Cruise Control example

This is a simple document to demonstrate how RAMN can be used to safely study automotive systems with open-source tools. We program one of the ECU (the powertrain ECU), so that it controls the throttle of a car simulated by CARLA (https://carla.org) and makes it maintain a constant speed. **This is for education and not for automotive use**.

We follow a similar approach to this tutorial from the University of Michigan:

https://ctms.engin.umich.edu/CTMS/index.php?example=CruiseControl&section=ControlPID

And this handbook:

https://www3.nd.edu/~pantsakl/Publications/348A-EEHandbook05.pdf

System modeling

For our study, we consider the following (simplified) model.

$$F_{friction} = \mu . v \qquad F_{powertrain} = \beta . throttle$$

The engine applies a positive force proportional to the throttle ($\boldsymbol{\beta . throttle}$) and the friction of the road/wind applies an opposite force proportional to the speed of the vehicle ($\boldsymbol{\mu . v}$).

According to Newton's second law of dynamics we expect:

$$\boldsymbol{\beta . throttle - \mu . v \; - m\frac{dv}{dt} = 0}$$

Where v is the speed of the vehicle, and m is the mass of the vehicle.

We can use the Laplace transform to get rid of the derivation operator.

$$\mathbf{\beta . throttle(s) - \mu . v(s) - mv(s) . s = 0}$$

Which allows us to find the relationship between the speed and the throttle:

$$\boldsymbol{v(s) = H(s) . throttle(s)}$$

Where H is the transfer function of the car and is defined as:

$$\boldsymbol{H(s) = \frac{\beta}{\mu}\frac{1}{1 + \frac{m}{\mu}s}}$$

This is a well-studied “first order system”. When a constant throttle is applied, we expect the vehicle to follow an exponential curve to a maximum speed of $\frac{\beta}{\mu}$ with a time constant of $\tau = \frac{m}{\mu}$.

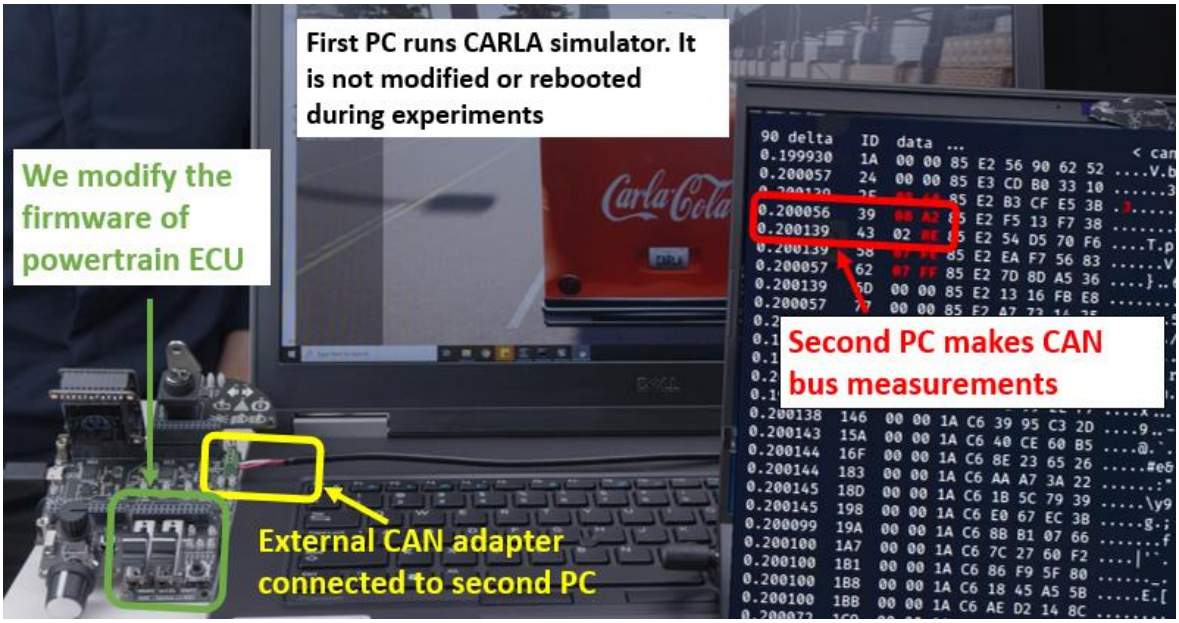
$$v(t) = \frac{\beta}{\mu}(1 - e^{\frac{-t}{\tau}})$$

This result is valid from a car, that is not moving at t=0, and that drives on a float road with a constant throttle. We make the assumption that as soon as we reach an upward slope, if the throttle is constant, the vehicle will lose speed because the added “gravity” force will slow down the vehicle. At least those are our expectations – we should verify them with RAMN.

Verification with RAMN

We can make measurements and try out different ECU algorithms with the following setup.

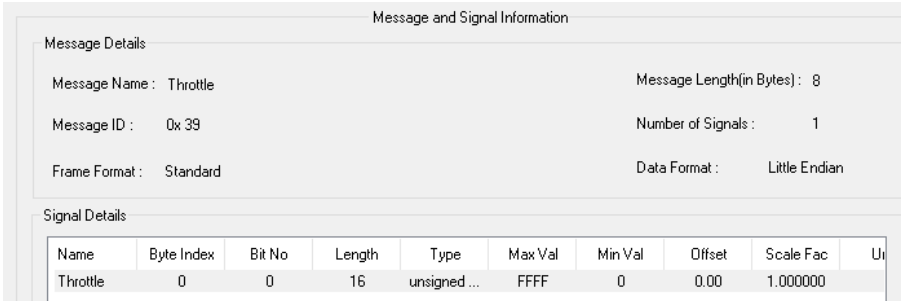
On the first PC, we run CARLA in closed-loop simulation with RAMN. We connect RAMN to a second PC, using an external CAN adapter. On that second PC, we run BUSMASTER, an open-source tool for CAN bus analysis (https://rbei-etas.github.io/busmaster/)



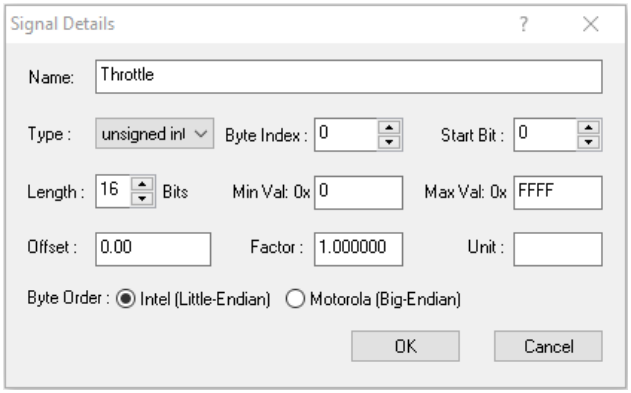
To make CAN bus measurements, we drive around the virtual city and look for a suitable place to experiment. We select the following setup: a flat road, followed by an upward slope, where we expect the vehicle to lose speed because of gravity.

We program the powertrain ECU to apply a constant throttle.

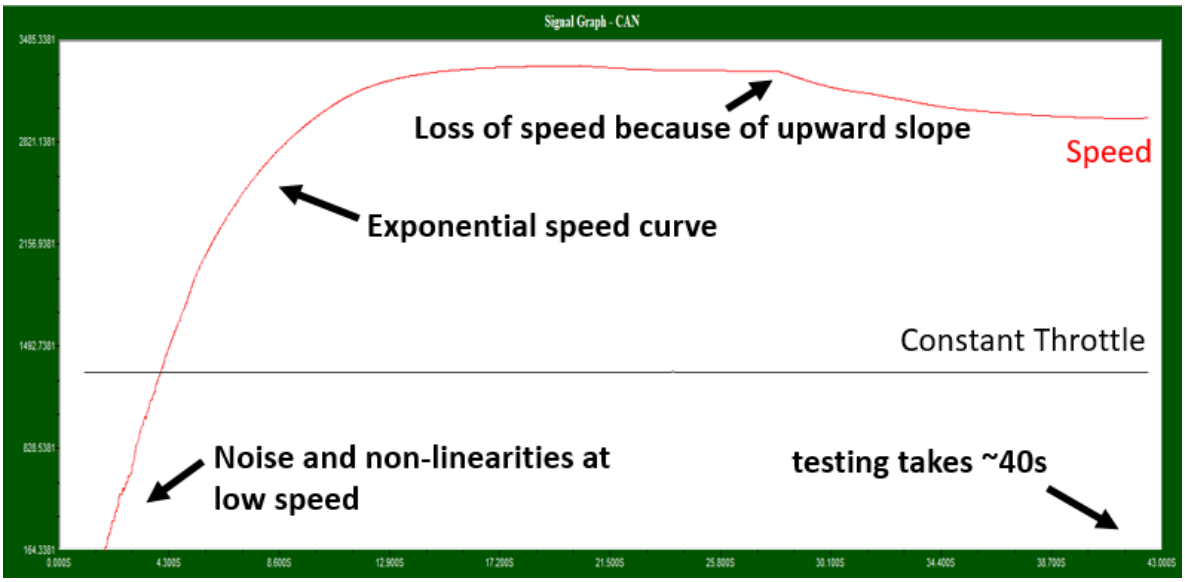
In BUSMASTER, we specify the format of the CAN messages we want to receive: **throttle** (ID 0x39) and **speed** (0x43).



We also need to specify the format of the payload: 16-bit unsigned integer, Little-Endian.



Once we have defined the format of CAN messages, we can plot the evolution of speed and throttle over time.



We make the following remarks:

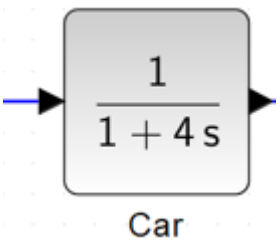
- As expected, the speed follows an exponential curve.
- There are some noise and non-linearities at low speed, but overall, the behavior of the car confirms our expectations from the system model (exponential curve – first order system).
- We are losing speed as soon as we reach the slope, because of gravity.
- Testing takes around 40s.

Since testing takes 40s, we do not want to spend that much time waiting for results every time we make a small change in our control algorithm. Therefore, we first use Scilab (https://www.scilab.org/) to model the behavior of the car.

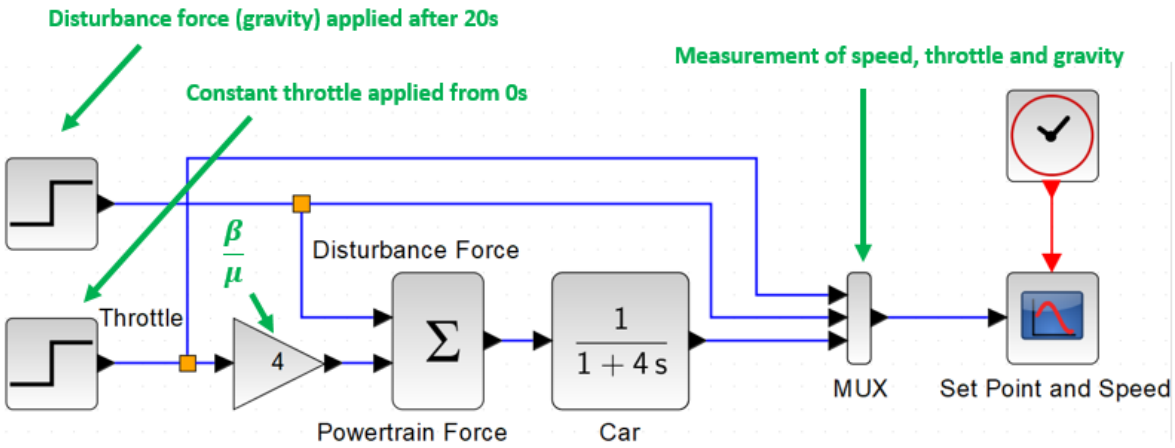
Simulations with Scilab

From our measurements, we notice that it takes around 4s for the car to reach 63.2% of its maximum speed, we therefore evaluate the time constant of the system as $\tau = 4$ seconds.

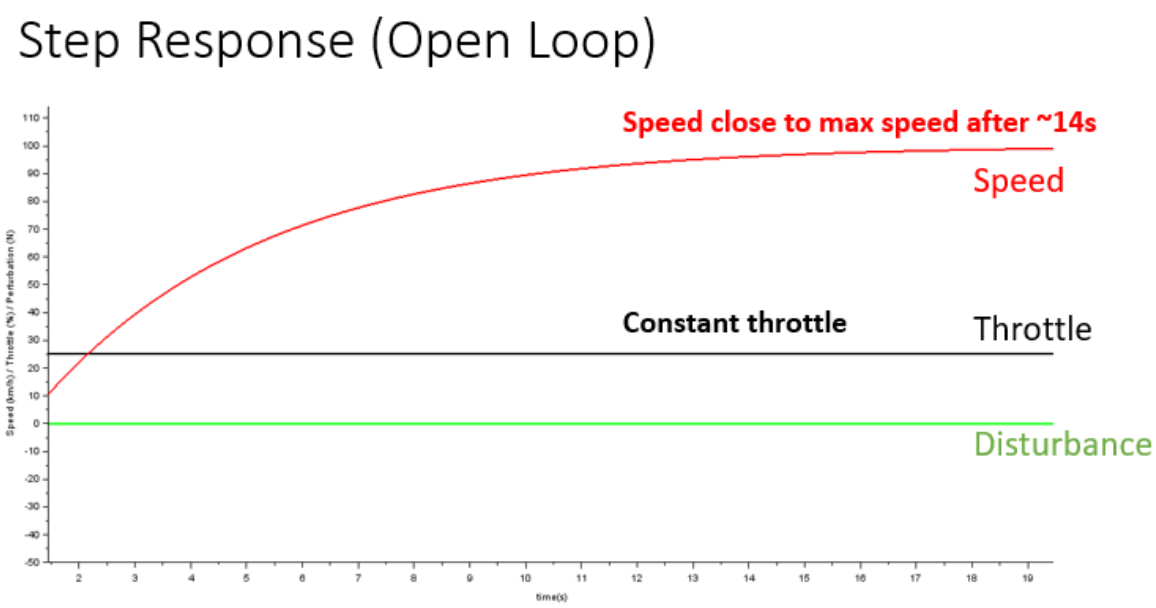
We can therefore model our car in Scilab with the following block (a first-order system with time constant of 4 seconds).

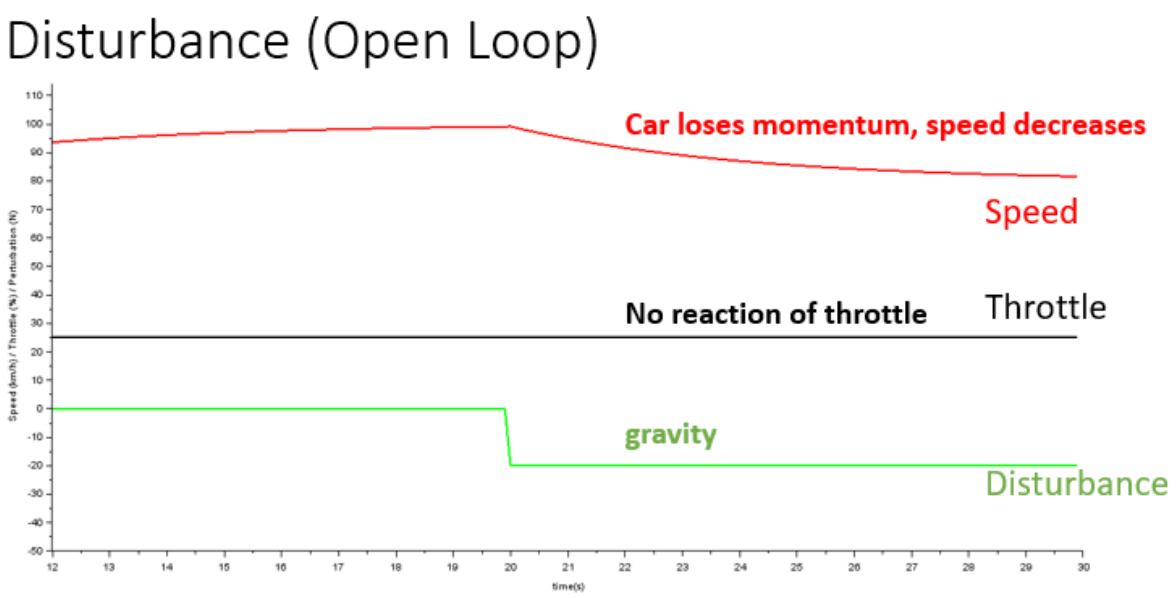


We can use Scilab to simulate the conditions of our testing:



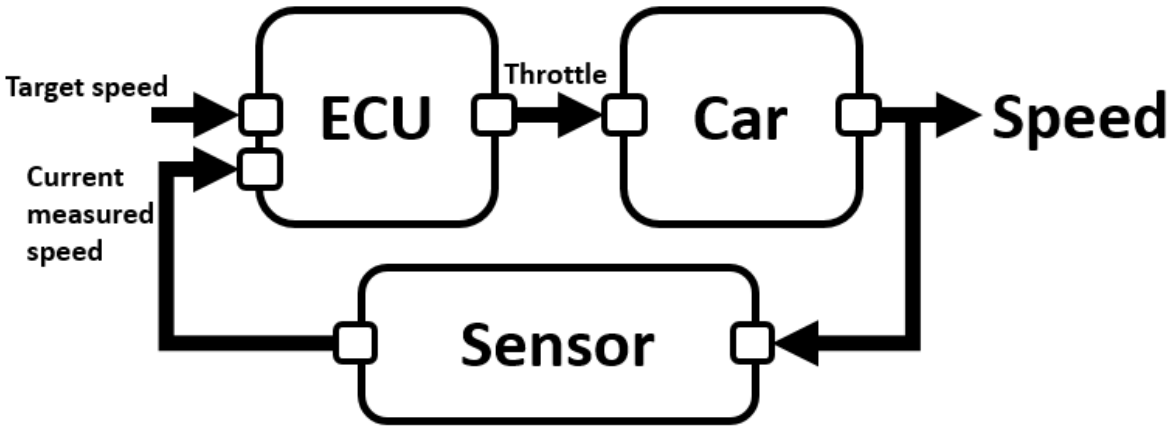
Where we can verify that the simulation shows the same behavior as from the measurements.





Now that we can simulate our car in Scilab, we can experiment with various control strategies, simulate them, then tune the parameters to optimize them. Once we have decided the parameters of the control algorithm, we can implement it in the powertrain ECU, and try it in CARLA.

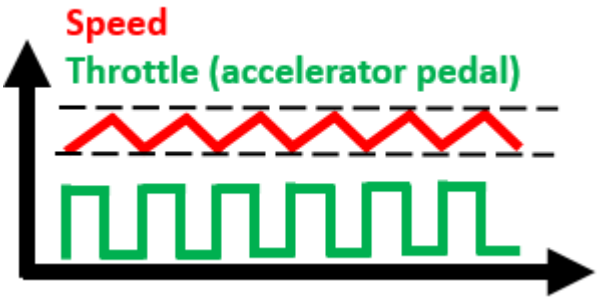
To ensure that we do not lose speed when there is gravity or additional friction (e.g., wind or different road), we need to use feedback from the speed sensor – which data is available to the powertrain ECU from the CAN bus.



First approach: Bang-Bang Control

https://en.wikipedia.org/wiki/Bang%E2%80%93bang_control

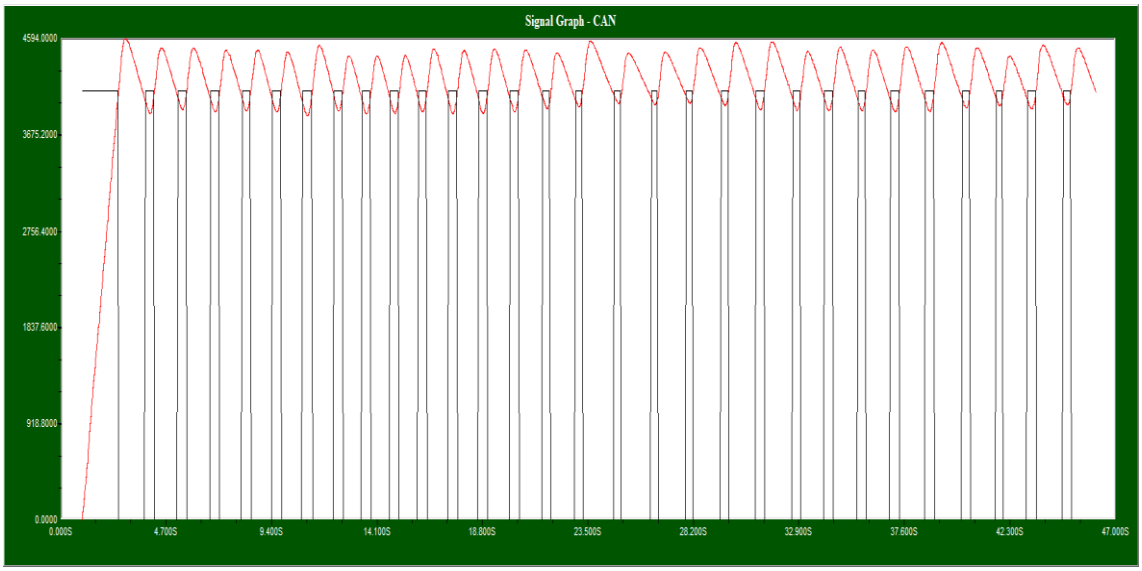
Bang Bang Control is a simple approach: If the car is above the target speed, the ECU releases the throttle. If the car is below the target speed, the ECU applies the throttle again.



It is also is easy to implement in C.

```
#ifdef BANGBANG
        if (throttle != 0)
        {
            if (current_speed >= target_speed +5) throttle = 0;
            else throttle = 0xFFF;
        }
        else
        {
            if (current_speed < target_speed - 5) throttle = 0xFFF;
            else throttle = 0;
        }
#endif
```

And we can verify that we do not lose speed anymore when we reach the slope. However, it can be observed that the car speed is oscillating, which is far from ideal for both the engine and for the passengers.



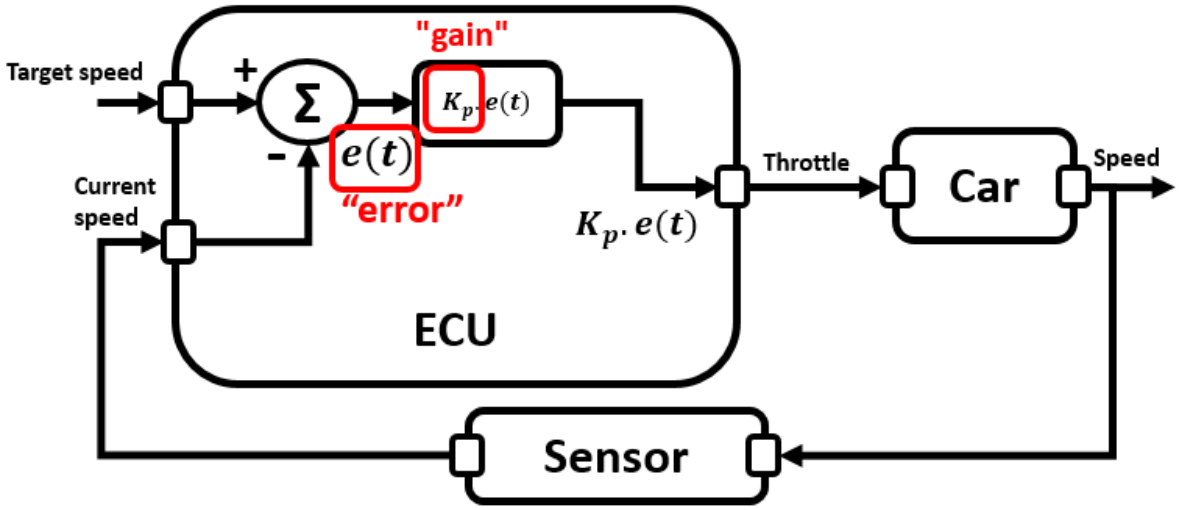
Those oscillations are particularly visible when watching from the driver's point of view.

While bang bang approach may be a valid for some applications (e.g., heating) it is clearly not a good approach for Cruise Control.

Second approach: Proportional Control

https://en.wikipedia.org/wiki/Proportional_control

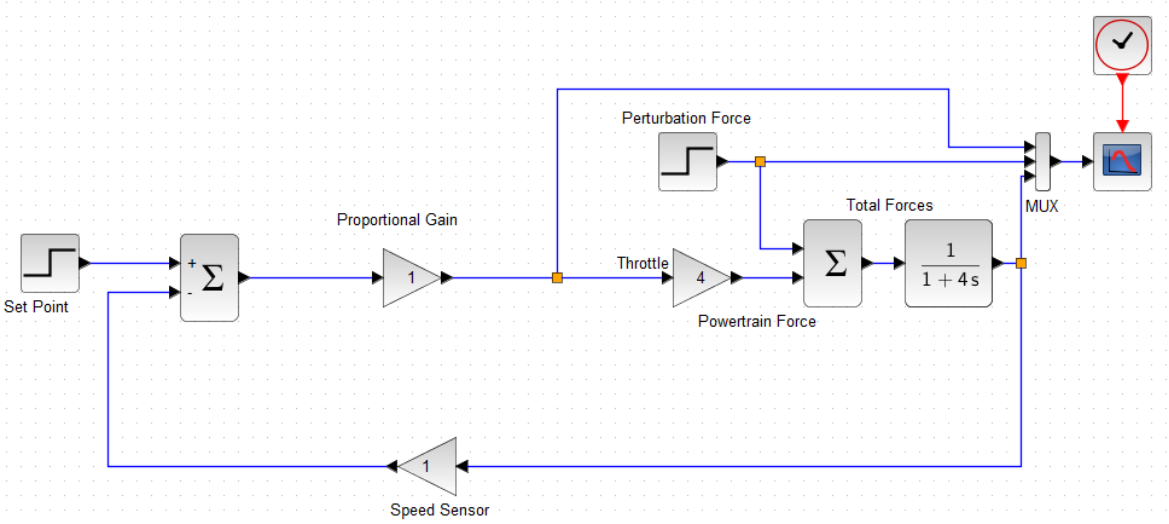
Proportional Control is the approach of controlling the throttle with a value proportional to the difference of the target speed and the current speed ("error"). The factor by which we multiply the "error" to compute the throttle is called "gain".



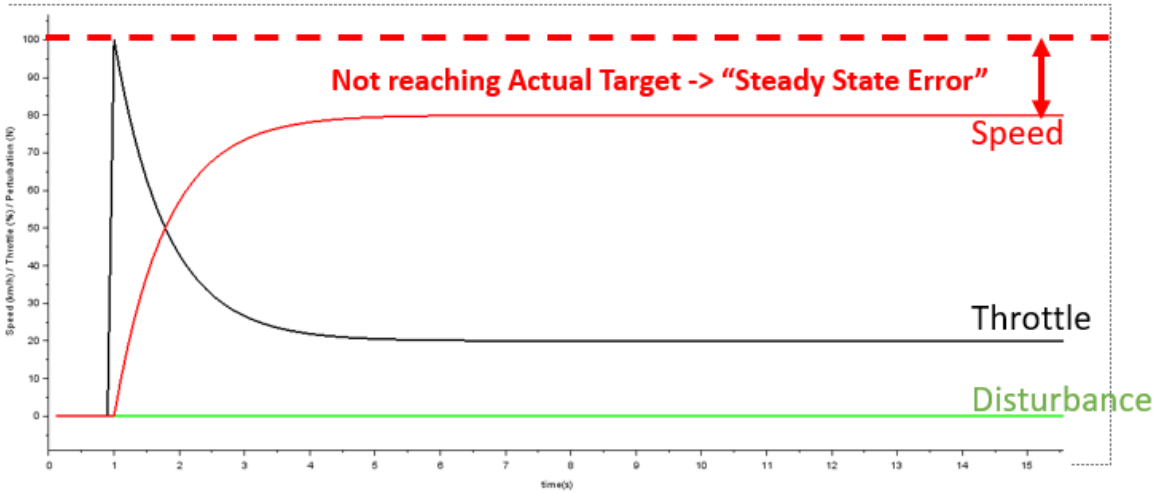
It is easy to implement in C.

```
#ifdef P_CONTROLLER
        const uint16_t Kp = 1;
        throttle = (Kp * (target_speed - current_speed);
#endif
```

And it can also be simulated with Scilab, using the following diagram.



And the results of the simulation are as shown below.

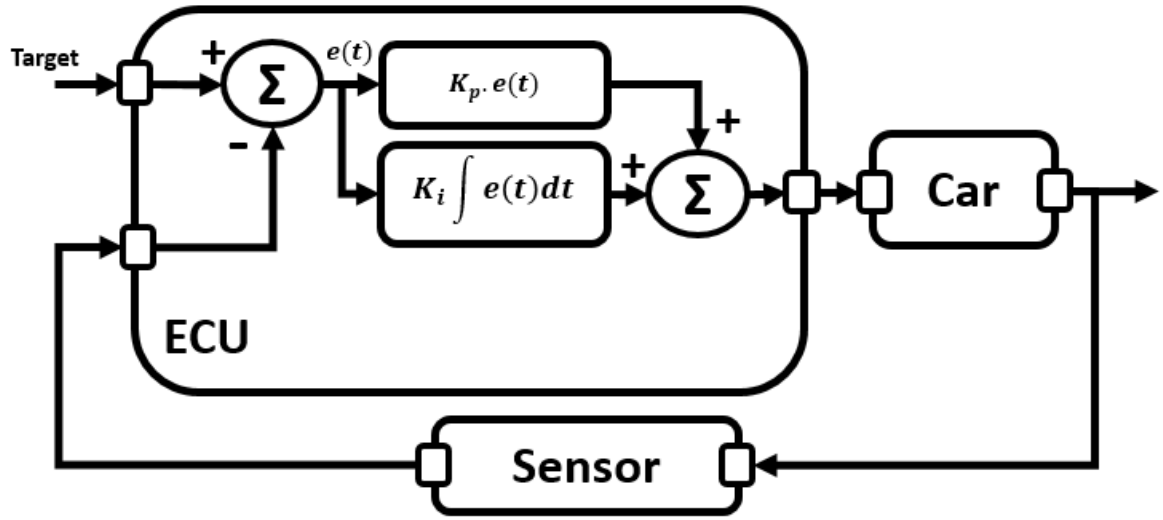


We can see that with proportional control only, we fail to reach the target speed – there is what is called a "steady state error" ("offset error"). There is no need to test on CARLA, because we already know from the simulations that this approach will not work, saving us a lot of time.

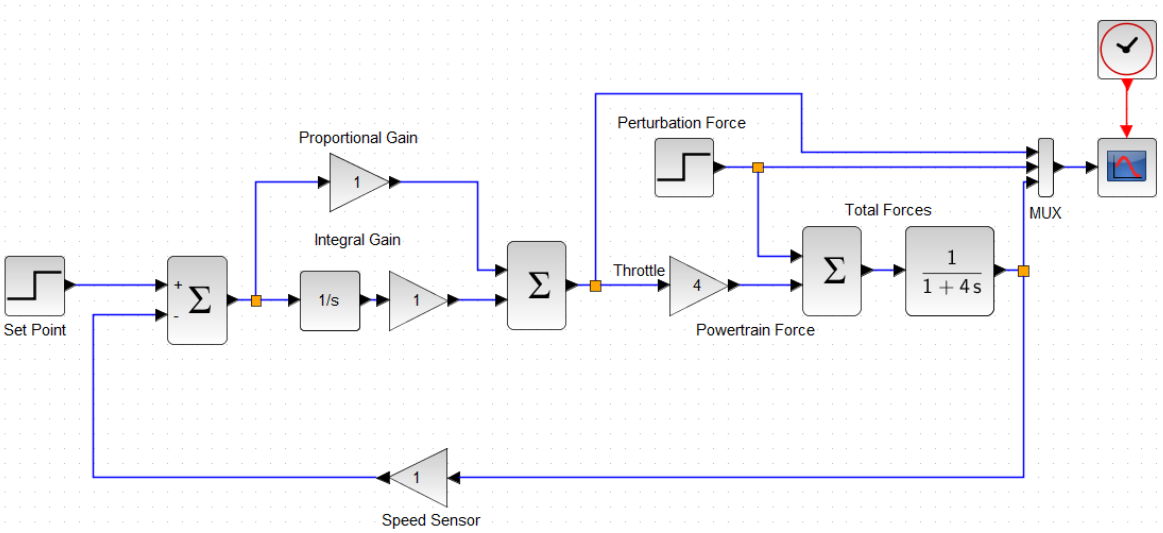
Third Approach: Proportional Integral Control

Third approach we can try is Proportional Integral Control (https://en.wikipedia.org/wiki/PID_controller#PI_controller).

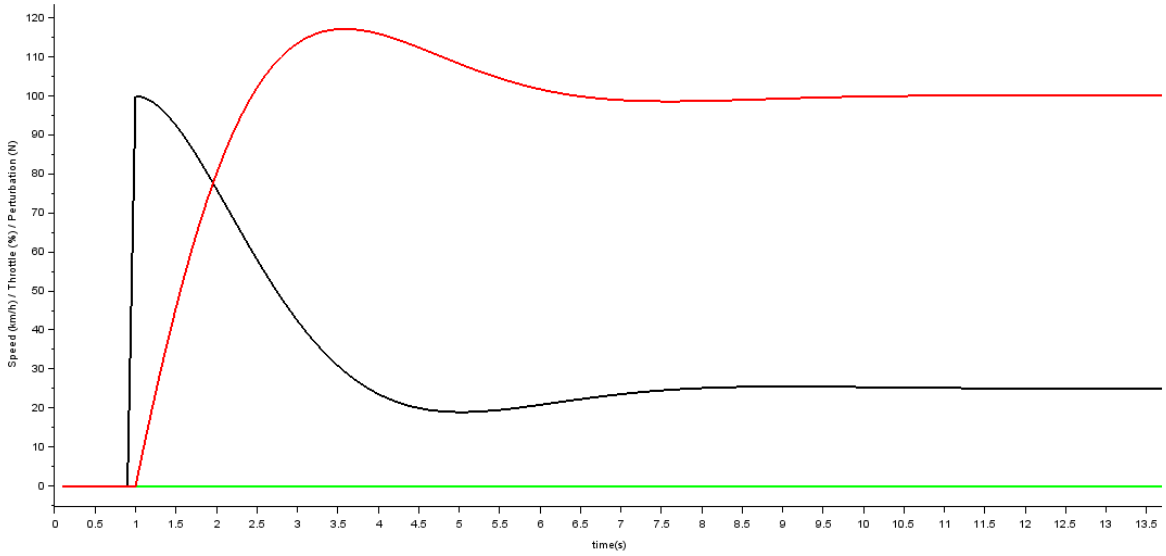
We control the throttle based on the sum of a value proportional to the error and a value proportional to the integration of the error over time.



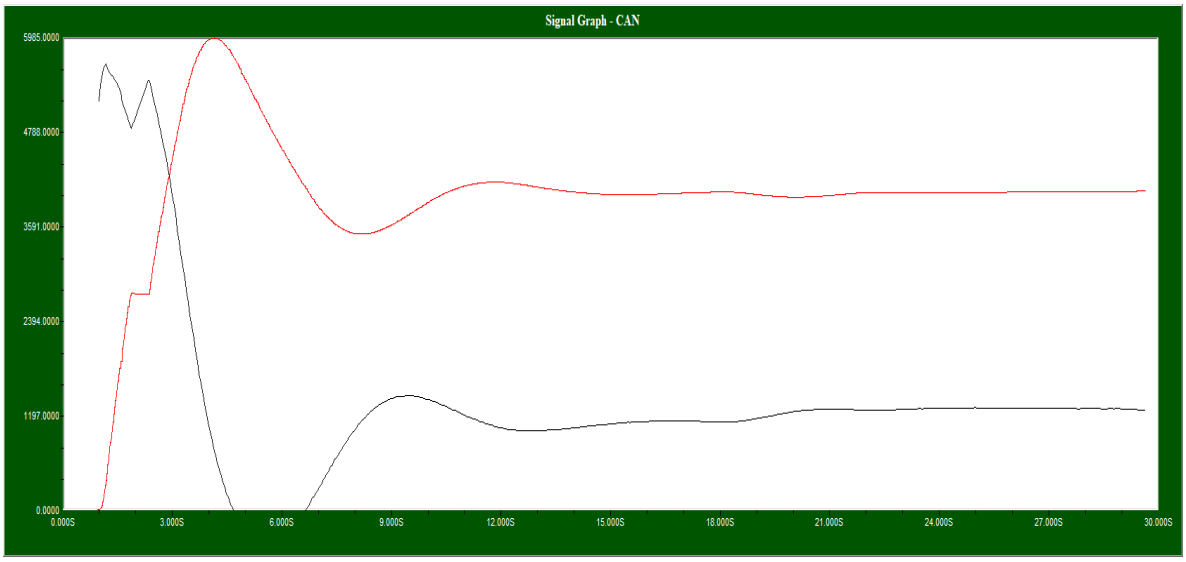
This is also something we can simulate on Scilab. Integration in time domain is equivalent to a division by s in the Laplace transform, hence the "1/s" block.



The simulation gives us the following results.



We can speed that this time we are overshooting the target speed by 20% - but we do eventually get there and maintain the speed even when there is a slope. When testing on CARLA, we get the following measurements.



We can see some deviations from the simulation:

- At the beginning, there is a short interval during which the speed is constant – which we could attribute to a gear change in the car. We can see that the throttle increases during the gear change because of the “integral” control.
- The throttle goes all the way to 0 after overshooting the maximum speed.

While we did not take into account those irregularities, we can see that overall, the speed seems to be similar to the results of our simulations. The ECU is now able to maintain the speed when reaching the slope – but we overshoot the target speed by 20%, which is not acceptable.

We could tune the parameters to fix the overshooting issue, or we could try the fourth approach.

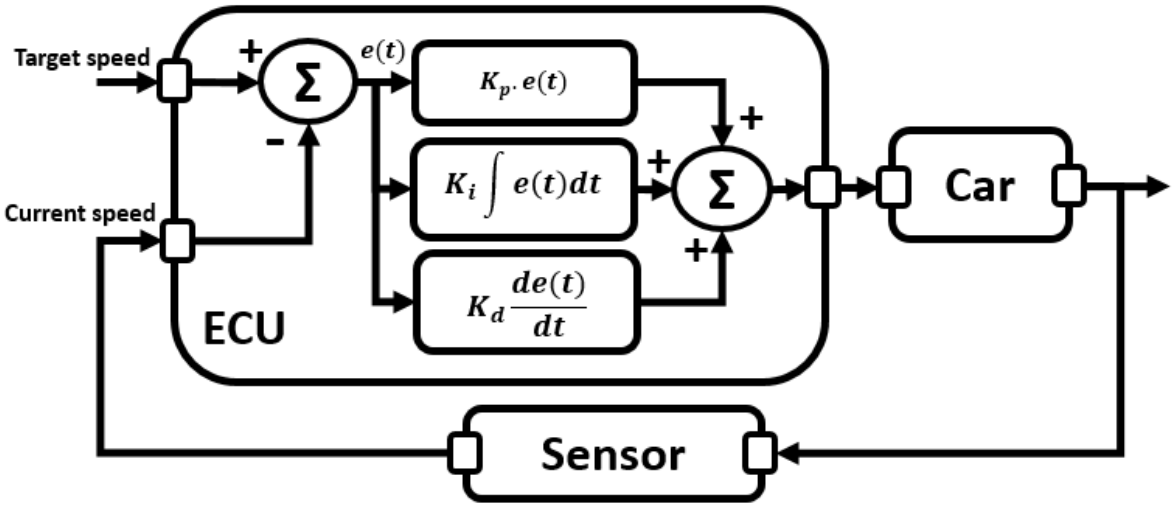
Fourth Approach: Proportional Integral Derivative Control

PID (Proportional Integral Derivate) control is one of the most famous control algorithms.

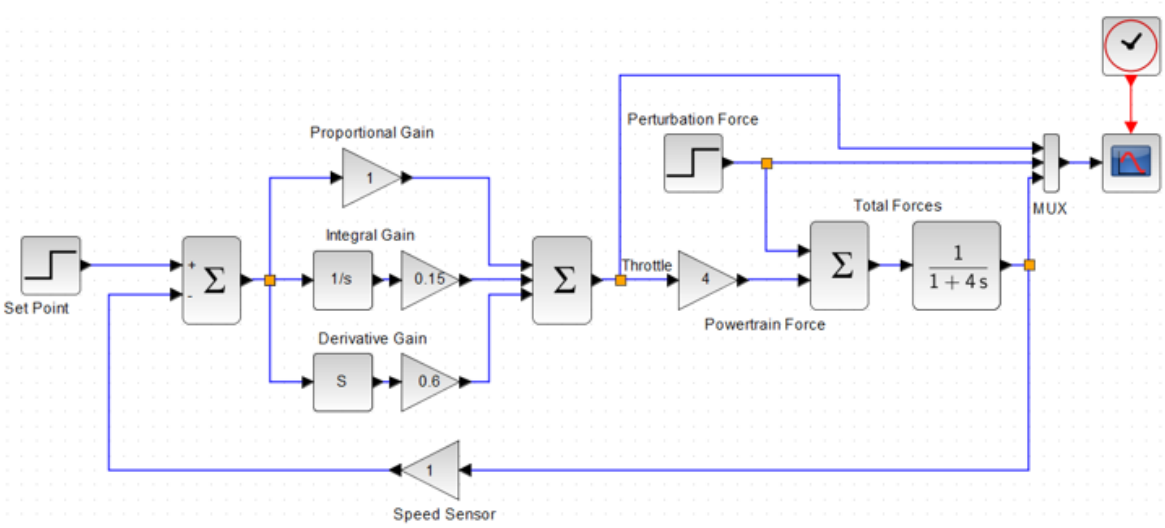
http://en.wikipedia.org/wiki/PID_controller

https://aeroastro.mit.edu/videos/controlling-self-driving-cars

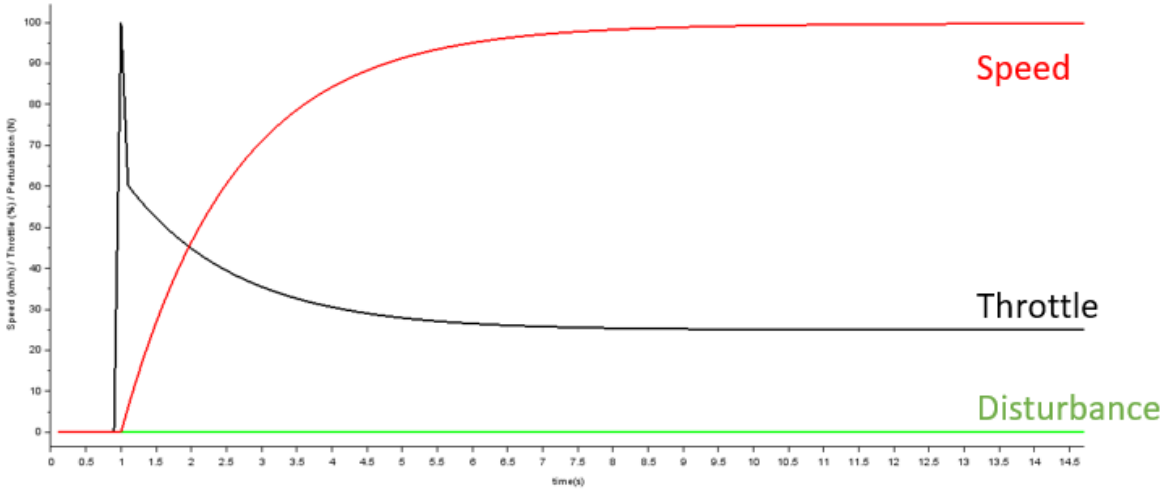
This approach is the same as the proportional Integral approach, but this time we add a "derivative" term as well.



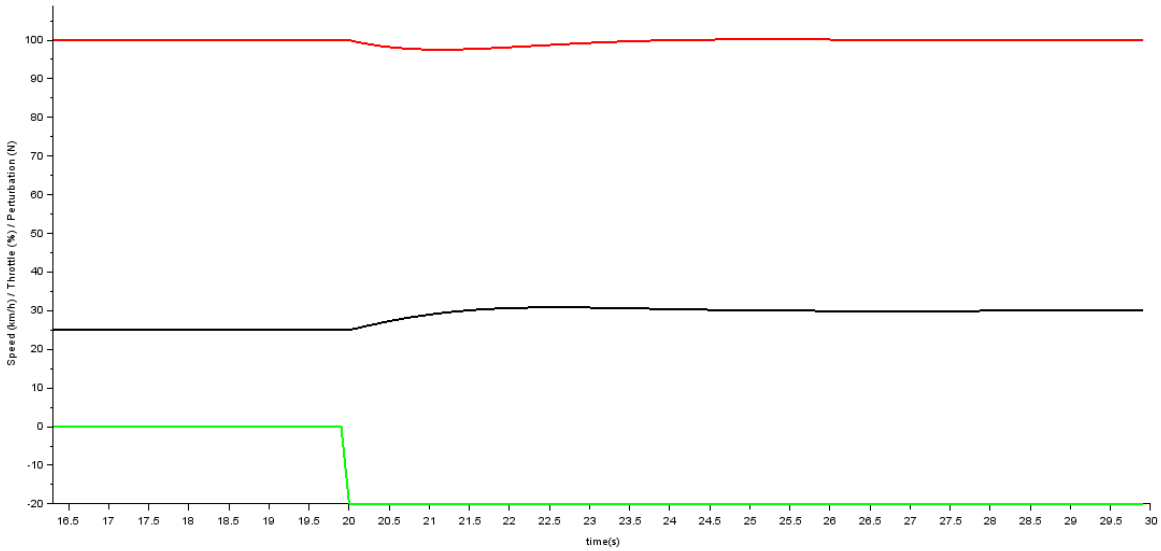
This is also easy to simulate with Scilab. A derivation in the time domain is equivalent to a multiplication by s in the Laplace Transform, hence the block that multiplies by s.



With this simulation, we obtain the following behavior.



We observe no more overshooting of the target speed.



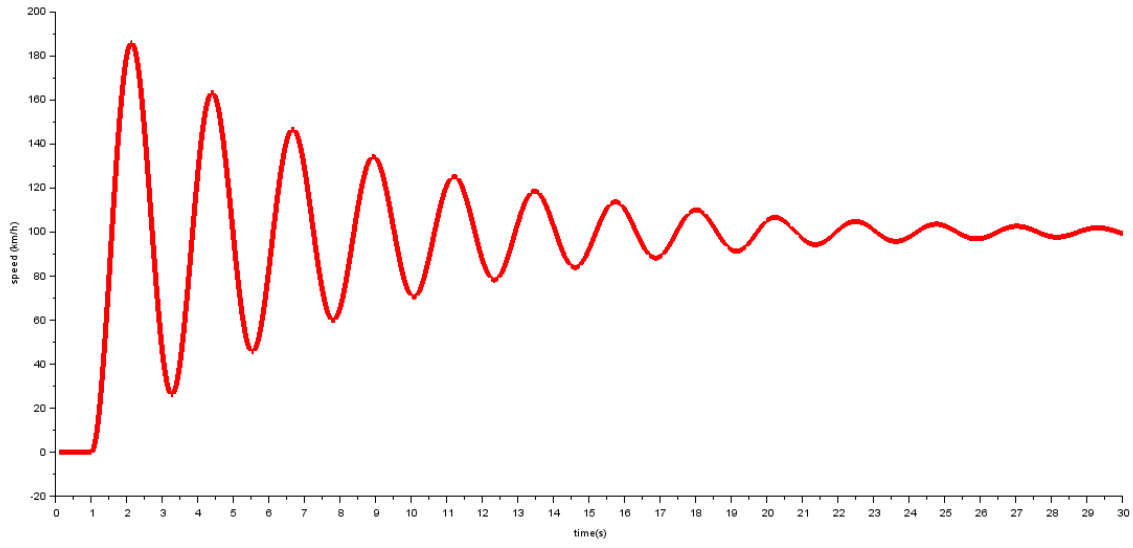
And we see that the ECU reacts quickly to the slope – we do not fail to maintain the target speed anymore.

Those waveforms are the result of a simulation for the parameters

$$K_p = 1 \quad K_i = 0.15 \quad K_d = 0.6$$

However, if we select the wrong parameters, we may get something different, unstable, or dangerous. For example, below are the results of the simulation with the same algorithm, but just one different parameter:

$$K_p = 1 \quad K_i = 10 \quad K_d = 0.6$$



This means that if we select to use PID control, our ECU will need to have active countermeasures to prevent such scenario from happening.

To implement a PID controller in an ECU is not straightforward. We need to create program it so that it can compute the throttle based on the following formula:

$$\boldsymbol{Throttle(t) = K_p.e(t) + K_d\frac{de(t)}{dt} + K_i\int e(t)dt}$$

"Integral" and "Derivative" operations are only possible for analog signals (continuous time). Our ECU is processing the "speed sensor" signal every 10ms, it is a "discrete controller". We can approximate the PID from our simulations with many solutions. We will just detail two here:

First solution: approximate integral and derivative terms.

https://www.wescottdesign.com/articles/pid/pidWithoutAPhd.pdf

We can implement the PID algorithm with the following approximations:

$$\boldsymbol{K_d\frac{de(t)}{dt} \approx \frac{(e[n] - e[n-1])}{T_{sampling}}}$$

$$\boldsymbol{K_i\int e(t)dt \approx T_{sampling}\sum_{i=0}^{n} e[i]}$$

Therefore, every 10ms ($\boldsymbol{T_{sampling}}$), the ECU will compute and output the throttle based on the following formula:

$$\boldsymbol{Throttle[n] = K_p.e[n] + K_i.T_{sampling}\sum_{i=0}^{n} e[i] + \frac{K_d}{T_{sampling}}(e[n] - e[n-1])}$$

Therefore, the throttle can be computed using the current error, the previous error, and the running sum of all errors. The ECU can implement this algorithm using the following code.

Initialization:

```
int32_t error_sum = 0;
int32_t previous_error = 0;
```

Control Loop:

```
#ifdef PID_CONTROLLER
        const float Kp = 1;
        const float Ki = 0.01f;
        const float Kd = 1;
        float output = 0;

        int32_t error = (int32_t)(target_speed - current_speed);
        error_sum += error;
        output = Kp*error + Ki*error_sum + Kd*(error - previous_error);
        if (output > 0xFFF) output = 0xFFF;
        if (output < 0) output = 0;
        throttle = (uint16_t)output;
        previous_error = error;
#endif
```

Second solution: use Z-transform

https://en.wikipedia.org/wiki/Z-transform

A PID controller's Z transform can be approximated using Tustin's transformation

https://ctms.engin.umich.edu/CTMS/index.php?aux=Extras_PIDbilin

$$Throttle(s) = K_p.e(s) + \frac{K_i}{s}e(s) + K_d s.e(s)$$

$$s = \frac{2}{T}\frac{z-1}{z+1}$$

$$Throttle(z) = \frac{\left(K_p + \frac{K_i T}{2} + \frac{2K_d}{T}\right) + \left(K_i T - \frac{4K_d}{T}\right)z^{-1} + (-K_p + \frac{K_i T}{2} + \frac{2K_d}{T})z^{-2}}{1 - z^{-2}}e(z)$$

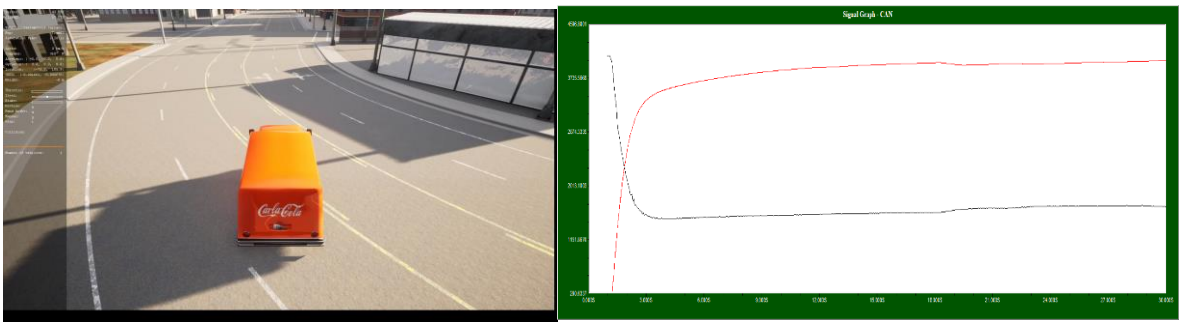
The inverse transform (recursive form) of this transfer function is

$$throttle[n] = throttle[n-2] + \left(K_p + \frac{K_i T}{2} + \frac{2K_d}{T}\right)e[n] + \left(K_i T - \frac{4K_d}{T}\right)e[n-1] + \left(-K_p + \frac{K_i T}{2} + \frac{2K_d}{T}\right)e[n-2]$$

Which is another way to implement the control loop – a combination of the output from two iterations ago, current error, previous error, and the error from two iterations ago.

While those two implementations may look different, they achieve the same control algorithm.

We measure the following waveforms on the CAN bus



Confirming our ECU is now able to make the car maintain its speed on the slope, by applying a dynamic control to the throttle.

What if it was a real ECU?

System level

This algorithm works in the scenario we are testing here (flat road followed by upward slope). But it has many flaws:

- It applies the throttle from 0% to 100% without “slow-start”; it accelerates too fast.
- It does not adapt the speed to the vehicle in front
- It cannot brake on a downward slope
- It does not feature an “integral windup” protection (https://en.wikipedia.org/wiki/Integral_windup)
- etc.

A real ECU would need to go through many more steps at concept and system design phases. For example, engineers should prepare:

- HAZOP Hazard and Operability Analysis
- HARA Hazard Analysis and Risk Assessment
- FMEA Failure Mode and Effects Analysis
- STPA System Theoretic Process Analysis
- etc.

Our model is also too simple – we should be using a more advanced one taking into account the transmission, the dynamics of each wheel, etc.

https://www.mathworks.com/help/physmod/sdl/ug/vehicle-with-four-wheel-drive.html

https://www.mathworks.com/help/ident/ug/modeling-a-vehicle-dynamics-system.html

Software level

The implementation is terrible – there is no typecasting or overflow checks. Real ECU software would need to follow MISRA-C guidelines, as well as other guidelines, for example from ISO26262-6

Additionally, the software should make use of safety features found on the microcontroller. Safety features of the STM32L4 and STM32L5 are summed up in this document: https://www.st.com/resource/en/product_presentation/stm32_stm8_functional-safety-packages.pdf

Here, for example, the software implementation should ensure that important variables are stored in sections of memory where redundancy features (ECC/parity) are available. Where possible, write-protection should be activated. Concretely, all variables should feature an __attribute__((section("SRAM2")) attribute to ensure they will be placed in SRAM2, where such features are available, and not SRAM1, where they are not.

STM32L4 and STM32L5 are not automotive grade controllers, they should not be used for automotive applications.

Conclusion

We experimented with Cruise Control to demonstrate how RAMN can be used to study and research automotive systems. The resulting algorithm is unsafe, both at system, software, and hardware levels – **this is for education and not for automotive use**.