# Integrating Temperature in the Dynamo Engine Map & Automating the Tuning Process for Stable Improvements of Fuel Efficiency

Irene Marie Malmkjær Danvy (s163905)
Frederik Ettrup Larsen (s163920)
Bachelor of Science in Engineering
2019

DTU Electrical Engineering
Department of Electrical Engineering

**Integrating Temperature**
**in the Dynamo Engine Map**
**& Automating the Tuning Process**
**for Stable Improvements of Fuel Efficiency**

**Report written by:**
Irene Marie Malmkjær Danvy (s163905)
Frederik Ettrup Larsen (s163920)

**Advisor(s):**
Søren Hansen
Claus Suldrup Nielsen

**DTU Electrical Engineering**
Technical University of Denmark
2800 Kgs. Lyngby
Denmark

elektro@elektro.dtu.dk

Project period: 4. February 2019- 14. July 2019

ECTS: 15

Education: B.Sc.

Field: Electrical Engineering

Class: Public

Edition: 1. edition

Remarks: This report is submitted as partial fulfillment of the requirements for graduation in the above education at the Technical University of Denmark.

# Summary

We conducted this project in the context of DTU Roadrunners, a team that builds, maintains, and improves an energy-efficient car, Dynamo, and that has repeatedly won its category in the Shell Eco-marathon Europe, including the last two years. For these last two years, the engine control has been manual and imprecise. This project aims to improve this control. We improved engine map accuracy by adding a temperature dimension to two lookup tables, which had two consequences, a technical one and a practical one. Technically, the extension made it possible for us to automate and simplify the process of tuning the engine map. And practically, the resulting engine map was more accurate than all of its previous versions, making the tuning significantly more stable. These changes allowed the engine to be tuned several days before a race, which is a significant improvement over having to do it at the last minute in the queue before each race. Our engine control solution proved superior to the previous one, as the car drove more efficiently at the Shell Ecocar Marathon than last year, in part due to our contribution.

# Preface

This bachelor report was prepared at the department of Electrical Engineering at the Technical University of Denmark in fulfillment of the requirements for acquiring a Bachelor of Science degree in Electrical Engineering.

# Acknowledgements

We would like to thank all our proofreaders: Ture Larsen, Olivier Danvy, Karoline Malmkjær, Ella Grandjean and Oda Byskov Aggerholm, for their outstanding performance under uncalled-for time pressure.

We would also like to thank our advisors Søren Hansen and Claus Suldrup Nielsen, for their boundless patience and willingness to answer sometimes repetitive questions.

This project was conducted as a part of the work of DTU Roadrunners, who are generally wonderful people to work with, and surprisingly amicable about the electrical engineers breaking the mechanical aspects of the car and not being able to fix it again. We would specifically like to thank our driver Sarah, team manager Christoffer and pit boss Tobias, without whom the trip to London certainly would not have gone as well as it did. Special thanks are also owed to our follow Roadrunner, Simon Friborg Mortensen, who discovered a two year old error in the $\lambda$ measurements making our job significantly easier, and whose mechanical input and patient answers proved invaluable to our final design, as well as former Roadrunner Henning Si Høj, for much needed guidance and assistance in the expansion of the ECU UI, as well as his prompt responses to any questions and pleas for help, despite living in Switzerland for the last year.

Finally we would like to thank Ella and Oda, for putting up with our lack of free time in the last couple of months. We promise to try spending less time playing with engines going forward.

# Acronyms

| Acronym | Meaning |
|---|---|
| AF | Air/Fuel Ratio |
| CANbus | Controller Area Network bus. |
| CFD | Computational Fluid Dynamics |
| COM-port | Communication port |
| DTU | Technical University of Denmark |
| DWC | Drivers World Championship (a competetion at SEM) |
| ECU | Engine Control Unit |
| EEPROM | Electrically Erasable Programmable Read-Only Memory |
| ICE | Internal Combustion Engine |
| INJ | Injection length |
| IGN | Ignition time |
| KERS | Kinetic Energy Recovery System |
| LHV | Lower Heating Value[ $\frac{\text{kJ}}{\text{kg}}$] |
| LOLIMOT | Local Linear Model Trees |
| LOPOMOT | Local Polynomial Model Trees |
| LUT | Lookup Table |
| NI | National Instruments |
| OLED | Organic Light-Emitting Diode |
| PC | Personel Computer |
| RAM | Random Access Memory |
| cRIO | compact Reprogrammable In/Out (industrial controllers, made by NI) |
| RPM | Rotations Per minute (The unit of the rotational crank speed) |
| RS232 | A communication protocol. |
| SD card | Secure Digital card |
| SEM | Shell Eco-Marathon |
| UI | User Interface |
| USB | Universal Serial Bus |

Table 1: Glossary of the acronyms used in this project.

# Contents

# List of Figures

# List of Tables

# CHAPTER 1

# Introduction

The student-driven project DTU Roadrunners builds and maintains the fuel-efficient ICE car, Dynamo, and competes with it in the Shell Eco-Marathon. Internal combustion engines are extremely delicate machines, whose functioning is highly dependant on environmental factors such as temperature and pressure in and around the engine. Therefore accurate models and precise control strategies are necessary for the engines to run smoothly. Over the last seven years, the engine control for Dynamo has gone through many different iterations, all of which had an engine map consisting of three one-dimensional lookup tables. This model was tuned by manually adjusting the values in these tables. The engine map did not describe the engine very precisely in conditions different than ones the tuning was performed in. To make up for this, tuning was done as closely as possible to a race, in order for the map to reflect the conditions on track at least semi-accurately. Until 2016, this process was assisted by a very slow form of closed loop control, adjusting the values in the map while the car was being driven.

Our project has two main goals: Firstly to design a system which can automatically tune the LUTs in order to avoid the slow process of adjusting the values by hand. Secondly to expand the engine model in such a manner that the car will perform well in the Shell Eco-Marathon 2019, without live control and without having to tune the car in the queue before each race.

To describe how these goals will be achieved in more detail, our problem statement is as follows:

## 1.1 Problem Statement

Currently, DTU Roadrunners' energy efficient car has no system that ensures that the engine operates optimally, except for a simple LUT, which is adjusted by hand. A new engine map and system for automatically tuning this engine map will be designed. The system should fulfill the following requirements:

- When tuned, the air/fuel ratio measured in the exhaust should remain steady across different operating conditions.
- At the very least, it must be able to automatically tune the car as well as when the car is tuned by hand.

- It must have a user interface, allowing the operator to monitor and control the process.
- The tuning process must be reliable.
- It must be possible to change the parameters of the tuning, for instance to tune the engine to run either lean or far.
- It must be possible to overwrite the result of the tuning by hand.

## 1.2 Roadmap

This report is structured as follows.

**Chapter 2** The project's physical context.

**Chapter 3** The project's theoretical context.

**Chapter 4** The design, implementation and testing of the first concept, as well as a discussion of the results and revision of the first concept.

**Chapter 5** The design, implementation and testing of the revised concept, as well as a discussion of the new results.

**Chapter 6** The conclusion.

**Appendices** In-depths descriptions of the old UI, the new UI, the final software implementation of the project, and rule details.

# CHAPTER 2

# The Team, The Car, The Competition

## 2.1 Shell Eco-Marathon

The Shell Eco-Marathon is an annual competition, held by Royal Dutch Shell plc. They invite teams from high schools and universities world wide to build cars in accordance with their rule set, travel to one of their events, complete their technical inspection and compete against each other.

There are six different categories: the cars can be either Battery Electric, Hydrogen or ICE and either Prototype or Urban Concept. For each category the goal is the same: to drive as efficiently as possible within the constraints of the competition.

DTU Roadrunners is the team from the Technical University of Denmark. We drive in the urban concept internal combustion engine category, with ethanol as our fuel supply. In order to make the results comparable, all energy and fuel used in the ICE has its equivalent volume of gasoline calculated. That's how teams like DTU Roadrunners, who use ethanol as a fuel, can compete in the same category as teams like Lycée Louis Delange who use diesel as fuel. Both team's cars use internal combustion engines. Examples of the conversion from ethanol to gasoline can be seen in appendix A.

The best three cars in each of the three Urban Concept categories, as well as the invited Urban Concept winners from the other Shell Eco-Marathons held around the world, qualify for an extra competition: the Driver's World Championship. In this competition the cars are given a "budget" of fuel corresponding to what was used in the mileage competition, and must then race each other. This competition allows the cars from the different engine categories to race each other, and it is as common for cars to lose due to using up their fuel budget before completing the race as it is too lose due to simply not coming first.

Shell Eco-Marathon Europe 2019 was held in London from the 2nd to 5th of July [Ecoa][Ecob].

## 2.2 DTU Roadrunners

DTU Roadrunners is a student driven group, formed with the purpose of participating in and winning the Shell Ecocar Marathon. We participate in the Autonomous Challenge, the Mileage Challenge in the category of Urban Concept ICE, and the DWC if we succeed in the Mileage Challenge.

In order to do this we design, build, maintain and improve an energy efficient car that we have named the DTU Dynamo.

## 2.3 Description of Dynamo

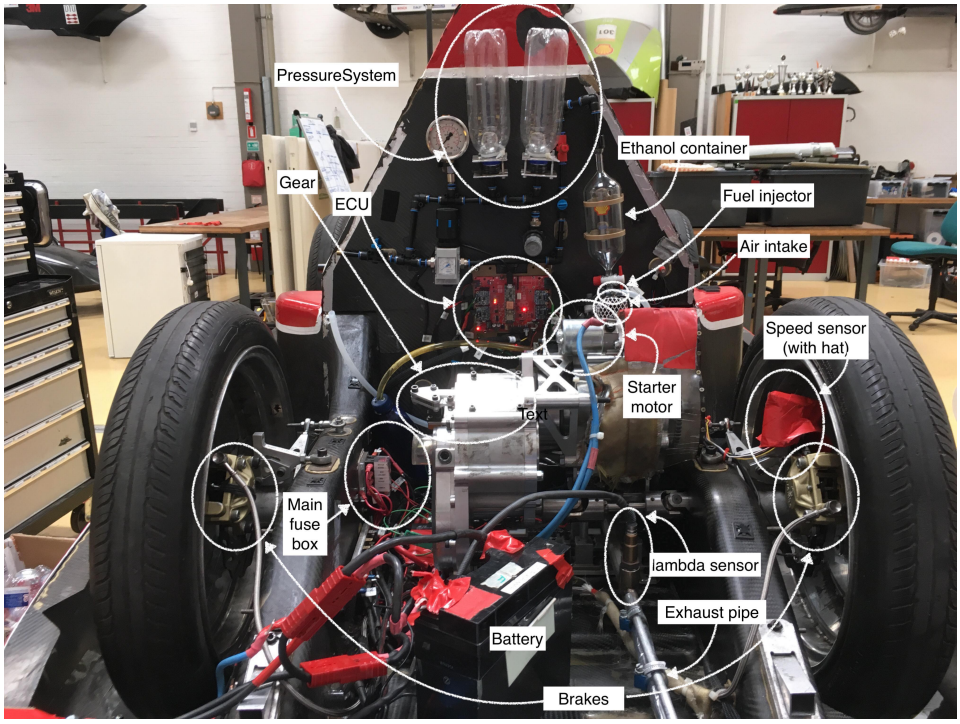
Dynamo is a very small car, that is built to (1) conform to the rules for competing vehicles in the Shell Eco-Marathon and (2) be as fuel and energy efficient as the DTU Roadrunners team can make it while still conforming to (1).

### 2.3.1 A Basic Description of the Car

The car weighs 128 kg, is 1.30 m wide, 3.30 m long and 1.10 m high. The engine provides torque for the back right wheel, there are brakes mounted on all four wheels, two gears and no reverse gear. The car in its entirety can be seen in figure 2.2, and opened up and labeled versions can be seen in figure 2.1.

As this car must pass the requirements to participate in the Urban Concept category of the Eco-Marathon, it must, among other things:

- Have exactly four wheels in constant contact with the road.
- Have idling capabilities.
- Have two side doors out of which the driver can exit in less than 10 s.
- Have a seat allowing the driver to sit upright.
- Have a windscreen wiper.
- When driving in the mileage challenge the car is also required to make a full stop once each lap (1.4km), in order to simulate driving in a city.
- Have front and back lights, blinking capabilities, brake lights and hazard lights.




(a) Dynamo from the back. Note that the engine only provides torque to the right back wheel.


(b) Dynamo from the side. Note that it's standing on the Chassis Dynamometer.

Figure 2.1: Dynamo from the back 2.1a and side 2.1b with top shell removed and elements of interest labelled.


Figure 2.2: Dynamo, assembled and on track at Shell Eco-Marathon 2019.

### 2.3.2 The Engine

The car runs on a heavily modified Yamaha Nios 64 47cc four stroke moped engine from 2008. The primary modification was to convert the engine from using gasoline to using bio-ethanol as fuel.

This engine has two control vectors: ignition and fuel-injection.

Ignition is controlled purely in terms of timing relative to the engine cycle.

The injection valve only has two settings, open and closed. Therefore injection is controlled in two ways: in terms of injection timing relative to the engine cycle and in terms of injection length, by which is meant the amount of time the valve is kept open.

### 2.3.3 The Electrical System and Electronics

The car is outfitted with a 12V car battery, which supplies the starter engine, the sensors, the lights, the wiper and the four control units in the car.

#### 2.3.3.1 The Sensors

The engine sensors are: Three encoders, a $\lambda$ sensor, and four temperature sensors.

One encoder is industrial, mounted on the crank shaft, used for determining engine cycle position, and will be described below. The other two encoders (one infra red and one inductive) are mounted on the wheel and measure speed. The temperature sensors measure the exhaust air temperature, intake air temperature,[1] engine oil temperature and the temperature of the water in the cooling system.

As the water temperature sensor, industrial encoder and $\lambda$ sensors are the three most used sensors in this project, they will be described in slightly more depth now.

The temperature sensor, like the other temperature sensors in the car, is an Adafruit Universal Thermocouple Amplifier MAX31856 Breakout. The MAX31856 allows readings as high as +1800°C and as low as -210°C. It has 19-Bit, 0.0078125°C thermocouple temperature resolution. In the range $-20$°C to 85°C it has internal cold-junction compensation and can measure thermocouple voltage with a $\pm 0.15\%$ accuracy[Max15]. Please note that we will use as high as 90°C, which is slightly outside the sensors optimal operating range, but given that we work in $5^o$C intervals, that shouldn't have a significant effect on our results.

The engine encoder is mounted on the crankshaft, and is a Type SCA50 from Scancon Industrial Encoders. It is optical and provides a resolution up to 12500 pulses per revolution,[Sca14] see figure 2.3. It's signals are interpreted by an inbuilt decoder in the Teensy 3.6, only triggering interrupt routines when relevant[**MotorStyring2018**].

The $\lambda$ sensor is mounted on the exhaust pipe and measures the excess air ratio (also called the air–fuel equivalence ratio) in the exhaust gas, meaning the ratio of the actual fuel air ratio to the fuel air ratio for a chemically correct mixture of fuel and air, also called the stoichiometric mixture. Under stoichiometric conditions the mixture is such

[1] Intake air temperature sensor was only mounted for tests, as it was placed in the air filter which had to be removed during competition.

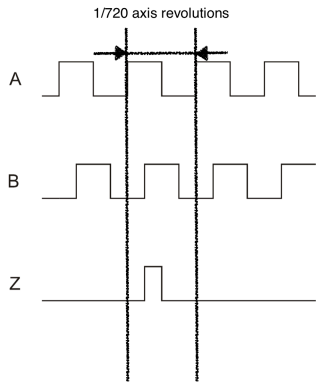



Figure 2.3: The outputs of an axle encoder. The period of the A- and B-pulses correspond to $\frac{1}{720}$'th of a crank shaft revolution, or half a degree of change. The Z-pulse's period corresponds to an entire crank shaft revolution, meaning there is one Z-pulse, whose signal is high for half as long as an A- or B-pulse, every complete rotation.

that the all of the air and fuel will react.[Sor11].

$$\lambda = \frac{AF}{AF_{stoch}} = \frac{\frac{fuel\,mass}{air\,mass}}{\left(\frac{fuel\,mass}{air\,mass}\right)_{stoch}} \tag{2.1}$$

If $\lambda > 1$ the mixture is said to be lean as there is excess air, and if the $\lambda < 1$ the mixture is said to be fat, as there is excess fuel.

$\lambda$ sensors exist in two kinds, narrow band, which can only tell you if there is a balanced, rich, or lean mixture, and wide band, which can measure the $\lambda$ value in its entire range of operation.

A narrow band sensor works by placing a ceramic membrane between the exhaust and the atmospheric air, and heating up this membrane until ions can pass through it. These ions create a voltage over the membrane, which is amplified and acts as the sensor's output, how many ions pass through from the atmosphere into the exhaust, and thus the output, depends on the fuel/air equivalence ratio.

A wide band sensor works in a similar manner, but has a secondary ceramic membrane called a pump cell. Between the two membranes is a measurement chamber. Oxygen-ions defuse between the measurement chamber and the atmosphere, while a voltage is applied to the pump cell, making ions diffuse through it in the direction which causes the measurement chamber to maintain stoichiometric conditions. The voltage applied over the pump cell is converted to the sensor's output[al00].

Wide band sensors give more information than narrow band sensors, but must also be calibrated more often. The wide band $\lambda$ sensor used in Dynamo is an Innovate enginesports LC-2 Digital Air/Fuel Ratio ($\lambda$) Sensor with a range from 0.8-1.51, whose manual suggest calibrating the sensor before every race if used by a race car, and once every three months if used in a commercial car[Inn].

In commercial cars the catalytic exhaust gas treatment functions best at $\lambda = 1$, which is why most commercial $\lambda$-based control systems have the goal of keeping $\lambda$ about 1[Ise12]. Dynamo is not subject to emission regulations, and therefore doesn't have any exhaust treatment. Our goal is to keep the $\lambda$ steady at the value for which the car runs most efficiently. This has previously been determined to be 1.3.

It should be noted that the $\lambda$ sensor measures the air/fuel ratio at the point it is in, regardless of if any combustion has happened in the engine. If ethanol is poured down the exhaust, the $\lambda$ sensor would think the engine was running very fat. Likewise it should be noted that our sensor is mounted inside the engine room, therefore the "atmospheric" air the exhaust is being compared to might not be particularly clean after driving for 39 minutes (the maximum length of a valid run according to Eco-Marathon rules[Ecob]). Lastly, it should be noted that while very useful for tuning, the sensor is usually removed during races, as it accounts for roughly half the car's electric power use, which counts towards energy consumption, see appendix A.

#### 2.3.3.2 The Control Units

There are four control units in the car: the front light board, the back light board, the ECU and the steering wheel. Each board is a custom PCB print, and has a Teensy 3.6 microcontroller (see table 2.1 for specifications). These boards communicate using an implementation of FlexCan for Teensy and Arduino.

The front and back light boards control the lighting and in the case of the front light board also the wind screen wiper's actuator.

The steering wheel board takes input from the driver and transmits it to the other boards, and gets sensor data from the ECU which is displayed to the driver on a small OLED screen. It also keeps track of the current time, which is also displayed on the OLED screen.

The ECU interprets the output of every sensor and communicates the results over CANBUS. It uses the sensor data to control the ignition and injection timing, as well as injection length, based in the engine's position in the combustion cycle as interpreted from the industrial encoder data.

It also logs the sensor and driving data on a mini-SD card and can communicate over serial with the UI in Json format. The ECU is the board that previously contained the old engine map, and now contains the new engine map as well as the tuning program designed and implemented in this thesis.

### 2.3.4 Some Things Most People Might Expect a Modern Car to Have that the Ecocar Does Not

Just as the engine has no variable fuel injection, it also doesn't have air intake control, nor does it have flow-meter to measure air intake volume.[2] Therefore the car also has no gas pedal. The engine is either running at full capacity or not running at all. It is, as per Shell Ecomarathon regulations, controlled by a dead man switch on the steering wheel, or from the UI if a PC is connected.

The engine doesn't have a knock sensor, any pressure sensors, nor fuel or oil level sensors, nor an oil flowmeter. It has neither an oil filter, nor an air filter, as the car (perhaps somewhat in contradiction of the state purpose of the Shell ecomarathon) doesn't have to comply with any emission regulations.

The car has no ventilation and very limited shock absorbance. The shell is designed to be light and aereodynamic, and as such doesn't have any built in driver protection should it crash, aside from seatbelts and a helmet for the driver, as required by Shell.

[2]The strategy of taking in as much air as possible, described in section 2.4.3, would be hampered by an attempt at either of these. A prototype flow meter was built in 2013, as part of a larger project, but it was never directly implemented[Dag13]. The author of that project does conclude that it can be helpful for engine mapping, but unfortunately that prototype has been lost to time since then. See section 3.4.2 for details on that project.

| **Teensy 3.6** |
|---|
| MK66FX1M0VMD18, core with an 180MHz, 32 bit Cortex-M4F processor |
| 1024 kbytes flash memory |
| 256 kbytes RAM |
| 4096 bytes EEPROM |
| 58 pins |
| 32 channels for direct memory access |
| 19 timers |
| Options for USB, I2C, SPI, CAN-bus, ethernet and SD card. |

Table 2.1: Technical specifications of the Teensy 3.6 [Sto].

## 2.4 Tools and Strategies for the Car

### 2.4.1 The ECU UI

The ECU UI is a piece of software that can communicate with the ECU. If connected it can be used to control the car, tune the car and change relevant software constants in the ECU, such as the wheel diameter used to calculate speed or the content of the engine maps.

The UI has the tabs: Cockpit, Configuration, LUT, Test, Charts, Debug, Log, CAN, RS232, Encoder and Tune.

The tabs LUT and Tune were added as part of this project, with the goal to support the extended engine map and the new tuning process. They are described in appendix D.

The tabs from the original UI, along with the rest of the visual interface, are described in appendixC.

The UI was originally written in JavaFX in the fall of 2018. It can be run on any computer with a Windows operating system and connects to the ECU with USB Serial communication through a COM-port.

The communication over serial is written in a simplified version of Json, with typically only one object per array, except when transmitting engine maps. In that case each element of the map is made into a single object with attributes rpm, temperature, ignition timing, injection length in first gear and injection length in second gear, and the entire map is sent as an array containing all these objects. An object in such an array has the form {”RPM”: 2500, ”TEMP”: 65, ”IGN”: 20, ”INJ1”: 5091, ”INJ2”: 6800}.

### 2.4.2 The Chassis Dynamometer

The Chassis Dynamometer, sometimes called the Rolling Road, is not a part of the car, but is extremely useful. It is a mount for Dynamo, allowing the team to simulate driving conditions. Please see figure 2.4. It is typically used to test driving strategies and heating the engine while in London, and was used extensively for tuning in this project. It is essentially a large table with a big drum on it. The drum can spin with a resistance that matches the friction of the car driving on an asphalt road. It should be noted that this friction has not been calibrated for a few years. Therefore the results on the Rolling Road might vary from results on a real road.

### 2.4.3 The Driving Strategy

The team's driving strategy is called "coast and burn", and is pretty common for ICE vehicles in the Shell Eco-Marathon. The principle is to run the engine as little as possible, meaning that two or three times per lap the driver accelerates up to around 35 $\frac{\text{km}}{\text{h}}$, then decelerates down to around 22 $\frac{\text{km}}{\text{h}}$ and then accelerates up again. This means that our average speed is around 25 $\frac{\text{km}}{\text{h}}$. Using this strategy, the car drove 374 $\frac{\text{km}}{\text{l}}$ in 2018 and 429 $\frac{\text{km}}{\text{l}}$ in 2019.

The two main contributing factors to Roadrunner's success is: How well the car rolls when no torque is applied from the engine, and how effectively it accelerates.

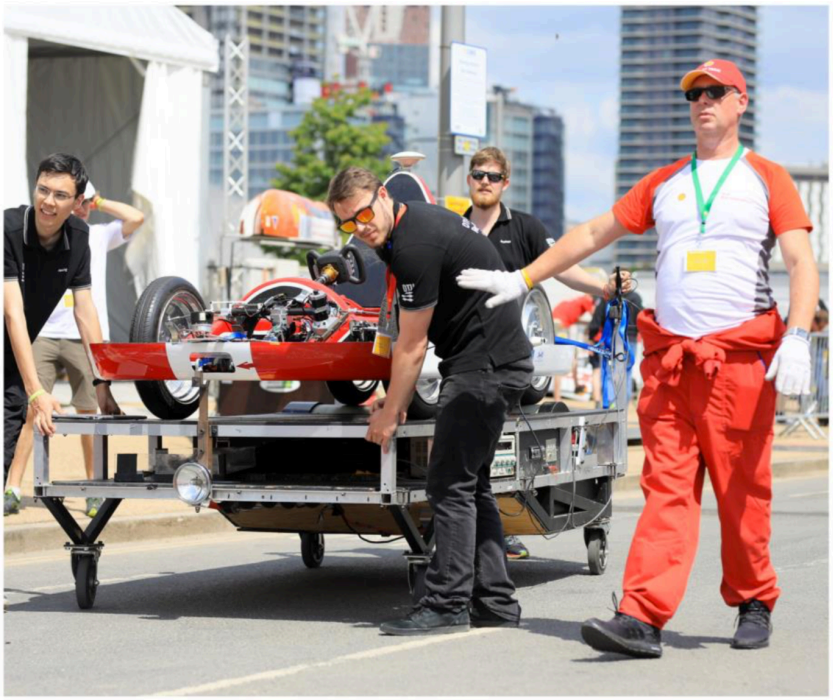


Figure 2.4: The Chassis Dynamometer, with Dynamo mounted, being pulled to the track by DTU Roadrunner members at Shell Eco-Marathon Europe 2016.

This strategy has the consequence that the engine only ever runs at full acceleration, or not at all.

## 2.5 The Engine Map and the Manual Car Tuning

Until 2018 the engine map for Dynamo has consisted of three one dimensional arrays: One for ignition times, and two for injection times (one for each gear). All of these arrays are indexed by RPM, with values going from 0 to 6000, each index corresponding to 250 RPM wide bucket. It should be noted that the car usually operates within the 1750 to 4000 interval. When the car is driving, the ECU constantly calculates the relevant ignition and injections times, by linear interpolation between the two RPM values closest to the current RPM. The array for ignition is fairly stable, and thus is rarely tuned, which is why we have generally ignored it in this project. The two arrays for injection, however, are quite unstable, as the ideal injection times changes with a lot of factors such as temperature or humidity. For this reason the injection arrays have to be tuned regularly.

The process of tuning a car is the process of adjusting all of the values in the engine map to optimize for some output of the engine. This might mean more speed, more torque, or as is our case, better fuel efficiency.

The process of tuning Dynamo before this project was as follows:

- Either mount the engine on a test stand, or mount the car onto the Chassis Dynamometer.
- The engine ”burns” (accelerates) from 0 to the maximum RPM in second gear.
- The operator looked at the $\lambda$ values and adjust the injection times, this follows a simple rule of thumb: If $\lambda$ is below the desired value for a given RPM interval, it means that the there is going too much fuel into the engine, so the injection time will have to be decreased. If $\lambda$ is above the desired value, it means that there is not enough fuel going into the engine, so the injection time will have to be increased. The operator repeats this until they are satisfied with the $\lambda$ values for each RPM interval. How much the injection times has to be increased or decreased is generally determined by the intuition and experience of the operator.
- The operator then burns the engine again, to see if the $\lambda$ values are acceptable, if not, the injection times are adjusted again.

## 2.6 Other Work Done on Dynamo in 2019

Aside from our project, other electrial and mechanical engineering students have also been working on the car this year. Here is list of their work:

- Mechanical
  - New shell
    A new carbonfiber shell has been made for the car, this was mostly done to comply with the rule, that the car needs two doors.
  - New engine
    A new engine has been developed and tested over the last couple of years. It should be more effecient than our current engine, we did however not get to test the engine on track this year.
- Electronic
  - New steering wheel
    A new steering wheel has been made, as the previous steering wheel was too big, used to much power and froze when the horn was used.
  - New batteries
    The current use of the car has been measured, based on these measurements it was possible to change the old heavy battery to a much lighter battery.
  - Preliminary work for a KERS
    All of the theoretical work for a kinectic energy recovery system has been done, this should in theory improve our mileage by 26km/l. The system has however not been implemented this year.
  - Autonomous driving system
    This year the car also participated in the autonomous challenge, which we unfortunately did not win. In preperation for this the autonomous work done last year has been extended.

It should be noted, that our project is the only project which had a significant impact on the mileage of the car this year.

# CHAPTER 3

# Engine Models and Control Strategies in Industry and for DTU Roadrunners

## 3.1 A Very Brief Historical Overview of Electronics and Mechatronics in Commercial Cars

Ever since electronic sensors and control vectors have been put into cars, electronic control has been necessary.[Ise12][Rol16]

Until the 1960s, gasoline engines were controlled mechanically, with transistor-triggered electromagnetic coil ignition. Throughout the 60s and 70s, an increasing amount of electric and mechatronic systems were introduced into engines of consumer cars, such as electric fuel pumps, electrically controlled carburetors, electronically controlled fuel injection, knock sensors and knock control, $\lambda$ sensors and $\lambda$-control. By the early 80s, most cars had two to five manipulated variables controlled with microprocessors. Throughout the late 80s, 90s and 00's, the existing electrical systems were joined by on-board diagnostic systems, CANbus communication system[1], electronic throttle control[2], valve timing control, electromagnetic injectors, superchargers, turbochargers[3], and more advancements have been made since then.

[1]The communication system used in modern cars, which allows a single net with a single wire going to each control unit, instead of having dedicated communication wires between every control unit that needs to communicate. Information can be published and pulled by each unit, with every unit simply repeating all information in the network with a certain frequency to maintain a constant stable information stream.[Cha]

[2]Electronic throttle control is the system connecting the accelerator pedal to the throttle. Its aim is to make the use of the accelerator pedal consistent irrespective of other conditions affecting the engine, such as temperature, added weight to the car in the form of passengers and luggage or altitude. Typically, an electric motor controls the throttle and is in turn controlled by an electronic control unit, based on accelerator pedal position, weather, load, etc.

[3]Super and turbochargers are air compressors, increasing the density of air supplied to the combustion process, thus supplying more oxygen to the reaction.

Advancement in control of ICE cars are driven both by opportunity, in form of technological advances, and by necessity, in the form of new emission regulations. The increase in electronically controlled functions necessitated a corresponding increase in sensors, activators and control units, which in turn necessitates improved communication protocols, mainly CANbus in the case of cars.

Most gasoline engines built around 2014 had 15-25 sensors, 6-8 main manipulated variables and powerful micro-controllers with 80-120 look-up tables and many control algorithms.

## 3.2 A Brief Overview of Engine Models and Control

Based on [Ise12] and [Rol16]. The first step in designing a control scheme for a system, is to model the system, based on this model, the control scheme can be designed. Which control system can be executed is dependent on the nature of the system model. Therefore we will now consider some simple common engine models, and which models can be used based on said models.

### 3.2.1 Simple Engine Models

Modelling an engine can be done in two main ways: Theoretically and experimentally. When modelling any system, one can talk about a white-box, black-box, and gray-box models. White-box models are completely theory based, black-box completely experiment based, and gray-box are a combination of the two. How light or dark gray a given "box" is depends on how well known the system being modelled is.

Modelling an engine or engine process theoretically is like the theoretical modelling of any system: Express the fundamental equations of the system, remembering to distinguish between distributed parameters (whose space/time dependency must be considered) and lumped parameters (whose space/time dependency can be neglected), and then balance and connect them. However, these models are often extensive and complicated, and thus simplifications are made, such as linearization, reduction of model order, or approximating distributed parameters as lumped.

The engine processes that directly influence combustion must be modelled as crank-angle dependant. These could be combustion pressure or temperature, air charge, fuel injection, ignition,[4] or valve phasing. Processes outside the cylinder can be considered time-dependant, such as emissions, turbo charger or speed. Any models neglecting the effect of the working stroke induced fluctuations are mean-value models.

[4]Both the injection and ignition timing control in Dynamo are crank angle dependant[**MotorStyring2018**].

Not every part of an internal combustion engine can be modelled theoretically, either because the mathematical formulation of the process is unavailable, or because the existing models are simply too complex and computationally expensive for the control hardware using the model to react as fast as necessary. Therefore the construction and use of gray- or black-box models is widely used for engine control[Rol16].

In [Ise12] Isermann discusses a number of simple methods of experimental modelling of engine behaviours, which will now be discussed. It should be noted that in commercial cars these are often combined for more complex structures.

For static processes, linear and unlinear alike:

- Grid based LUT, with bilinear interpolation between existing datapoints. Widely accepted in the field of non-linear control, LUTs are useful when computational power and storage are limited, but real-time constraints must be met. They can be constructed as completely black-box models based on experiments, or from gray-box models that are simulated beforehand and then filled from the simulated output. Likewise they have the advantage of easy adaptation of single data-points due to changing environment conditions. The main disadvantage is the exponential growth size as the number of inputs increase.
- Parametric model representations, such as polynomials and neural networks, that have significantly smaller storage demands, but also require more complex computations than the bilinear interpolation of the LUTs to produce outputs. The more complex parametric models can also be used as basis for filling LUTs, as explained above.

For dynamic processes:

- Correlation functions are useful for determining impulse response of stationary stochastic or periodic signals on a linear process with a single in- and output. The main advantage is that no model need be assumed. The main disadvantages is the slow and massive calculation.
- Parameter estimation for linear dynamic processes, in which a single input, single output dynamic process is assumed to be describable by a linear difference equation, which can then be solved a number of ways.
- Parameter estimation for nonlinear dynamic processes, can either be estimated with classical nonlinear equations, with extended neural networks, or using operating point-dependant local models.

A comparison of non-linear model structures for emissions control was done in [Rol11]. The article compared the models produced by two machine learning algorithms (LOLIMOT and LOPOPOT), an adaptive polynomial structure, and a grid-based LUT.

It concludes that the machine learning models were flexible and did not require as much memory as the other models, but required several partitions drastically increasing calculation time. The LUTs performed best with regards to small computation times, but suffered in precision unless the grid points were concentrated in regions of interest. However, they quickly took up large amounts of memory for complicated models. The adaptive polynomials were found to have computing times only slightly higher than the LUT and significanlty less memory use, especially for larger models.

### 3.2.2 Control Strategies

Controlling an ICE subsystem is like controlling any complex system. The type of control possible depends on the system model.

For processes that can be modelled linearly, linear feedforward and feedback systems can be applied. Extra care should be taken with feedback systems, so as to (1) avoid wind-up as many sensors (like the $\lambda$ sensor) have maximum and minimum saturation and (2) dampen the systems more than appears necessary in models, as most linear models in ICEs tend to be fairly inaccurate. Often a combination of controllers becomes necessary, especially in the case of multi-variable control.

Nonlinear control can either be static, or dynamic, depending on the nature of the model.

Nonlinear static engine control is usually done with feedforward control, and those control schemes inevitably resembles engine models, as this scheme involves directly applying the output of the model without measuring the effect on the system. The two primary ones are:

1. LUTs: Typically only one- or two-dimensional, but rarely higher due to the inherent visualization problems that would entail. However several LUTs can be combined, either through multidimensional static functions whose inputs are taken from different LUTs or through LUTs containing smaller LUTs within them. The author suggests regularizing using regression, if the input data was noisy or certain areas of the LUT was based on insufficient data, causing undue irregularities.

2. Polynomial models: With the option of simulating them and producing a LUT, using more storage space but requiring fewer calculations on the ECU itself.

Nonlinear dynamic engine control can be done both with feedforward and feedback control, though not directly on the system. In [Ise12] two main strategies are suggested:

1. Local linear control: The nonlinear system is locally approximated to linear systems, for each of which a linear controller is designed. Depending on the system's current state, each of these are weighted and applied to a different degree. While quite time-consuming to produce manually, these local approximations can be constructed using machine learning schemes.

2. Nonlinear control for static nonlinearities: By assuming static nonlinearity, the nonlinearities can be spilt into a linear and unlinear part, the latter of which can be described as a polynomial or LUT, the unlinearity can be compensated for by adding and inverted model of it's expression/LUT. A linear controller can then be designed for the linear part of the model.

## 3.3 A Short Note on $\lambda$ and Injection Control as it is Typically Done in Commercial Cars

Conventional $\lambda$-control is modelled parametrically, and controlled with a feedback system in steady state[Ise12]. However, the response time of most conventionally available $\lambda$ sensors is too slow for a well functioning feedback system during transients. Therefore, in industry, the $\lambda$-control schemes implemented for acceleration between steady states are typically based on static feedforward systems.[Elb13][Dag13].

## 3.4 Control Strategies and Engine Maps Used by DTU Roadrunners, 2012-2018

Below is presented a short history of the state of Dynamo's engine mapping and control strategies, going back to 2012. The results and control strategies are summarized in table 3.1.

### 3.4.1 Gray Box Model Approach, 2012

In 2012 the first and only gray-box approach to engine modeling in DTU Roadrunners history was attempted, and it is described in the Master Thesis of Jacob Mac

| Year | Place | Result | Control strategy |
|---|---|---|---|
| 2012 | 1 | 611km/l | Parametric engine model |
| 2013 | 1 | 612km/l | Closed loop feedback |
| 2014 | 1 | 599km/l | Closed loop feedback |
| 2015 | 1 | 665km/l | Closed loop feedback |
| 2016 | - | - | Closed loop feedback |
| 2017 | 2 | 449km/l | Simple LUT |
| 2018 | 1 | 374km/l | Simple LUT |

Table 3.1: The different results and control strategies over the years. Please note that the closed loop was not exactly live. It still require manual tuning before each race, and only regulated between burns.

Rygaard[Ryg12]. A two-dimensional LUT, acting as the engine map, was developed and named the $\alpha N$-matrix. Each element indicated the desired fuel length as a function of throttle angle $\alpha$[5] and engine speed $N$. This matrix was to be filled out, not by experiments as would be the case in later years, but calculated from a second model. Initially this model was determined using Computational Fluid Dynamics, but upon seeing the resulting $\alpha N$-matrix, CFD was abandoned in favour of a second model, whose parameters were partially determined experimentally.

In this second model the measurements of pressure and temperature in the inlet manifold are used to determine the injection length in the following manner:

$$\text{injection length} = \left(786.7080745 \frac{1.287950635 \cdot 104 \cdot p_{signal} C_1}{1.322531488 \cdot T_{signal} C_2 19.21618414} C_3\right) \tag{3.1}$$

Where $T_{signal}$ and $p_{signal}$ are the inputs from the temperature and pressure sensors mounted on the throttle, and $C_1$, $C_2$ and $C_3$ are constants that can be changed. The author encourages future students optimize them through repeated testing [Ryg12].

Rygaard initially described the correlation between the mass of injected fuel and the injection length as:

$$m_{fuel} = 8.138 \cdot 10^4 \, \frac{\text{mg}}{\mu\text{s}} \cdot \text{injection length} - 0.6695 \, \text{mg} \tag{3.2}$$

And the relationship between the fuel mass flow and fuel mass is then:

$$m_f = \dot{m}_f \cdot \frac{60 \, \frac{\text{s}}{\text{min}} \cdot 2 \frac{\text{revolutions}}{\text{injection}}}{N} 1000 \, \frac{\text{mg}}{\text{g}} \tag{3.3}$$

The air mass flow can be estimated from a speed density function:

$$\dot{m}_{air} = N \frac{V_d}{60x} \rho_{in} \eta_v \tag{3.4}$$

Where $N$ is engine efficiency, $\eta_v$ is volumetric efficiency, and $\rho_{in}$ is $\rho_{air\,in\,cylinder}$. $\rho_{air}$ is found from $T_{signal}$ and $p_{signal}$ using the ideal gas law. $\eta_v$ is assumed to constant. To compensate for the errors introduced by these assumptions, the three calibration factors $C_1$, $C_2$ and $C_3$ were introduced.

In [Dag13], the author Nikolaj Dagnæs-Hansen who was responsible for $\lambda$-optimization the following year, describes his predecessor's work. He calls the three calibration factors $k_{temperate}$, $k_{pressure}$ and $k_{final}$. The structure of the final implementation of 3.1, is explained by Dagnæs-Hansen through a figure reproduced in figure 3.1.

[5]The throttle in 2012 was a butterfly type, and has since been replaced.

Rygaard's engine map was used in the 2012 Shell Eco-Marathon, with a result of 611 $\frac{\text{km}}{\text{l}}$. An overview of $\lambda$ values for one of the attempts at that race can be seen in figure 3.2. While fairly effective, it should be noted that while this method doesn't have the manual construction of engine maps we see in later years, it does require a fair amount of testing to identify $C_1$, $C_2$ and $C_3$.

### 3.4.2 "Closed Loop" $\lambda$ Control System, 2013

In 2013 an injection control system was designed and implemented by DTU Bachelor student and DTU Roadrunners team member Nikolaj Dagnæs-Hansen. This system did two things: Firstly, it implemented the simple engine map consisting of an array of different injection-lengths, each corresponding to a different engine rotation speed, where linear interpolation was done to find injection lengths for rotational speeds not represented in the array. This map was still used, mostly unchanged, until 2019 when the authors of this thesis replaced it. Secondly it implemented a very simple and fairly slow, but live, $\lambda$ control scheme. The coast and burn diving strategy was also used in 2013, and the first version of this scheme would average the $\lambda$ values during an entire burn and calculate a correction factor based on this average as follows:

$$n_m = \frac{\lambda_{avg,m}}{\lambda_{desired}} \prod_{i=0}^{m-1} n_i \quad , n = \text{correction factor}, m = \text{number of burns since engine start} \tag{3.5}$$

This correction factor was then multiplied with the entire map before the next burn. This version was found to be under-dampened and a new, dampened version was implemented as follows:

$$n_m = \frac{\frac{\lambda_{avg,m}}{\lambda_{desired}} + 1}{2} \prod_{i=0}^{m-1} n_i \tag{3.6}$$

The results from using this strategy can be seen in figure 3.3. The author concluded that either the map or the control scheme on their own was lacking, but together the result was acceptable. He did stress that (1) the creation of a good map was imperative, as otherwise the $\lambda$-control would simply move a bad map around, and (2) the creation

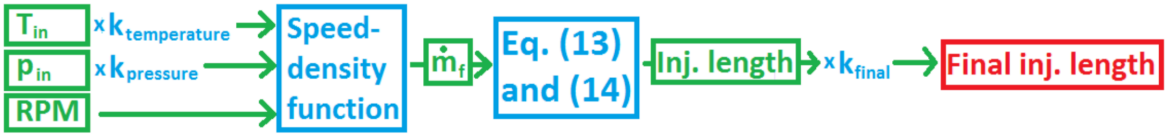



Figure 3.1: A graphical explanation behind 3.1, by the author of that equation's successor[Dag13].

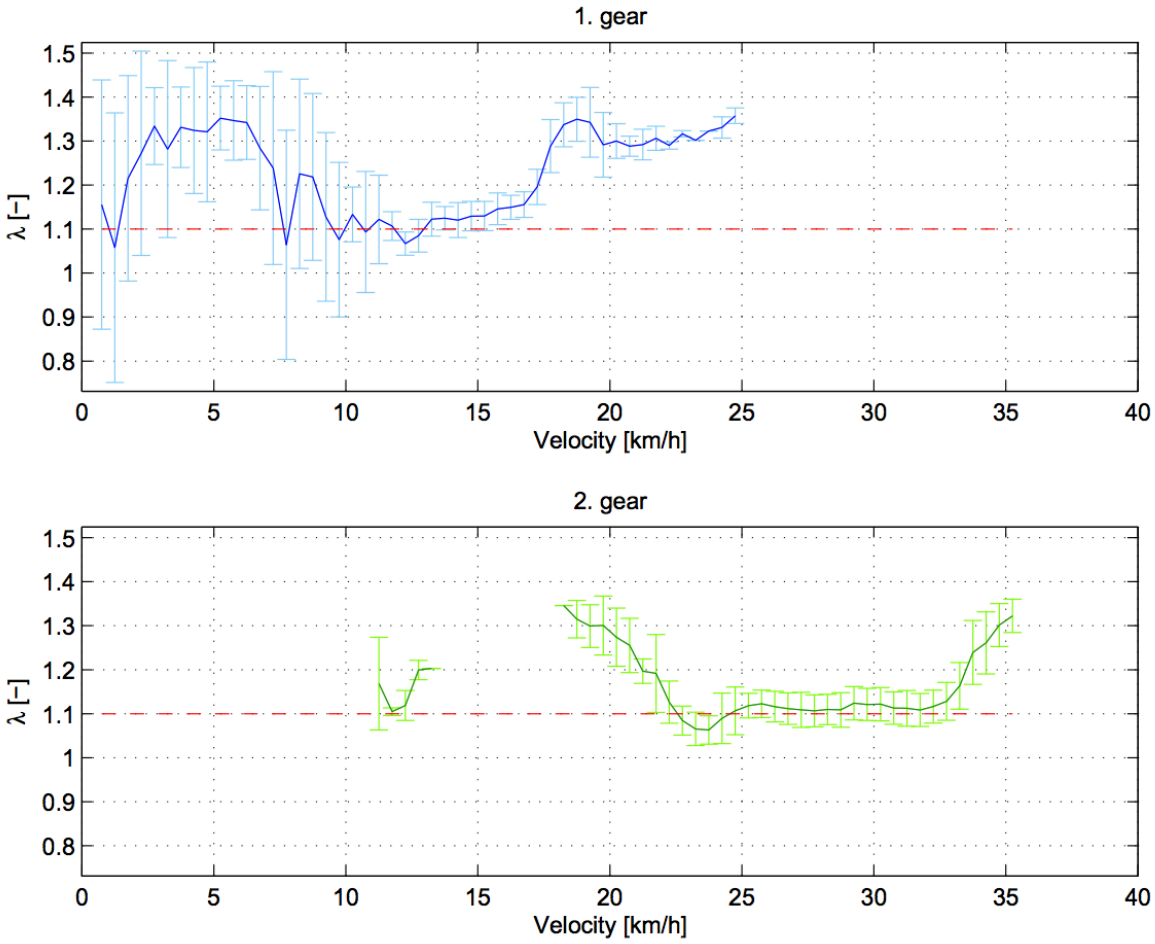



Figure 3.2: A graph of the $\lambda$ values during an attempt at the Eco-Marathon 2012, when Rygaard's system was used, taken from [Ryg12]. While the results don't look very convincining at first glance, if one considers that the car then used a similar coat and burn strategy in 2012 as it does now, and thus spent the majority of the race at a speed of $22 - 35 \frac{\text{km}}{\text{h}}$, it is clear that the control scheme was quite successful.

of such a map was quite labour-intensive, and future students should start in good time. He further advised future students to implement a flowmeter on the engine's air intake and that basing any future maps on air-intake might speed up the process. This has not yet been done.

### 3.4.3 Improvements to The $\lambda$ Closed Loop System, 2014-2016

The basic structure of the 2013 implementation was kept in the following years. Where in 2013 the desired $\lambda$ was set to 1.1, and in 2015 it was raised to 1.2, and later to 1.25,

The main difference implemented in these years was first attempted in 2014: Students introduced a method of regulating each element of the engine map individually, as opposed to the entire map together.[s1215]. There was, however, one big issue that had to be addressed before this could be done.

A certain amount of delay incurred from when the $\lambda$ sensor started taking a measurement to when that measurement was read by the cRIO (the previous ECU, a National Instruments real-time embedded industrial controller, programmed in LabView) and again until that measurement was stored in the correct array. In 2013 this total delay was measured to be less than 300 ms and was dealt with by simply averaging all $\lambda$ values from a burn. In 2014 a $\lambda$ array was created during each burn. This array would contain the average $\lambda$ values for smaller angular RPM buckets, each of size 50 RPM. The group implementing this array measured the delay from sensor to array to be 500 ms. The work-around chosen to deal with that delay was to simply place newly arrived $\lambda$ values

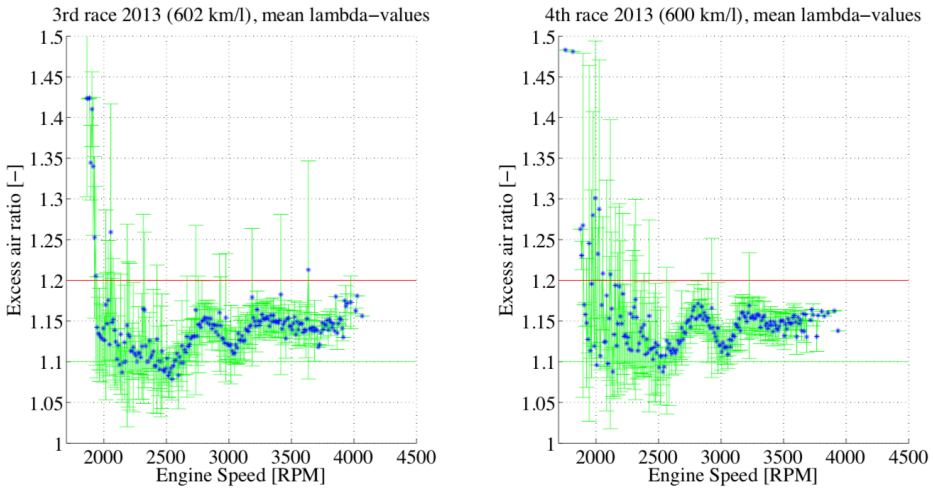



Figure 3.3: The recorded $\lambda$ values from Dynamo's 3rd and 4th race in 2013[Dag13]. The earlier races did not drive with the final version of the control scheme, and thus the data is not included here.

in the buckets corresponding to the RPM the engine had 500 ms earlier. Additionally, to reduce noise, all $\lambda$ values above 1.4 or below 1.1 were ignored.

Why the $\lambda$-array was chosen to have 50 RPM resolution is unknown, as the engine map still had 250 RPM buckets, and indeed the first thing done after creating a new $\lambda$-array was to downscale the resolution to 250. If this was done by averaging the values of every four buckets, or simply by taking every fourth element, or in some other manner, is unclear.

Once a map of $\lambda$ values corresponding to the engine map was constructed, the method similar to the one described in equation 3.6 was applied individually to each pair of measured $\lambda$ and injection corresponding to the same RPM bucket. This method can be described as follows:

$$n_i = \begin{cases} \frac{\lambda_{meas}}{\lambda_{desired}} > 1.1\lambda_{desired} : & \frac{1.1\lambda_{desired}+1}{2} \\ 0.9\lambda_{desired} < \frac{\lambda_{meas}}{\lambda_{desired}} < 1.1\lambda_{desired} : & \frac{\frac{\lambda_{meas}}{\lambda_{desired}}+1}{2} \\ 0.9\lambda_{desired} < \frac{\lambda_{meas}}{\lambda_{desired}} : & \frac{0.9\lambda_{desired}+1}{2} \end{cases} \tag{3.7}$$

Where $n_i$ is correction factor for bucket with RPM $i$ and $i$ is 2500, 2750, 3000... 6000.

This system was only implemented for angle velocities of 2500 and above. Furthermore it was only applied to the second gear. An overview of the implementation can be seen, on figure 3.4.

Despite the improvement of the control software, the students of 2015 stressed the importance of doing the final tuning of the engine map as close to the actual race as possible, as the map had a tendency to drift with changing weather conditions[s1215]. The same report suggested tuning in the queue to go on track before each run while also warming the engine.

How effective this system was at regulating $\lambda$ values during runs is unknown, as it wasn't documented. We can observe that the team's overall milage improved steadily until 2015, when DTU Roadrunnes set our record of 665 km/L. That was probably also due to the many other improvements done, as well as the two separate bachelors written on driving strategy between 2013 and 2016. Likewise when the the record wasn't improved in 2016, it probably had significantly more to do with the wheel mount breaking and the car not completing any valid runs, and not the effectiveness of the engine control.

### 3.4.4 Abandoning Feedback Control, but Also Not Changing the Engine Map, 2017

In 2017 the entire engine code was rewritten with the goal of simplification[Kar17]. As a part of this process the injection feedback control was moved into a separate VI (the LabView term for a script) and was brought to work on a test-bench, but that was

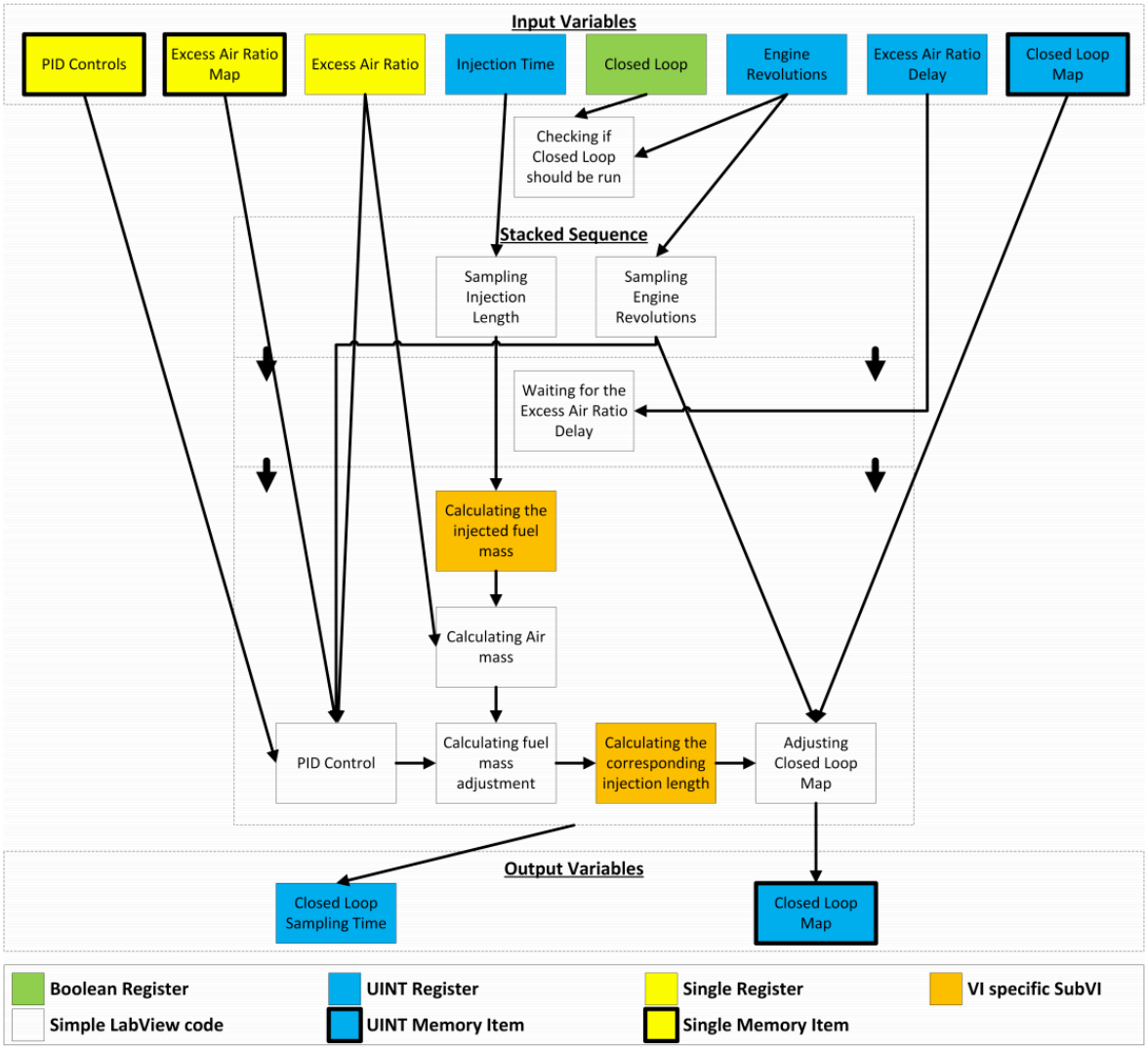



Figure 3.4: The flowchart of the closed loop control of injection length included in a 2016 guide to the engine control program, borrowed from the appendix of a Master's thesis[Fra16]. The control strategy and code were not changed from 2015 to 2016, simply documented.

considered too unstable[6] to be used on track.

There is no data for how stable the $\lambda$ value is kept, but again the overall results are known: 449.2 $\frac{km}{l}$. Removing the feedback control may take part of the blame for the comparatively poor result, however, many other factors could have affected this result.

### 3.4.5 Porting the ECU to a New Format, Bringing the Engine Map but Leaving the Feedback Control Behind, 2018

In 2018 power was considered for the first time in Shell Eco-Marathons history. In order to save power[7] in the car, the NI cRIO ECU was replaced with a Teensy 3.6 on a custom PCB. This meant the entire engine control was rewritten from the bottom, in teensyduino (an extension of the arduino language) and C++. Most functionalities were ported as closely as possible, as the group doing the new coding consisted entirely of 3rd semester electrical engineering students, who a priori didn't know anything about how ICEs work. The closed loop control did not get ported, due partially to not existing in a stable enough form to be used anyway, and partially due to it's functionality and implementation not being well documented.

Data is available from 2018, but not the collection circumstances of said data, making the use slightly limited, it is attempted in section 5.8.1.

In 2018 we won SEM with a mileage of 374.2 km/L. This reduction compared to the previous year is likely due to the introduction of a new way of computing fuel mileage, that included not only fuel but also power. Without the telemetry data from the that year it's difficult to say how much of that is due to this new rule.

## 3.5 Discussion of Engine Model Choice

The goal of this thesis was never closed loop $\lambda$ control. In this section, we explain why that decision was taken.

#### 3.5.0.1 Power Consumption of the $\lambda$ sensor

Until 2018 the power consumption of ICE vehicles was not considered by the organizers of the Shell Eco-Marathon. Therefore power-use was optimized in Dynamo for the first time in 2018. The biggest change for that occasion was that the old ECU (a 4kg NI RIO and a "motor board" interfacing the RIO with the rest of the car) was replaced with

[6] It is unknown if "unstable" is meant in the control theory sense of the word, or is simply meant to imply buggy, unfinished code. Having seen the code, we believe it to be the latter.

[7] More details on the power saving process in section 3.5.0.1.

the current ECU, a single PCB with the same micro controller as the old motor board, effectively using the same amount of power as the old motor board. See section 3.4.5 for details.

However, this was not the biggest draw on power. The $\lambda$ sensor is the biggest draw on power by far, as it heats up to around 300 degrees Celsius, and uses a lot of energy to do so. A very simple experiment has been done to evaluate the approximate energy drawn for the $\lambda$ during a normal 39 minute run, by simply turning the car on and measuring the current for 39 minutes, and then repeating the procedure with the $\lambda$ sensor turned off. This told us it used just over 20 kJ during a normal race. If we consider the two official runs seen in appendix A, we see that approximately 20 kJ separates their electric energy consumption corresponding to one race being run with the sensor and one without.

It's therefore clear that any closed loop solution would have to be not only better that an open loop solution, but better by a margin larger than the disadvantage imposed by the $\lambda$ sensor. Exactly how much better that is, of course depends on the closed loop implementation, and since we didn't make one, we can't compare it to the solution we did implement. We can see how well a possible solution would have to out perform the one implemented, and we can compare results with previous years.

Over the course of a single run, the energy that goes to the $\lambda$ sensor can be converted to the volume of gasoline, following the official Shell conversion:

$$V_p = \frac{P}{0.25 \times 0.75 LHV_{gas}\rho_{gasoline}} = \frac{20kJ}{0.25 \times 0.75 \times 42900\frac{\text{kJ}}{\text{kg}}0.7646\frac{\text{kg}}{L}} = 3.33\,\text{ml} \tag{3.8}$$

Converted to ethanol volume we find:

$$V_{ethanol} = \frac{V_p LHV_{gas}\rho_{gas}}{LHV_{ethanol}\rho_{ethanol}} = 5.11\,\text{ml} \tag{3.9}$$

Any closed loop control system would need to save us 5.11 ml ethanol more than a open loop control of similar quality. For context, in the best run of 2019, we used approximately 40 ml of ethanol.

### 3.5.1 Evaluating the Requirements

In [Ise12], the author says the following about ECUs: "The design and implementation of engine control functions has developed into a sophisticated and labor-intensive procedure. This is for many reasons, among them the high multi-variable complexity of engine control, the high performance requirements of suppliers, manufacturers and customers, and legislative certification limits for fuel consumption and emissions, and competition. "

Reading this paragraph it is apparent how different the situation of the DTU Roadrunners team is from that of a commercial manufacturer. We do not have "high multivariable complexity" in our engine, nor do we have "high performance requirements of suppliers, manufacturers and customers", the only performance requirements are the minimal tasks required to pass the technical inspection of the Shell Eco-Marathon and that of winning and ideally surpassing the performance of the previous year that the team sets for itself. Likewise we are not subject to "legislative certification limits for fuel consumption and emissions".

Therefore we can set the requirements for the model and control scheme chosen relatively freely:

- Stable.
- Does not have complicated computations at injection time.
- Fits in the EEPROM of a Teensy 3.6.
- Easy to understand, use, and modify by future students.
- More convenient to tune prior to races than previous solutions.

Given the driving strategy of accelerating as much as possible, or not running the engine at all, it is clear that the engine does not behave linearly while running, and a linearization around an operating point is not possible, as it is never actually going to hold any state for very long. The long and massive calculations required for local linearisations may have been possible on the NI cRIO, but given the hardware limitations introduced by moving the ECU to a microcontroller, the two most obvious models become: An adaptive polynomial or a grid based LUT.

Considering that the last attempt at a parametric model of any kind was in 2012 and was abandoned after a year, we do not believe that it would satisfy the requirement of being easy to understand and modify by future students. Additionally any parametric model would likely suffer from the fairly limited amount of sensors in the car just as the 2012 model did. If a future generation of Roadrunners manage to install a flowmeter on the air intake, a reliable parametric model may become an achievable concept and we hope it will be attempted.

This leaves us with the engine model the car already had when this thesis was started: LUTs. The main disadvantage of LUTs in modern commercial cars is the sheer size and number of LUTs necessary to control the many actuators based on the many sensors, see section 3.1. Dynamo's engine doesn't have 6-7 actuators, and therefore doesn't need 60-80 LUTs. It needs three: The ignition timing table, the injection timing for first gear, and the injection timing for second gear. Until our car sees the same rise in complexity seen in commercial cars over the last 30 years, we don't need to introduce the complex control algorithms commercial cars need today.

Due to the performance requirements of any live feedback control system, see section 3.5.0.1, combined with the difficulty of constructing such a system, see section 3.3, a feedforward control system was chosen.

Dynamo already had a LUT based control system when this project started, and as just stated we did not believe changing the nature of the control system would be beneficial. We did still wish to improve performance both on and off track. We had observed the time and labour necessary each year, both during the course of the year and in London, for the creation or improvement of the engine map.

This lead us to the two main goals of this thesis: Improvement of the engine map itself, and simplification of the tuning process.

# CHAPTER 4
# Simulated Hand Tuning

## 4.1 Aim

The first iteration of our solution will be detailed in the next chapter. This iteration was designed with purpose of achieving the project statement as originally stated in March at the start of this project:

## 4.2 Problem Statement

As it stands, tuning DTU Roadrunners energy efficient car is done by hand, which is a slow and at times imprecise process. A system will be designed for automatically tuning the car. The system should fulfill the following requirements:

- At the very least it must be able to automatically tune the car as well as when the car is tuned by hand.
- It must have a user interface, allowing the user to monitor and control the process.
- The tuning process must be reliable, both in efficiency and duration.
- It must be possible to change the parameters of the tuning, for instance to tune the engine to run either lean or far.
- It must be possible to overwrite the result of the tuning by hand, if certain results are found to be undesirable.

## 4.3 Project Outline

In this project we will design a system for automatically tuning the Dynamo's engine control look up tables in the engine control unit (ECU).

## 4.4 Motivation

When tuning by hand, the manual approach, as discussed in Section 2.5, is to accelerate up to the maximum number of rotations per minute, with the auto-gear turned on, which will change the gear during the acceleration. From this, an operator can then look at the $\lambda$ values and estimate if the engine is running lean or fat, and thus if the injection times should be increased or decreased. How much the injection times should be increased or decreased is something that the operator will often estimate, based on their experience and intuition. Our aim with simulating hand tuning is to implement a system which can replicate this process, and compete with the intuition of an experienced operator.

## 4.5 Design

The first iteration of this project essentially emulates what the operator does. It accelerates up to the maximum RPM in second gear, or sweeps the RPM values. It then looks at the $\lambda$ values compared to the injection times, and then adjusts the times as necessary. To make the data more manageable, and in part to reduce the uncertainty introduced by the delays introduced by the $\lambda$ sensor, the RPM-range is split up into RPM buckets that each span 250, using the following conversion

$$RPM_{index} = \frac{RPM}{250} - 6 \tag{4.1}$$

The $\lambda$ values recorded are sorted into these buckets according to the RPM at which they were collected. The average $\lambda$ values is found for each bucket, and these average $\lambda$ values are then used for calculating the new injection time for each RPM index.

### 4.5.1 Flowchart for First Iteration of the Project

In order to approximate the process of hand tuning, a state machine was implemented, the state machine follows the flowchart, as seen in figure 4.1. The machine has the following states:

- **NOT_TUNING**
  This is the default state for the car, when the car is operating normally.
- **INIT_SWEEP**
  Here a burn is requested, gear is set to automatic, and the variables used for tuning are reset. Additionally, the sweep counter is incremented.
- **SWEEPING**
  In this state, the car is burning and thus RPM is going from 0 to maximum, while the gear changes. This state saves data for further processing, and checks if the

car is in second gear and above 4000 RPM. If it is, the state machine goes to the next state. Otherwise, it remains in this state.

- **PROCESS_DATA**
  In this state, the data saved in the **SWEEPING** state is processed, the $\lambda$ values for each RPM bucket is calculated, and the injection times are modified. If any of the buckets have a $\lambda$ value outside of a predefined interval, it goes to **SPEED_DOWN**, otherwise it goes to **EXIT_TUNING**. Additionally, the processed data is logged.

- **SPEED_DOWN**
  The state machine will stay in this state until the speed sensor records 0 $\frac{\text{km}}{\text{h}}$. Usually this is achieved by having an operator apply the brakes, although hypothetically it could be left to time. It then goes to **WAITING**.

- **WAITING**
  The system remains in this state for 5 seconds, and plays a song. The state machine then goes to **INIT_SWEEP**. This state is here to ensure that the operator has time to stop applying the brakes, before the engine begins again.

- **EXIT_TUNING**
  In this state, the state machine leaves the tuning process, it also logs the data from the final sweep, and sets the sweep counter to 0.

### 4.5.2 The Challenge of Overheating

The biggest problem with this implementation of the system is overheating, as the water would quickly reach its boiling point when we repeatedly swept. When tuning by hand, the operator can simply monitor the various temperatures, and then from this decide roughly when to wait and let the car cool down. In order to emulate this, a system which can monitor either the temperature of the water used to cool the cylinder or the oil temperature has been implemented. If the monitored temperature goes above or below the desired temperature by $5^o$C the system will wait, until the temperature is within $1^o$C of the desired temperature. This solution requires an operator to maintain the temperature when it goes out of bounds, either by cooling the engine with fans or patience, or by heating it with an oil heater or by burning.

### 4.5.3 Flowchart for Second Iteration of the Project

This iteration is similar to the first iteration, with the main difference being the **WAITING** state. The flowchart can be seen in figure 4.2. The system can either be set to tune around a water temperature, or an oil temperature. In this example, the oil temperature is tuned around. The new **WAITING** state works as follows:

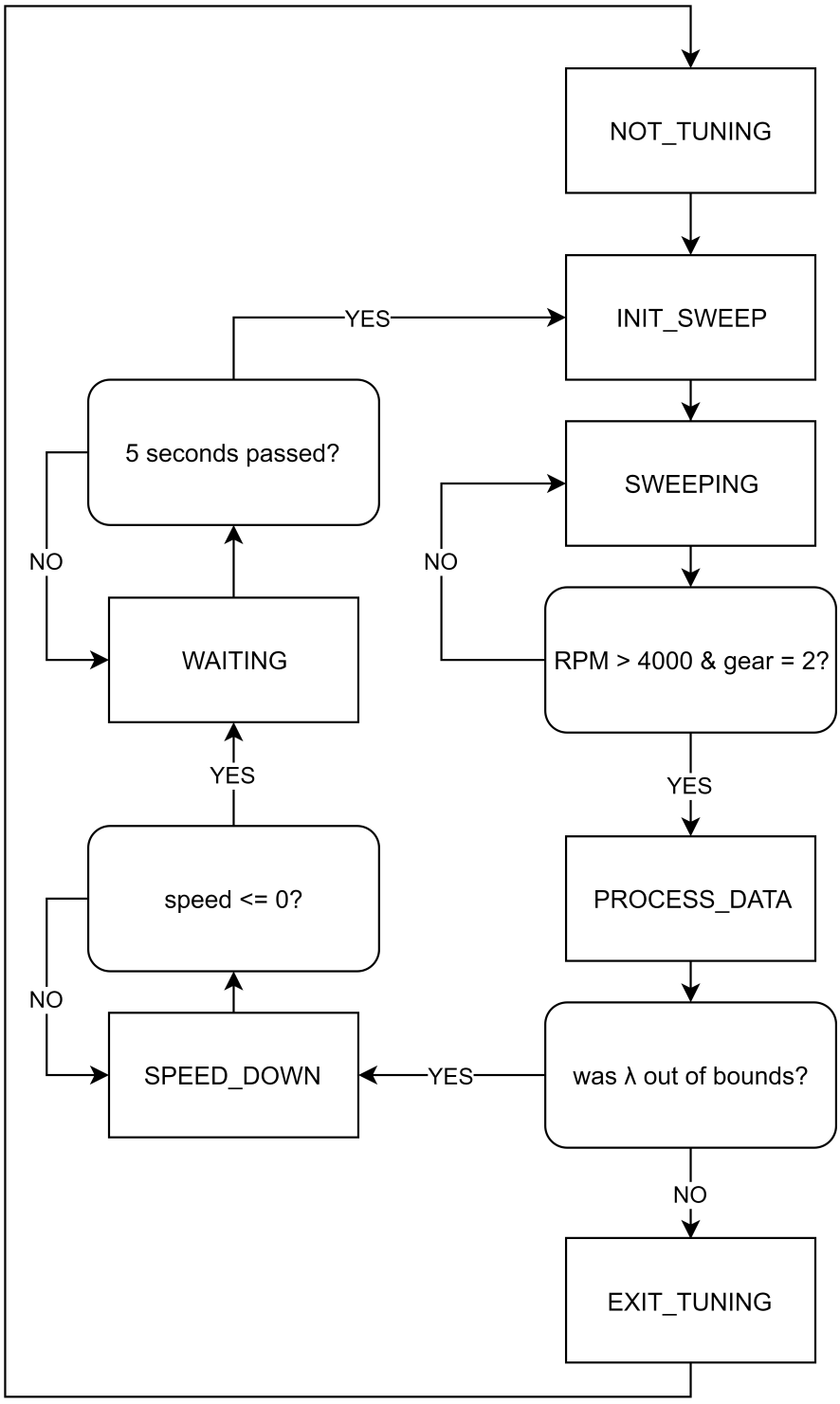



Figure 4.1: The state machine for the first iteration of our engine control. This flowchart is meant to give an overview of how the state machine goes between states

- **WAITING**
  The system remains in this state for 5 seconds. It is then checked if the oil is too warm or too cold. It will then wait until the temperature is within acceptable parameters. An operator is expected to be present in order to supervise the heating or cooling as necessary. Once the temperature is acceptable, the car will go to **INIT_SWEEP**.

### 4.5.4 Computing New Injection Times

As the injection array is indexed by RPM in intervals of 250, it was decided that the simulated hand tuning would also handle data in this manner. However, it was noted that the engine does not run at every RPM during normal operation; it mainly operates from 1500 to 4000. The system will be designed with this in mind.

A few data points are collected during each sweep: The temperature of the air in the intake, the temperature of the oil, and the temperature of the water cooling the cylinder. Technically the system should also measure the temperature in the exhaust, but due to unknown reasons, this temperature measurement was faulty, as it only returned $-199^o$C. Additionally, the $\lambda$ values were also saved.

The sum of all of the data in one RPM bucket was stored in a data array, along with the number of measurements taken by the system. Using this, the average value for each index could be computed. The conversion from RPM to index is as follows:

$$i = \frac{RPM}{250} - 6 \tag{4.2}$$

Which yields 0 for RPM 1500, and 10 for RPM 4000. It of course only works for RPM above or equal to 1500, all values below 1500 is ignored in this iteration. Once a sweep is finished, the various temperatures are stored on an onboard SD card, and the new injection times are calculated, if $\lambda$ is out of bounds. This calculation is done using the following equation:

$$t_{inj,new}[i] = t_{inj,old}[i] + (\lambda_{desired} - \lambda_{avg}[i]) \cdot p \tag{4.3}$$

Where $i$ goes from 0 to 10, p is a factor that can be changed as desired, we had it around 100 to 500 when testing.

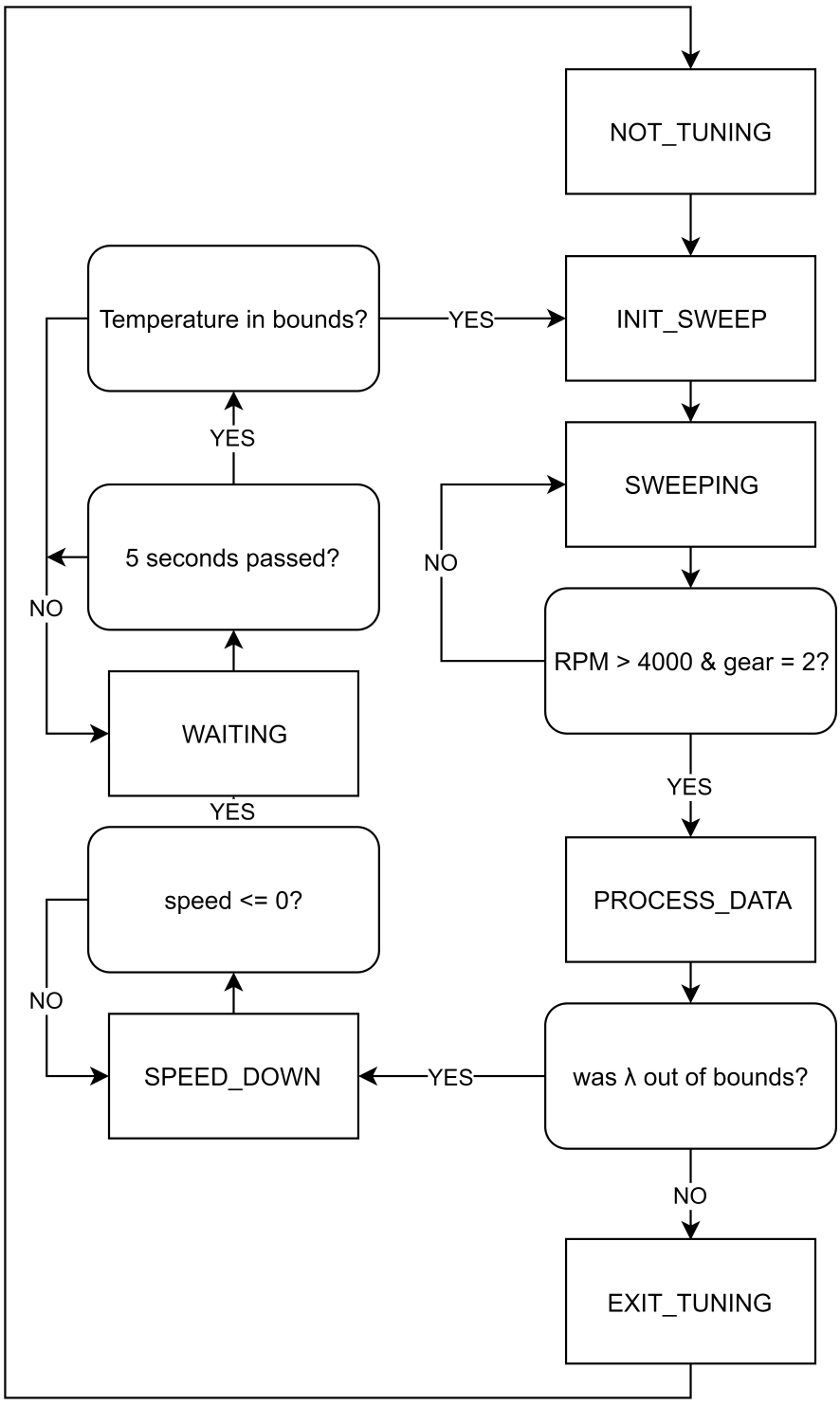



Figure 4.2: The state machine for the second iteration of our engine control, when regulating oil temperature. This flowchart is meant to give an overview of how the state machine goes between states

## 4.6 From Tuning At Temperatures to Tuning An Engine

To implement this design, it would be necessary to look at how the $\lambda$ values change between sweeps when we don't change the injection times, as this could show which temperature has the biggest effect on $\lambda$. If such a temperature exists, it would be an obvious choice for an axis when expanding the LUT.

Of course this experiment could also reveal that $\lambda$ has no correlation with temperature, in which case another solution would have to be implemented.

## 4.7 Results of Simulated Hand Tuning

In this section we will discuss the results from the various iterations of simulated hand tuning. But first we will briefly discuss the $\lambda$ measurements.

### 4.7.1 A Note on $\lambda$ Measurements

Another member of DTU Roadrunners realized that for the last couple of years we had not been using the correct conversion between the $\lambda$ probe's output voltage, and the actual $\lambda$ values the voltages corresponded to. This means that some of our data has had to be converted from the wrong range (which was [0.25; 1.47]) to the correct range (which is [0.8; 1.51]). The narrower range means that we have a higher resolution, and that we won't be able to detect if the engine is running fatter than 0.8. In the following data sets, we tried to tune around $\lambda = [0.9; 1.1]$ which then becomes $\lambda = [1.1783; 1.2947]$ for our data using the wrong $\lambda$ conversion. The tests depicted in figure 4.3 and 4.4 were made using the wrong conversion.

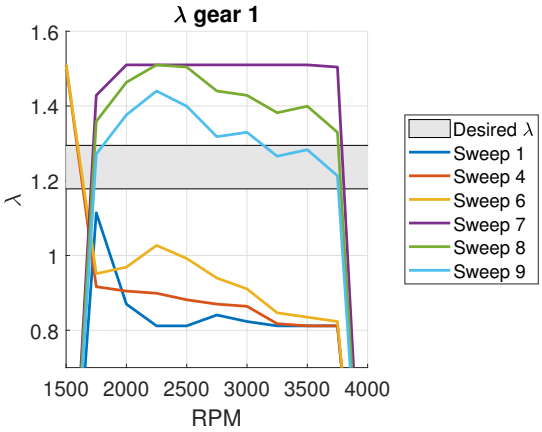
### 4.7.2 First Iteration

An example of the first iteration in action can be seen in figure 4.3. In this iteration, there is no temperature regulation. Here we applied the brakes whenever the sweep was finished, which meant that a speed of 0 was reached in around 30 to 40 seconds. This means, that there is roughly 40 seconds between each sweep, except for between sweep 6 and 7, as we let the car coast for a few minutes to see how stable our tuning would be. There is about 3.5 minutes between those sweeps.

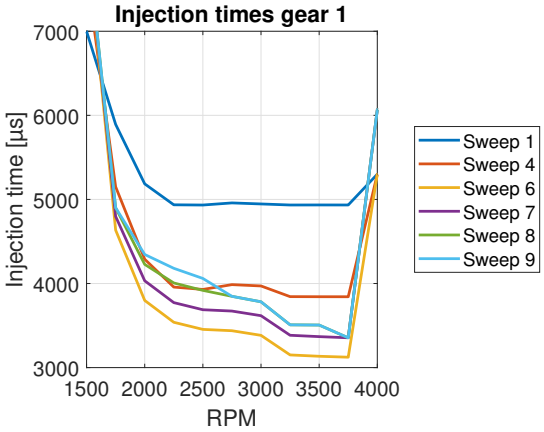
It is clear, that the $\lambda$ jumps quite a lot between the sweeps. The specific reason for this change is hard to explain, especially because we didn't measure the temperatures at this point. But needless to say, if the $\lambda$ values drifts this much in 3 minutes, it is not a reliable tuning.

### 4.7.3 Second Iteration with Oil Temperature Regulation

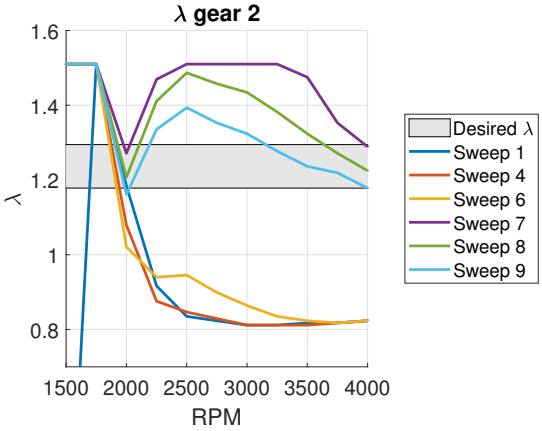
In figure 4.4, the result of regulating the oil temperature can be seen. Now the car waits between sweeps for the temperature of the oil to go down. This process takes about 2.5 minutes. Here we do manage to keep $\lambda$ significantly more steady over time than we did with the first iteration.



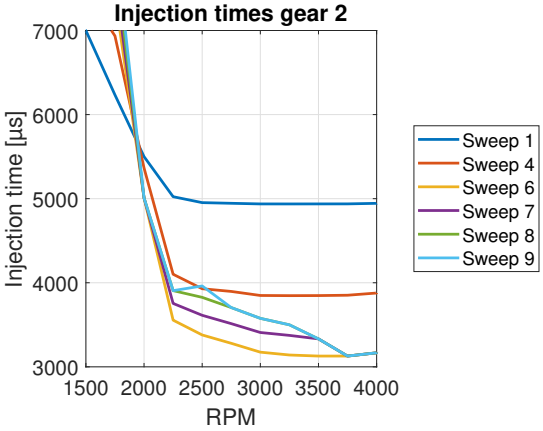

(a) How the $\lambda$ values changes for the first gear, in the first iteration of our tuning program.




(b) How the injection times changes for the first gear, in the first iteration of our tuning program.

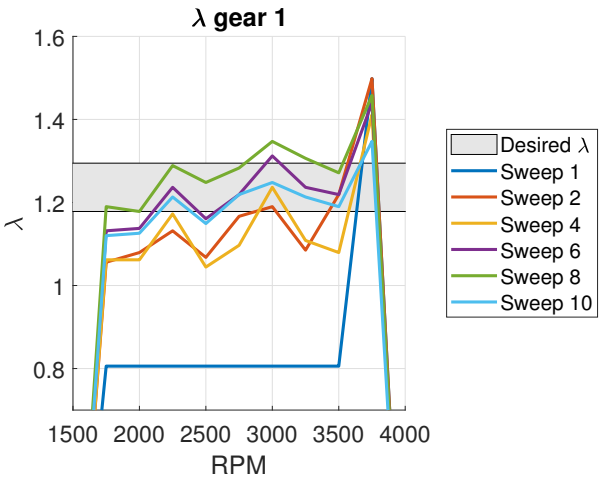



(c) How the $\lambda$ values changes for the second gear, in the first iteration of our tuning program.

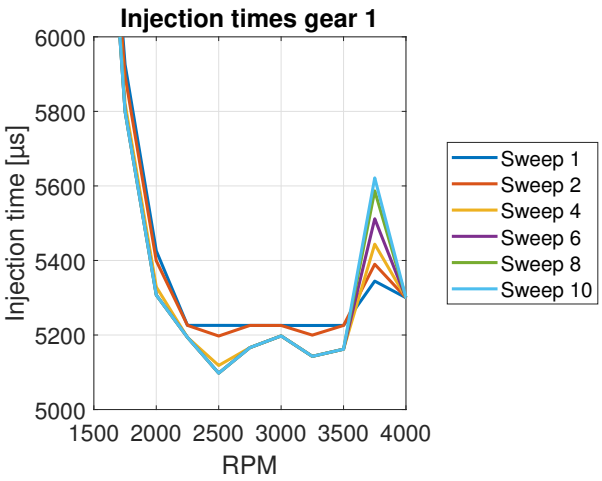



(d) How the injection times changes for the second gear, in the first iteration of our tuning program.

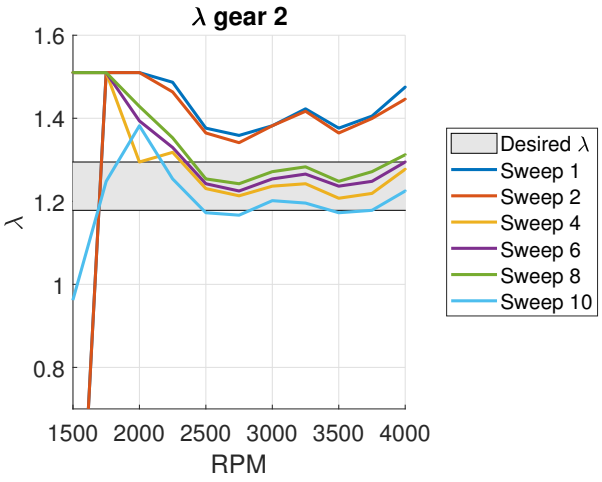
Figure 4.3: The various $\lambda$ values and injection times for a tuning session using the first iteration of simulated hand tuning. Note that there is a 3.5 minute break between sweep 6 and 7, while there is slightly less than a minute between all of the other sweeps. In this 3.5 minute gap, the $\lambda$ values all jump drastically as the engine goes from running very fat to very lean. It should also be noted how little the injection times changes from sweep 6 to 7. This demonstrates just how much the $\lambda$ values can be effected by conditions outside injection times, which is one of the fundamental problems with just using hand tuning alone.



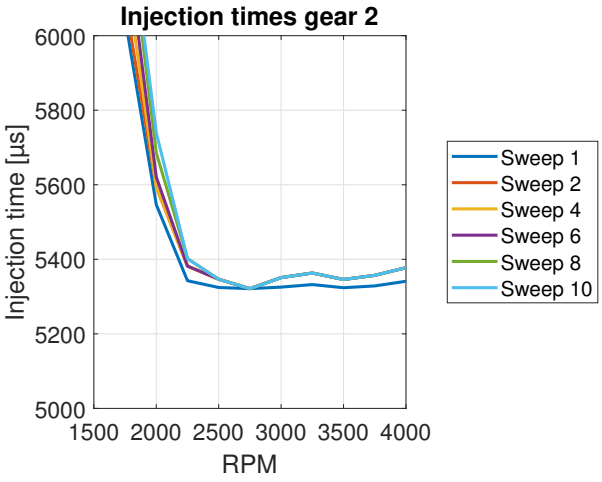

(a) The $\lambda$ values for the first gear in the second iteration when regulating oil temperature.




(b) The injection times for the first gear in the second iteration when regulating oil temperature.




(c) The $\lambda$ values for the second gear in the second iteration when regulating oil temperature.




(d) The injection times for the second gear in the second iteration when regulating oil temperature.

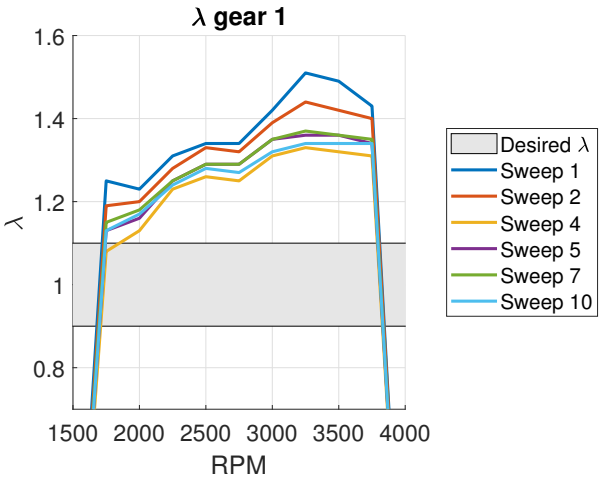
Figure 4.4: The various $\lambda$ values and injection times when the oil temperature is regulated. There is roughly 2.5 minutes between each sweep. Note that in spite of the time between the sweeps, the $\lambda$ values does not drift drastically between sweeps like in figure 4.3. It should also be noted, that there still is some small undefined behavior, especially in second gear, where $\lambda$ slowly drifts up from sweep 4 to 8 and then drifts a bit down for sweep 10. This implies that while oil might be a better indicator of how the engine will perform than only RPM, the is still some other factor that effects the engine behavior.

### 4.7.4 Second Iteration with Water Temperature Regulation

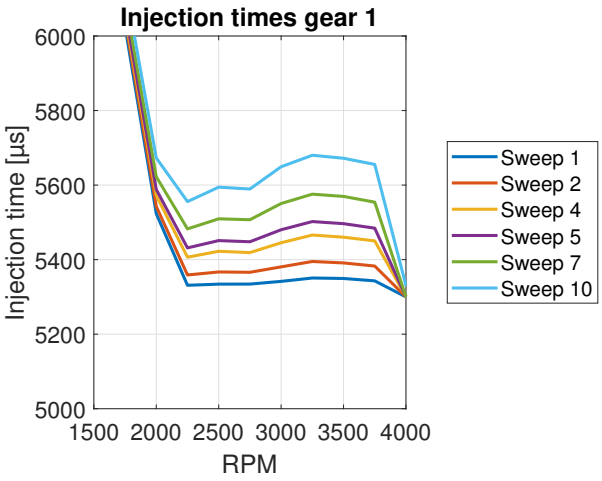
This run is a bit more interesting since the time between sweeps varies greatly; however, it does provide some promising information regarding the stability of the system when regulating water temperature. The time between sweeps can be seen in table 4.1. The data from the tuning can be seen in figure 4.5. It is especially interesting, that between sweep 4 and 5, which have more than 8 minutes between them, $\lambda$ barely drifts at all.

| Sweep | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Time passed | - | 42.5s | 38.8s | 42.4 | 521.7s | 38.1s | 104.2s | 99.5s | 54.4s | 220.1s |

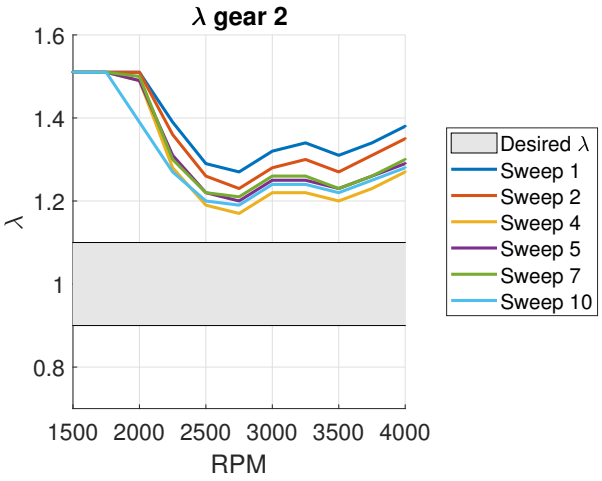
Table 4.1: The time passed since last sweep for every sweep, in the second iteration, when regulating water temperature



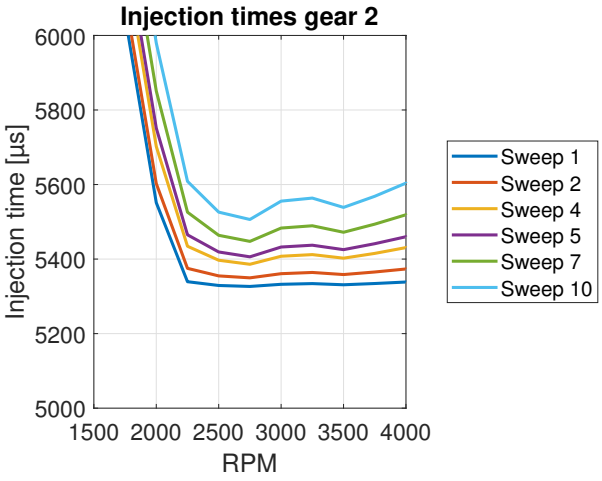

(a) The $\lambda$ values for the first gear in the second iteration when regulating water temperature.




(b) The injection times for the first gear in the second iteration when regulating water temperature.




(c) The $\lambda$ values for the second gear in the second iteration when regulating water temperature.



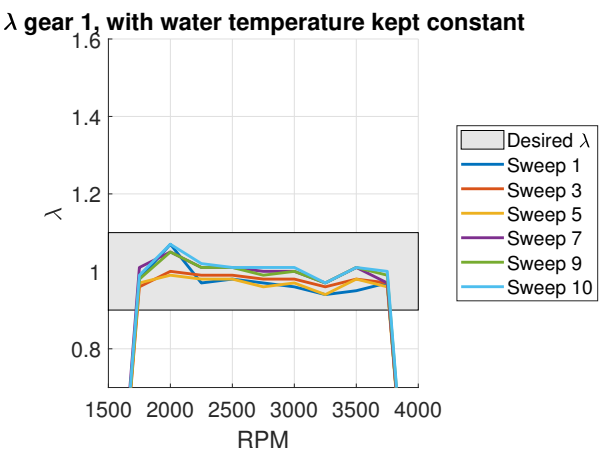

(d) The injection times for the second gear in the second iteration when regulating water temperature.

Figure 4.5: The various $\lambda$ values and injection times when the water temperature is regulated. This is our first measurement with the proper $\lambda$ measurements. The time between sweeps is very irregular. Note that the $\lambda$ values almost are shifted up and down the same amount at each index, this is a good indicator that water temperature is one of the most important indicators of how the engine operates.

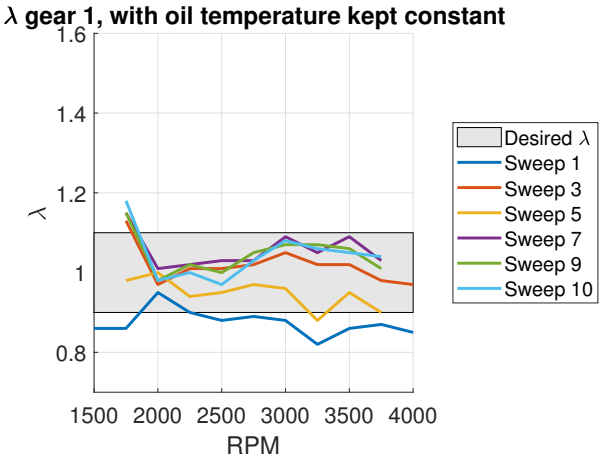
### 4.7.5 Test of the Stability of Oil And Water Temperature

In figure 4.6 the $\lambda$ values of two separate sessions can be seen. For these sessions, a modified version of the code is used, which doesn't adjust the injection times. This means that all of the variations in the $\lambda$ values are the result of the engine's operating conditions changing ever so slightly. The intent of this experiment is to see how keeping

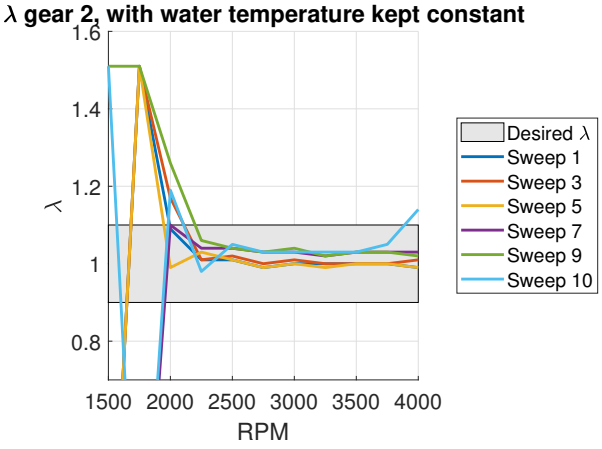
the water and oil temperatures regulated will affect $\lambda$. It is clear that the $\lambda$ remains much more stable when we regulate the temperature of the water. This is what we would expect, as the temperature in the cylinder is an important factor in determining the $\lambda$ value[Sim19], and the water temperature follows the cylinder temperature as it used to cool the cylinder.



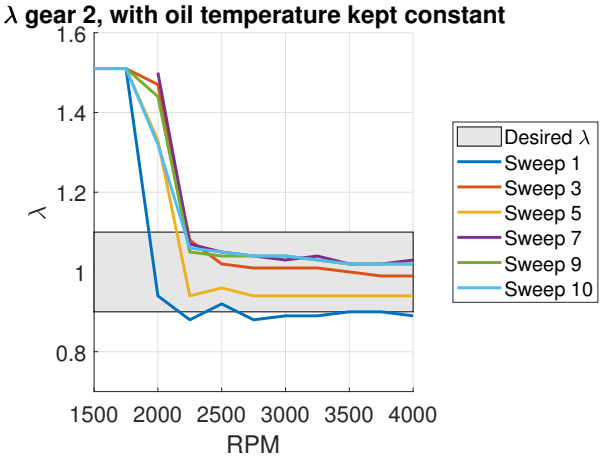

(a) The $\lambda$ values for the first gear, when water temperature is regulated




(b) The $\lambda$ values for the first gear, when oil temperature is regulated




(c) The $\lambda$ values for the second gear, when water temperature is regulated




(d) The $\lambda$ values for the second gear, when oil temperature is regulated

Figure 4.6: The various $\lambda$ values when keeping injection times constant, and regulating first water, then oil temperature. This test was meant to give an idea of how the temperatures impact the stability of the $\lambda$ values. Our test indicates that the water temperature has the biggest impact, given how steady $\lambda$ is when we regulate water temperature. The data from this test also gives a good idea of which RPM values the engine operates. From 1750 RPM to 3750 RPM in first gear, and from 2000 RPM to 4000 RPM in second gear. This will be taken into account going forward.

### 4.7.6 Discussion

#### 4.7.6.1 Extending the Engine Map

From the first iteration it is clear that tuning the engine solely based on RPM is far from enough; if the tuning is to be reliable it has to have something compensating for the drift when the external conditions change. As we have seen from previous iterations of the engine control, this can be done with closed loop control, but as this is not an option for us, we have chosen to expand the tuning. Based on how stable the engine performs when water temperature is regulated, we have decided to use it for the new dimension of the LUT.

#### 4.7.6.2 Which RPMs Should We Tune At?

From almost all of the $\lambda$ plots, it is quite clear that the engine often runs very lean, or very fat at low RPMs and high RPMs in first gear. This is due to how the engine operates in these areas. The engine rarely goes to 4000 RPM in first gear, as it usually has geared up at that point. This means that we have few measurements in this region and they almost always have data from a point where the engine is still running at a very high RPM, but the injections have stopped, so there is no fuel in the exhaust, leading to a very lean measurement. The clutch also ensures that the engine isn't actually driving the car, until RPM is above 1800, which means that the $\lambda$ values below 1800 almost always are very lean, as the car is being driven solely by the electrical starter engine. Finally the second gear rarely dips below 2000 RPM. For these reasons, going forward, we should keep in mind that in first gear we are only interested in RPM [1800;4000], and in second gear [2000;4000].

# CHAPTER 5

# Tuning by RPM and Cylinder Temperature

## 5.1 Goal of Tuning by RPM and Cylinder Temperature

As we implemented the first and second iterations of our code, it became clear to us, that even though we had fulfilled the project as outlined in our initial project statement, this would not be a useful solution for the DTU RoadRunners team. For this reason, we expanded the project statement:

## 5.2 New Problem Statement

Currently, DTU Roadrunners energy efficient car has no system that ensures that the engine operates optimally, except for a simple LUT, which is adjusted by hand. A system will be designed for automatically tuning the car, and ensuring better engine operation. The system should fulfill the following requirements:

- When tuned, the air/fuel ratio measured in the exhaust should remain steady across different operating conditions.
- At the very least it must be able to automatically tune the car as well as when the car is tuned by hand.
- It must have a user interface, allowing the operator to monitor and control the process.
- The tuning process must be reliable, both in efficiency and duration.
- It must be possible to change the parameters of the tuning, for instance to tune the engine to run either lean or fat.
- It must be possible to overwrite the result of the tuning by hand, if certain results are found to be undesirable.

This was done to reflect our desire not only to tune the car, but ensure that the tuning would not drift over differernt operating conditions.

## 5.3 New Project Outline

Over the course of our project, our goal went from being ”design a system for automatically tuning the Dynamo’s engine control LUTs in the ECU”, to ”design a system for automatically tuning the Dynamo’s engine control LUTs in the ECU and expand the engine control to ensure optimal engine operation across different conditions”, where optimal engine operation is understood to mean steady $\lambda$ value in the exhaust.

## 5.4 Introduction

To facilitate tuning by both RPM and temperature the simulated hand tuning code was expanded, so that it would repeat the existing tuning process around each chosen temperature range, until every temperature the engine operates at has been accounted for. The resolution of the temperature axis was chosen to be $5^o$C, as this was already the temperature limits in the temperature regulation code from the last iteration of simulated hand tuning.

## 5.5 Design

### 5.5.1 The Two Dimensional LUT

As a system for tuning for controlling temperatures had already been implemented for the last iteration of simulated hand tuning, it was decided to expand upon this system. The system now tunes at one temperature, at which point it increments the temperature tuned at by five, and then repeats the process for this temperature. The injection times are then stored in a LUT, similar to the one seen in table 5.1. It was also decided to increase the resolution of the RPM indices in an attempt to make the process faster.

| | 0RPM | 1750RPM | 1875RPM | ... | 3750RPM | 3875RPM | 4000RPM |
|---|---|---|---|---|---|---|---|
| $60^o$C | $t_{inj}[0][0]$ | $t_{inj}[0][1]$ | $t_{inj}[0][2]$ | | $t_{inj}[0][16]$ | $t_{inj}[0][17]$ | $t_{inj}[0][18]$ |
| $65^o$C | $t_{inj}[1][0]$ | $t_{inj}[1][1]$ | $t_{inj}[1][2]$ | | $t_{inj}[1][16]$ | $t_{inj}[1][17]$ | $t_{inj}[1][18]$ |
| ⋮ | | | | | | | |
| $85^o$C | $t_{inj}[5][0]$ | $t_{inj}[5][1]$ | $t_{inj}[5][2]$ | | $t_{inj}[5][16]$ | $t_{inj}[5][17]$ | $t_{inj}[5][18]$ |
| $90^o$C | $t_{inj}[6][0]$ | $t_{inj}[6][1]$ | $t_{inj}[6][2]$ | | $t_{inj}[6][16]$ | $t_{inj}[6][17]$ | $t_{inj}[6][18]$ |

Table 5.1: The general structure of the new injection LUTs.

### 5.5.2 Bi-linear Interpolation

The specific injection time for a given operating condition, is then found by bi-linear interpolation between indices, using the following equations:

First it uses the following functions to convert from water temperature and RPM to indices in the LUT:

$$W(t_{water}) = \begin{cases} 0 & t_{water} < 60 \\ 6 & t_{water} > 90 \\ \frac{t_{water}}{5} - 12 & \text{otherwise} \end{cases} \tag{5.1}$$

$$R(RPM) = \begin{cases} 0 & RPM < 1750 \\ 18 & RPM > 4000 \\ \frac{RPM}{125} - 14 & \text{otherwise} \end{cases} \tag{5.2}$$

The upper and lower index for both RPM and water temperature is found, as well as the increment:

$$RPM_{lower} = \lfloor R(RPM) \rfloor \tag{5.3}$$
$$RPM_{upper} = RPM_{lower} + 1 \tag{5.4}$$
$$RPM_{increment} = R(RPM) - RPM_{lower} \tag{5.5}$$
$$water_{lower} = \lfloor W(t_{water} \rfloor \tag{5.6}$$
$$water_{upper} = water_{lower} + 1 \tag{5.7}$$
$$water_{increment} = W(t_{water}) - water_{lower} \tag{5.8}$$

These values are used when calculating the interpolation:

$$y_1 = \tag{5.9}$$
$$(t_{inj}[water_{lower}][RPM_{upper}] - t_{inj}[water_{lower}][RPM_{lower}]) \tag{5.10}$$
$$\cdot RPM_{increment} + t_{inj}[water_{lower}][RPM_{lower}] \tag{5.11}$$
$$y_2 = \tag{5.12}$$
$$(t_{inj}[water_{upper}][RPM_{upper}] - t_{inj}[water_{upper}][RPM_{lower}]) \tag{5.13}$$
$$\cdot RPM_{increment} + t_{inj}[water_{upper}][RPM_{lower}] \tag{5.14}$$
$$t_{inj,calc} = (y_2 - y_1) \cdot water_{increment} + y_1 \tag{5.15}$$

How the engine operates based on the conditions can be seen in figure 5.1.

### 5.5.3 Flowchart for Tuning with LUT

For a more in depth description of our code, please refer to appendix B. In order to tune the LUT, the simulated hand tuning state machine was further extended to handle

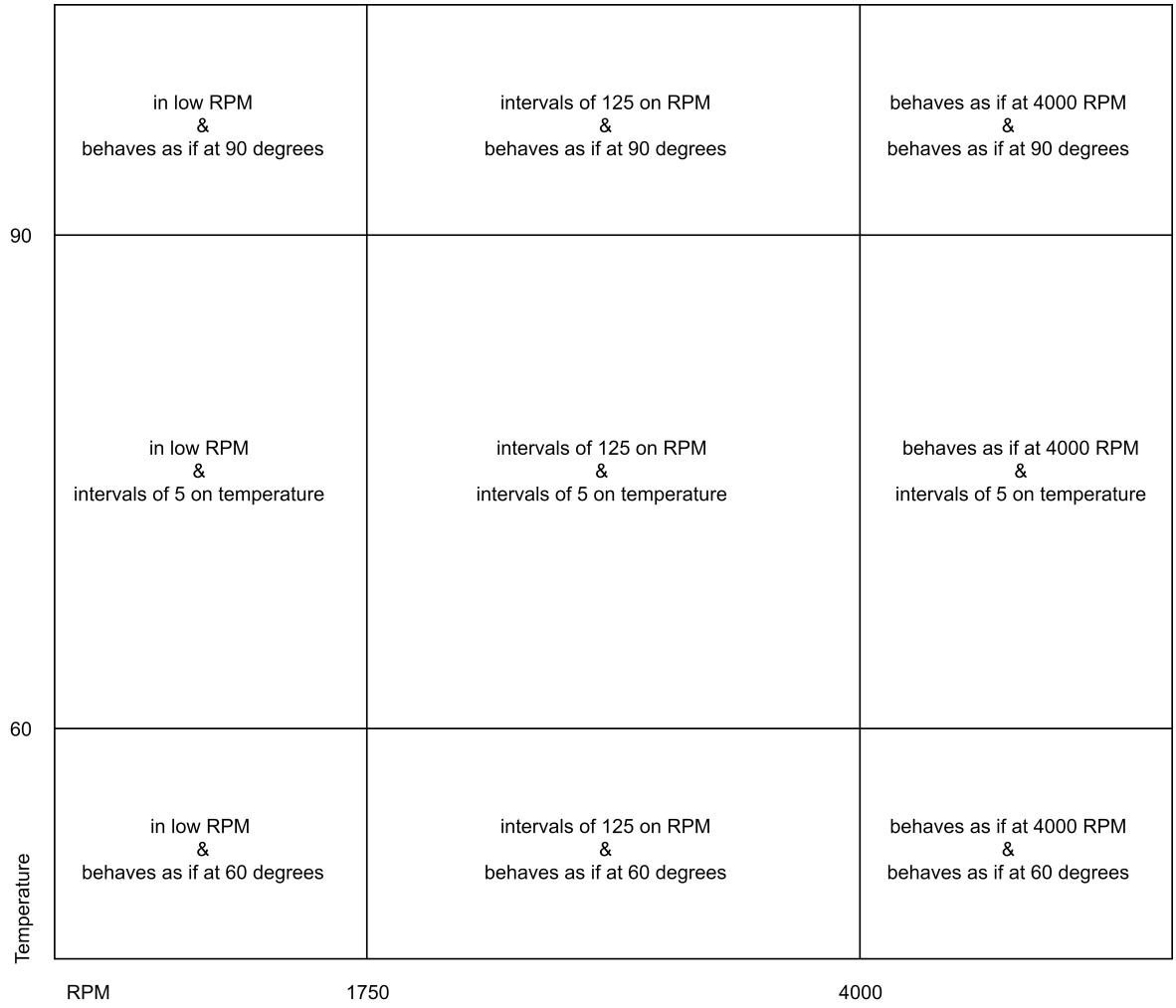



Figure 5.1: The different conditions the car can operate in. Generally speaking the car will be operating in the middle, which is why this is the area we focus on when we tune it. This is meant to visualize the how the $W(t_{water})$ and $R(RPM)$ functions effect how the engine operates

temperature. This is mainly done through an internal variable which tracks which temperature we are currently tuning at. The system takes advantage of the fact that the engine generally will be warm enough to be in the next temperature index whenever it finishes a sweep. Additionally, the state machine was extended to handle various additional features, such as tuning from a specific temperature, or pausing an ongoing session. The flowchart can be seen in figure 5.2. The state machine follows:

- **NOT_TUNING**
  This is the default state for the car, when the car is operating normally.
- **INIT_SWEEP**
  This state works as it did in simulated hand tuning.
- **SWEEPING**
  This state works as it did in simulated hand tuning, except for the added feature to pause the sweep, which sends it back to **NOT_TUNING**.
- **PROCESS_DATA**
  Processes data and adjust injection as before, with a few changes. It now processes data according to temperature and RPM, injection times will now be adjusted even if within the desired $\lambda$ band, and if a sweep has been accepted it will go to the **NEXT_TEMP**, it will also go to this state if the sweep has been approved by the operator. If the current temperature is higher than the upper temperature limit, it will go to **EXIT_TUNING**, if the sweep is not accepted, it will go to **SPEED_DOWN**.
- **SPEED_DOWN**
  Works as it did in simulated hand tuning.
- **NEXT_TEMP**
  This state increments the current temperature and checks if the current temperature is higher than the upper limit. According to this it goes to either **EXIT_TUNING** or **SPEED_DOWN**.
- **WAITING**
  This state works as it did in simulated hand tuning, with an additional check for if temp has been approved by the operator. If it has, it goes to **NEXT_TEMP**.
- **EXIT_TUNING**
  Works as it did in simulated hand tuning.

## 5.6 Optimizing Tuning Time

The biggest problem with this design, is that it takes a long time to tune the entire map. In order to optimize time we have made some adjustments to the correction calculation.

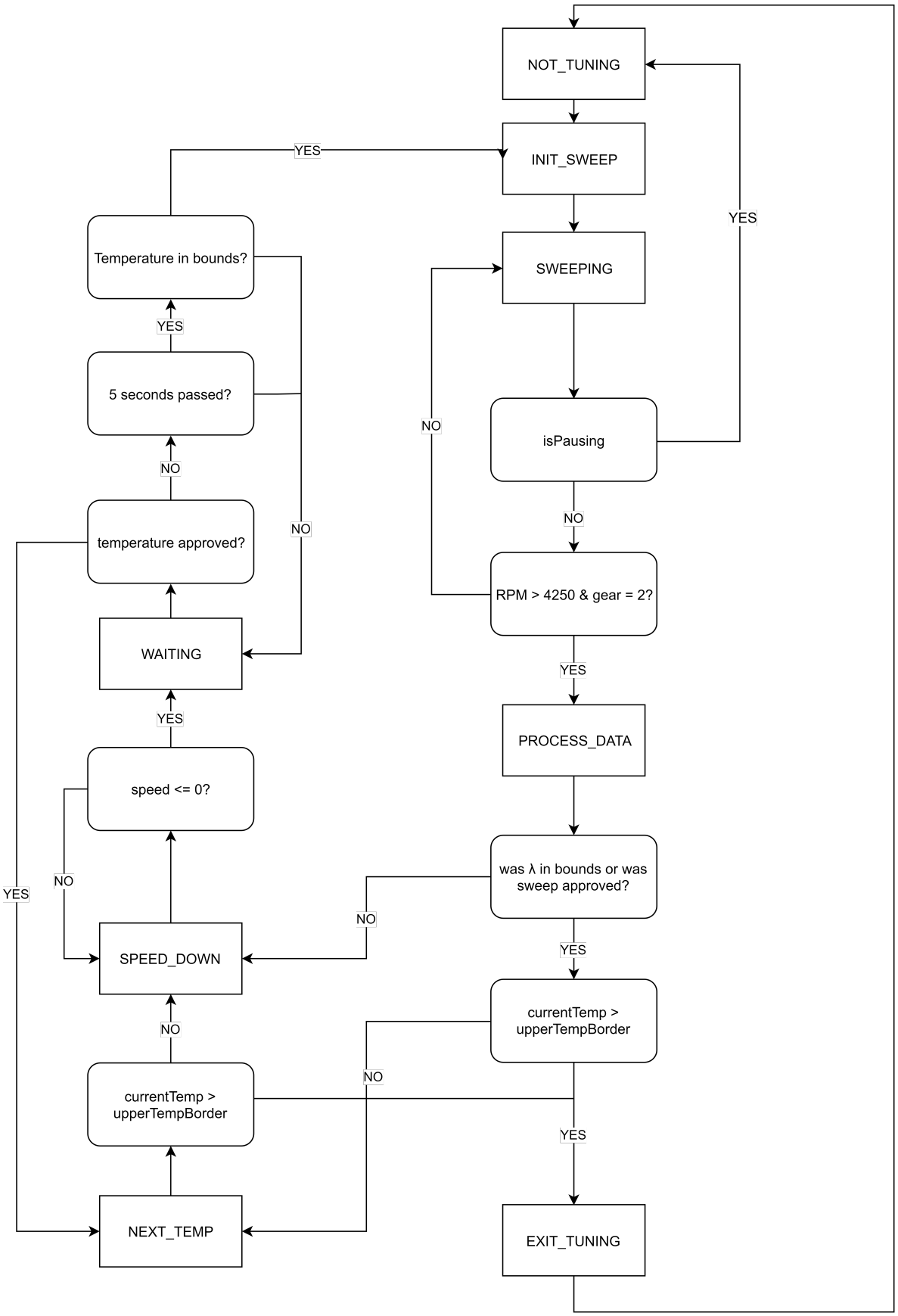



Figure 5.2: The flowchart for the final iteration of our tuning program. This flowchart is meant to give an overview of how the state machine goes between states

First the correction factor has been changed into an array, so that different corrections can be chosen for different RPM values. This was done as we noticed, that for some RPM values, the $\lambda$ value would change drastically between sweeps, while on others it would only change by a little bit. With this setup, we can set correction to be fairly high, for the RPM values that change slowly, and smaller for the values that change faster. This should shorten the tuning duration.

As mentioned earlier, we also increased the resolution of the RPM buckets, as we believe that this would increase tuning time, as it makes it easier to give every bucket the right correction.

Finally, we altered the calculation of new injection time, so that if the $\lambda$ is 1.51 ie. out of bound, the correction will be multiplied by 10. This has not been done when the $\lambda$ is 0.8, as we feared the situation where $\lambda$ shifts between 1.51 and 0.8. Additionally, as we usually tune around 1.2 or 1.3, the difference between the desired value and 0.8 and the desired $\lambda$ tends to be larger than the difference between 1.5 and the desired $\lambda$. All of these adjustments mean that injection times are now calculated like so:

$$t_{inj,new}[j][i] = \begin{cases} t_{inj,old}[j][i] + (\lambda_{desired} - \lambda_{avg}[j][i]) \cdot p[i] & \lambda_{avg}[j][i] < 1.51 \\ t_{inj,old}[j][i] + (\lambda_{desired} - \lambda_{avg}[j][i]) \cdot 10 \cdot p[i] & \text{otherwise} \end{cases} \tag{5.16}$$

where

$$i = \lfloor R(RPM) \rfloor \tag{5.17}$$

$$j = \lfloor W(t_{water}) \rfloor \tag{5.18}$$

The correction values for each gear can be seen in table 5.2. It should be noted that as it stands, we have not had the time to find the optimal values for each bucket, the values have just been copied for each gear, and we have tried to make the correction larger at high and low RPMs, to compensate for the slower tune time in each end of the RPM range.

| RPM bucket | $p$ 1. Gear | $p$ 2. Gear |
|---|---|---|
| 0 | 2000 | 2000 |
| 1 | 2000 | 2000 |
| 2 | 2500 | 2500 |
| 3 | 2000 | 2000 |
| 4 | 2000 | 2000 |
| 5 | 2000 | 2000 |
| 6 | 2000 | 2000 |
| 7 | 1000 | 1000 |
| 8 | 1000 | 1000 |
| 9 | 1000 | 1000 |
| 10 | 1000 | 1000 |
| 11 | 1000 | 1000 |
| 12 | 1000 | 1000 |
| 13 | 1000 | 1000 |
| 14 | 1000 | 1000 |
| 15 | 1000 | 1000 |
| 16 | 5000 | 5000 |
| 17 | 5000 | 5000 |
| 18 | 5000 | 5000 |

Table 5.2: The various correction factors for each index.

All of these changes where made before London, except for increasing the resolution, which we did while at the competition, as we were still not satisfied with the tuning time. This means the effectiveness of that measure hasn't been tested in controlled circumstances.

## 5.7 The Shell Ecocar Marathon 2019

### 5.7.1 Preparation

We finalized the LUT iteration of our program a week before SEM began, however, as there was a lot of work that needed to be finished on the car, we did not have time to test or debug our code on the car, before we where at SEM, and the time we had at SEM was limited. The car was tested with our tuning from Sjællandsringen, which was around 1.3, and as the mechanical team was satisfied with its performance, it was decided to tune a map for a different $\lambda$. We therefore tuned a second map for $\lambda = 1.2$, and tested it. This test revealed that the car drove better with 1.3-tuning, and it was decided to use this tuning during the race. The 1.3-map was not entirely accurate around $85^o$C and $90^o$C, due to a bug in the interpolation. We therefore re-tuned the car around those temperatures for 1.3 before going on track for the race. While debugging we also

accidentally changed the injection times for 60$^o$C. which lead to these values running very fat when below 65$^o$C in the first race. As it only ran so cold for the very first burn, this luckily did not impact the race much.

### 5.7.2 The Race

Each car in the Urban Concept category gets four attempts. To facilitate this, there are four time slots for Urban Concept cars to race in, each time slot is 1 hour and 30 minutes long. As a valid race takes at most 40 minutes it is technically possible to drive twice in the same block. The original plan, as we understand it, was to drive twice in the first block, once with the $\lambda$ probe to collect data about the current tuning, and one without to avoid loosing mileage to the probe. However, after we finished the first race, which went very well and gave us a mileage of 429 $\frac{\text{km}}{\text{l}}$, so much time had passed with the technical team measuring fuel consumption, that it was unclear if we even would have time to finish another race. Instead it was decided to replace the car's engine, with the intention of racing in the time block six hours later. As the new engine would require an entirely different LUT, and as we only had six hours to change the engine, tune it, and test it to figure out the best driving strategy, we decided to try and optimize the tuning process, to make the program tune faster. This was done by increasing the resolution of the RPM indices from 250 RPM to 125 RPM. Before the second race the new engine broke, and we did not have the required spare part, so the rest of the evening was used to change the engine back.

The next day, we did not make it in time to race in the morning. We did however tune the car from 70$^o$C to 90$^o$C at $\lambda = 1.3$ immediately before the second race block that day, and that race was driven without a $\lambda$ probe. The race itself did not go very well, as the driver had to make a few more burns than would be optimal and the brakes where malfunctioning. In spite of all of these problems we made it with a mileage of 447km/l. This result was later invalidated due to problems off-track.

### 5.7.3 Drivers World Championship

Despite losing our best run, our team did still win our category, qualifying it for the DWC. We did not have the time to tune the car in preparation for the DWC. This means the car drove with our tuning from the invalided run. We ended up third in this race, and we ran out of fuel just as we crossed the finish line.

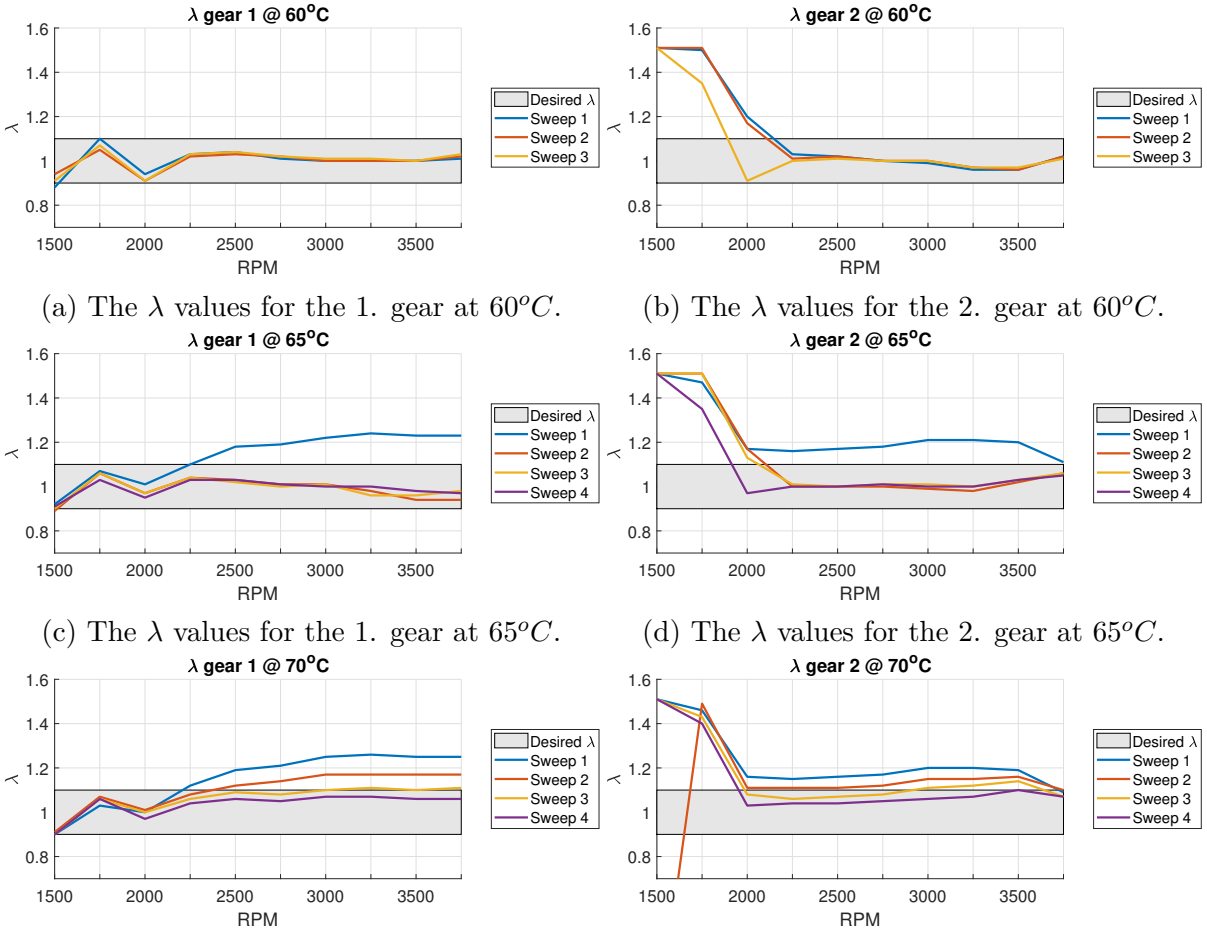
## 5.8 Results of Tuning With LUT

### 5.8.1 Testing of LUT tuning

Figure 5.3 displays a test of the final iteration of our LUT tuning. We did not complete this tuning, we just let it go through the first couple of temperatures in order to see how well it worked.

The final sweep at each temperature is the sweep which was accepted by the program.

From the text it is clear that the tuning itself does work, as it does move all of the $\lambda$ values towards the desired band. However, a few other things can also be observed. From figure 5.3b and figure 5.3d, it is clear that the $\lambda$ value at 2000 RPM in second gear can be difficult to control, as it in both cases doesn't move at all, until it all of a sudden jumps into the accepted region. This alone can drastically increase the tuning time, where all of the effort effectively goes to tuning the car at one RPM index. From figure 5.3e and figure 5.3f, it is also clear that the engine does not behave similarly at different temperatures. As the $\lambda$ value steadily decreases across the board at $70^oC$, where the $\lambda$ values at the other temperatures becomes almost tuned immediately except for at a few RPM indices.




(a) The $\lambda$ values for the 1. gear at $60^oC$.

(b) The $\lambda$ values for the 2. gear at $60^oC$.

(c) The $\lambda$ values for the 1. gear at $65^oC$.

(d) The $\lambda$ values for the 2. gear at $65^oC$.

(e) The $\lambda$ values for the 1. gear at $70^oC$.

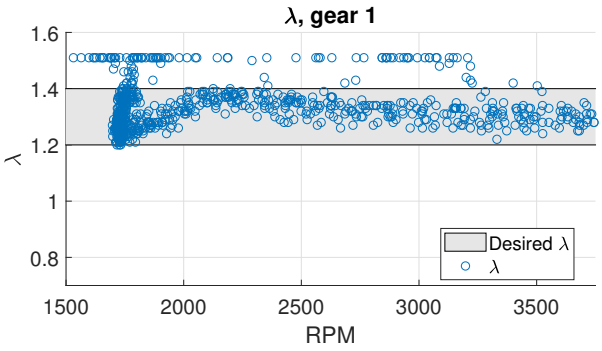
(f) The $\lambda$ values for the 2. gear at $70^oC$.

Figure 5.3: A test of our program for tuning the LUT. The test demonstrates how the injection times and $\lambda$ values change for each temperature. This is of course not a complete tuning, as it only goes to $70^oC$, but we believe that it is a clear demonstration of the program working with more temperatures. It is interesting to note that at $65^oC$ it becomes tuned in almost one sweep, except for at 2000 RPM in second gear, which takes it three additional attempts to tune.

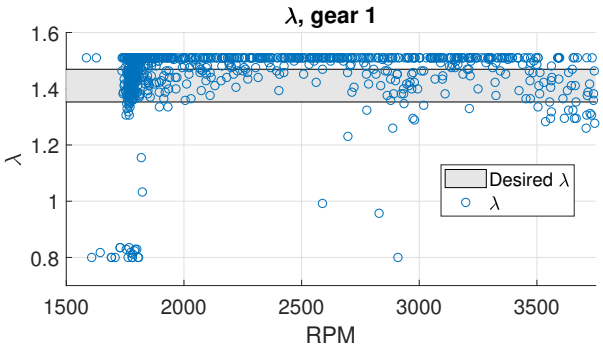
### 5.8.2 Comparison With Last Year's results

In figure 5.4, a comparison of our tuning and the tuning used last year can be seen. A few things should be noted. First of all, after discussing with the technical team, we decided to tune at 1.3 $\lambda$ as it is believed to be the optimal $\lambda$ value for our purposes. We fully tuned the LUT during the night of June 22nd 2019. The following day we drove the length of a full run on Sjællandsringen, which FDM kindly allowed us to test on. The data is the $\lambda$ values for each RPM throughout the race. This data is compared to one of our attempts from last year at SEM 2018. The $\lambda$ from the old run has been converted to the actual values compensating for the problem described in subsection 4.7.1, and

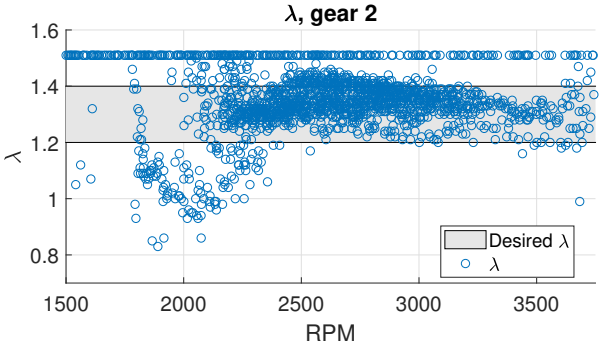
the band tuned for has also been adjusted. We do not know how well tuned the car was before the 2018 race, but we do know that it was tuned the night before. Therefore we think this is a reasonable comparison. It should be noted that there are generally many $\lambda$ measurements that are too lean. This is the result of the system taking measurements when the engine is not running, and the crank is speeding down. These measurements would of course be read as very lean, as there are no fuel in the exhaust when the engine isn't running.



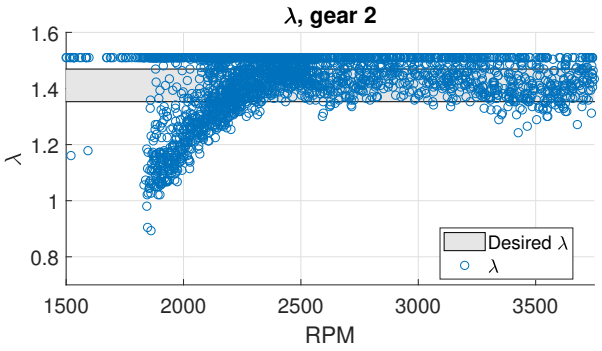

(a) The $\lambda$ values for the 1. gear at Sjællandsringen.




(b) The $\lambda$ values for the 1. gear from last year.




(c) The $\lambda$ values for the 2. gear at Sjællandsringen.




(d) The $\lambda$ values for the 2. gear from last year.

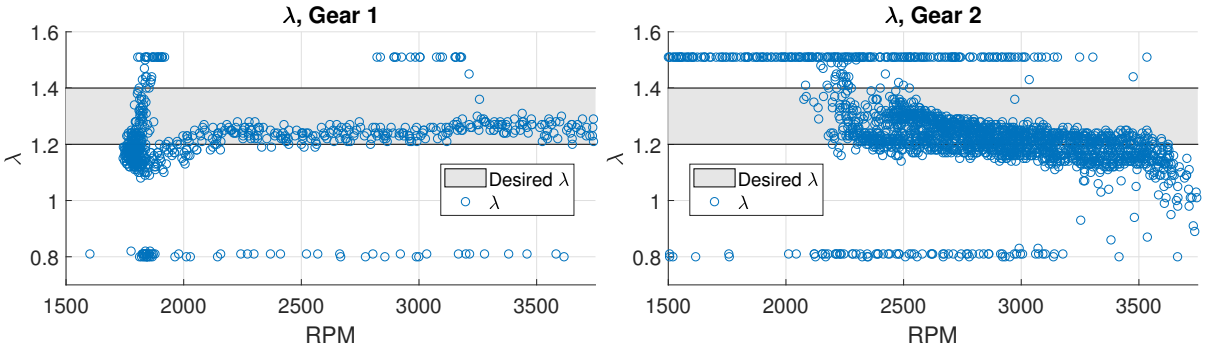
Figure 5.4: A comparison between a race from SEM 2018, and a test run on Sjællandsringen. Note that the results from last year uses the old conversion and such was effectively tuned to drive at roughly $\lambda = 1.4$. The most immediate improvement of our program is the reduction of the "handle" going from 2000 RPM to 2500 RPM in second gear from last year, as it in our case has been turned into a more of a cloud with fewer points, implying that our system spends less time in this state.

## 5.9 Results from The Shell Ecocar Marathon

The results for our one valid race can be seen figure 5.5.

We have compared our results from the race with the results from Sjællandsringen.

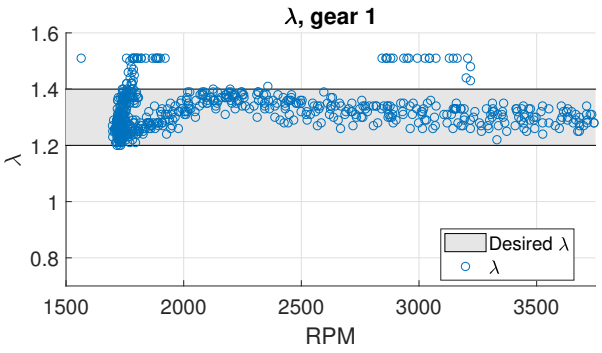
We believe this comparison is reasonable as the car drove with roughly the same tuning at both occasions, the only difference being the temperatures 85$^o$C, 90$^o$C and 60$^o$C, which we had changed since Sjællandsringen. In figure 5.6 the two runs are compared, ignoring the three temperatures that differ.



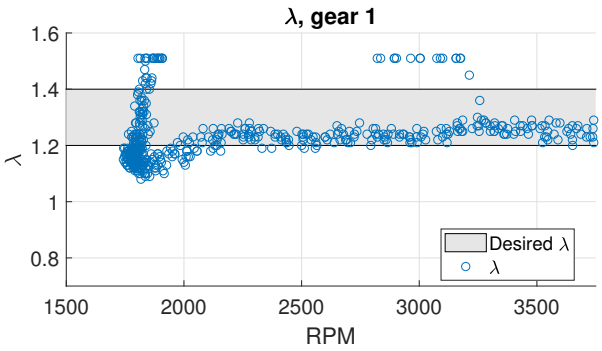

(a) The $\lambda$ values for the first gear.

(b) The $\lambda$ values for the second gear.

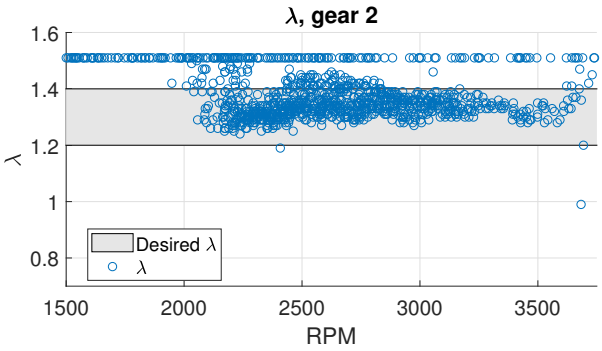
Figure 5.5: The various $\lambda$ values for first and second gear in our race with a mileage of 429km/l. It should be noted that the first burn which was at 60$^oC$ was poorly tuned and was far too fat, which is why there are a few $\lambda$ measurements at 0.8, but apart from this, it is clear that the first gear is still well tuned, and the second gear just on the edge of the accepted



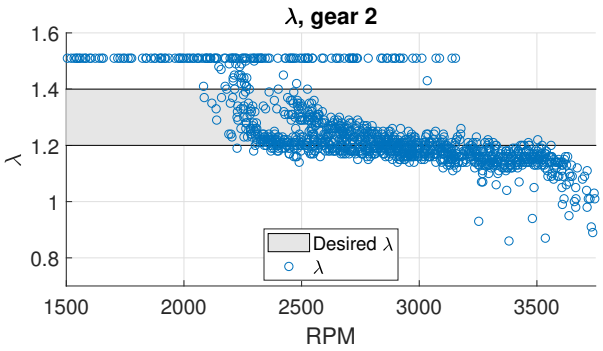

(a) The $\lambda$ values for the 1. gear at Sjællandsringen.




(b) The $\lambda$ values for the 1. gear from the race




(c) The $\lambda$ values for the 2. gear at Sjællandsringen.




(d) The $\lambda$ values for the 2. gear from the race.

Figure 5.6: A comparison between our valid race, and a test run on Sjællandsringen. It should be noted that we are only looking at values where the water temperature is in the range of $65^oC$ to $80^oC$ as the car used the exact same LUT in this temperature range. It then becomes clear that the tuning in general has drifted down, and especially in second gear it has drifted further at the high RPMs. It is also clear that the $\lambda$ values at 2500 RPM starts to split, this indicates that the engine can behave two different ways when in second gear.

It can be seen that for second gear $\lambda$ has drifted quite a bit, and it seems that the high RPM area has drifted further then the low RPM area. The low RPM area has also started to split, as if the engine is operating in to distinct conditions while in second gear. Figure 5.7 contains an example of a lap on the track. Here it can be seen that the car operates in second gear under two conditions: the first where it gears up from first gear, and the second where it starts in second gear, as the car is already driving at speed. To illustrate how the engine behaves differently in these two conditions we have graphed how $\lambda$ and RPM from every lap behaves in second gear in the two conditions, see figure 5.8.

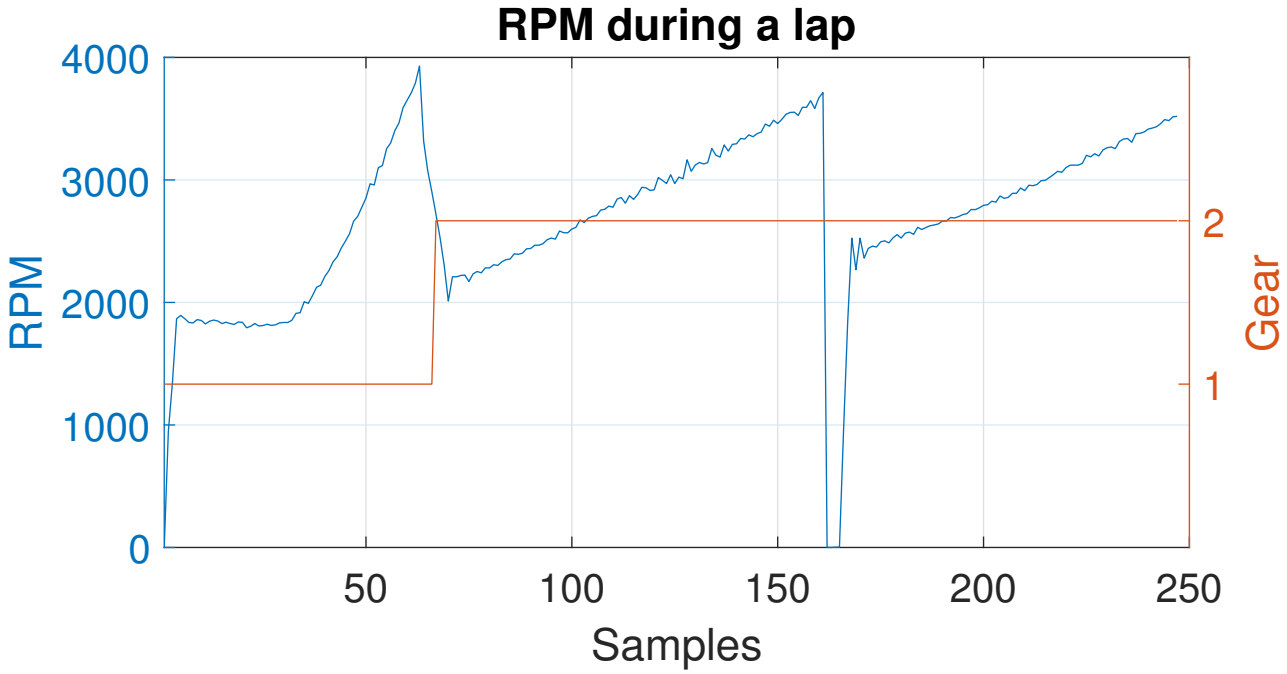
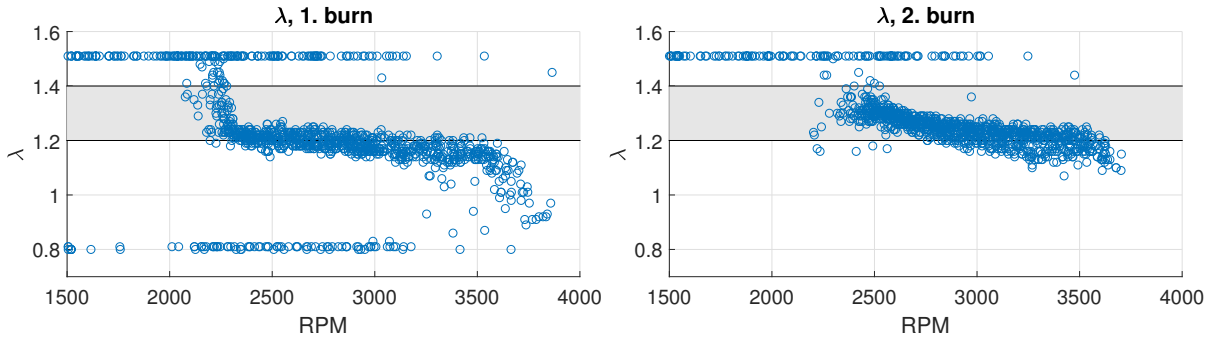



Figure 5.7: An example of how RPM and gear changes on a lap on the track. The first two RPM spikes are the initial burn, there is a time gap in the interval when RPM is zero, this time gap has been shortened for the ease reading. The third RPM spike is the final burn. This is could be the two conditions that the car drives in second gear under, changing from first to second gear, and having to start in second gear.

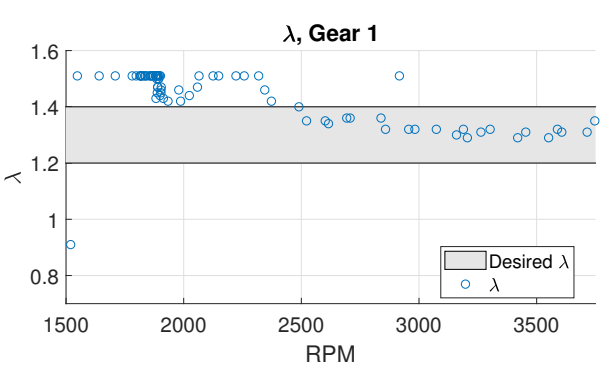



(a) The $\lambda$ values for the 1. burn.

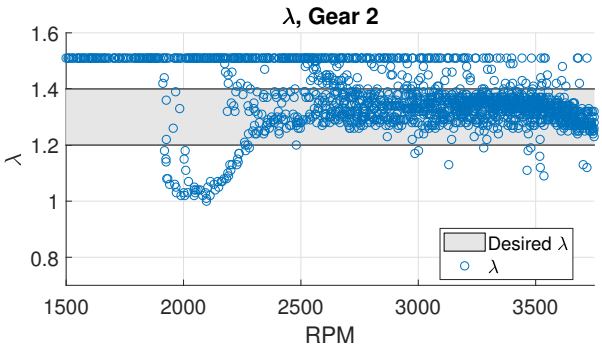
(b) The $\lambda$ values for the 2. burn.

Figure 5.8: A comparison between the second gear of both burns from our valid race. The $\lambda$ values does at first glance look fairly similar, bu there is a key distance, as the $\lambda$ values for the first burn, except for the spike in the beginning, follow an almost straight line up until 3500 RPM, where the RPM drops off. In comparison, in the second burn, the $\lambda$ values moves down on a slope from 2500 RPM to 3500 RPM.

Our $\lambda$ values from DWC can be seen in figure 5.9. We did not drive very fuel efficient, and we spend the entire race, except for the first burn, in second gear.




(a) Data from the single burn made in first gear.




(b) Data from the remaining burns made during DWC.

Figure 5.9: $\lambda$ across RPM during DWC. We drove the DWC with a complete tuning around 1.3 made the day before. This tuning has a low RPM handle similar to the one seen in figure 5.4d, aside from the handle, the $\lambda$ values are very consistently within the band, which is interesting, given that the engine was in second gear for almost the entire race, and thus constantly operating in the "second burn" condition.

## 5.10 Discussion of Results

From figure 5.3, it is clear that although our tuning does work, it could be faster, as it often has a lot of problems tuning low RPMs. Looking at the data from figure 5.4, it is clear that our system does keep the $\lambda$ values within the desired band for first gear. The second gear is mostly within the band, with a the main difference between our solution and the old solution being the slope from 2000 RPM to 2500 RPM, which we have almost eradicated. It is hard to tell just how accurate the tuning in the old solution was, as so many of the measurements are outside of the scope of the probe, and it seems that this also includes measurements while the engine is running.

Because of this, we cannot be entirely certain as to the nature of the $\lambda$-values of the old tuning, as we do not know exactly how high $\lambda$ goes. However in the most generous case, where $\lambda$ is 1.52 whenever the probe is out of scope, our tuning would still be better. Our $\lambda$ values only go out of the band around 2500 RPM, where the old solution has $\lambda$ values out of the band from 2000 RPM and down.

Our results of $429\,\frac{\text{km}}{\text{l}}$ is $55\,\frac{\text{km}}{\text{l}}$ above last years result of $374\,\frac{\text{km}}{\text{l}}$ and appendix $A$ indicates that had the $\lambda$-sensor been unplugged during the race, the result may have been over $500\,\frac{\text{km}}{\text{l}}$. As our project was the only project which significantly improved the mileage, this result on its own seems to indicate that our system works. Especially considering that we set that results with the tuning from Sjællandsringen, which was over a week old at the time. Of course direct comparisons between mileages should not be used as more than a guideline to give an idea of the state of the car, as many factors outside our control also impacts mileage. Such as the fact that the track last year was shorter, and had a sharp turn at the start, both of which had a negative effect on the

mileage.

From figure 5.6 is it clear so see that the $\lambda$ values for the first gear does drift a bit, but fairly evenly across RPM. The important part is that our $\lambda$ remains constant across RPMs, which means that our LUT can keep the engine operating with a stable output, even if our output drifts. From figure 5.8, we can see that both types of burns starts very lean. This could be the result of the delay from the driver pressing burn, and there being any fuel in the exhaust. Up until there being any fuel in the exhaust, $\lambda$ would just read 1.51. The $\lambda$ of the first burn quickly settles into an almost horizontal line which drifts off at high RPMs. These $\lambda$s are fairly consistent with what we would expect from a well tuned engine. Meanwhile the $\lambda$ of the second burn almost linearly goes down across the RPMs. This, of course, does not align with our desire for an $\lambda$ which remains as constant as possible across RPMs.

When we tune the car, we are burning until we reach the maximum RPM in second gear. This is essentially the same burn as the first burn on a lap, therefore it makes sense that this burn would be the most well tuned. It should also be noted that given that the this difference in burns was not apparent on from Sjællandsringen, it could be possible that this problem becomes worse as the tuning drifts over time. It could also just be that the conditions were better overall on Sjællandsringen. Our results from DWC points towards the first case, as it would seem that we spend most of the race within the accepted $\lambda$ band. The $\lambda$ does go down at low RPMs, but the $\lambda$ values does not seem to drift downwards at higher RPMs, so it would seem that even though the car is effectively operating in the second burn for the entire race, our program does manage to keep $\lambda$ in bounds. This implies that when the car has recently been tuned the $\lambda$ does behave fairly well, aside from the sag at low RPMs. It would be interesting to further test this, and perhaps implement a system so that both types of burns works well. This could for instance be achieved with a third engine map for the second burn.

# CHAPTER 6

# Future Work

## 6.1 Measure Air Intake

Future studies could investigate if introducing a flowmeter on the air intake would allow the expansion of engine control to also take the air intake into account. This could be through an even larger LUT or be the basis for a gray-box model, by having the air intake play in to some of the injection time calculations.

## 6.2 Code Cleaning

While documenting our code we have discovered quite a few pieces of legacy code which has been used to maintain earlier builds of our program, or used to debug our program. Additionally the ECU still has all of the functions necessary to run the old engine control.

## 6.3 Extending UI Functionality

In current version the $\lambda$ value around which we tune, as well as the width of the accepted band, are both hard coded constants in the ECU. It would be very useful to make fields where this could be manually by the operator, either in the Tune Tab or Configuration Tab.

In current version of the ECU UI has a function for extracting the Normal-Use log, see section 2.4.1. However the tuning log is stored in a different file, and can only be accessed by manually removing the SD card from the ECU Teensy. An extraction function for the tuning log through the UI would be extremely useful.

Additionally, a thorough test of the various functionalities of the UI should be conducted before SEM 2020, as we did not have the time to test them all.

## 6.4 Test of the Different Types Of Burn

As discussed in section 5.10, the operating condition in which the tuning drifts the most, is when the burn starts in second gear. As it stands, it seems that when the car has been tuned recently, this operating condition is not a problem, but it does drift over

time. We recommend a future study to figure out how the tuning drifts, if it drifts more in the second burn, and if it is possible to somehow compensate for this drift.

## 6.5 Optimization of Tuning Process

As it stands now, tuning the car takes around 1 hour. It does however use much of this time struggling with tuning a few RPM indices. Implementing something that could avoid this would help reducing tuning time greatly. This could be done by expanding on how we calculate the injection time correction, by making it better at recognize when these problems occur and working out how to avoid them. The easiest way to implement this would be by completely tuning the car many times, under many different conditions. Additionally the two arrays of correction factors could be experimented on to figure out the optimal values.

## 6.6 Further Expansion of LUT

As discussed in appendix B.3, we are currently using 25.97% of the EEPROM for storing the LUT. It is possible to expand the LUT even further, by perhaps increasing resolution or by adding more dimensions to the LUT. Another possibility we discussed with the mechanical team after DWC, would be to have several LUTs tuned for different $\lambda$ values, and some way for the driver to chose between them. This would allow the driver to chose between a fat map for when it is necessary to drive fast, and a lean map when it is necessary to drive more efficient. As we are limited by EEPROM size, it would not be feasible to have more than three maps stored, unless the memory usage was somehow optimized, or the maps made smaller.

# CHAPTER 7

# Conclusion

We have developed, implemented, and tested several iterations of an engine tuning program, in order to improve engine map stability, tuning ease, and tuning speed.

First, we documented the hand-tuning process for the car and then we implemented a semi-automated version. We tested this version and found the stability of the resulting engine maps to be unacceptable: it produced large drifts over short periods of time. After consulting our advisers, we ran tests to determine whether the maps drifted when other factors were fixed, specifically oil and water temperature. The test showed that water temperature is a sufficiently good predictor of engine map stability. We therefore designed a new engine map to take not only RPM but also water temperature as input, and we extended the automated tuning to handle the new map. After a complete tuning, we conducted a test at Sjællandsringen that confirmed the effectiveness of the new engine map. We made some additional changes to increase tuning speed, and before traveling to London with the team, we developed the UI to ease its use by people unfamiliar with the ECU's inner workings.

It is too early to assess the actual usability of the UI, since the schedule did not give outside operators a chance to interact with it. However, it does allow the operator to monitor and control the tuning process, as well as to manually overwrite the results, thus satisfying the requirements in the problem statement.

Our team won the race with a result of 429 $\frac{\text{km}}{\text{l}}$, giving a 55 $\frac{\text{km}}{\text{l}}$ improvement compared to 2018. As no other changes made to the car would have a significant impact on mileage, this result must in large part be owed to our new engine control system.

The calculations in appendix A suggest that had we been able to complete a run with the $\lambda$-sensor unplugged, this result might have surpassed 500 $\frac{\text{km}}{\text{l}}$.

# Bibliography


[al00] Horst Bauer et al. *BOSCH Automotive Handbook 5th edition*. SAE society of Automotive Engineers, 2000, page 117.

[Cha] SparkFun Electronics's YouTube Channel. *SparkFun According to Pete #55 - How CAN BUS Works*. URL: https://www.youtube.com/watch?v=YvsGuK9Up0E (visited on March 10, 2018).

[Dag13] Nikolaj Dagnæs-Hansen. "Bachelor Project: Closed-loop Lambda Control of Ecocar Engine". In: (2013), pages 43–72.

[Ecoa] Shell Eco-marathon. *2019 OFFICIAL RULES CHAPTER I*. URL: https://www.shell.com/make-the-future/shell-ecomarathon/europe/for-europe-participants.html. (accessed: 12.07.2019).

[Ecob] Shell Eco-marathon. *2019 OFFICIAL RULES CHAPTER II*. URL: https://www.shell.com/make-the-future/shell-ecomarathon/europe/for-europe-participants.html. (accessed: 12.07.2019).

[Elb13] Elbert Hendricks. Personal communication with Nikolaj Dagnæs-Hansen, author of [Dag13]. March 2013.

[Fra16] Benjamin Arnold Krekeler Hartz & Rasmus Frausing. "Master Project: Design, Construction and Testing of a Small IC Engine". In: (2016).

[Gmb] DDBST Dortmund Data Bank Software Separation Technology GmbH. *Saturated Liquid Density*. URL: http://ddbonline.ddbst.de/DIPPR105DensityCalculation/DIPPR105CalculationCGI.exe (visited on July 9, 2019).

[Inn] Innovate Motorsports. *LC-2 Digital Air/Fuel Ratio (Lambda) Sensor Controller Manual*. Online; accessed 12th July 2019.

[Ise12] Rolf Isermann. *Engine Modeling and Control - Modeling and Electronic Management of Internal Combustion Engines*. Springer, 2012. ISBN: 978-3-642-39933-6.

[Kar17] Andreas Ulrich Møbius & Benjamin Lucas Lewis Karlson. "Untitled Engine Control documentation for DTU Roadrunners". In: (2017), pages 2–4.

[Max15] Maxim Integrated Products, Inc. *Precision Thermocouple to Digital Converter with Linearization*. 2015. URL: https://cdn-learn.adafruit.com/assets/assets/000/035/948/original/MAX31856.pdf. Online; accessed 13th July 2019.

[Rol11] Heiko Sequenz Rolf Isermann. “Emission Model Structures for an Implementation on Engine Control Units”. In: (2011).

[Rol16] Heiko Sequenz Rolf Isermann. “Model-based development of combustion-engine control and optimal calibration for driving cycles: general procedure and application”. In: (2016).

[Ryg12] Jacob Mac Rygaard. “Master Project: Optimization of the fuel/air ratio in an SI engine”. In: (2012).

[s1215] Nicklas Høgh Iversen (s123092) & Andreas Riis Christiansen (s123128). “Motorstyring slutrapport - DTU roadrunnes årgang 2014/2015”. In: (2015), pages 9–10.

[Sca14] Scancon Industrial Encoders. *Engine Encoder Scancon SCA50-720*. September, 2014. URL: `https://www.scancon.dk/media/1194/sca50-specifications-17.pdf`. Online; accessed 11th July 2019.

[Sim19] Simon Friborg Mortensen. Conversion with Simon Friborg Mortensen, who wrote a bachelor where he tested the new engine. May 2019.

[Sor11] Spencer C. Sorenson. *Engine Principles and Vehicles*. ©Spencer C. Sorenson, Department of Mechanical Engineering, Technical University of Denmark, 2011, pages 183–186.

[Sto] Paul J. Stoffregen. *Teensy Technical Specifications*. URL: `https://www.pjrc.com/teensy/techspecs.html`. (accessed: 18.06.2018).

# Appendices

# APPENDIX A

# Fuel calculations according to Shell ecomarathon rules

The Shell Eco-Marathon rules do not contain exact formulas to follow and recreate their results. They do contain examples, and when the formulas extrapolated from those examples are followed the results yielded are in the same ballpark as the official results.

If the fuel used isn't gasoline the volume converted equivalent volume gasoline according to the following formula:

$$V_{fuel_equiv_gas} = \frac{V_{fuel}\rho_{fuel}LHV_{fuel}}{\rho_{gas}LHV_{gas}} \tag{A.1}$$

Power is converted to equivalent volume gasoline according to the following formula:

$$V_{power\_equiv\_gas} = \frac{E}{1000 \times 0.25 \times 0.57\rho_{gas}LHV_{gas}},\ E = \text{energy} \tag{A.2}$$

Fuel efficiency is then found in the straight foreward manner:

$$\eta = \frac{d}{V_{power\_equiv\_gas} + V_{fuel\_equiv\_gas}},\ d = \text{distance} \tag{A.3}$$

The lower efficient heating values of ethanol and gasoline as well as the density of gasoline (at 15$^o$C for an unknown reason) are given in the eco-marathon rules, but for some reason the density of ethanol is not. However the density of ethanol at 15$^o$C is fairly easily found to be approximately 0.7937 $\frac{\text{km}}{\text{l}}$ [Gmb].

At the Shell eco-marathon each team's car is mounted with official telemetry equipment, including a joulemeter and, if you're in the ICE category, a flowmeter. The official telemetry data for each team's car is available to that team, and no other, after each

run. To illustrate the difference between the results found when combining the above formulas with the telemetry data and the official results, let us consider the two runs from 2019 for which we have both an official result and telemetry data.

### A.0.1 The official run, conducted on the 3rd of July 2019 around 9:00 (local time)

According to the telemetry data received from Shell, the team drove 15.44 km, used 47947 J of energy from the car battery, and 0.040291338411315756 l of ethanol. For practicality's sake we will limit ourselves the 4 significant digits.

$$V_{fuel\,equiv\,gas} = \frac{V_{ethanol}\rho_{ethanol}LHV_{fuel}}{\rho_{gas}LHV_{gas}} \tag{A.4}$$

$$= \frac{0.0403\,\text{l} \times 0.7937\,\frac{\text{km}}{\text{l}} \times 26900\,\frac{\text{kJ}}{\text{kg}}}{0.7646\,\frac{\text{km}}{\text{l}} \times 42900\,\frac{\text{kJ}}{\text{kg}}} = 26.23\,\text{ml} \tag{A.5}$$

$$V_{power\,equiv\,gas} = \frac{E}{1000 \times 0.25 \times 0.75\rho_{gas}LHV_{gas}} \tag{A.6}$$

$$= \frac{47950\,\text{J}}{1000 \times 0.25 \times 0.75 \times 0.7646\,\frac{\text{km}}{\text{l}} \times 42900\,\frac{\text{kJ}}{\text{kg}}} = 7.796\,\text{ml} \tag{A.7}$$

$$\eta = \frac{d}{V_{power\,equiv\,gas} + V_{ethanol\,equiv\,gas}} = \frac{15.44\,\text{km}}{26.23\,\text{ml} + 7.796\,\text{ml}} = 459.1\,\frac{\text{km}}{\text{l}} \tag{A.8}$$

The official result for this run was 429 $\frac{\text{km}}{\text{l}}$.

An interesting calculation to do is to consider how much is lost to electric power:

$$\eta_{no\,power} = \frac{d}{V_{ethanol\,equiv\,gas}} = \frac{15.46\,\text{km}}{26.23\,\text{ml}} = 595.6\,\frac{\text{km}}{\text{l}} \tag{A.9}$$

$$loss = \eta_{no\,power} - \eta = 593.0\,\frac{\text{km}}{\text{l}} - 454.4\,\frac{\text{km}}{\text{l}} = 136.5\,\frac{\text{km}}{\text{l}} \tag{A.10}$$

The extremely large loss to energy is due the $\lambda$-sensor still being plugged in due to a last minute misunderstanding.

### A.0.2 The invalidated run, conducted on the 4th of July 2019 around 13:30 (local time)

According to the telemetry data received from Shell, the team drove 15.46 km, used 27501 J of energy from the car battery, and 0.044042196413858971 l of ethanol. For prac-

ticality's sake we will once again limit ourselves the 4 significant digits.

$$V_{fuel\,equiv\,gas} = \frac{V_{ethanol}\rho_{ethanol}LHV_{fuel}}{\rho_{gas}LHV_{gas}} \tag{A.11}$$

$$= \frac{0.04404\,\text{l}0.7937\,\frac{\text{km}}{\text{l}}26900\,\frac{\text{kJ}}{\text{kg}}}{0.7646\,\frac{\text{km}}{\text{l}}42900\,\frac{\text{kJ}}{\text{kg}}} = 28.48\,\text{ml} \tag{A.12}$$

$$V_{power\,equiv\,gas} = \frac{E}{1000 \times 0.25 \times 0.75\rho_{gas}LHV_{gas}} \tag{A.13}$$

$$= \frac{27501\,\text{J}}{1000 \times 0.25 \times 0.75 \times 0.7646\,\frac{\text{km}}{\text{l}} \times 42900\,\frac{\text{kJ}}{\text{kg}}} = 4.472\,\text{ml} \tag{A.14}$$

$$\eta = \frac{d}{V_{power\,equiv\,gas} + V_{ethanol\,equiv\,gas}} = \frac{15.46\,\text{km}}{28.48\,\text{ml} + 4.472\,\text{ml}} = 466.5\,\frac{\text{km}}{\text{l}} \tag{A.15}$$

The official result for this run was $447\,\frac{\text{km}}{\text{l}}$.

Once again it is interesting to see how much is lost to electric power:

$$\eta_{no\,power} = \frac{d}{V_{ethanol\,equiv\,gas}} = \frac{15.46\,\text{km}}{28.67\,\text{ml}} = 539.3\,\frac{\text{km}}{\text{l}} \tag{A.16}$$

$$loss = \eta_{no\,power} - \eta = 539.2\,\frac{\text{km}}{\text{l}} - 466.5\,\frac{\text{km}}{\text{l}} = 72.76\,\frac{\text{km}}{\text{l}} \tag{A.17}$$

The loss to energy is significantly lower that above, due the $\lambda$-sensor being removed in this run. However significantly more ethanol is used than in the official run. This is due to the brakes sticking in the first lap, as well as congestion on track, together meaning the driver had to make four more burns than should have been necessary, on a clearer track with less traffic and with functioning brakes.

None the less this result is useful, as we can use it estimate how well we might have performed in the official run, had we remembered to remove the $\lambda$-sensor before the race.

$$\eta_{ideal} = \frac{d}{V_{fuel\_equiv\_gas\_official} + V_{power\_equiv\_gas\_unofficial}} = \frac{15.44\,\text{km}}{26.23\,\text{ml} + 4.472\,\text{ml}} = 508.9\,\frac{\text{km}}{\text{l}} \tag{A.18}$$

### A.0.3 Discussion

Shell does warn that the delivered telemetry data may not correspond precisely with the official result. It is common knowledge among the participating teams that it probably won't, as demonstrated here. Shell Eco-Marathon organizers have been contacted with questions about where exactly the discrepancy originates, but they have not responded.

# APPENDIX B

# Implementation of LUT tuning on a Teensy 3.6

## B.1 Overview of Old Engine Control

This project has expanded on the engine control program that we helped develop last year. In order to facilitate the tuning process, we have added two header files, autoTune and autoTuneStateMachine, we have also expanded on the config header file.

## B.2 config

The config header file handles setting and changing all of the variables used by the car. It has functions for saving and loading variables in the EEPROM, as well as functions to calculate the interpolation used in the engine map. In this file we added the following functionality to support our expanded LUT:

- struct injectionLutHandler_t
  This structure contains a 7x19 matrix, which is the LUT for one gear. Two global instances of this structure exists, one for each gear. We decided to make this a structure containing a matrix instead of just a matrix, as structs are far easier to pass it between functions than matricies.
- float injection2LutFlat & float injection2LutFlat
  Two arrays of length $7 \cdot 19 = 153$, they contain every value in the two LUTs. They are used when reading from and writing to the EEPROM.
- void matrixToArray(injectionHandler_t*,float*)
  Takes one of the injectionHandler_t structures, and stores it in one of the flat LUTs
- void arrayToMatrix(injectionHandler_t*,float*)
  Takes one of the flat LUTs, and stores it in one of the injectionHandler_t structures
- void defaultInjectionLut1() & void defaultInjectionLut2()
  Sets the respective injectionLutHandler_t to the default values.

- injectionLutHandler_t getInjection1Lut() & injectionLutHandler_t getInjection2Lut()
  Returns the respective injectionLutHandler_t.

- void setInjection1Lut(float, int, int) & void setInjection2Lut(float, int, int)
  Changes the injection time to the float in the index addressed by the two integers. The new injection time is also saved in the EEPROM.

- void setInjection1LutFast(float, int, int) & void setInjection2LutFast(float, int, int)
  Changes the injection time to the float in the index addressed by the two integers. This new values is not saved in the EEPROM. This was done as we change the injection times the same place in the code as we check if our lambda values are in bound. In this place, we are not interested in wasting time writing to the EEPROM, especially when working on a teensy which lowers the clock rate when writing to the EEPROM.

- void saveInjectionLut()
  This function saves both of the injection LUTs to the EEPROM. It is called during tuning at a less demanding part of the tuning process, where we can afford to lower the clock rate of the teensy.

- struct INTERPOL_t
  From last year's version we have inherited the INTERPOL_t structure. This structure exists to facilitate the interpolation between RPM indices. We have expanded it to also handle the interpolation between water temperature indices. It now contains the following values:

  - int RPMlower
  - int RPMupper
  - float RPMincrement
  - int waterTemperaturelower
  - int waterTemperatureUpper
  - float waterTemperatureIncrement

- INTERPOL_t calculateInjectionInterpolation(float, float)
  This function used to only set RPMlower, RPMupper and RPMincrement. We have expanded it to also set waterTemperaturelower, waterTemperaturUpperlower and waterTemperatureIncrement. The function takes the engine RPM, and the water temperature as inputs. They are converted as follows:

$$RPM_{lower} = \left\lfloor \frac{RPM}{125} \right\rfloor - 14 \tag{B.1}$$
$$RPM_{upper} = RPM_{lower} + 1 \tag{B.2}$$
$$RPM_{increment} = \frac{RPM}{125} - RPM_{lower} \tag{B.3}$$
$$water_{lower} = \left\lfloor \frac{t_{water}}{5} \right\rfloor - 12 \tag{B.4}$$
$$water_{upper} = water_{lower} + 1 \tag{B.5}$$
$$water_{increment} = \frac{t_{water}}{5} - 12 - water_{lower} \tag{B.6}$$

These values are used when calculating the interpolation:

$$y_1 = \tag{B.7}$$
$$(t_{inj}[water_{lower}][RPM_{upper}] - t_{inj}[water_{lower}][RPM_{lower}]) \tag{B.8}$$
$$\cdot RPM_{increment} + t_{inj}[water_{lower}][RPM_{lower}] \tag{B.9}$$
$$y_2 = \tag{B.10}$$
$$(t_{inj}[water_{upper}][RPM_{upper}] - t_{inj}[water_{upper}][RPM_{lower}]) \tag{B.11}$$
$$\cdot RPM_{increment} + t_{inj}[water_{upper}][RPM_{lower}] \tag{B.12}$$
$$t_{inj,calc} = (y_2 - y_1) \cdot water_{increment} + y_1 \tag{B.13}$$

- float temp2tempIndex(float)
  This is a helping function that takes a temperature and returns the corresponding index, using the following conversion:

$$water_{index} = \begin{cases} 0 & t_{water} < 60 \\ 6 & t_{water} > 90 \\ \frac{t_{water}}{5} - 12 & \text{otherwise} \end{cases} \tag{B.14}$$

  It is used multiple placed in the program, in order to ease the conversion from temperature to index, whenever it is necessary.

- float rpm2rpmIndex(float)
  This is a helping function that takes an RPM and returns the corresponding index, using the following conversion:

$$RPM_{index} = \begin{cases} 0 & RPM < 1750 \\ 18 & RPM > 4000 \\ \frac{RPM}{125} - 14 & \text{otherwise} \end{cases} \tag{B.15}$$

  Like temp2tempIndex, it is used multiple times in the program in order to make the conversion easier.

- float tempIndex2temp(float) & float rpmIndex2rpm(float)
  These function handles the conversion the other way of temp2tempIndex and rpm2rpmIndex, using the following conversions:

$$t_{water} = (water_{index} + 12) \cdot 5 \tag{B.16}$$
$$RPM = (RPMindex + 14) \cdot 125 \tag{B.17}$$

## B.3 EEPROM

The entirety of our LUT is stored in the EEPROM. As our EEPROM is 4096 bytes, and as our LUT is filled with floats, each index takes up 32 bit, or 4 bytes. This means that the memory consumed by our LUT is:

$$\text{LUT\_SIZE} = \text{INDEX\_SIZE} \cdot \text{LUT\_HEIGHT} \cdot \text{LUT\_WIDTH} \cdot \text{NUMBER\_OF\_LUTS} \tag{B.18}$$
$$= 4 \cdot 7 \cdot 19 \cdot 2 = 1064\text{bytes} \tag{B.19}$$

Which this takes up about one fourth of the EEPROM:

$$\text{MEMORY\_USAGE} = \frac{\text{LUT\_SIZE}}{\text{EEPROM\_SIZE}} = \frac{1064\text{bytes}}{4096\text{bytes}} = 25.97\% \tag{B.20}$$

We store 185 additonal floats in the EEPROM, corresponding to 18.07% of the total memory. This means that we can roughly make the LUT three times as big before we run in to any memory problems.

## B.4 Autotune

The autotune header file handles all of the data processing related to tuning. It contains a class, tuningDataHandler, which contains the following functions and variables:

- Variables
  The tuningDataHandler contains the following variables, all of which are private.
  - int whatGearIsThisFor
    There are two instances of the tuningDataHandler class in the program, one for each gear. This is used to keep track of them.

  - float lambdas[7][19]
    Matrix containing the most recent average lambda values for all temperatures and RPMs, is used to calculate the new injection times, and overwritten when a new sweep at the same temperature begins. Saved on the miniSD-card for the tuningLog.
  - float oilTemps[7][19]
    Matrix containing the most recent average oil temperatues for all temperatures and RPMs. Saved on the miniSD-card for the tuningLog.
  - float airTemps[7][19]
    Matrix containing the most recent average intake air temperatues for all temperatures and RPMs. Saved on the miniSD-card for the tuningLog.
  - float exhaustTemps[7][19]
    Matrix containing the most recent average exhaust air temperatues for all temperatures and RPMs. Saved on the miniSD-card for the tuningLog.
  - float waterTemps[7][19]
    Matrix containing the most recent average water temperatues for all temperatures and RPMs. SSaved on the miniSD-card for the tuningLog.
  - float dataPoints[7][19]
    Matrix containing the number of data points for each set of measurements, used for finding the averages when done sweeping. Saved on the miniSD-card for the tuningLog.

- Functions
  The tuningDataHandler contains the following functions, all of which are public.
  - void reset(int)
    Takes all of the private matrices and sets all of the RPMs at a temperature given by the input integer to zero.
  - tuningDataHandler()
    A constructor.
  - tuningDataHandler(int)
    A constructor that also sets the gear this instance of the tuningDataHandler is operating with.
  - void getTuningData(uint32_t)
    Takes the sum of all of the varying data points at each index of the tempera-

ture currently tuning at for each RPM

$$dataPoints[water_{index}][RPM_{index}] = n \tag{B.21}$$

$$lambdas[water_{index}][RPM_{index}] = \sum_{i=0}^{n} \lambda_i \tag{B.22}$$

$$oilTemps[water_{index}][RPM_{index}] = \sum_{i=0}^{n} t_{oil,i} \tag{B.23}$$

$$airTemps[water_{index}][RPM_{index}] = \sum_{i=0}^{n} t_{air,i} \tag{B.24}$$

$$waterTemps[water_{index}][RPM_{index}] = \sum_{i=0}^{n} t_{water,i} \tag{B.25}$$

$$exhaustTemps[water_{index}][RPM_{index}] = \sum_{i=0}^{n} t_{exhaust,i} \tag{B.26}$$

– dataProcess(int)
Takes all of the data sums from getTuningData and divides them with the number of data points, to get the average. This average is stored in the same place as the sum, to save place, and to avoid having double the necessary amount of variables.

$$lambdas[water_{index}][RPM_{index}] = \frac{lambdas[water_{index}][RPM_{index}]}{dataPoints[water_{index}][RPM_{index}]} \tag{B.27}$$

$$oilTemps[water_{index}][RPM_{index}] = \frac{oilTemps[water_{index}][RPM_{index}]}{dataPoints[water_{index}][RPM_{index}]} \tag{B.28}$$

$$airTemps[water_{index}][RPM_{index}] = \frac{airTemps[water_{index}][RPM_{index}]}{dataPoints[water_{index}][RPM_{index}]} \tag{B.29}$$

$$waterTemps[water_{index}][RPM_{index}] = \frac{waterTemps[water_{index}][RPM_{index}]}{dataPoints[water_{index}][RPM_{index}]} \tag{B.30}$$

$$exhaustTemps[water_{index}][RPM_{index}] = \frac{exhaustTemps[water_{index}][RPM_{index}]}{dataPoints[water_{index}][RPM_{index}]} \tag{B.31}$$

– tuneMap(int)
This function checks all of the lambda values in the lambdas matrix, at the temperature given by the integer in the input. It the calculates the new injection time, and tracks if lambda was within the accepted band. If all lambdas was within the accepted band, it returns true. The RPMs looked at are determined by gear. In first gear it looks at 1750 RPM through 3750 RPM. In

second gear it looks at 2000 RPM through 4000 RPM. This function uses the setInjectionLut1Fast(float,int,int) & setInjectionLut2Fast(float,int,int) functions, as it repeatedly changes a lot of injection times.

- get functions for all of the data points
  These functions take RPM index and a water temperature index and returns the value in the matrix at the given index.

- functions not in the class
  - correction(float, float, float)
    Used to actually calculate the new injection time, the injection time, the correction factor and the average lambda, it then calculates the new injection times like so:

$$t_{inj,new}[j][i] = \begin{cases} t_{inj,old}[j][i] + (\lambda_{desired} - \lambda_{avg}[j][i]) \cdot p[i] & \lambda_{avg}[j][i] < 1.51 \\ t_{inj,old}[j][i] + (\lambda_{desired} - \lambda_{avg}[j][i]) \cdot 10 \cdot p[i] & \text{otherwise} \end{cases} \tag{B.32}$$

## B.5 Autotune Statemachine

The autotune statemachine header handles the state machine described in the flowchart in figure 5.2. This is done with a class called tuningStateMachine. The header has the following functions:

- Outside the class
  - enum tuningState_t
    Used to tell which state the machine should be in.
  - enum uiCommunicationsTypes_t
    Used for communication with the user interface.

- variables
  The state machine has a few variables stored privately:
  - tuningStat_t tuningState
    Tracks the state the machine is in.
  - bool isPausing
    Used to pause tuning.
  - bool goingToExit
    Used to overwrite that the tuning should stop when the current sweep is finished.
  - bool logTuning
    Used to tell when the data from the most recent sweep should be stored on the SD-card.

  - bool lambdaInBound1 & bool lambdaInBound2
    Used to track if the average lambda was inside the band in first and second gear.
  - uint32_t sweepCount
    Used to track which sweep we are at.
  - uint32_t waitTime
    Used in the **WAITING** state to wait 5 seconds
  - bool water2Hot & bool water2Cold
    Used to ensure that the water is within the desired temperature band.
  - uint32_t timeSinceLastGetDataForTuning
    Used in **SWEEPING** to wait a millisecond between running getTuningData(uint32_t).
  - bool exitOrSpeedDownConditions
    Used to check if the condtions are met to either go to **EXIT_TUNING** or **SPEED_DOWN**.
  - uint32_t currentTemp
    Used to track the temperature index we are currently tuning at
  - uint32_t upperTemperatureBorder
    Used to track the maximum temperature index, if it for instance is desired to tune up to $70^oC$ degrees. By default it is 6 ($90^oC$)
  - bool sweepApproved
    Used to track if an operator has approved the current sweep.
  - uint32_t tuneUICommunicationFlag
    Used for communication with the UI.

- Functions
  All of these functions are for the tuningStateMachine class, and they are all public.
  - tuningStateMachine()
    The constructor, initializes all of the variables used when going through the state machine.
  - void stopTuning()
    Completely stops the tuning. Stops the engine, sets currentTemp and sweepCount to 0.
  - void approveSweep()
    Sets sweepApproved to be true.
  - tuningState_t getTuningState()
    return the current state of the statemachine.
  - int goToNextTuningState()
    This state contains the actual state machine, which behaves as described in

the flowchar in figure 5.2. Here is an overview of when the functions from the config header and and from the tuningDataHandler class is called:

  * **SWEEPING**
    In this state getData(uint32_t) is called for the relevant gear once every 1millisecond.
  * **PROCESS_DATA**
    In this state dataProcess(int) is called for each gear, followed by tuneMap(int) for each gear.
  * **WAITING**
    Inthis state, saveInjectionLut() is called, as the car is just waiting in this state, and there has time to write alot to the EEPROM.

- bool requestTuningStart(uint32_t,uint32_t)
  If the car is not already tuning, it sets currentTemp and upperTempBorder to be the two input unsigned integers, and starts tuning. It also returns true if it starts a tuning, and false if a tuning is already in progress.
- bool requestTuningEnd()
  Sets goingToExit true, and returns true, if the car is already is tuning, otherwise it does nothing and returns false.
- bool currentlyTuning()
  Returns true if the car is tuning.
- bool getIsPausing()
  Returns isPausings
- void setIsPausing(bool)
  Rets isPausing to the input boolean.
- int getSweepCount
  Returns sweepCount.
- void incrementSweepCount()
  Increments sweepCount.
- void resetSweepCount()
  Sets sweepCount to 0.
- void setLogTuning(bool)
  Sets logTuning to the input boolean.
- bool getLogTUning()
  Returns logTuning.
- getCurrentTemp()
  Returns currentTemp.

APPENDIX C

# The Visual Interface of the Classic UI

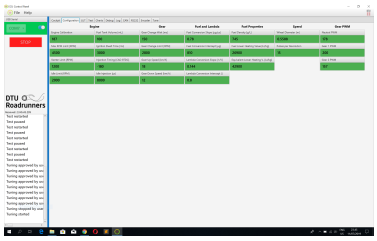
The UI has the tabs: Cockpit, Configuration, LUT, Test, Charts, Debug, Log, CAN, RS232, Encoder and Tune. LUT and Tune were produced by as a part of this project and are discussed in appendix D.

RS232 is a leftover from when the engine control was done by a NI RIO, which communicated with the rest of the control units indirectly by communicating over a RS232 protocol with the engine-board. This board is now no longer in use. This tab should therefore be removed.

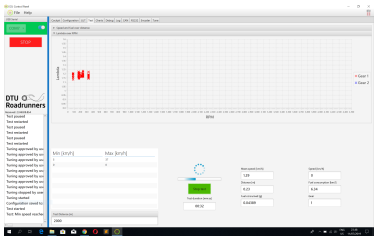
The rest of the tabs can be seen in figure C.1.

The upper right corner contains a toggle-button and a drop down menu underneath the text ”USB Serial”. The drop-down menu contains a list of all available COM-ports with a USB Serial connection. The toggle-button attempts to connect to over the chosen COM-port. The Big Red Button underneath breaks the connection to the ECU. The ECU will automatically stop burning upon the UI disconnection.

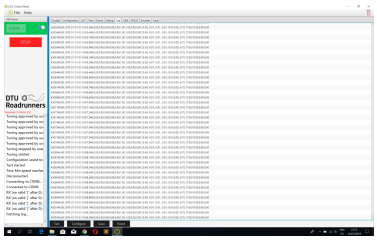
The bottom right corner contains a communications window through which the messages and error messages are printed. When a successful connection is established or broken, this is printed. Anything received over the Serial connection that cannot be identified as a Json object is printed. If a button is pressed, a message indicating if the action was successfully completed or not is printed.



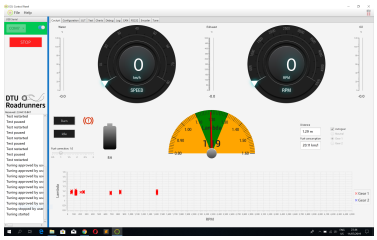
(a) Configuration allows the user to set constants use to calculate relevant data (such as giving the wheel diameter the ECU uses to calculate speed) and to control the car (such as the starter limit, which defines the engine crank rotational velocity at which the starter engine stops running and the ICE engine takes control).



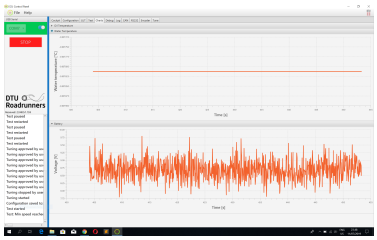
(b) Test allows the user to simulate competition. The intention is to mount the car onto the rolling road, preprogram a series of burns indicating the speed at which each burn starts and ends, as well as the length of the track to run. Once the test is started the relevant data is displayed and all data is logged.



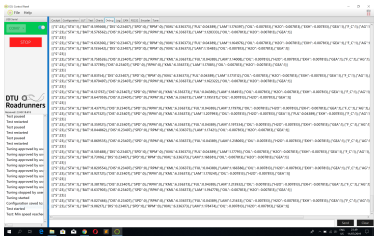
(c) Log allows the user to retrieve, save, and clear the logs taken and stored on the SD card by the ECU, as well as change the ECU's logging frequency and set the current time so the logs will be dated accurately and not simply at milliseconds since the Teensy last started up.



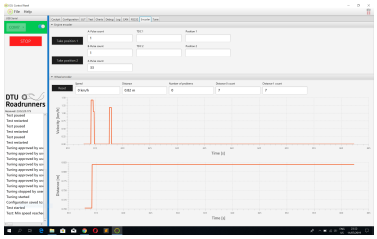
(d) Cockpit displays sensor data, distance and consumed fuel, as well as giving control over burn, idle and gear from the PC.



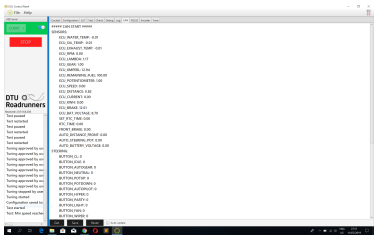
(e) Charts display oil temperature, water temperature and battery voltage over time, is updated as long as the UI is connected.



(f) Debug allows the user see what is sent over Serial from the Teensy as well as manually typing Json messages (and other things if they so desire) to send to the ECU.



(g) Encoder gives information from the engine and wheel encoders, mainly used for finding configuration values when a new wheel encoder is implemented or the crank shaft encoder is mounted on the engine in a different position than earlier.



(h) CAN allows the user to spy on the CAN communication between the control units, as well as send custom messages through the ECU.

Figure C.1: The tabs that existed in the UI before 2019

# APPENDIX D

# Extending the UI to support the new engine map and support user control of and interference with the tuning

In order to facilitate the interaction with the engine map and tuning program for others, the existing UI was extended. Please see section 2.4.1 for a description of the existing UI.

Two new tabs were introduced, the LUT tab and the Tune tab. Both can be seen in figure D.1.

As was the case in the previous UI, almost all actions cause a small message to appear in the lower left corner window. These will tell if the action attempted was successful or not.

## D.1 The Tune Tab

The Tune tab has the following features:

- Graphs of $\lambda$-values over RPM values, speed over time and distance over time. Each graph can be hidden, enlarging the size of the others. When a new tuning is started, the graphs are reset.
- The Start/stop tuning the button. The text displayed is ”Start Tuning” if tuning isn’t currently happening, and ”Stop Tuning” if an ECU is connected and a tuning is in progress. Pressing ”Start Tuning” sends the signal for the tuning process to begin if an ECU is connected. Otherwise it has no effect. ”Stop Tuning” sends a signal to the ECU causing it to finish the current sweep and then end the tuning.
- The Pause/Restart tuning button. The text displayed is ”Pause Tuning” unless a tuning is paused, in which case the text displayed is ”Restart tuning”. Pressing

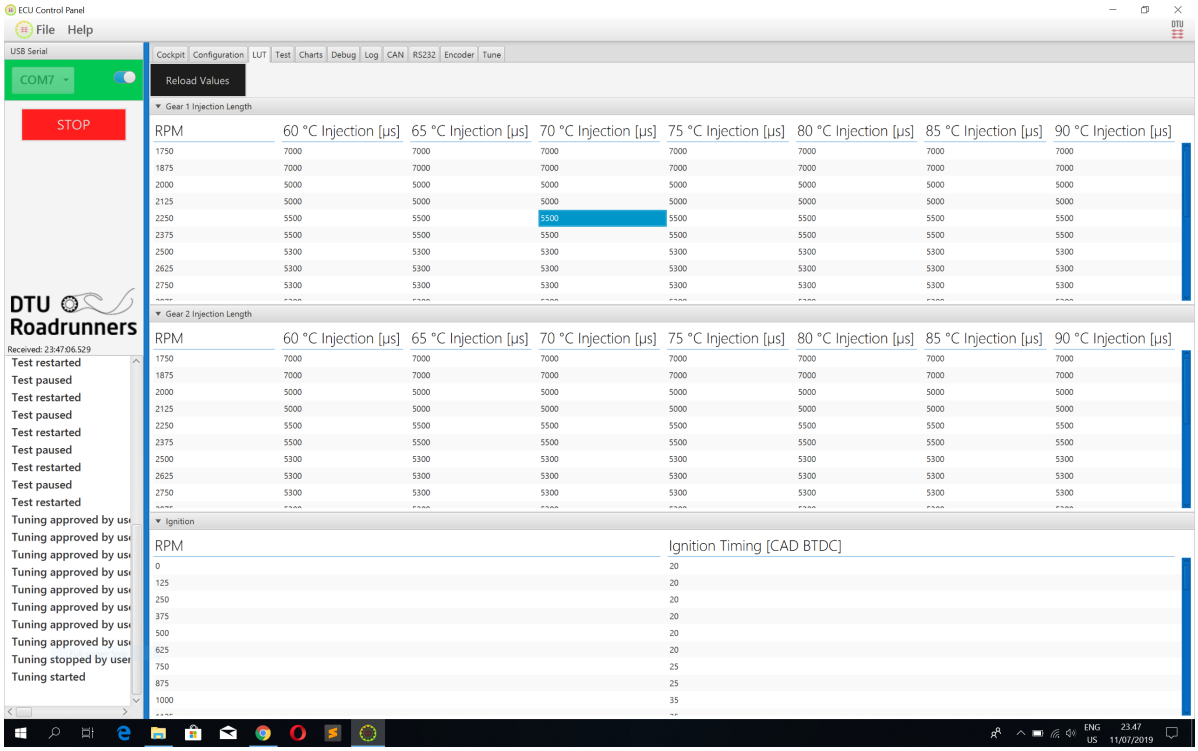
"Pause tuning" has no effect if no ECU is connected or no tuning is in progress. If an ECU is connected, a message is sent to ECU causing tuning to immediately pause and, if a sweep is in progress, halt that sweep and discard the data collected from it. No sweep can start while a tuning is paused. "Restart Tuning" causes the ECU to once again be startable.

- The cluster of text-fields in the lower right corner. While a tuning is in progress, each field displays the information whose name it is labeled with.
- The Approve Temp button. If a tuning is in progress, pressing this button will cause the ECU to finish the current sweep, save the map for the temperature that sweep had been occurring at, and move on to tuning the injection lengths at the next temperature. If no tuning is in progress, or no ECU is connected, it has no effect.
- Lower and Upper limit for tuned temperature. These two drop down menus allow the user to choose which range of temperatures to tune in. They should be set before pressing start tune, as changing their state them afterwards will have no effect. Not setting them before a tuning is started, will simply cause the start a tuning that will move through all available temperatures. Setting them so that the the Upper Limit is lower than the Lower Limit, and then pressing the Start tuning button, will cause the upper limit to be pulled up to the value of the lower limit, and a tuning for that temperature will be done.

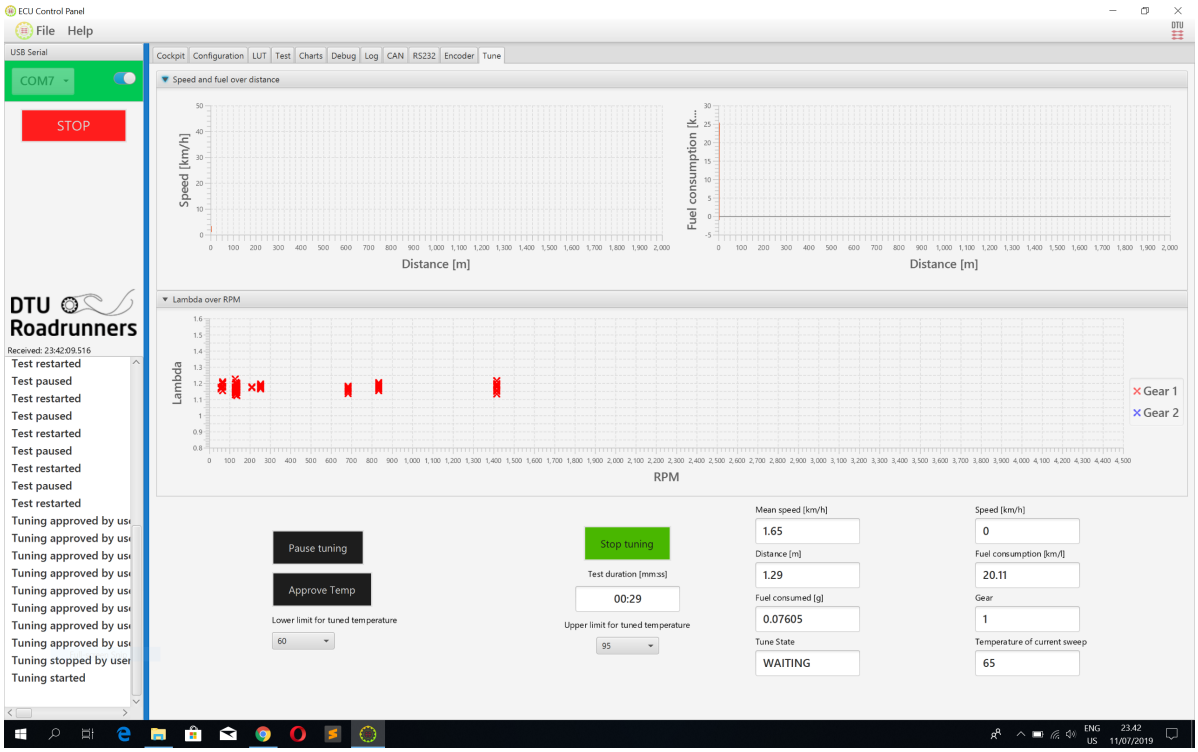
## D.2 The LUT tab

The LUT tab has the following features:

- A display of the content of each of the three LUTs in the ECU: The two 2-dimensional ones controlling injection-length as a function of temperature and engine rotational speed for each gear, and the 1-dimensional array containing the ignition timing as a function of engine rotational speed.
- Each value can be manually changed by double-clicking on the corresponding cell, typing in the new desired value and then pressing enter.
- A map on the ECU can be saved to the PC running the program by right-clicking anywhere on the arrays and choosing Export in the pop-op menu.
- A map can be loaded from the PC, onto the ECU by right-clicking anywhere on the arrays and choosing Import on the pop-op menu.




(a) LUT gives an overview of the content of each LUT in the memory.




(b) Tune is acts as the control interface for conducting a tuning.

Figure D.1: The two tabs added to ECU UI to facilitate the change in engine map and tuning process introduced in this project. Please see section D.1 and D.2 for a detailed description of their functionalities.

## D.3 Extension of ECU communication code

The ECU code code for PC communication, all done in the files PCSync.cpp and PCSync.j was extended to respond to the new UI features in two ways. Firstly the switch-case for interpreting and reacting to incoming Json messages was extended. Secondarily the the size-limit for messages was increased to allow for the newly sized LUTs to be sent in a single package.

**DTU Electrical Engineering**
**Department of Electrical Engineering**
Technical University of Denmark

Ørsteds Plads
Building 348
DK-2800 Kgs. Lyngby
Denmark
Tel: (+45) 45 25 38 00

www.elektro.dtu.dk