TSFS09 1B

September 18, 2025

1 Introduction

In project 1A, we selected our component models and investigated what measurement signals and parameters were needed to characterize them. In this project, we will use measured engine data to find appropriate values for our unknown parameters. We will use two different techniques for two types of models, respectively:

- Least-squares regression for models of static type.
- Visual inspection for models of dynamic type.

By dynamic and static type, we mean models that include time differentials and those who do not.

1.1 Least-squares Regression

Here's a model:

$$y = kx + m.$$

It's a linear model, dependent on the unknown parameters k and m, as well as the signals x and y. With two unknown parameters (as you have learned in introductory math classes) we need two values each for x and y to solve for k and m. This will make a determined system of equations:

$$\begin{aligned} y_1 &= kx_1 + m \\ y_2 &= kx_2 + m, \end{aligned}$$

for which we can shuffle the variables to find k and m.

But: we might not have two values for x and y: we might have less than two, which leaves us with nothing to work with, or we might have more, which leads us to **least-squares regression**.

If we have more than two values each for x and y, the system of equations is overdetermined, and we can ask instead: what is the best choice of k and m?

In that case, we need to specify what "best" means. Typically, we mean "best" in the "least-squares" sense, that is, a choice of k and m that minimizes the sum

$$\sum_{k=1}^{N}[y_k - (kx_k + m)]^2,$$

if we assume we have N values for x and y.

As you will soon learn, we have 237 measurement points in our engine data. That means we have 237 equations to solve for each model, and generally only one, two, or a few parameters that are unknown. Therefore, we must use **least-squares regression** to find parameters for all static models.

1.2 Visual Inspection

There are two dynamic models we need to find time-constants for. As we know from basic control theory, a time-constant can be read from time-resolved step-response data (time from the beginning of the step to when the step has reached 63 % of the final value) which we measured during the engine lab session.

By visual inspection, we simply mean graphing the step signal (control input) versus the output response. From these signals, a time constant can be determined.

2 Project Files

On the LISAM course page you will find a project template folder. Here we will describe the files, what they do, and how you may use them.

First, a description of the folders and what they contain:

- **data:** Contains the engine data (`engine_data.mat`), a script for loading the data into the workspace (`load_engine_map.m`), and a script for loading the drive cycle we will use in Project 1C.
- **emissions:** Contains emissions calculation functions that we will use in Project 1C.
- **models:** Contains the Matlab functions where we will write our models. There is one function file per model. The `injectorModel.m` function is completed and may be used as an example.
- **parameterization:** Contains the Matlab scrips where we will parameterize our models. There is one script per model. The `injector.m` script is completed and may be used as an example.
- **simulink:** Contains Simulink files which will be used in Project 1C.

Note: Be sure to include the folders on your Matlab path. If the folders are grey, they are not included on the path. Right-click and add them.

3 Fuel Injector Example

Let's begin by looking at the provided example.

The fuel injector model we have chosen returns the fuel mass (per cycle per cylinder) m_{fi} as a function of the injector opening time t_{inj} in the following way:

$$m_{fi} = c_{fi}\ (t_{inj} - t_0)$$

Currently, we do not know what values should be assigned to c_{fi} and t_0. We can use the engine data to find the "best" values by least-squares regression.

There are multiple ways to do linear least-squares regression, but perhaps the easiest and most iconic (Mathworks even puts it on T-shirts) is the *backslash-operator*.

3.1 First of all, why backslash?

Consider this equation of scalars,

$$a\ x = b,$$

i.e., we have only one value for a and b, and we want to find x.

What do we do? Divide both sides by a:

$$x = b/a.$$

If we have a system of N equations instead:

$$\begin{aligned} a_1\ x &= b_1 \\ a_2\ x &= b_2 \\ &\vdots \\ a_N\ x &= b_N \end{aligned}$$

we could write it in matrix form and get

$$Ax = b.$$

If we want to find x, what do we do now? There is no division for matrices. The closest thing to division is a multiplication by the inverse (A^{-1}), which unfortunately doesn't exist for overdetermined systems. Thus, we must again consider the "best" value for x, in a least-squares sense.

In Matlab, we can use the backslash operator for this:

$$x = A \backslash b$$

The use of backslash is to indicate the relationship to division, as we saw in the scalar case.

3.2 Applying it to the fuel injector

To use the backslash operator, we need to shuffle our model into the form

$$A\ x = b.$$

Expanding the fuel injector model, we get

$$m_{fi} = c_{fi}\ t_{inj} - c_{fi}\ t_0.$$

As we can see, the model is not linear in the parameters c_{fi} and t_0, but with a slight change of variables $x_1 = c_{fi}$ and $x_2 = c_{fi} \cdot t_0$, we get

$$m_{fi} = x_1\ t_{inj} - x_2,$$

and since we have 237 values for m_{fi} and t_{inj}, we get the system of equations:

$$\begin{aligned} m_{fi,1} &= x_1\ t_{inj,1} - x_2 \\ m_{fi,2} &= x_1\ t_{inj,2} - x_2 \\ &\vdots \\ m_{fi,237} &= x_1\ t_{inj,237} - x_2 \end{aligned}$$

We can then write the model in our desired form by letting b be the vector of fuel mass data, and choosing an appropriate A:

$$b = \begin{pmatrix} m_{fi,1} \\ m_{fi,2} \\ \vdots \\ m_{fi,237} \end{pmatrix}, \qquad A = \begin{pmatrix} t_{inj,1} & -1 \\ t_{inj,2} & -1 \\ \vdots & \vdots \\ t_{inj,237} & -1 \end{pmatrix}$$

Note that with this choice of A and b, we have written the model on the desired form.

Now we can execute the backslash command and find our values for x. Finally, we only need to divide x_2 by x_1 to find t_0. Here is the Matlab code that does this:

```
% load engine data
load_engine_map;
```

```
% target
b = m_fi;

% regressors
A = [t_inj, -ones(237,1)];

% least squares fit
x = A\b;

% extract parameters
c_fi = x(1);
t0   = x(2) / c_fi;
```

Parameter fitting with this technique mainly involves wrangling the model into the form $Ax = b$, and finding the appropriate matrix A.

3.3 Model Validation

Now that we know how to find values for our model parameters, we need a way to determine if those values (and perhaps the model structure itself) is accurate enough.

Statisticians use measures for accuracy: root mean-squared error (RMSE), R-values, p-values, and so forth; we will not deal with any of these, because in our models, we have almost zero noise. A visual inspection of the model prediction versus the measured data is sufficient.

However, we must discuss what "accurate enough" means. We approach this course as automotive controls engineers, and so, the context of our modelling work is for the purpose of *developing controller*s. Thus, our models must be accurate enough to *be beneficial* in automotive control systems. How accurate is that? It's impossible to say without using the models in our control loops, which we will do in Project 1C. Unfortunately, determining the required accuracy of a model, and refining that accuracy if insufficient, is a cyclical process that involves engineering experience, which you will only have after finishing this course.

Some models will be dead-on accurate. Some will be on a threshold that might make you suspicious. Try to use intuition when determining which models are fine, and which are not. For example: when would a driver want accurate control of the throttle air flow? At high flows, when cruising on the highway, or at low flows, when manoeuvring into a parking space?

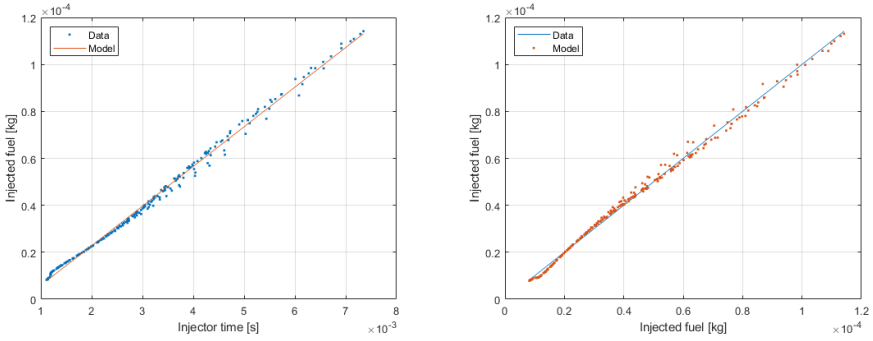
There are two ways to validate the model graphically:

- Plot input on the x-axis, and output on the y-axis.

- Plot the output on x and y-axes.

The first method is only possible for models depending on one input variable, while the second method works for all models we use in this course.

The idea of the second method is that the model error is represented as a deviation from a diagonal line, running through the figure. The fit-to-data of our fuel injector models are shown in Figures 1a and 1b.



(a) Validation of the fuel injector model with the first method.

(b) Validation of the fuel injector model with the second method.

Figure 1: Validation figures.

We can see that the model fit is quite accurate over the whole operating region, and thus, our model is sufficient. Here is the code that produced the figures:

```
figure;
plot(t_inj, m_fi, '.');
hold on
grid on
plot(t_inj, m_fi_mod);
xlabel('Injector time [s]');
ylabel('Injected fuel [kg]');
L = legend('Data', 'Model');
L.Location = 'northwest';

figure;
plot(m_fi, m_fi);
hold on
grid on
plot(m_fi, m_fi_mod, '.');
xlabel('Injected fuel [kg]');
ylabel('Injected fuel [kg]');
L = legend('Data', 'Model');
```

```
L.Location = 'northwest';
```

Note that to produce the diagonal line in Figure 1b, we have plotted the measured data against itself.

Study the fuel injector example and use it to guide the parameterization of the remaining models.

4 Tips

- To make elements-wise operations when working with vectors or matrices a "." is put before the operator (e.g .* for multiplication ./ for division). If only the operator is used, matrix operation will be performed!
- To save a figure from a Matlab plot the command `save` can be used. You can use this to get your scripts to automatically save the generated figures.
- Sometimes the data contains unwanted entries that need to be removed before using the data. This can easily be done in Matlab. For example, the following example shows how we can extract the positive values in a vector:

  ```
  y   = sin(1:10)
  idx = y >= 0;
  x   = y(idx);
  ```

  Try it for yourself and see what happens to x.