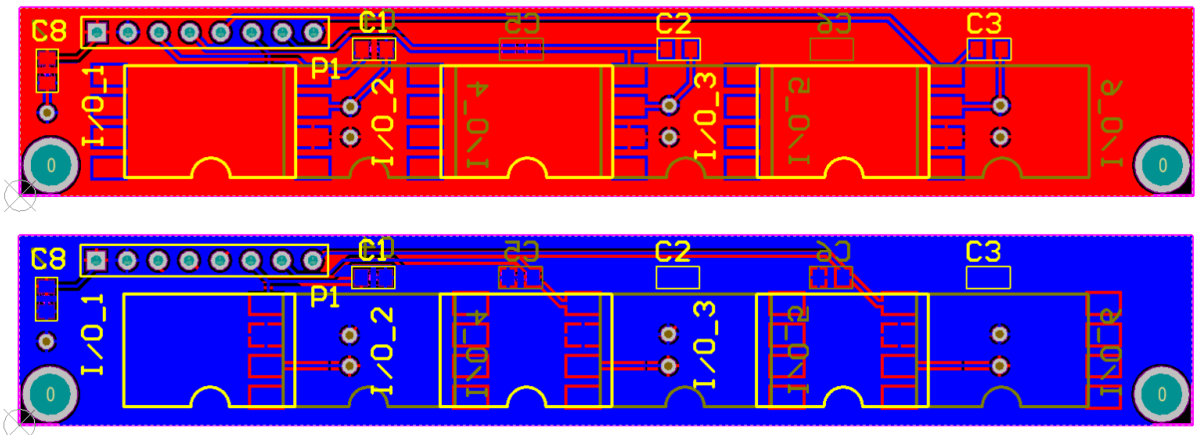


Figure 26 shows both sides of the PCB as it was printed.

Once the system was electrically assembled the challenge became connecting the fragile transducer ports to steel M3 barbs for easy and robust tube connections via a solid header. The early attempts to build a header used ABS plastic which was 3D printed primarily because it allowed for internal channels to connect the reference lines. The ABS failed to be strong enough to hold the M3 Barbs and to be stiff enough when the header was tightened so the ABS solution was taken as a prototype and an aluminum equivalent was CNC machined without the common header for the reference lines.

This also meant that the header and O-ring compression plate were stiffer which meant that the O-rings would be held under more consistent pressure and provide a good seal even for the transducers furthest from the compression plate bolts.

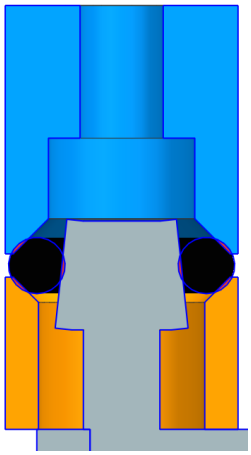


Figure 27 shows a detailed cross section of the sealing interface used to connect the taps port with the M3 barbs.

Generating a proper seal was one of the biggest challenges with producing the DPMS because the ports on the transducers were so small that non custom O-rings wouldn't fit the ports and because sealing required the very precise positioning of the O-ring. The ports on the transducers had to be expanded with some 4mm ID Nylon tube from SMC to fit the 5mm O-rings with enough pressure to seal. The chamfers seen in Figure 27 also helped seal the transducers by compressing the O-ring as the blue section was tightened to the orange section. This means that during assembly the taps had to be clamped down before the O-ring compression plate was tightened or the transducers would not seal properly.

3D printed bracing bars were used to hold the transducers onto the header before the compression plate was tightened to produce the best seal. Once tightened it is suggested that the braces are left on even though the O-rings are sufficiently tight to hold the transducers in place. Once the sensors are properly mounted to the aluminum header. If the system is wired correctly, before bolting down the cover the internal connector which connects the connecter – which is attached to the case – to the header, transducer and microcontroller assembly must be connected, with the wires colors matching on either side of the connector – there is no unidirectional connector because it is too bulky. Finally, the case can be tightened down onto the header and the edge can be sealed with tap to improve weather sealing.

Before all this the Arduino program needs to be loaded onto the Teensy 3.5 so that it knows what to do when the power is switched on. The program consists of two parts which are separated by the background colors in Figure 28 with the blue representing the setup stage and the two grey section representing normal operation and debugging.

The debugging code allows the Teensy to be accessed once it has already been installed which can allow for the zeroing of data, taking a sample for calibration or just to check the system is operating as expected, display the timer information for the system to ensure that the system is logging at the correct rate and a final debugging display that informs the user of how many messages are sent with each set of samples.

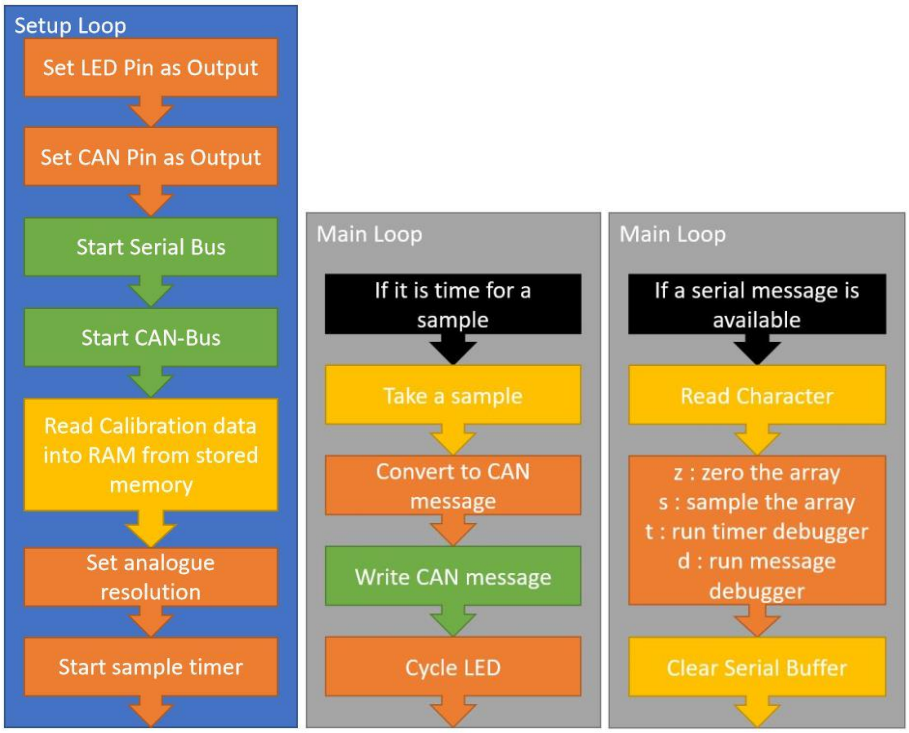


Figure 28 shows simple graphical representations of the Arduino program which runs on the microprocessor.

Table 6 shows the variables that might be changed in the Arduino program to adjust the program to match different system setups.

Variable	**Description**
N_TAPS	Sets the number of taps to read and write to the CAN bus.
RATE	The rate is in Hz and sets the write rate so how often the CAN message is sent.
SAMPLES	Samples defines how many times to read each tap for a sample. The equation below should be used to get a maximum rate for the cleanest data. Samples = 34,752/(RATE*N_TAPS)
bd	Is the baud rate of the CAN bus so for 1Mb/s use 1000000.
tx, rx	Set which pins the CAN adapter is plugged into and should usually be set to 1 for both.

Each CAN message is 64 bits (or 8 bytes) which is enough to carry the pressure value for 4 taps which are stored as the pressure in Pascals with a default offset of 2000 added. This is done because M1

Build would not read in signed integers properly so 2000 was added to the pressure value in Arduino then 2000 was subtracted in M1 Build. To transmit 24 values this meant that 4 messages had to be used to transmit all the 24 transducers values.

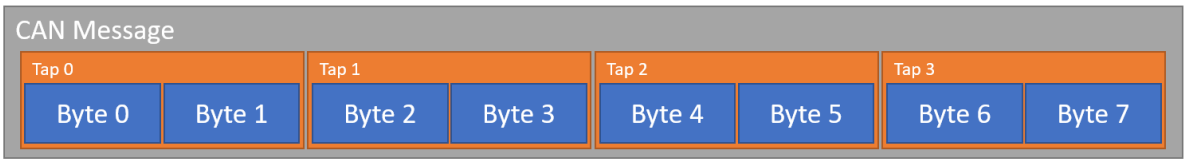


Figure 29 shows the breakdown of a CAN message sent by the microcontroller to the CAN bus.

Once the pressure tapping box is completed the MoTeC data logger needs to be told how to read and log the data sent over the CAN bus by the pressure array. This programming is done in M1 Build which is MoTeCs software for programming their ECUs and data loggers. Appendix 9.8 contains all the setup information for M1 Build to get the system to work. In short there are 24 channels which are updated by a scheduled function which just reads the CAN messages then saves them to the channel. All of these elements are contained within a group for neatness and organization but which also contributes the variables names. To get the data logger to store the values M1 Tune must be used to add the Pressure Array variables to the logged data.

Figure 30 shows the completed pressure array ready to be installed on the vehicle.

Once completed the box shown in Figure 30 can be plugged into any port on the car and will automatically be logged. This allows the system to be used in a wide range of situations to test any aerodynamic component on the vehicle. Some data was collected to prove the system worked before the completion of this report however a lack of time has prevented any valuable data from being extracted. Figure 31 shows some values plotted in MoTeC as an overlay of the track to help add context to the location of the pressure values.

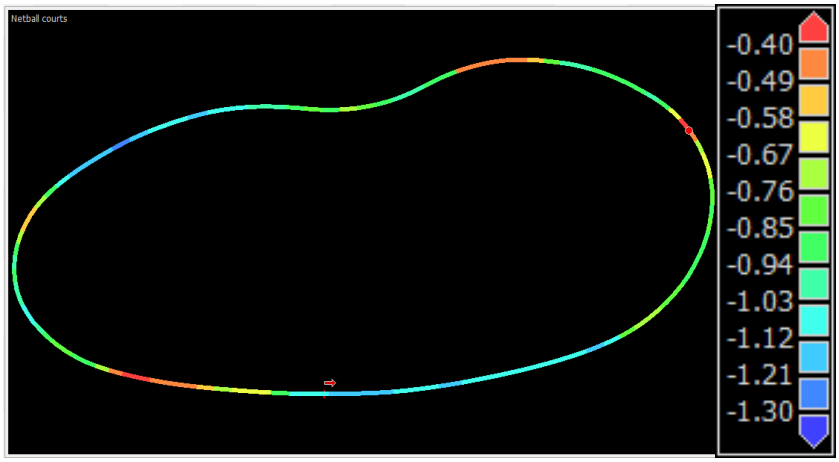


Figure 31 shows the coefficient of pressure value for one of the transducers over a lap of a short 150m track involving a 10m slalom (top) and two 20m radius corners.

This last paragraph will add some recommendations for the future of the pressure array so that it can be developed and become an even more usable tool for the team when designing with aerodynamics. The current system involves a lot of soldered wires which are prone to failure from fatigue so one of the suggestions is to remove all the wire connections and replace them with PCBs and connectors to improve reliability. The weather sealing of the current box is highly questionable so it would also be suggested that the box be either checked or redesigned to be water tight.

6. CONCLUSIONS

In this project an aero map was successfully produces using Fluent as a fluid simulation package with assistance from the MonARCH cluster and a python script for batch job submission and results processing. This process was coupled with a variety of analysis tools in both MATLAB and MoTeC i2 Pro which allowed for the maximum value to be extracted from the aero map during the design phase of development and during the tuning and setup phase of development.

Wind tunnel data was also collected to help validate the simulated values and to gain an understanding of where the simulation was predicting the flow well and where it was failing to predict the flow. A system for collecting on-track data to overcome the wind tunnels limitation on producing results for the vehicle at different attitudes but also to help develop and understand the aero map during real world driving.

7. ACKNOWLEDGEMENTS

I would like to acknowledge the following for their contribution to this project:

Scott Wordley as the advisor for the Monash Motorsports team and the Academic advisor for this project

Bryce Ferenczi for his assistance with the development of the PCB for holding the pressure transducers.

Thejana Abeykoon for his help and guidance with setting up M1 Build and Tune so that the pressure array would work properly with the vehicles.

Simon Michnowicz from MonARCH for allowing the MMS team to use the cluster and for showing me how to get setup on the cluster.

8. REFERENCES

Franklyn J, Kelecy, Coupling Momentum and Continuity Increases CFD Robustness, 2008, https://www.ansys.com/-/media/ansys/corporate/resourcelibrary/article/aa-v2-i2-coupling-momentum-and-continuity.pdf

Hashan Mendis, Better meshing using ANSYS Fluent Meshing?, 2018, https://www.linkedin.com/pulse/better-meshing-using-ansys-fluent-hashan-mendis/?published=t

Mark Keating, Accelerating CFD Solutions, 2011, https://pdfs.semanticscholar.org/b209/aceb802f67e862bd49dc4f1b2748a7ff9a17.pdf

Xin Zhang, Willem Toet and Jonathan Zerihan,2006, Ground Effect Aerodynamics of Race Cars

9. APPENDICES

9.1 Python Scripts defining Sweep Class

```
import os
import shutil
from pickle import dump, load
from itertools import count

name_format = '{baseline:02d}.{set:02d}.{run:02d}'

attitude_string = '''; Define attitude variables
(define frh {frh}) ; Front Ride Height
(define rrh {rrh}) ; Rear Ride Height
(define roll {roll}) ; Roll Angle
(define steer {steer}) ; Steer Angle
(define yaw {yaw}) ; Yaw Angle
; Define Vehicle Parameters
(define wb {wb}) ; Wheel Base
(define tw {tw}) ; Track Width
(define rh {rh}) ; Base Ride Height
(define rch {rch}) ; Roll Center Height'''

class sweep:
    def __init__(self, base, attitude = attitude_string, baselineval = 0, setval = 0):
        self.base = base.copy()
        self.baseline = baselineval
        self.set = setval
        self.cases = []
        self.cases.append(base.copy())
        self.result = {}
        pass

    def add(self, **kwargs):
        case = self.base.copy()
        bool_base = False

        for key, value in kwargs.items():
            case[key] = value
            if self.base[key] is not value:
                bool_base = True
            pass

        if bool_base:
            self.cases.append(case)
```

```
            pass
        return case

    def get(self, n):
        return cases[n]

    def get_attitude(self, n):
        case = self.cases[n]
        return attitude_string.format(**case)

    def save(self, filename = 'sweep.data'):
        with open(filename, 'wb') as filehandle:
            dump(self.cases, filehandle)
        pass

    def load(self, filename = 'sweep.data'):
        with open(filename, 'rb') as filehandle:
            self.cases = load(filehandle)
        pass

    def setup(self, save = True, start = True):

        home = os.getcwd()

        if save:
            self.save()
            pass

        for i, case in enumerate(self.cases):
            name =  name_format.format(baseline = self.baseline, set = self.set, run = i)

            os.mkdir(name)
            shutil.copy('sweep.sh', '{}/sweep.sh'.format(name))

            os.chdir(name)

            with open('attitude.jou', 'w+') as file:
                file.write(attitude_string.format(**case))
                pass

            if start:
                os.system('sbatch sweep.sh')
                pass

            os.chdir(home)
            pass
        pass

    def collect(self, save = True, load = True):
        home = os.getcwd()

        if load:
            self.load()
            pass

        for i, case in enumerate(self.cases):
            name =  name_format.format(baseline = self.baseline, set = self.set, run = i)
            data = {}

            os.chdir(name)
            os.chdir('out')

            keys = ['force', 'moment']
            dims = ['x', 'y', 'z']
            for key in keys:
                with open('{}.txt'.format(key)) as filehandle:
```

```
                    lines = filehandle.readlines()
                    pass
                for line in lines:
                    if line[0:3] == 'Net':
                        total = line
                        break
                    pass

                n = 0
                strValue = ''
                values = []
                for char in total:
                    if n > 5:
                        nn = 0
                        if char == ')':
                            values.append(float(strValue))
                            break
                        elif char == ' ':
                            values.append(float(strValue))
                            strValue = ''
                            pass
                        else:
                            strValue = strValue + char
                            pass
                        pass
                    elif char == '(':
                        n = n + 1
                    pass


                for dim, value in zip(dims, values):
                    valName = '{key}.{dim}'.format(key = key, dim = dim)
                    self.cases[i][valName] = value
                    pass
                pass

            os.chdir(home)
            pass

        if save:
            self.save()
            pass

    def export(self, filename = 'map.csv'):
        lines = []
        line = ''
        keys = []
        for key, value in self.cases[0].items():
            line = line + key + ','
            keys.append(key)
            pass

        line = line[:-1] + '\n'
        lines.append(line)

        for case in self.cases:
            line = ''

            for key in keys:
                line = line +  str(case[key]) + ','
                pass

            line = line[:-1] + '\n'
            lines.append(line)
            pass
```

```
        with open(filename, 'w+') as file:
            file.writelines(lines)
            pass
        pass
    pass
```

9.2 Python Script defining Map Setup

```
from sweepClass import sweep

# Define sweep values
rhs = [15, 25, 35, 45, 55, 65]
rolls = [0, 1, 2]
yaws = [0, 4, 8, 12, 16]
steers = [0, 6, 12, 18, 24]

# Define baseline attitude
base = {'frh' : 35,
        'rrh' : 35,
        'roll' : 1,
        'steer' : 8,
        'yaw' : 4,
        'wb' : 1550,
        'tw' : 1200,
        'rh' : 40,
        'rch' : 30}

# Create sweep object
swp = sweep(base)

# Add the ride height map
for frh in rhs:
    for rrh in rhs:
        swp.add(frh = frh, rrh = rrh)
        pass
    pass

# Add the roll sweep
for roll in rolls:
    swp.add(roll = roll)
    pass

# Add the steer sweep
for steer in steers:
    swp.add(steer = steer)
    pass

# Add the yaw sweep
for yaw in yaws:
    swp.add(yaw = yaw)
    pass

# Start the simulations
swp.setup()
```

9.3 Python Script defining Data Collection

```
from sweepClass import sweep

swp = sweep({})
swp.collect(load = True, save = True)
swp.export('map.csv')
```

9.4 Shell Script for Simulations

```
#!/bin/env bash
#SBATCH --job-name=mms-cfd-aeromap
```

```
#SBATCH --time=12:00:00
#SBATCH --partition=comp,short
#SBATCH --mem=98000
#SBATCH --ntasks=16
#SBATCH --ntasks-per-node=16
#SBATCH --cpus-per-task=1

module load ansys/19.1

master='/mnt/lustre/projects/eh14/sweep/master'
origin=`pwd`

mkdir out

cp $master/* $origin/
fluent 3d -g -meshing -t${SLURM_NTASKS} -i run.jou

cd $origin
cp slurm* solution* attitude* *.txt $origin/out/
rm *
```

9.5 Aerodynamic Map MATLAB FUNCTION

```
function [ force, moment ] = function_map( map, attitudes, velocities )
%FUNCTION_MAP This function takes a map, an attitude (or list of attitudes)
%and velocity (or list of velocities) and returns the forces and moments at
%those attitudes and velocities note that the dimensions of attitudes and
%velocities must be factors of each other.

% Assume that if no velcity is provided the users is after coefficients
if nargin == 1
    attitudes = map.attitude;
    dynamic_pressure = 1;
elseif nargin == 2
    dynamic_pressure = 1;
else
    dynamic_pressure = 0.5*1.225*velocities.^2;
end


% Create matricies for frh and rrh
[frh,rrh] = meshgrid(map.rhs.frh, map.rhs.rrh);

% Get the size of the inputs velocities and attitudes
szeAtt = size(attitudes);
szeDyn = size(dynamic_pressure);

% Create empty matricies
force_rhs = NaN(szeAtt(1), 3);
moment_rhs = NaN(szeAtt(1), 3);

% Sweep through the dimensions for the 2d interpolation tool
% Get the front and rear ride height forces
for a = 1:3
    force_rhs(:,a) = interp2(frh, rrh, map.rhs.force(:,:,a), attitudes(:, 1), attitudes(:,
2));
    moment_rhs(:,a) = interp2(frh, rrh, map.rhs.moment(:,:,a), attitudes(:, 1), attitudes(:,
2));
end

% Get the roll forces
force_roll = interp1(map.roll.roll, map.roll.force, attitudes(:, 3));
moment_roll = interp1(map.roll.roll, map.roll.moment, attitudes(:, 3));

% Get the steer forces
force_steer = interp1(map.steer.steer, map.steer.force, attitudes(:, 4));
moment_steer = interp1(map.steer.steer, map.steer.moment, attitudes(:, 4));

% Get the yaw forces
force_yaw = interp1(map.yaw.yaw, map.yaw.force, attitudes(:, 5));
moment_yaw = interp1(map.yaw.yaw, map.yaw.moment, attitudes(:, 5));

% Sum the forces
force = (force_rhs + force_roll + force_steer + force_yaw - repmat(3 * map.bsl.force,
[szeAtt(1), 1]));
moment = (moment_rhs + moment_roll + moment_steer + moment_yaw -  repmat(3 * map.bsl.moment,
[szeAtt(1), 1]));
```

```
% Scale the matricies so that their dimensions will agree
% If there are more attitudes than velocities
if szeAtt(1) >= szeDyn(1)
    % Repeat the number of velocities required
    dynamic_pressure = repmat(dynamic_pressure, [szeAtt(1)/szeDyn(1), 3]);
else
    % Otherwise scale the attitudes we also need a velocity for each dim
    dynamic_pressure = repmat(dynamic_pressure, [1, 3]);
    force = repmat(force, [szeDyn(1)/szeAtt(1), 1]);
    moment = repmat(moment, [szeDyn(1)/szeAtt(1), 1]);
end

% Generate the output forces
force = force .* dynamic_pressure;
moment = moment .* dynamic_pressure;
end
```

9.6 Script to batch process figures from the aero map

```
init()

%% Read in the map data
load('Outputs/map.mat')

% This is the only line you need to change !!!
params = {map.roll, map.steer, map.yaw}; % Works with roll, steer, yaw

%% Loop
for a = 1:3
    param = params{a};

    %% Generate the calculated outputs
    cr = param.moment(:,2)./-1.535;
    cf = param.force(:,3)-cr;
    bal = cf./param.force(:,3) * 100;
    szeParam = size(param.force);
    bsl_force = repmat(map.bsl.force, [szeParam(1),1]);
    loss = (param.force - bsl_force)./bsl_force * 100;

    %% Get Parameter Names
    name = param.name;

    %% Generate Plots
    fig = initParams();
    plot(param.values, param.force(:, 3), 'rs:', 'MarkerSize',10, 'LineWidth',2)
    title(sprintf('%s Sensitivity Sweep of Lift', name.short))
    xlabel(sprintf('%s (%s)', name.long, name.units))
    ylabel('C_Z.A')
    saveas(fig,sprintf('Outputs/Figures/%s_Lift_sens.bmp', name.short));

    fig = initNewFigure();
    plot(param.values, param.force(:, 1), 'rs:', 'MarkerSize',10, 'LineWidth',2)
    title(sprintf('%s Sensitivity Sweep of Drag', name.short))
    xlabel(sprintf('%s (%s)', name.long, name.units))
    ylabel('C_X.A')
    saveas(fig,sprintf('Outputs/Figures/%s_Drag_sens.bmp', name.short));

    fig = initNewFigure();
    plot(param.values, bal, 'rs:', 'MarkerSize',10, 'LineWidth',2)
    title(sprintf('%s Sensitivity Sweep of Balance', name.short))
    xlabel(sprintf('%s (%s)', name.long, name.units))
    ylabel('Balance (%)')
    saveas(fig,sprintf('Outputs/Figures/%s_Balance_sens.bmp', name.short));


    fig = initNewFigure();
    plot(param.values, loss(:, 3), 'rs:', 'MarkerSize',10, 'LineWidth',2)
    title(sprintf('%s Sensitivity Sweep of Lift', name.short))
    xlabel(sprintf('%s (%s)', name.long, name.units))
    ylabel('% Loss to straightline')
    saveas(fig,sprintf('Outputs/Figures/%s_Lift_sens_percent.bmp', name.short));
end
```

9.7 Arduino Code for Pressure Array

```
#define N_TAPS 4
#define MAX_TAPS 24
#define MSG_SZE 8
#define BYTE_SZE 8
#define READ_SZE 2
#define RATE 10
```

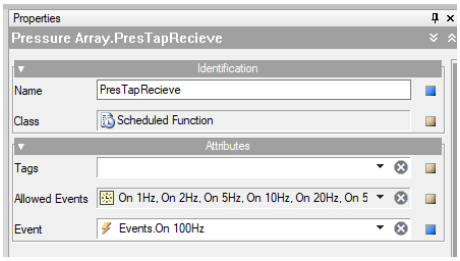
```
#define BASE_ADDRESS 0
#define ZERO_SAMPLES 1000
// Maximum rate is RATE * SAMPLES * N_TAPS < 34 752
#define SAMPLES 860

#include <EEPROM.h>
#include <FlexCAN.h>

// CAN Setup
// Create the message and the message mask
CAN_message_t msg;
CAN_filter_t defaultMask;

// Set the baud rate
uint32_t bd = 1000000;

// Set the Teensy Pins
uint8_t tx = 1; // 0 is 3, 1 is 29
uint8_t rx = 1; // 0 is 4, 1 is 30

// Set the CAN BUS id
uint16_t id = 0x600;

// Create the variables which hold the number of CAN messages and the number of taps per message
uint16_t n_msg = 0;
uint16_t n_taps = 0;

// General Setup
// Create the timer and the timing interval
uint64_t t = 0;
uint64_t t_i = 1000000/RATE;

// Performance timers
uint64_t t_sample;
uint64_t t_all;

// Create the led variable for flashing led during operation
int led = HIGH;

// Create an array to hold the pressure values
double data[MAX_TAPS];

// Set which ADC pin refers to which port
uint8_t adc_pins[] = {A0, A1, A2,
                      A3, A4, A5,
                      A6, A7, A8,
                      A9, A10, A11,
                      A12, A13, A14,
                      A15, A16, A17,
                      A18, A19, A20,
                      A21, A22, A23};

// Create arrays for the calibration values
double intercepts[MAX_TAPS];
double slopes[MAX_TAPS];

// Set the EEPROM memory locations
unsigned int int_address = BASE_ADDRESS;
unsigned int slp_address = BASE_ADDRESS + MAX_TAPS;

void setup() {

  // Set the led pin and switch it off
  pinMode(13, OUTPUT);
  digitalWrite(13, LOW);

  // Set the CAN adapter to on
  pinMode(28, OUTPUT);
  digitalWrite(28, LOW);

  // Start the Serial bus for debugging, zeroing and calibrating
  Serial.begin(9600);

  // Set the default mask values
```

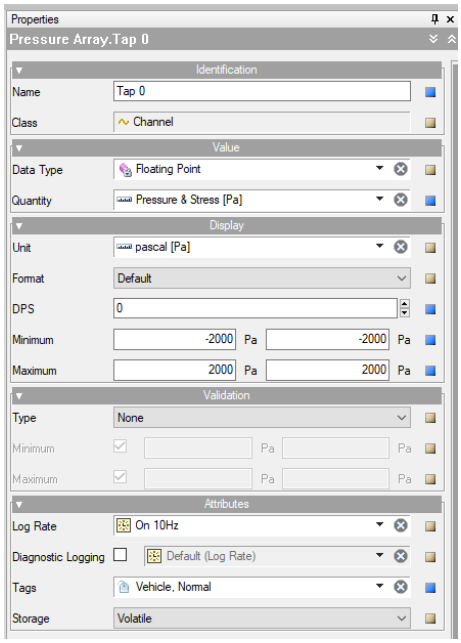
```
  defaultMask.flags.remote = 0;
  defaultMask.flags.extended = 0;
  defaultMask.id = 0;

  // Setup the CAN message with the data length and switch of extended id
  msg.len = 8;
  msg.ext = 0;

  // Start the CAN BUS
  Can0.begin(bd, defaultMask, tx, rx);

  // Read the calibration arrays from the memory
  EEPROM.get(int_address, intercepts);
  EEPROM.get(slp_address, slopes);

  // Set the slope calibration values (should be removed in next version)
  for (unsigned int i = 0; i < N_TAPS; i++)
  {
    slopes[i] = 0.050354;
  }

  // Set the analogue read resolution
  analogReadResolution(READ_SZE * BYTE_SZE);

  // Determine the number of taps in a CAN message
  n_taps = 64 / READ_SZE / BYTE_SZE;
  // Determine the number of CAN messages required for all the taps
  n_msg = N_TAPS / n_taps;

  // Set the first write time
  t = micros() + t_i;

  // Turn the led on
  digitalWrite(13, led);

}

void loop()
{
  // When the write time is exceeded
  if (t < micros())
  {
    // Time the write process
    t_all = micros();

    // Take a data sample
    sample(data, intercepts, slopes, N_TAPS, SAMPLES);
    t_sample = micros() - t_all;

    // Write the samples to the CAN BUS
    // Loop through the messages
    for (unsigned int i = 0; i < n_msg; i++)
    {
      // Loop through the taps within this message
      for (unsigned int j = 0; j < n_taps; j++)
      {
        // Determine the data point for this message and tap combination
        unsigned int ii = i * n_taps + j;

        // Retrieve and convert the data point
        uint16_t value =  data[ii] + 2000;

        // Split the data into two bytes
        msg.buf[2*j+1] = lowByte(value);
        msg.buf[2*j] = highByte(value);
      }

      // Set the message id and send it!
      msg.id = id + i;
      Can0.write(msg);
    }

    // Blink the led at each write
    led_blink();
```

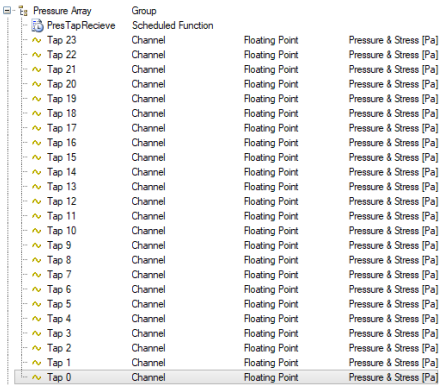
```
    // Set the next write time
    t += t_i;

    //Time the write process
    t_all = micros() - t_all;
  }

  // If there is a serial message this means we are debugging
  if (Serial.available() > 0)
  {
    // Set the led to on
    digitalWrite(13, led);

    // Read the command which is a sing/e character
    char cmd = Serial.read();

    // Determine task based on command
    switch (cmd)
    {
      // Zero command (z)
      case 'z':
        // Determine the intercepts to zero the data
        zero(intercepts, N_TAPS, ZERO_SAMPLES);
        // Save the intecepts to the EEPROM memory
        EEPROM.put(int_address, intercepts);
        // Don't break and take a sample as well

      // Sample command (s)
      case 's':
        // Take a sample
        sample(data, intercepts, slopes, N_TAPS, ZERO_SAMPLES);

        // Print it to the Serial Bus
        for (uint16_t i = 0; i < N_TAPS; i++)
        {
          uint16_t value = data[i] + 2000;
            Serial.print(value);
            if (i < N_TAPS - 1)
            {
              Serial.print(',');
            }
        }

        Serial.println();
        break;

      // If debugging timer (t)
      case 't':
        Serial.print("Rquired time(ms) : ");
        Serial.println(int(t_i));
        Serial.print("Total time(ms) : ");
        Serial.println(int(t_all));
        Serial.print("Sample time(ms) : ");
        Serial.println(int(t_sample));
        break;

      // Message debugging (d)
      case 'd':
        Serial.print("Number of messages : ");
        Serial.println(n_msg);
        Serial.print("Number of taps per : ");
        Serial.println(n_taps);
        break;

      // Return and error (e) if the command is not recognised
      default:
        Serial.println('e');
    }

    // Clear the serial memory
    while (Serial.available() > 0)
    {
      Serial.read();
```

```
    }

    // Return to normal operation
    // Set the led back to blinking position
    digitalWrite(13, led);

    // Set the first write time
    t = micros() + t_i;
  }
}

/************ FUNCTIONS ************/
void led_blink()
{
  if (led == HIGH)
    led = LOW;
  else
    led = HIGH;

  digitalWrite(13, led);
}

void zero(double *intercepts, unsigned int n_taps, unsigned int samples)
{
  for (unsigned int i = 0; i < N_TAPS; i++)
  {
    intercepts[i] = 0;
  }

  for (unsigned int s = 0; s < samples; s++)
  {
    for (uint16_t i = 0; i < N_TAPS; i++)
    {
      uint8_t pin = adc_pins[i];
      intercepts[i] += double(analogRead(pin)) / samples;
    }
  }
}

void sample(double *data, double *intercepts, double *slopes, unsigned int n_taps, unsigned int samples)
{
  for (unsigned int i = 0; i < N_TAPS; i++)
    {
      data[i] = 0;
    }

  for (unsigned int s = 0; s < samples; s++)
  {
    for (unsigned int i = 0; i < N_TAPS; i++)
    {
      uint8_t pin = adc_pins[i];
      data[i] += ((double(analogRead(pin)) - intercepts[i]) * slopes[i]) / samples;
    }
  }
}
```

9.8 M1 Build Setup for Pressure Array







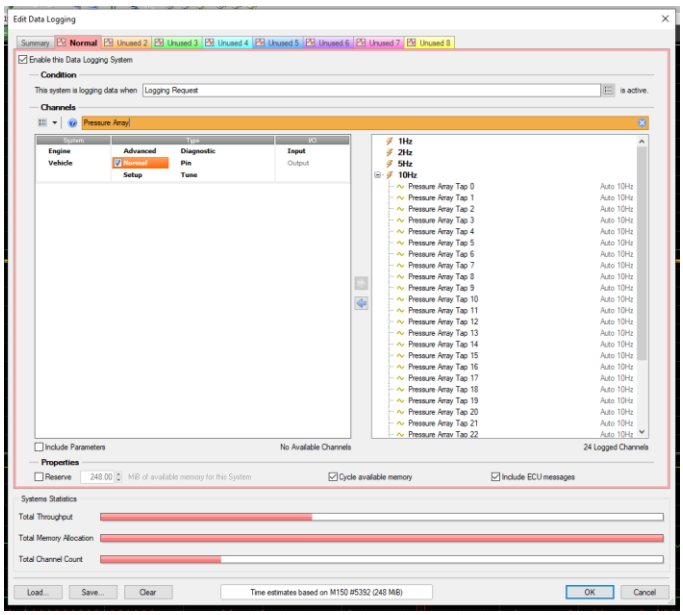


Figure 32 Shows the MoTec M1 Build and Tune parameters.

```
local PresTapCAN0 = CanComms.RxOpenStandard(0, 0x700, 0x0, true);
local PresTapCAN1 = CanComms.RxOpenStandard(1, 0x700, 0x0, true);
if(CanComms.RxMessage(PresTapCAN0))
```

```
{
        local value = CanComms.GetUnsignedInteger(PresTapCAN0, 0, 16);
        Tap 0 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 16, 16);
        Tap 1 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 32, 16);
        Tap 2 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 48, 16);
        Tap 3 = value - 2000.0;
}
else if(CanComms.RxMessage(PresTapCAN1))
{
        local value = CanComms.GetUnsignedInteger(PresTapCAN1, 0, 16);
        Tap 0 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 16, 16);
        Tap 1 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 32, 16);
        Tap 2 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 48, 16);
        Tap 3 = value - 2000.0;
}

PresTapCAN0 = CanComms.RxOpenStandard(0, 0x701, 0x0, true);
PresTapCAN1 = CanComms.RxOpenStandard(1, 0x701, 0x0, true);
if(CanComms.RxMessage(PresTapCAN0))
{
        local value = CanComms.GetUnsignedInteger(PresTapCAN0, 0, 16);
        Tap 4 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 16, 16);
        Tap 5 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 32, 16);
        Tap 6 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 48, 16);
        Tap 7 = value - 2000.0;
}
else if(CanComms.RxMessage(PresTapCAN1))
{
        local value = CanComms.GetUnsignedInteger(PresTapCAN1, 0, 16);
        Tap 4 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 16, 16);
        Tap 5 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 32, 16);
        Tap 6 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 48, 16);
        Tap 7 = value - 2000.0;
}

PresTapCAN0 = CanComms.RxOpenStandard(0, 0x702, 0x0, true);
PresTapCAN1 = CanComms.RxOpenStandard(1, 0x702, 0x0, true);
if(CanComms.RxMessage(PresTapCAN0))
{
        local value = CanComms.GetUnsignedInteger(PresTapCAN0, 0, 16);
        Tap 8 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 16, 16);
        Tap 9 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 32, 16);
        Tap 10 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 48, 16);
        Tap 11 = value - 2000.0;
}
else if(CanComms.RxMessage(PresTapCAN1))
{
        local value = CanComms.GetUnsignedInteger(PresTapCAN1, 0, 16);
        Tap 8 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 16, 16);
        Tap 9 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 32, 16);
        Tap 10 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 48, 16);
        Tap 11 = value - 2000.0;
}

PresTapCAN0 = CanComms.RxOpenStandard(0, 0x703, 0x0, true);
PresTapCAN1 = CanComms.RxOpenStandard(1, 0x703, 0x0, true);
```

```
if(CanComms.RxMessage(PresTapCAN0))
{
        local value = CanComms.GetUnsignedInteger(PresTapCAN0, 0, 16);
        Tap 12 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 16, 16);
        Tap 13 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 32, 16);
        Tap 14 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 48, 16);
        Tap 15 = value - 2000.0;
}
else if(CanComms.RxMessage(PresTapCAN1))
{
        local value = CanComms.GetUnsignedInteger(PresTapCAN1, 0, 16);
        Tap 12 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 16, 16);
        Tap 13 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 32, 16);
        Tap 14 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 48, 16);
        Tap 15 = value - 2000.0;
}

PresTapCAN0 = CanComms.RxOpenStandard(0, 0x704, 0x0, true);
PresTapCAN1 = CanComms.RxOpenStandard(1, 0x704, 0x0, true);
if(CanComms.RxMessage(PresTapCAN0))
{
        local value = CanComms.GetUnsignedInteger(PresTapCAN0, 0, 16);
        Tap 16 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 16, 16);
        Tap 17 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 32, 16);
        Tap 18 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 48, 16);
        Tap 19 = value - 2000.0;
}
else if(CanComms.RxMessage(PresTapCAN1))
{
        local value = CanComms.GetUnsignedInteger(PresTapCAN1, 0, 16);
        Tap 16 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 16, 16);
        Tap 17 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 32, 16);
        Tap 18 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 48, 16);
        Tap 19 = value - 2000.0;
}

PresTapCAN0 = CanComms.RxOpenStandard(0, 0x705, 0x0, true);
PresTapCAN1 = CanComms.RxOpenStandard(1, 0x705, 0x0, true);
if(CanComms.RxMessage(PresTapCAN0))
{
        local value = CanComms.GetUnsignedInteger(PresTapCAN0, 0, 16);
        Tap 20 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 16, 16);
        Tap 21 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 32, 16);
        Tap 22 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN0, 48, 16);
        Tap 23 = value - 2000.0;
}
else if(CanComms.RxMessage(PresTapCAN1))
{
        local value = CanComms.GetUnsignedInteger(PresTapCAN1, 0, 16);
        Tap 20 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 16, 16);
        Tap 21 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 32, 16);
        Tap 22 = value - 2000.0;
        value = CanComms.GetUnsignedInteger(PresTapCAN1, 48, 16);
        Tap 23 = value - 2000.0;
}
```