TSFS09 1C

October 1, 2025

1 Introduction

In project 1A, we selected our component models. In project 1B, we parametrized our component models to measured engine data. In this project, we will put everything together (in Simulink) to create a full engine model. We will then use this engine model to develop a controller for the fuel injectors.

This project consists of three parts:

- Completing the `template_model.slx` Simulink file.
- Developing a fuel controller.
- Analyzing the performance of our engine in terms of emissions and fuel efficiency.

2 Description of `template_model.slx`

First, let's briefly familiarize ourselves with the template code.

Note: the script `init_model.m` assigns values to all the constants needed in `template_model.slx`.

The root level of `template_model.slx` is shown in Figure 1. As might be clear from the names of the subsystems, the simulation environment contains completed models for the driver, the gearbox, and the rest of the vehicle. You should not edit the driver, gearbox, and vehicle subsystems. You will find that

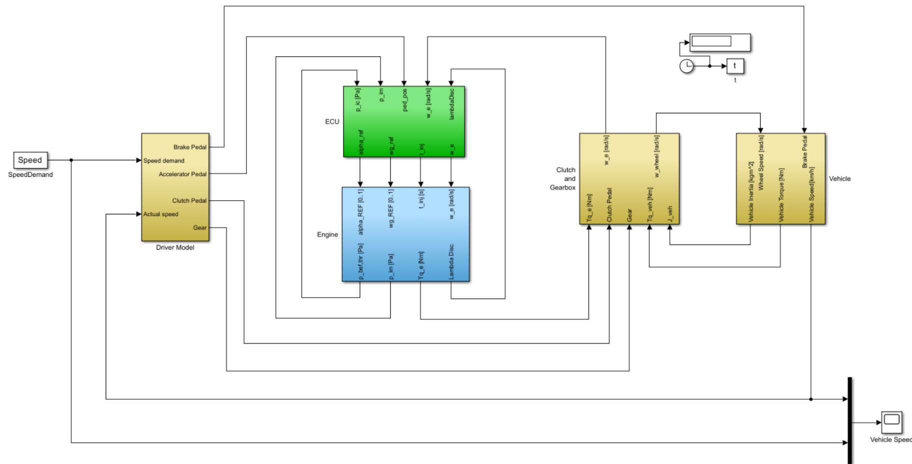

Figure 1: The root level of the template Simulink file. In this project, you will work in the `engine` and `ECU` subsystems.

the `engine` subsystem, shown in Figure 2, is unfinished. You are provided with some of the model "structure": a few of the required subsystems (throttle, intake, injector, and cylinder) and their corresponding inputs and outputs are already connected (while being empty). You are expected to finish the empty subsystems and create the rest on your own.

Note: To create a new subsystem, drag and click in Simulink: a rectangle will appear with some associated options. Choose "Subsystem".

The `ECU` subsystem contains the engine controllers. We will use the boost control subsystem in project 2: ignore it for now. The `Fuel Cont`-subsystem is empty: here we will develop our fuel controller based on the discrete lambda, intake manifold pressure, and engine speed signals.

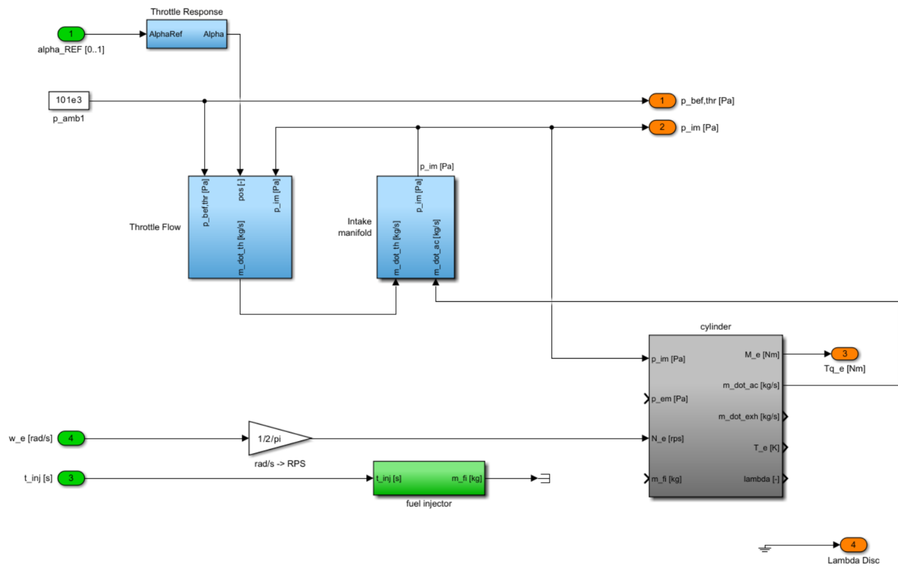

Figure 2: The unfinished `engine` subsystem.

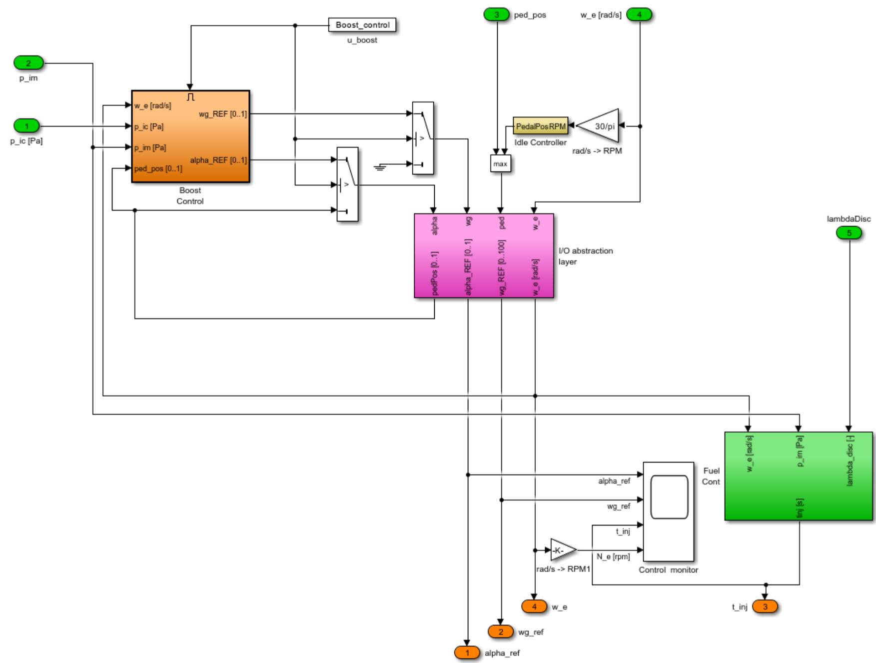

Figure 3: The ECU subsystem.

3 Fuel Injector Example

Continuing from previous projects, we provide the fuel injector as a completed example. The contents of the fuel injector subsystem are shown in Figure 4.

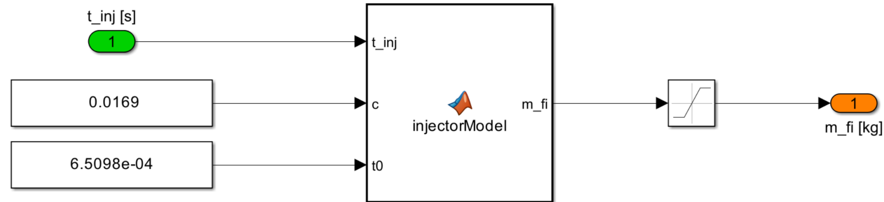

Figure 4: The completed fuel injector subsystem.

Here we have used a very convenient Simulink tool: the `Matlab Function`-block.

Naturally, Simulink understands Matlab code (in most places where you can write numbers, you can also write code). Open the `injectorModel`-block by double-clicking. You will then see a window where you can type standard Matlab code.

Since we have already completed the function files in project 1B, all we need to to is to copy the code over to Simulink!

Note: there is no requirement to *only* use Matlab functions in your model. You can of course also use Simulink equivalents (gains, sums, products) when it suits you.

- Follow the fuel injector example to complete the rest of the engine model!

4 λ-control

The task now is to develop a PI-controller that regulates the injector opening time, t_{inj}, to ensure that $\lambda = 1$.

Note: The task is very similar to the computer class, so make sure you have finished it (at least up until and including the fuel controller.)

Catalytic converters are used to remove harmful emissions from the exhaust. For catalytic converters to work well, we need to ensure stoichiometric combustion, i.e. $\lambda = 1$ (we inject precisely the right amount of fuel for the amount of air in the cylinder).

As you now know, we can measure λ of our exhaust, however, due to the time delay, our measurement of λ will always lag behind the actual value.

Suppose we use only feedback to make sure that $\lambda = 1$. Consider what happens when stepping on the gas pedal: additional air will instantly be supplied to the cylinder. For the duration of the time delay, λ will increase ($\lambda > 1$), and harmful emissions will slip through our catalytic converter.

What can we do about this?

4.1 Feedforward

Since we measure the engine speed N_e and intake manifold pressure p_{im}, we can use the models we have developed in project 1B to construct a *feedforward*. Essentialy, we invert our models to compute the appropriate t_{inj} from the measured N_e and p_{im}.

- Complete the `invertedInjector`-model in the models folder. Invert your fuel injector model: the inverted function should return an appropriate injector time given a fuel mass.
- Complete the feedforward part of your fuel controller. In the `Fuel Cont` subsystem, use the intake pressure and engine speed inputs along with your inverted injector model to compute an injector opening time. Use a constant volumetric efficiency ($\eta_{vol} = 0.7$).
- Run the simulation. Ensure that the vehicle is able to follow the drive cycle profile (assess in the vehicle speed scope). How does your λ behave? A proper feedforward controller should maintain a λ within a span of 0.7 to 1.5.

4.2 Feedback

In the feedforward part, we used a constant volumetric efficiency. The purpose of this is to simulate a model error: if we use the same volumetric efficiency

calculation in our controller and engine, the amount of injected fuel would result in a perfect $\lambda = 1$, and there would be no need for feedback. However, no model can perfectly describe the real volumetric efficiency, so there will always be an error, and consequently a need for feedback.

- Include a feedback part in your fuel controller. Use the discrete lambda signal as input to a PI-controller, with the K_p and K_i parameter values from the computer class.
- Run the simulation and assess λ. A properly tuned feedback controller should oscillate within the span 0.95-1.05. (It may deviate from this range during transients, i.e., very large acceleration).

5 Performance, Emissions & Fuel Consumption

With our model and controller complete, we can analyze the performance of our engine.

- Plot the velocity along with the reference speed of the drive cycle. Plot also the intake manifold and exhaust manifold pressure signals. Plot also the engine torque.

In the `emissions`-folder, you will find the function `calcEmissions()`, which you can use to estimate the produced emissions of your engine.

- Calculate the emissions produced by your engine during a 600s drive cycle (with and without feeback control). The minimum requirement to pass is EURO3 (using feedback). Use `To Workspace`-blocks to extract signals from Simulink into your Matlab workspace.

Note: If your engine fails to pass EURO3, your fuel controller might be faulty or poorly tuned.

- Finally, compute the fuel consumption of your vehicle in the unit litre gasoline per 100 km. Use $\rho = 780$ [kg/m^3] as density for gasoline.

6 Useful Simulink Blocks

- `Constant`: can be a vector.
- `Scope`: plots a signal versus time.
- `Input/Output`: creates inputs and outputs to the current subsystem.
- `Display`: shows the current value of a signal in your Simulink editor. Very useful to catch `inf` and `nan` which will not appear in a Scope.
- `Saturation`: limit a signal to stay within an upper and lower value. You can use `inf` or `-inf` as limits if the limit is one-sided (e.g., positive).
- `Matlab Function`: see description above.
- `To Workspace`: records a chosen signal for use in the Matlab editor. After a simulation is completed, a struct (typically called `out`) appears in your workspace with the recorded data. There are some formatting options (strucs, timeseries, etc.). Explore and use the format you find most convenient. **Note:** there are some irregularities with this block depending on the Matlab version: it might simply give you a variable instead of a struct. There is no issue using either.
- `Go To/From`: sometimes, connecting our models with lines can lead to messy models. With these connected blocks, we can let the signals "jump" between points.
- `Transport Delay`: delays a signal a given amount of time. Useful for modelling time delays.
- `PID`: a pre-built PID controller. Simply supply the tuning parameters for the proportional, integral, and derivative parts (zero values can be used to create a PI-controller).

7 Simulink Troubleshooting

Undoubtedly you will run into issues while using Simulink, and unfortunately, debugging a Simulink model can be very difficult. Since model outputs feed back as inputs to preceding models, errors propagate throughout the network and are sometimes challenging to find. In this section, we provide some techniques for you to try when encountering common errors.

7.1 Derivative of state is not finite

A very common error resulting when passing an infinite value to an integrator. Tricky to solve because **the reason why the signal carries an infinite value may have nothing to do with the subsystem in which the error is caught**. Simulink will warn for division by zero: be vigilant.

Potential causes:

- Division by zero.
- Forgetting to include initial conditions in your integrators.

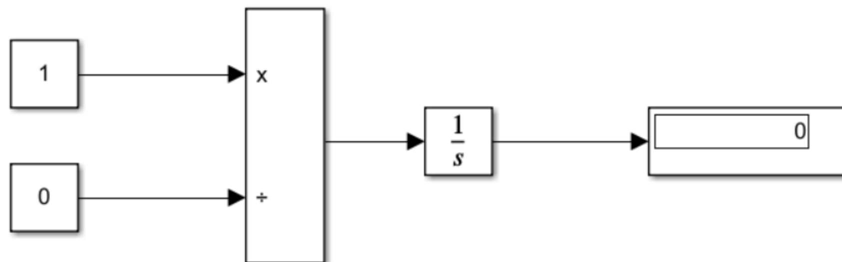

Figure 5: Division by zero before integration. Avoid!

7.2 Domain error

Simulink does not like complex numbers. If you use square roots in your Matlab function blocks, make sure to include appropriate guardrails against this: for example, using `min()`, or `max()` commands before the square root, or using `Saturation`-blocks on the function inputs.

7.3 Not enough information to determine output sizes

If we use matrices in Matlab function-blocks, we need to sometimes provide Simulink with our desired output sizes. This error can usually be fixed by

specifying, at the start of your function, the shape of the output in zeros.

7.4 Consecutive zero crossings

Simulink is particularly nervous about signals crossing from positive to negative and back. Many Simulink blocks include a counter for zero crossings. When the counter reaches a pre-determined value, the Simulation stops. Too many zero crossings within a certain time is considered, by default, as an error. Zero crossings can be ignored, but if you encounter this error, something is likely wrong in your model.

7.5 Algebraic loops

Feedback without integration: this leaves Simulink with having to solve the equation produced by the loop (See Figure 6, what is Simulink supposed to do here?). Sometimes, Simulink succeeds in solving the equation. Most often it fails. In this project we have no need for algebraic loops, so they should be avoided. Simulink will give you a warning if it finds one, but the simulation might not crash.

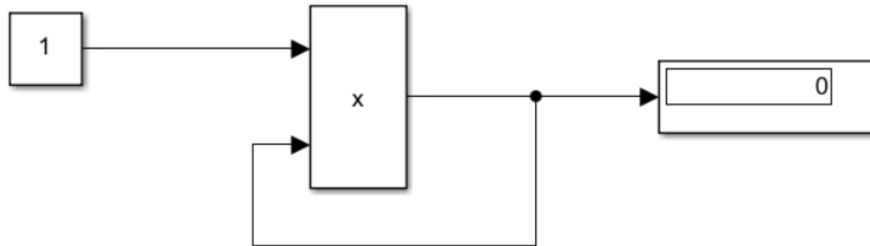

Figure 6: Algebraic loop. Avoid!

7.6 Solver failed to converge

We use an implicit solver in our template model (`ode23tb`, as shown in the lower right corner in Simulink). An implicit solver solves a non-linear system of equations every iteration, and sometimes, it just fails, without giving you any indication as to why. Usually, this error occurs after midnight: a potential solution is to go to bed and try again tomorrow.