DHBW
Duale Hochschule
Baden-Württemberg

# Documentation Parkassist

## Graphische Programmierung und Simulation

at the Cooperative State University Baden-Württemberg Stuttgart

07.04.2020

| | |
|---|---|
| **Time of Project** | 24.03.2020 - 07.04.2020 |
| **Student ID, Course** | 8540946; 6430174, STG-TINF17-ITA |
| **Lecturer** | Dr. Kai Pinnow |

# Contents

# Acronyms

| | |
|---|---|
| **NaN** | Not a Number |

# List of Figures

# List of Tables

# Listings

# 1 Introduction

In this project Matlab version R2019b and ASCET version 7.4 have been used. The Matlab and Simulink code is in the Matlab subfolder and the ASCET project is in the ASCET subfolder. The report is structured by devoting one chapter to each requirement.

In ASCET mainly graphical programming using block diagrams is used due to requirement D7. Even though D8 does not demand graphical programming, for consistency reasons D8 is being graphically modelled as well. Exceptions are the unit tests. The test are programmed in ESDL because some test have a lot of test cases which would lead to a huge block diagram that would likely be unclear.

Simulink: ode8 most precise runge-kutta solver Fixedstep

# 2 D1: Time estimate based on three point estimation

For the three point estimation we use the formuals from the lecture. The expected effort is computed using a weighted mean formula:

$$< T >= \frac{optimistic + 4 * likely + pessimistic}{6} \tag{2.1}$$

To compute the standard deviation we use the following equation:

$$\sigma = \frac{pessimistic - optimistic}{6} \tag{2.2}$$

In the following the time unit is hours.
Table 2.1 contains our effort estimates for the tasks at hand, the expected value, computed standard deviation and the actually needed time.

Table 2.1: Three point estimation of effort for meeting requirements

| **Requirement** | **Optimistic** | **Likely** | **Pessimistic** | **<T>** | $\sigma^2$ | **Actual** |
|---|---|---|---|---|---|---|
| D1 | 0.75 | 1 | 1.5 | 1.0417 | 0.0156 | 0.75 |
| D2 | 0.5 | 1 | 1.5 | 1 | 0.02778 | 1.25 |
| D3 | 1.5 | 2 | 3 | 2.08333 | 0.0625 | 4 |
| D4 | 2 | 3 | 4 | 3 | 0.11111 | 2 |
| D5 | 2 | 3 | 4 | 3 | 0.11111 | 3 |
| D6 | 2 | 2.5 | 3 | 2.5 | 0.02778 | 5 |
| D7 | 3 | 4 | 5 | 4 | 0.11111 | 3 |
| D8 | 1.5 | 2.5 | 3 | 0.0625 | 0.25 | 1 |
| D9 | 4 | 4.5 | 5 | 4.5 | 0.02778 | 2 |
| D10 | 3 | 3.5 | 4 | 3.5 | 0.02778 | 2.5 |
| D11 | 1 | 1.5 | 2 | 1.5 | 0.02778 | 2 |
| D12 | 6 | 7 | 8 | 7 | 0.11111 | 7 |
| D13 | 1 | 1.5 | 2 | 1.5 | 0.02778 | 1 |
| D14 | 1.5 | 2 | 2.5 | 2 | 0.02778 | 2 |
| Total | 29,75 | 39 | 48,5 | 39,04167 | 0.77951 | 36.5 |

The estimation of the 95 % estimate of the total duration is computed as:

$$\sum < T > +2 * \sqrt{\sum \sigma(T)^2} \tag{2.3}$$

This results in a 95 % estimate of 40.8 hours.
The total actually needed time is slightly less than the expected value of 39 hours. Some tasks, like D3 (human velocity profile extraction) and D6 (pulse signal in Simulink) took significantly longer than expected.

-other tasks less time

Nontheless the acutally needed time is well within 95 % confidence interval.

# 3 D2: Feasibility study

The aim of the feasibility study is to analyse whether it is possible to realise smooth braking with the introduced model and parameters based on the given formulas

$$\frac{\partial v}{\partial t} = -c - b * p \tag{3.1}$$

$$\frac{\partial x}{\partial t} = v \tag{3.2}$$

with $c = 1.5\ m/s^2$ and $b = 10\ m/s^2$. For that, in the next section a Simulink model is created, based on the equations above. After that, a test scenario to test the feasibility is described and in section 3.3 the results of the study are outlined.

## 3.1 Simulink model

Figure 3.1 shows the simulink model representing the differential equations above. The parameter $p$ is configured on execution of the simulation in Matlab. The initial velocity $v_0$ is set to 10 km/h and the initial car position $x_0$ is set to 0, corresponding to the scenario, that should be tested. Since the output of the first integrator is in m/s, $v_0$ needs to be divided by 3.6 to convert $v_0$ from km/h to m/s. The minimal velocity of 0.29 km/h from requirement R5 has been included in the model. Otherwise, the car would not come to a full stop because the velocity would become negative eventually. R5 is realised with a switch, that sets velocity to zero if the computed velocity is below 0.29 km/h, which corresponds to approximately 0.0806 m/s. The velocity is integrated to compute the location, following the second differential equation. The output parameters acceleration (a), velocity (v) and travelled distance (s) are output using the out block. Input parameters are input as constants. In this preliminary test a constant p is used, which is not time-dependent.

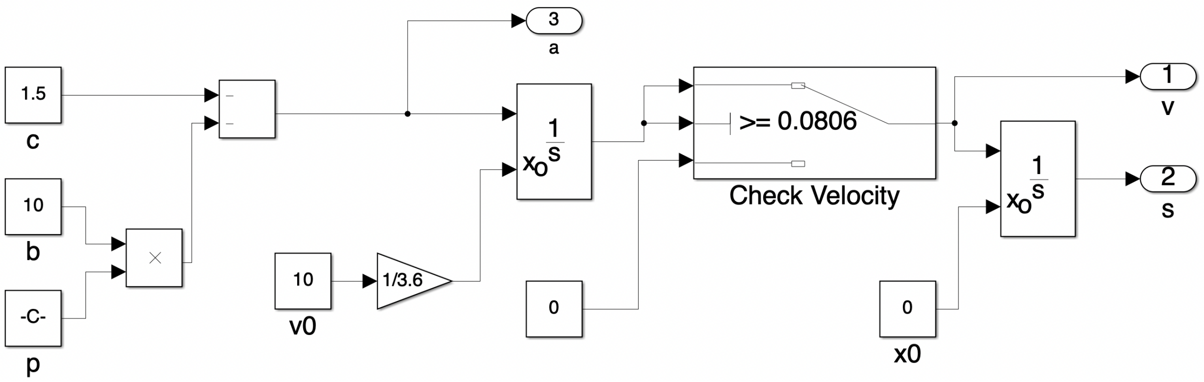


Figure 3.1: Simulink model of the differential equations

## 3.2 Test scenario

The goal of this scenario is to check whether it is possible, with the given model, to realize a smooth stop with a stopping location < 2 m. The idea is to demonstrate, that a full stop before 2 m can be realised with a reasonable brake pedal pressure and a reasonable deceleration. If, for example a stop before 2 m could only be realised with 100 % brake pressure, that would mean that a smooth stop can not be realised.
It is not necessary to implement a human-like smooth stop, but to show that this is possible. Therefore the proposed test scenario is to run the model and find a reasonable small brake pressure where the car comes to a full stop with a location < 2 m. This would show the feasibility of the task. Creating a human-like brake profile where the brake parameter p can be adjusted over time is not an objective of this feasibility study, this will be covered at a later stage.
The execution of this scenario is automated using a Matlab script, which parametrizes the model and displays the outputs. Listing 3.1 shows the script that executes the simulation and stores the results. The code to create the result plots has been excluded. In lines one to eight the simulation is parametrized. A stop time of 2 seconds is chosen because the car will come to a stop before.

todo solver begründen

The constant brake pressure is set to 5 %. This value results in a stop before the car traveled 2 meters of distance and is also reasonably small.

```
1 %% Parametrize Model
2 set_param('D2','StopTime','2');
3 %set solver
4 set_param('D2','Solver',['ode',sprintf('%d',8)]);
5 %set simulation step size
6 set_param('D2','FixedStep',sprintf('%f',0.01));
7 %set brake pressure parameter
```

```
set_param('D2/p','value',sprintf('%f',0.05));


%% Simulate and get output
res = sim('D2','SaveOutput','on','SaveState','on');
t = res.tout;
v = res.yout{1}.Values.Data;
s = res.yout{2}.Values.Data;
a = res.yout{3}.Values.Data;
%convert velocity to km/h
v = v*3.6;
```

Listing 3.1: Execution of the feasibility test

## 3.3 Results

The following figure 3.2 shows the simulation results with a constant brake pressure of 5%. This results in a constant deceleration of -2 m/$s^2$. Setting the velocity to zero, when reaching the minimal velocity, has no impact on the computed acceleration. This is the reason, why the acceleration still remains the same, even though the car is not moving anymore.

The velocity linearly decreases from $v_0$ = 10 km/h to zero (except the decrease from minimal velocity to zero, which was expected). The third subplot shows the covered distance of the car, which is clearly under the 2 m mark. Therefore, with a constant brake pressure of 5 % the car can be stopped before reaching 2 m travelled distance with an acceptable deceleration of 2 m/$s^2$. In the cruise control project we have discussed that an acceleration of less than 4 m/$s^2$ is not uncomfortable.

These simulation results demonstrate that it should be possible, with variation of the brake pressure over time, to realise a smooth stop with a human-like brake profile.

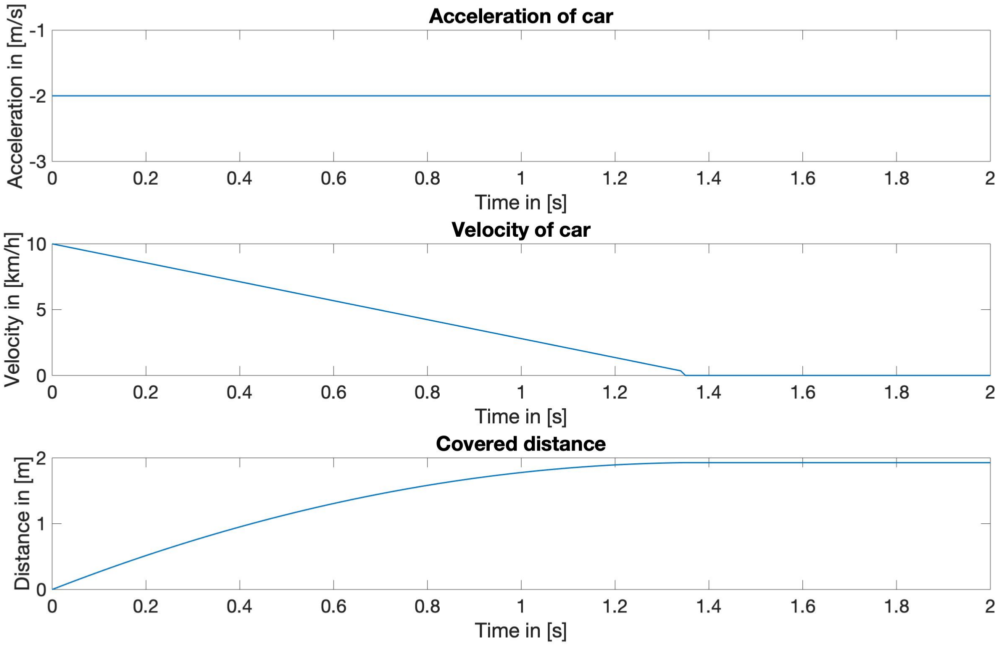


Figure 3.2: Simulation results of feasibility study

# 4 D3: Analysis of human velocity profile

In this section the provided human velocity profile is analysed in order to find a pattern that can be abstracted as a human-like brake profile. This pattern will be used in the following sections and will be adapted to the braking function of the ParkAssist. For the analysis Matlab functions are used, as demanded by requirement D3.

## 4.1 Import the measurement data

The human velocity profile contains a time dimension and the four wheel velocities. The following listing shows how the measurement data is imported in Matlab. A Matlab library function is used for the import of the textfile. This results in a standard numeric matrix.

```
%import velocity data
velocity_data = importdata('MeasuredVelocities.txt');
```

Listing 4.1: Import measurement data in Matlab

## 4.2 Data preparation

The aim of the analysis is to make conclusions that lead to a time dependant brake profile. Since the brake pedal has a linear influence on the acceleration (see model formula) the goal is to extract an acceleration curve from the human velocity profile, from which to make deductions on the brake profile. For that, a cars acceleration and in consequence the cars velocity is needed. Therefore a mean car velocity has to be extracted from the given four wheel velocities.

Listing 4.2 shows the Matlab script that reprocesses the data. As a first step, the four wheel velocities are extracted from the measurement data and a mean velocity is calculated with the following equation (lines 2 and 3 in listing 4.2):

$$v_{car} = (v_{w1} + v_{w2} + v_{w3} + v_{w4})/4 \tag{4.1}$$

This mean velocity serves as an estimate for the car velocity. It is only an estimate, because friction losses, other physical influences and cornering are not considered. The mean velocity is converted to [m/s] for further calculations in line 4 in listing 4.2. Before calculating the acceleration, a moving average filter is applied on the mean velocity (line 8 in listing 4.2). Further details about the moving average filter are given in the next paragraph. The acceleration can be calculated by differentiating the mean velocity (line 11 in listing 4.2).

```
 1 %compute mean velocity of all 4 wheels
 2 velocity_per_wheel = velocity_data(:,2:5);
 3 mean_velocity = mean(velocity_per_wheel,2);
 4 mean_velocity = mean_velocity/3.6;      %convert velocity from km/h to m/s
 5 raw_mean_velocity = mean_velocity;      %for demonstration purposes
 6 
 7 %apply moving average filter to smoothen the data
 8 mean_velocity = movmean(mean_velocity, 200);
 9 
10 %differentiate velocity to get acceleration
11 acceleration = diff(mean_velocity);
12 raw_acceleration = diff(raw_mean_velocity);     %for demonstration purposes
```

Listing 4.2: Preprocessing measurement data

As can be seen in the upper subplot of figure 4.1 the calculated acceleration is noisy due to noise in the measured velocity. Because of the noise it would be harder to analyse the braking behaviour. Therefore we used a moving average filter to smoothen the data. A moving average filter is calculating a mean value of neighbouring elements within a sliding window of a predefined size. Applying a moving average filter results in a filtered graph that can bee seen in the lower subplot of figure 4.1. However, applying a moving average filter distorts the acceleration values because the exact value is replaced by the average of the neighbouring elements. This reduces the difference between the values of neighbouring elements. As a result the differentiation of the velocity results in an acceleration that is is in our case approximately 10 times smaller than the non-filtered results. Although the values of the acceleration are smaller the course of the acceleration is not changed. Therefore the filtered data is not used to extract exact acceleration values but rather to get an understanding of the acceleration curve during braking.

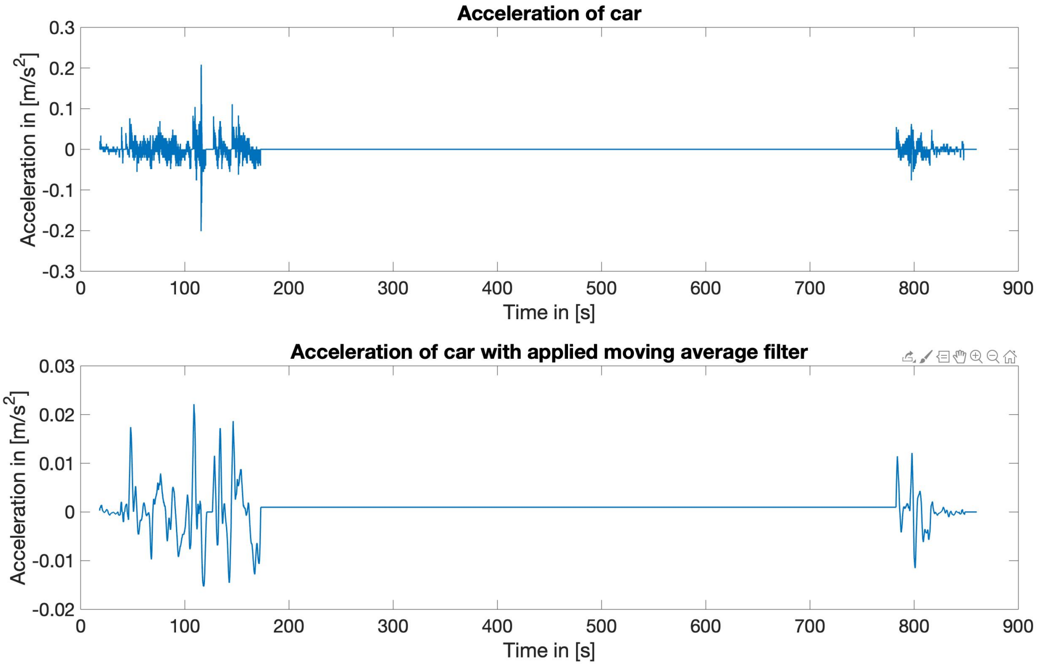


Figure 4.1: Impact of applying a moving average filter on the cars acceleration

Figure 4.2 shows the mean velocity of the car as well as the corresponding smoothened acceleration of the car over time.

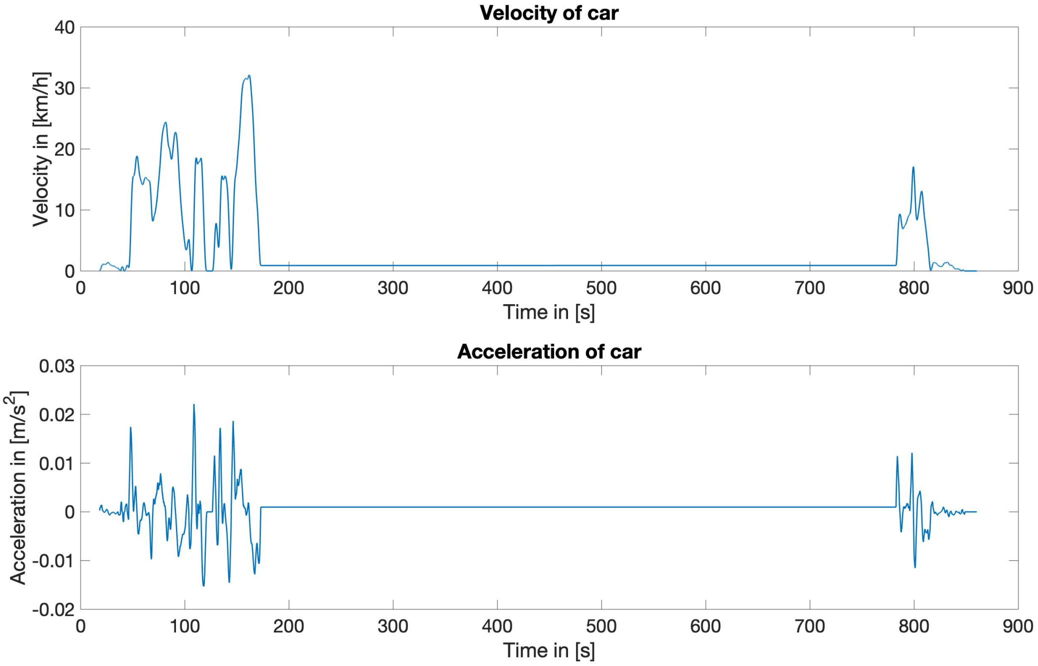


Figure 4.2: Extracted human velocity profile

## 4.3 Extracting negative acceleration

As the goal is to analyse the braking behaviour, only negative acceleration is relevant and is thus being extracted from the overall acceleration (line 5 in listing 4.3). Also, all velocities that are based on a positive acceleration are not considered anymore (line 6 in listing 4.3). We decided to set those values to Not a Number (NaN) because this way we are still able to plot the velocity and the acceleration over time without cut-outs. The result can be seen in figure 4.3.

```
1 %search for negative acceleration and set positive acceleration to NaN
2 %set all velocities that have a positive acceleration to NaN
3 neg_acceleration = acceleration;
4 decreasing_velocity = mean_velocity;
5 neg_acceleration(neg_acceleration>0) = nan;
6 decreasing_velocity(isnan(neg_acceleration)) = nan;
```

Listing 4.3: Extracting negative acceleration

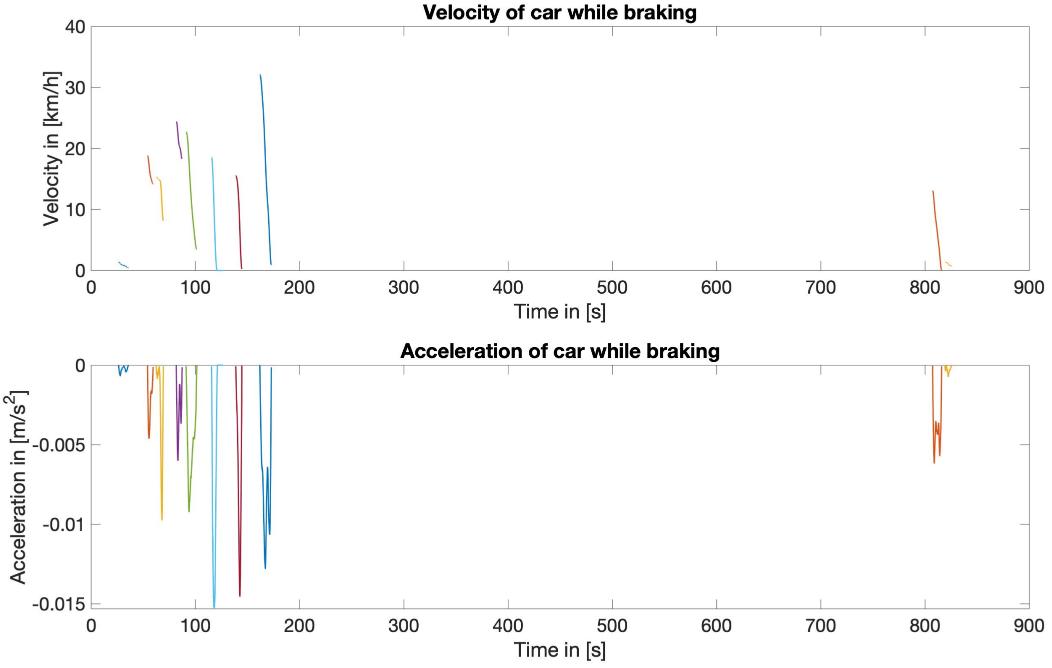


Figure 4.3: Braking manoeuvres in human velocity profile

## 4.4 Separating breaking sequences

For better visualisation, the breaking sequences that can be seen in figure 4.3 will be stored separately. One breaking sequence consists of multiple consequent velocity measurements. This means that the breaking sequences can be separated by searching for a gap in the velocity measurements (NaN values). Therefore, the indices of all velocities that are set not NaN are extracted (line 2 in listing 4.4) and differentiated (line 3 in listing 4.4). Consequently, very differentiation that is greater than 1 marks a gap between velocities because indices of one braking sequence are consequent (differentiation equals 1). Furthermore, short breaking sequences are not considered because they are most likely not relevant for recognizing a breaking pattern (line 13 in listing 4.4). Two individual breaking sequences can be seen in figure 4.4 and figure 4.8.

```
1  %find individual breaking sequences
2  notNaN_velocity = find(~isnan(decreasing_velocity));   % find index of every velocity that is not NaN -> one breaking sequence has consequent time steps -> one breaking sequence has consequent indices
3  diff_notNaN_velocity = diff(notNaN_velocity);          % differentiate indices -> if indices are not consequent (unequal 1), a new breaking sequence has begun
4  n = 1;                                                 % set start values
5  start = 1;
6  
7  %seperate breaking sequences with previous findings
8  for i=1:length(diff_notNaN_velocity)
9      %for every index check if indices are consequent -> equals 1
10     if diff_notNaN_velocity(i) > 1
11         %if not check if a breaking sequence consists of min 500 time
12         %steps, this way small breaking sequences are sorted out
13         if (i-start) > 500
14             %add breaking sequence to section
15             %i is end of sequence
16             section{n} = notNaN_velocity(start:i);
17             %start for new sequence is end of old sequence + 1
18             start = i+1;
19             %increment n
20             n = n+1;
21         else
22             %if breaking sequence is to small, set start for new sequence
23             %to end of old sequence + 1 anyway
24             start = i+1;
25         end
26     end
27 end
```

Listing 4.4: Separating breaking sequences

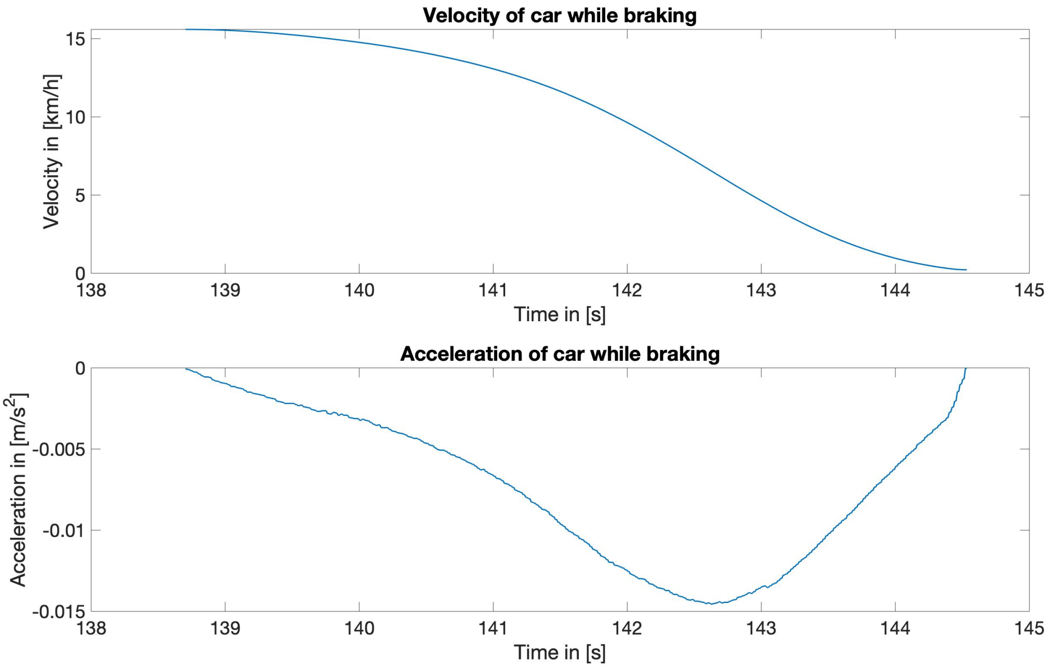


Figure 4.4: Individual braking manoeuvre 1 extracted from the human velocity profile

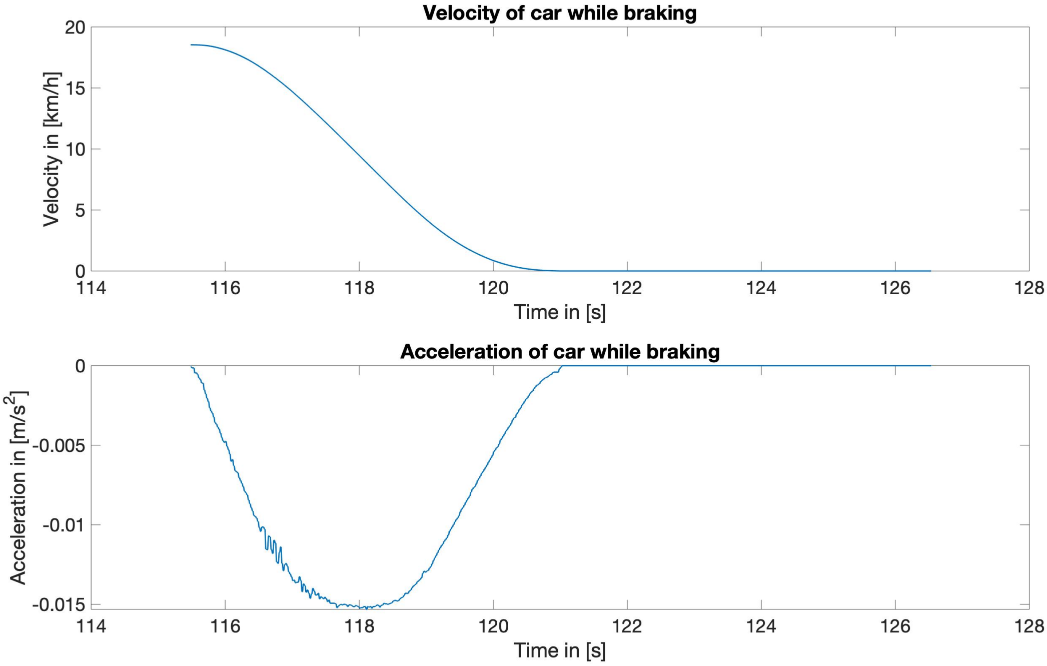


Figure 4.5: Individual braking manoeuvre 2 extracted from the human velocity profile

## 4.5 Results

Considering figures 4.4 and 4.8, it can be seen that the acceleration is approximately in the shape of a downward facing parable. This results in the velocity decreasing slowly in the beginning, then decreasing more severly and flattening in the end.

## 4.6 Development of p(t)

with no brake: -model from D2 -stopped after 1.8s, at approx 2.6m

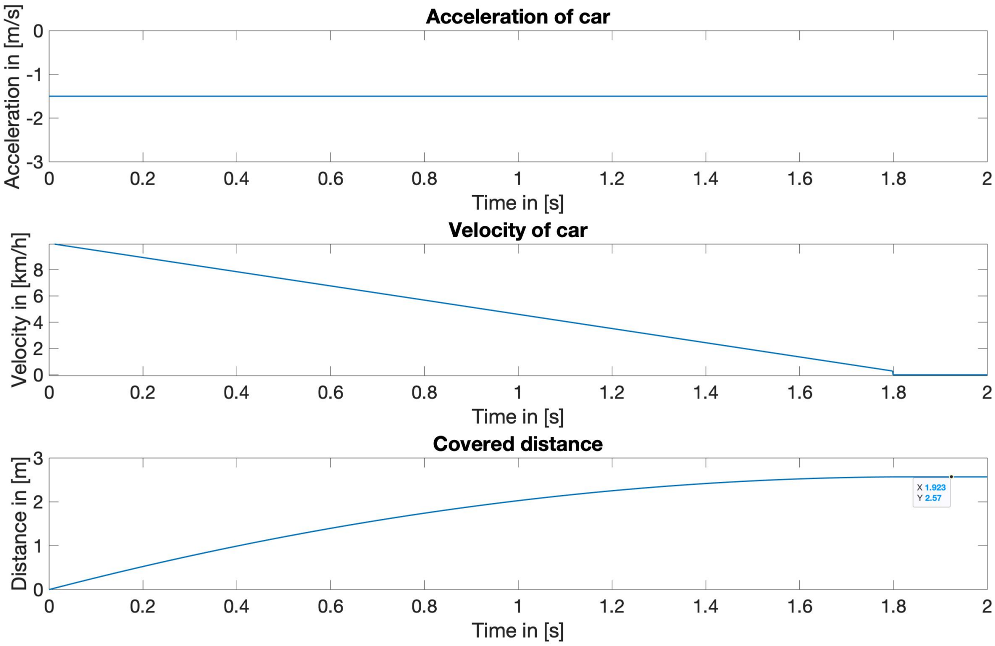


Figure 4.6: Simulation without braking

Now goal -p(t) time dependant -with acceleration curve similar to extracted above -start pressing brake increasing from start of brake maneuver and releasing pedal after half of maneuver

To simulate a time dependant brake pressure the before developed Simulink model has been extended with a lookup-table that contains the brake pressure parameters. The integrator that integrates the constant 1 provides the time input to the lookup table.

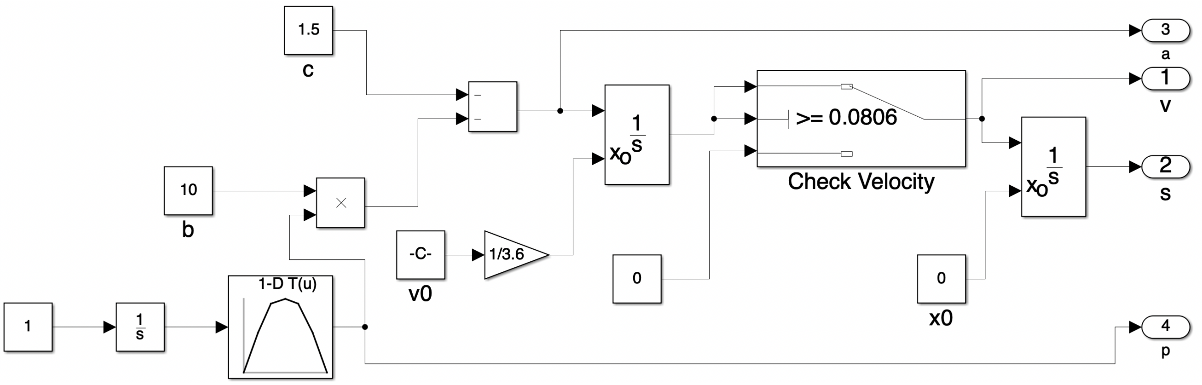


Figure 4.7: Simulation without braking

Table 4.1: Determined p(t) parameters

| Time in [s] | Brake pressure |
|---|---|
| 0 | 0 |
| 0.2 | 0.043 |
| 0.4 | 0.073 |
| 0.6 | 0.078 |
| 0.8 | 0.043 |
| 1 | 0.043 |
| 1.2 | 0 |

-$v_0$ 10 km/h -Stop at circa 1.92 meters -leave margin for inaccurate $v_0$, which will be analysed in chapter 6 todo check -Acceleration curve similar to human profile -let go of brake before coming to a stop, let friction do the rest

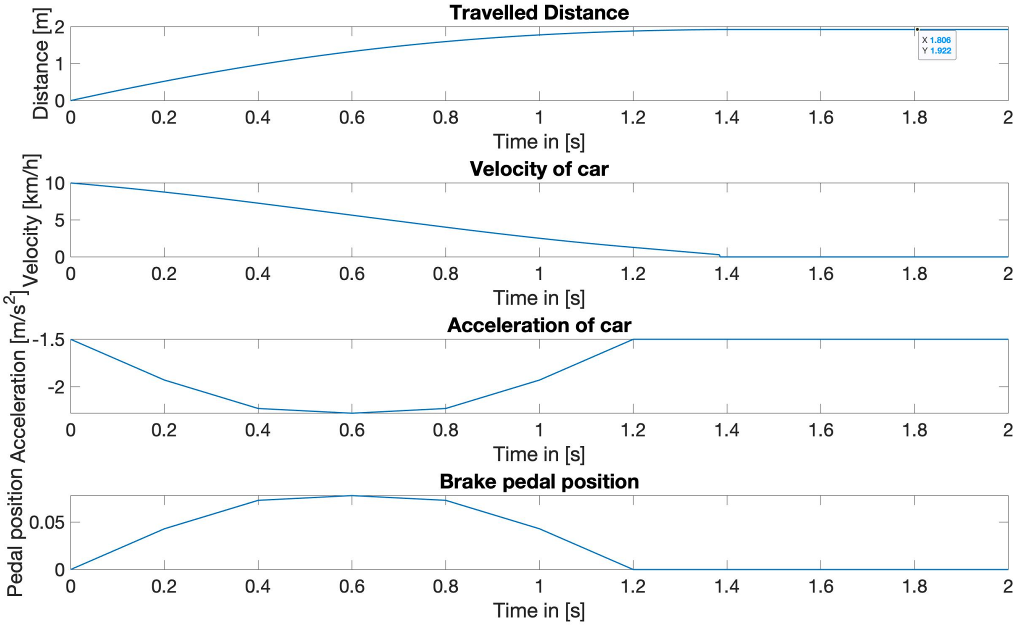


Figure 4.8: Simulation results with p(t)

# 5 D4*: Consideration of uneven parking spaces

-diagram

-forces need to be considered (hangabtriebskraft)

A negative slope would increase stopping time and position. Whereas a positive slope would make stopping time shorter. A negative slope critical since collision could occur.

-therefore brake power would have to be increased or decreased

-physical borders need to be considered

Also when the car is stopped it should not start to roll forwards or backwards. On a plain surface, the brake can be released, after the car is stopped. That might not be the case on an uneven parking space and the car could start rolling. To prevent that the handbrake would have do be engaged. A brake assist on a positive or negative slope would need to keep holding the brake or engaging the handbrake.

In the model without a minimum velocity, the velocity would become negative on braking instead of the car coming to a full stop.

# 6 D5: Discussion of inaccuracies in velocity measurement

## 6.1 Inaccuracy in car velocity

The velocity information of the car has an inaccuracy of velocity +/- 0.1 km/h. Also the minimal velocity is 0.29 km/h which is not realistic, because velocity does not drop from 0.29 km/h to zero in one step. The brake pedal function is not dependant on the velocity, therefore inaccuracies in the velocity measurement during the braking procedure do not influence the braking procedure. This has one exception. Considering the velocity measurement inaccuracy, the initial velocity could be in a range of 9.9 $km/h \leq v \geq$ 10.1 $km/h$. This would certainly influence the braking behaviour of the car. It would be dangerous if the initial velocity would be higher than expected, because the car would travel further than expected because of that. This might cause an accident. If it would be lower, the car might stop earlier but that would not lead to an accident.
A simulation is used to evaluate the danger of the initial velocity being higher than expected. The, highest initial velocity that is possible with R5 is 10.1 km/h. So this is used as $v_0$ for the impact evaluation. The model from Figure 4.8 from before is used for the evaluation, to see if the margin to 2 m is enough with a $v_0$ of 10.1 km/h.

As can be seen from the results of the simulation a -Figure shows that margin is enough, because stopping distance is 1.96 m.

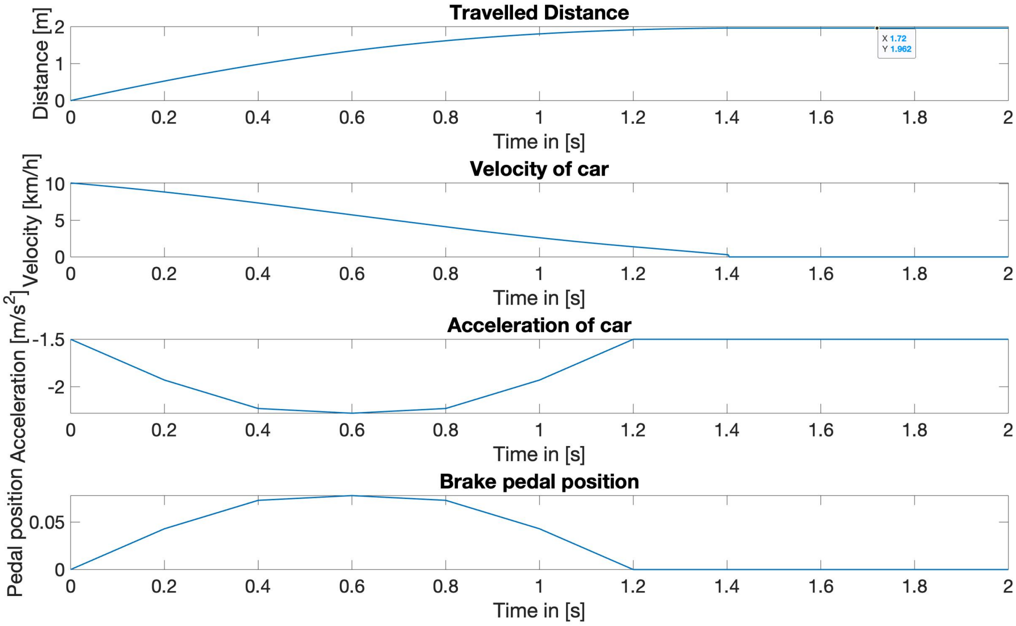


Figure 6.1: Simulation of inaccurate $v_0$

Uncertanties in the measured velcoity also influence the computed travelled distance. The velocity is needed to compute the travelled distance. The travelled distance is computed as the integral of the velocity. Therefore errors in the velocity carry into computed travelled distance. -integral could add up -if gaussian distribution overall error could be smaller

## 6.2 Inaccuracy in velocity measurement of human velocity profile

-in human velocity profile 4 tire velocities recorded -mean used to compute car velocity -because of inaccuracy in velocity computed acceleration also has inaccuracy -car driving around corner validate findings by numbers from simulation

Findings: -leave safety margin

# 7 D6: Implementation of pulse signal in Simulink

In D6 a pulsing information signal is described. This chapter documents the implementation of that signal. Both signals are needed for the frequency computation. The pulsing signal should only be present if the nonzero velocity of the car is $\leq 1$ m/s and the position of the car is between 1 and 2 meters. From a frequency of 1 Hz at 1 meter it should rise up to 9 Hz at 1.9 meters. If the traveled distance is greater than 1.9 meters, the signal should become continuous.

This requirement can be divided into two separate compontents with separate responsibilities.

In the first component it is determined in which state the signal is (off, pulsing, continuous) and in the case of a pulsing signal the frequency of that pulse is computed.

The second component is responsible for outputting the pulse signal corresponing to the output of the first component.

The components are implemented using simulink subsystems which enables re-usability, encapsulation and also creates a better overview when looking at the main simulink model.

## 7.1 Computation of pulse signal frequency

Figure 7.1 shows the simulink subsystem for the computation of the frequency of the pulse signal. The inputs are the velocity and position of the car. The if-condition determines in which state the signal is.

In the first case the nonzero car velocity is $\leq 1$ m/s and the position of the car is $> 1.9$ meters. In this case, the pulsing signal should provide a continuous output as demanded by requirement D6. The subsystem outputs a value of 10 in that case. This value is the indication for a continuous signal.

In the second case of the if condition the nonzero car velocity is also $\leq 1$ m/s but the position of the car is between 1 and 1.9 meters. In this case the frequency of the pulsing signal should be between 1 and 9 Hz depending on the location. For that a lookup table is used, that provides an output frequency that linearly increases from 1 to 9 Hz depending on the position from 1 to 1.9 meters. A linear increase is used, because then the driver

could estimate the position linearly by the signal.
When none of the above mentioned conditions are the case, the subsystem outputs 0, which indicates that the pulsing signal is not present.

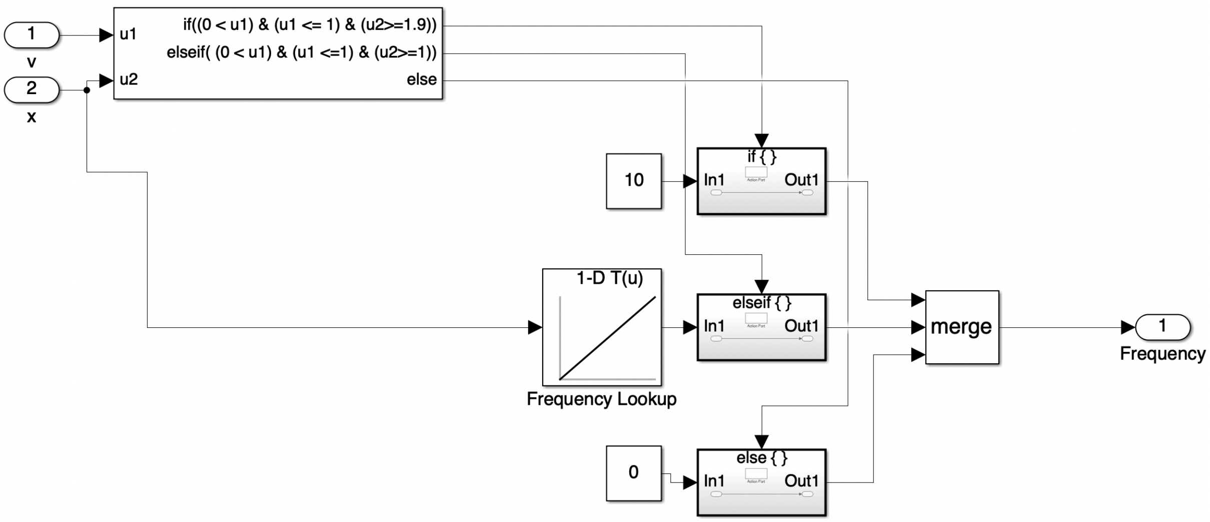


Figure 7.1: Simulink model of the frequency computation subsystem

todo output plot

## 7.2 Pulse signal generation

Figure 7.2 shows the simulink model of the pulse signal generator. This component provides a pulse signal with a variable frequency input. The input is the output of the before described component. Therefore if the input is 10, a continuous signal should be output. This is realised by the if-condition, which, if the input signal is 10 provides a continuous high output.
If the frequency is 0, 0 should be output.
If that is not the case either the pulse signal is generated. The frequency is the input to the integrator. The integrator has a reset input port and will be reset after each period. For each period the output of integrator will start at 0 and will go up to 1. Until the output reaches 0.5 a high pulse will be output (50 % duty cycle). This is the purpose of the less or equal 0.5 check. If the output of the integrator is greater that 0.5 a low pulse is output.

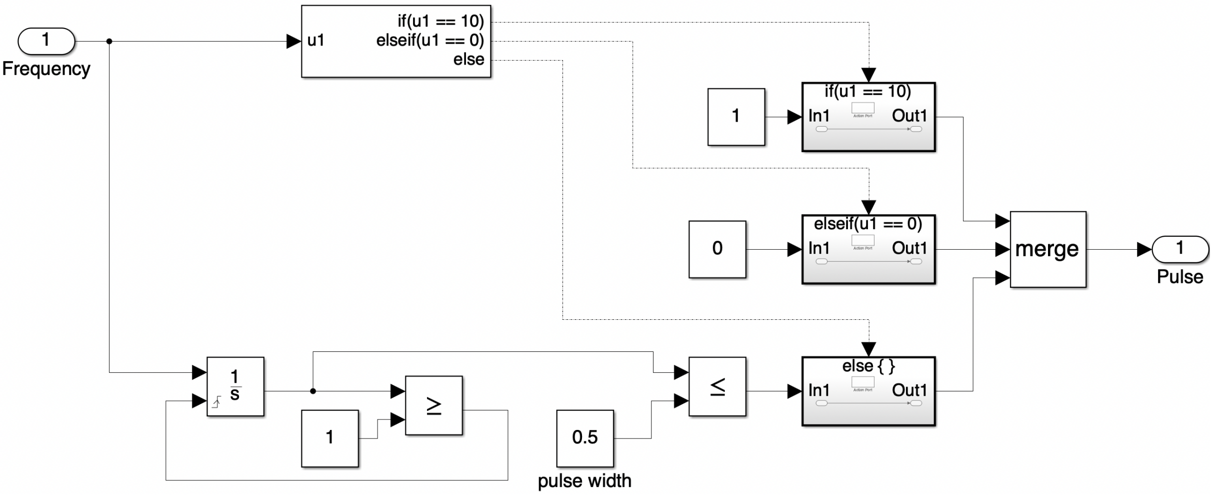


Figure 7.2: Simulink model of the pulse generation subsystem

## 7.3 Integration into car model

Using subsystems for the developed components makes it easy to include into and extend the existing model, which can be seen in figure 7.3.

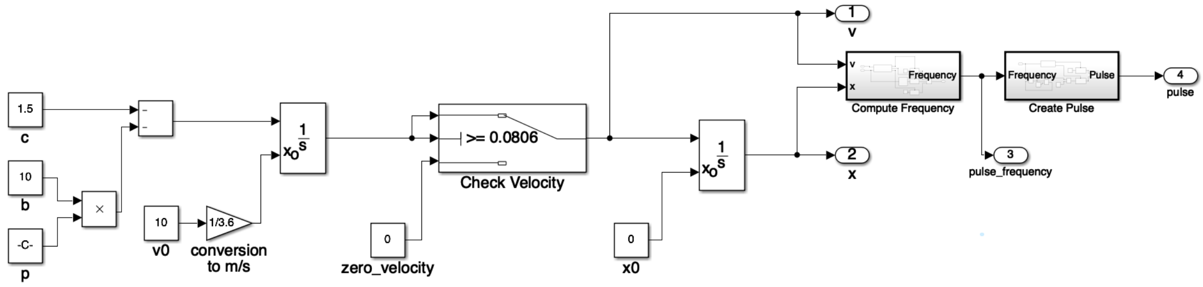


Figure 7.3: Simulink model of the car including the pulse signal

## 7.4 Signal demonstration

Figure 7.4 shows a demonstration of the implemented pulse signal. The Simulink car model is initialised with a constant brake pressure of 5 %. The frequency computation functionality can be seen in subplot 3. The frequency rises from zero, when the car reaches a velocity of 1 m/s (and also the distance is greater than 1m). Then the frequency increases as the car moves closer to 2 m. At 1.9 m traveled distance, the frequency rises to 10, which indicates a continuous pulse. When the car comes to a full stop at approximately 1.4 seconds the frequency goes back to zero, which indicates no pulse. The fourth subplot shows the output pulse signal. It starts when the frequency rises from zero, and the period of the pulses gets shorter with the time increasing. When the frequency starts to indicate a continuous pulse (from approximately 1.25 s) the pulse signal outputs a continuous high signal until the car comes to a stop. When the car stops, the continuous pulse stops and no pulse is output.

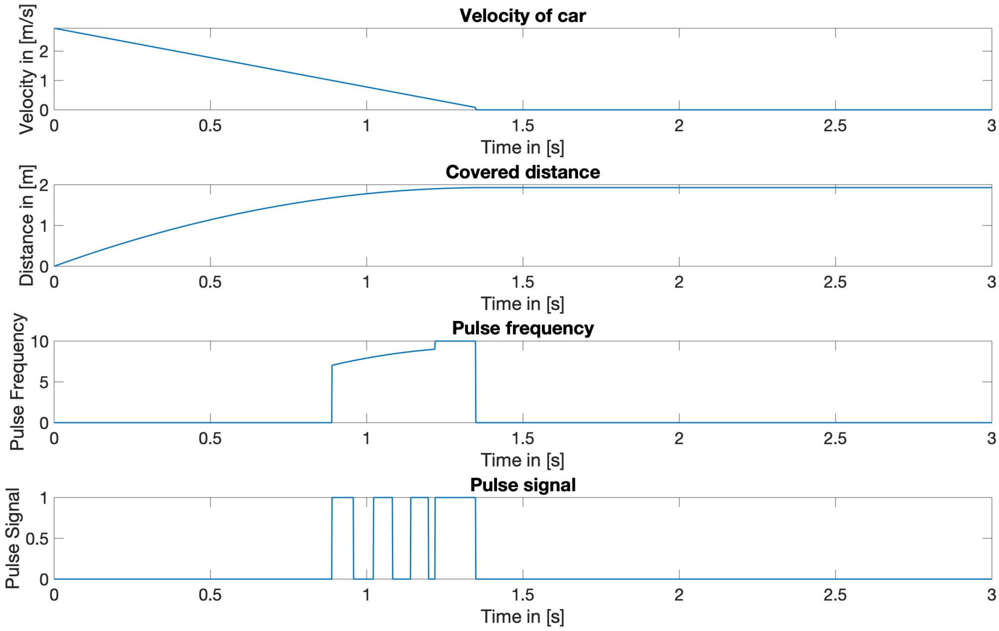


Figure 7.4: Demonstration of pulse signal

# 8 D7: Transfer of Simulink model to ASCET

This chapter describes how the Simulink model from chapter 3 is transferred to ASCET. Furthermore, as ASCET is time discrete execution slots for threads have to be defined.

First, messages are defined to describe the needed data interfaces. Listing 8.1 shows the two defined message interfaces. The first interface represents messages regarding the car including its velocity, its acceleration, its position, the ultrasonic distance and the pulse signal. The other interface contains a message that describes the current brake pedal position.

```
data interface CarMessages {
  real v = 0.0;
  real acceleration = 0.0;
  real position = 0.0;
  real ultrasonic_distance = 0.0;
  real pulse_signal = 0.0;
}

data interface ParkAssistMessages {
  real brake = 0.0;
}
```

Listing 8.1: Defined messages in ASCET

The car model is built using the static class *Car* whose block diagram is shown in Figure 8.1. The class consists of three thread functions: *drive*, *velocity* and *ultrasonic*. The *drive* function reads the brake pedal message and passes it together with a time constant to an instance of the *DriveModel* class (shown in Figure 8.2). Consequently, the function writes the acceleration message based on the *DriveModel* classes output. The *velocity* function writes the v message and the position message. The *ultrasonic* function writes the ultrasonic_distance message. The functionality of the *Car* class is split in three thread functions because of the given requirements demanding the ultrasonic_distance to be available every 2ms and the velocity to be available every 10ms. The implemented functions can be called individually in the specified time intervals.

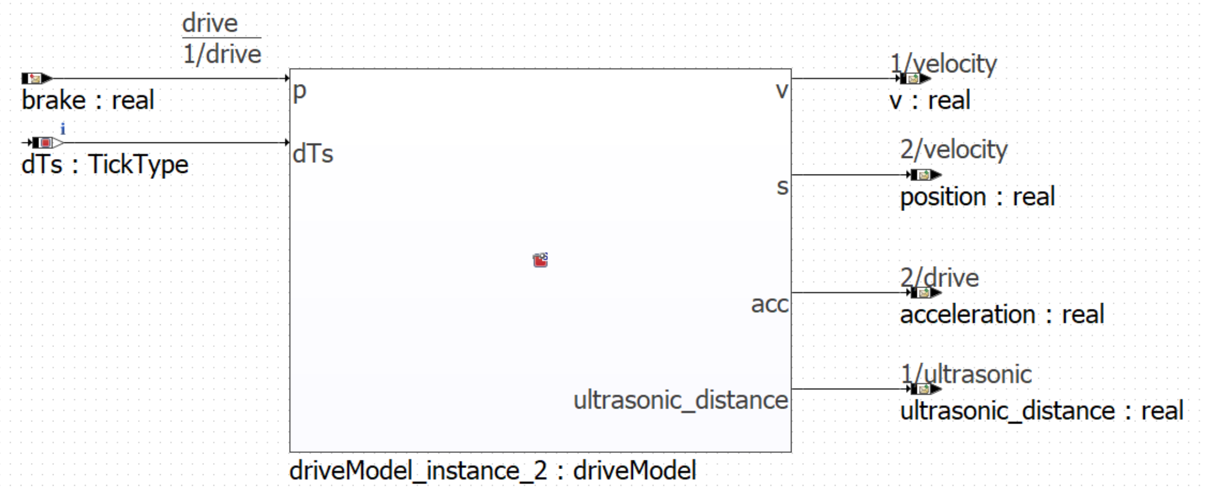


Figure 8.1: ASCET block diagram of static class *Car*

The *DrivingModel* class is responsible for calculating the acceleration, velocity and position of the car as well as the ultrasonic distance. These calculations are based on the input brake pedal position. The block diagram of the class is shown in Figure 8.2. Similar to the Simulink model, the acceleration is calculated by multiplying brake pedal position and braking constant and substracting it from the negative friction constant. The velocity is calculated by integrating the acceleration. Unlike Simulink ASCET does not possess integration blocks, so acceleration, time constant and the factor 3.6 to convert the acceleration from [m/s] to [km/h] are multiplied. The product is then added to the previous velocity. The default velocity is set to 10 km/h due to the requirements. If the velocity is less than 0.29 km/h it is set to zero based on requirement R5. The position is gained by multiplying velocity and time and diving it by 3.6 to get the position in meters and not in kilometres. The product is then also being added to the previous position whose default value is zero. The ultrasonic distance is gained by subtracting target position (2m) and position.

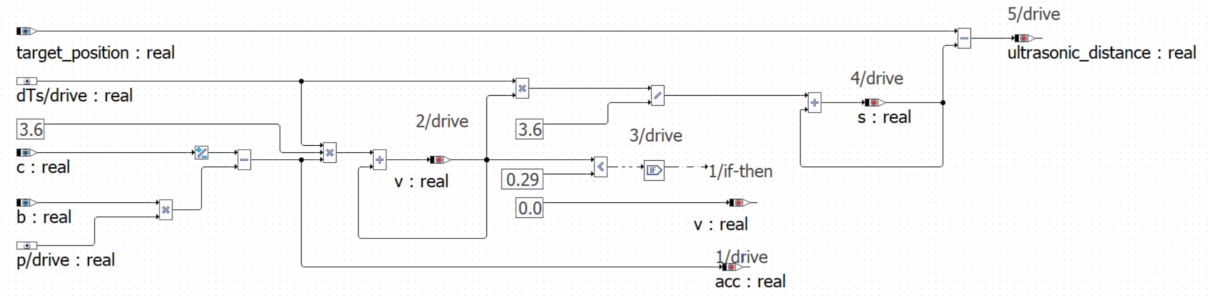


Figure 8.2: ASCET block diagram of class *DrivingModel*

Besides the car model, the static class *ParkAssistController* (see Figure 8.3) is implemented for generating the brake pedal position. Based on the elapsed time since starting the experiment, the class uses an instance of the class *ParkAssistControllerClass* to generate the brake pedal position and writes the brake pedal message afterwards.

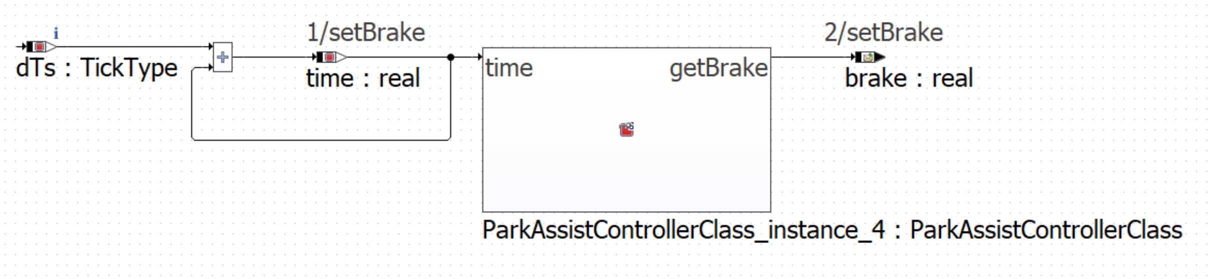


Figure 8.3: ASCET block diagram of static class *ParkAssistController*

The brake pedal position is determined based on a look-up table that represents the brake pedal function described in chapter 4. After determining the brake pedal position, the position is returned. The class *ParkAssistClass* is implemented as a non static class in order to unit test the class. The block diagram of the class is shown in Figure 8.4.

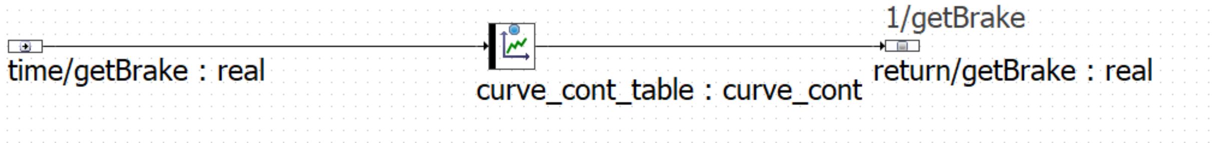


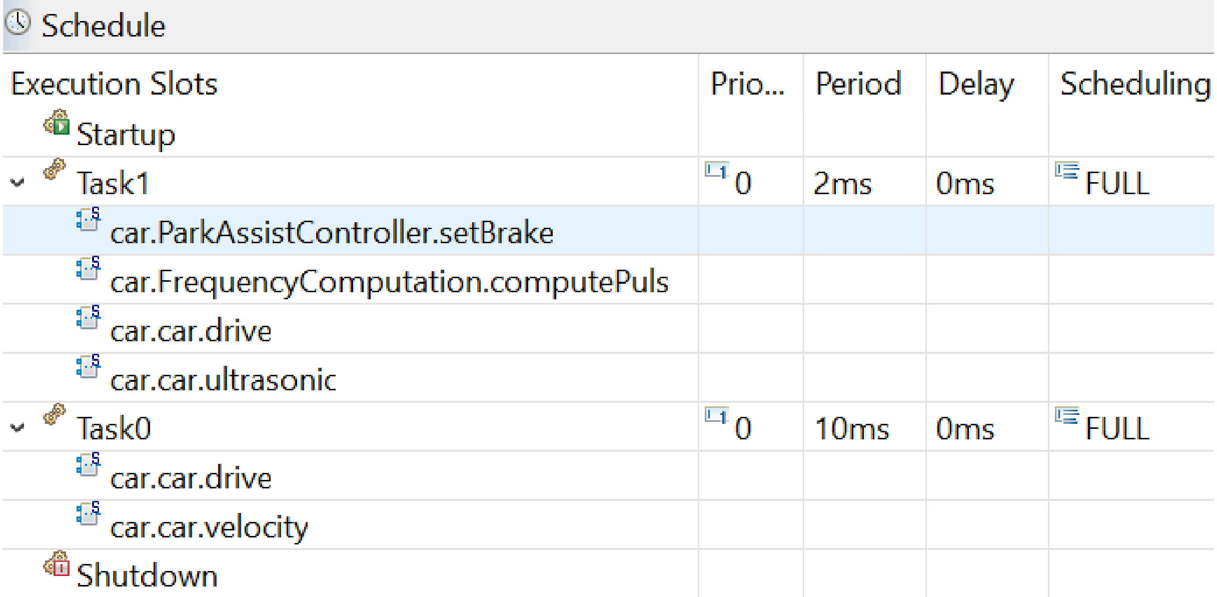
Figure 8.4: ASCET block diagram of class *ParkAssistControllerClass*

The braking pedal position look-up table is shown in Listing 8.2.

```
characteristic curve_cont curve_cont_table = {{0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2}, {0.0,
    0.043, 0.073, 0.078, 0.073, 0.043, 0.0}};
```

Listing 8.2: Braking pressure look up table

In order to execute the experiment, it is necessary to define the schedule. As can be seen in Figure 8.5, the experiment is divided in two tasks, one being executed every 2ms and the other task being executed every 10ms. As defined in requirement R4, the ultrasonic distance message is computed every 2ms. Alongside this, the ParkAssistController adjusts the brake pedal position and the pulse signal is computed also every 2ms for a higher precision. The velocity however is generated every 10ms as described in requirement R5.

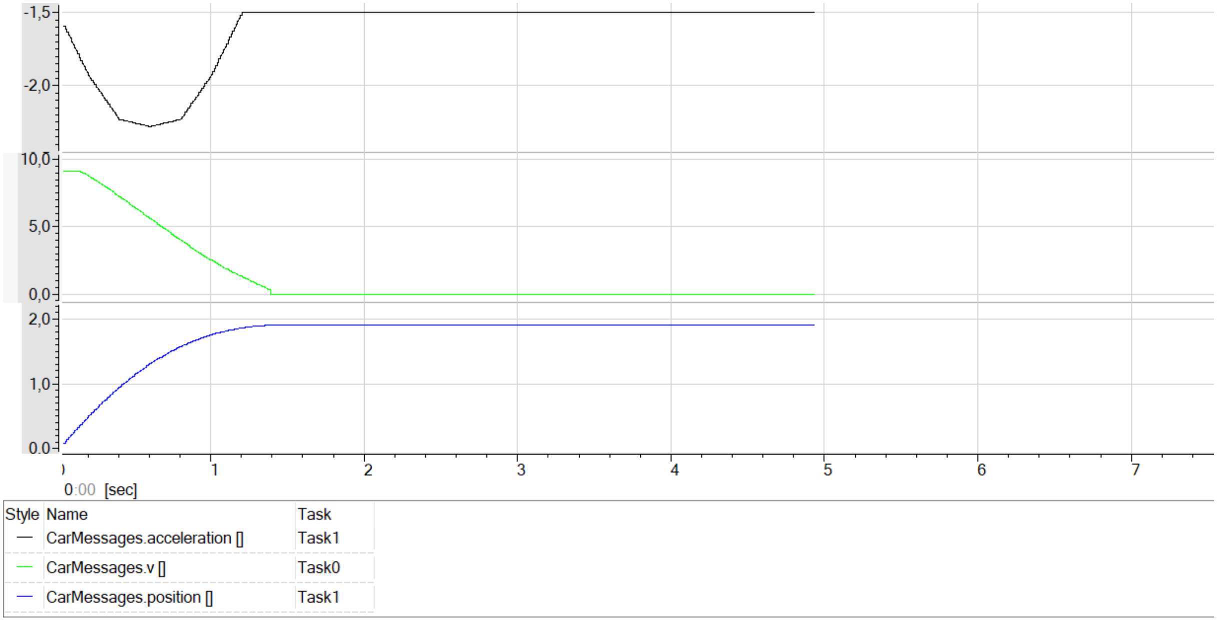


Schedule

| Execution Slots | Prio... | Period | Delay | Scheduling |
|---|---|---|---|---|
| Startup | | | | |
| Task1 | 0 | 2ms | 0ms | FULL |
| car.ParkAssistController.setBrake | | | | |
| car.FrequencyComputation.computePuls | | | | |
| car.car.drive | | | | |
| car.car.ultrasonic | | | | |
| Task0 | 0 | 10ms | 0ms | FULL |
| car.car.drive | | | | |
| car.car.velocity | | | | |
| Shutdown | | | | |

Figure 8.5: Schedule of ASCET model

The following Figure 8.6 shows the simulation results. The acceleration has the form of a parable as desired. Due to the friction and braking constant the velocity is steeper than the curve that can be seen in chapter 4. If one would like to have a similar velocity curve it would be necessary to apply gas in order to brake more slowly which is not possible in this model as there is no gas pedal. with a constant brake pressure of The third subplot shows the covered distance of the car, which is clearly under the 2 m mark.



| Style | Name | Task |
|---|---|---|
| — | CarMessages.acceleration [] | Task1 |
| — | CarMessages.v [] | Task0 |
| — | CarMessages.position [] | Task1 |

Figure 8.6: ASCET car model

# 9 D8: Implementation of pulse signal in ASCET

The pulse signal that has been implemented in Simulink (see chapter 7) is now transferred to ASCET. The requirements remain the same. The ASCET model consists of three sub-models described in three block diagrams. Two of the sub-models are regular classes that can be instantiated. This makes the whole model modular and the sub-models are able to unit test. One of the two classes calculates a frequency based on a given velocity and position of the car. The second class generates a pulse signal based on the calculated frequency. The third class is a static class that passes the calculated frequency to the pulse signal generating class. The static class is also responsible for receiving the velocity and position messages of the car and pass them to the frequency computation class as well as sending the generated pulse signal as a new message. This can be seen in Figure9.1 that shows the block diagram of the static class.

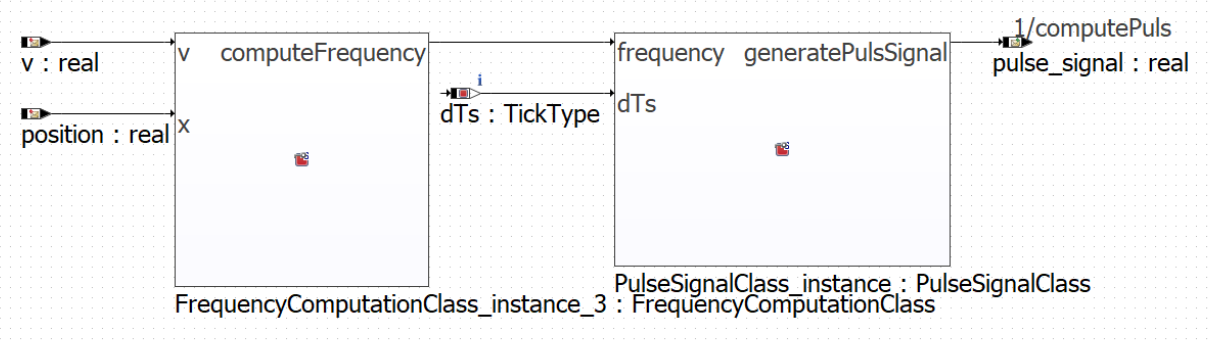


Figure 9.1: ASCET block diagram of static class *FrequencyComputation*

The frequency is calculated based on the Simulink implementation. However, in ASCET if-clauses with multiple conditions need to be implemented in series. The model can be seen in Figure 9.2 and the sequencing process is described in the following:

1. If $x \geq 1.9m$, $v \leq 1.0$ and $v > 0.0$ return 10 (**1a** in Figure 9.2)

2. Otherwise (**1b** in Figure 9.2) if $x \geq 1.0$, $v \leq 1.0$ and $v > 0.0$ find frequency in lookup table (**2a** in Figure 9.2). The lookup table is shown in Listing 9.1. If none of the conditions haven not been meet return 0 (**2b** in Figure 9.2)

```
characteristic curve_cont frequency_lookup = {{1.0, 1.9}, {1.0, 9.0}};
```

Listing 9.1: ASCET frequency lookup table

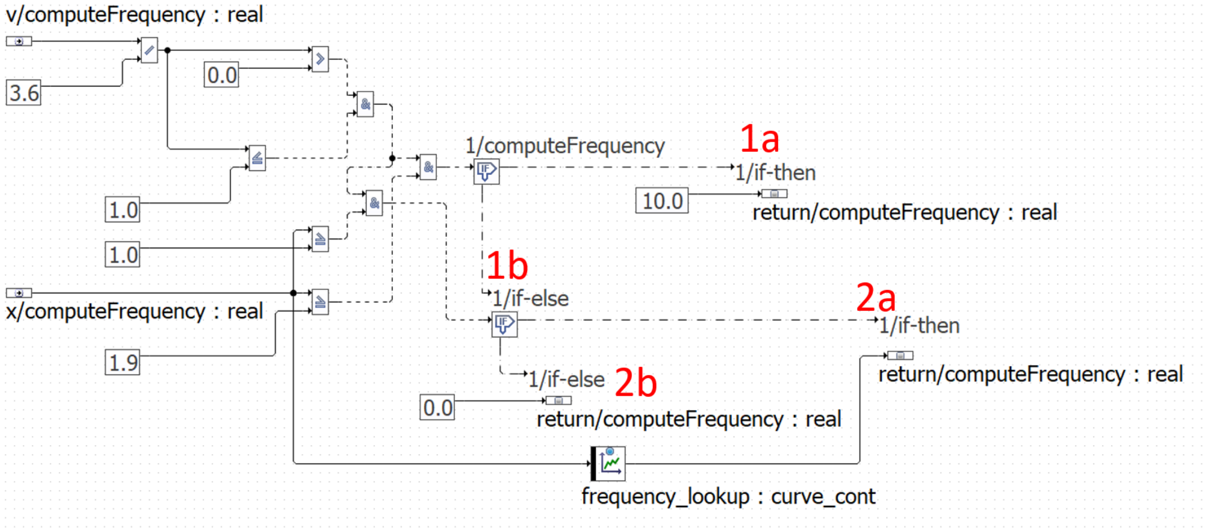

v/computeFrequency : real
3.6
0.0
1.0
1.0
x/computeFrequency : real
1.9
1/computeFrequency
1a
1/if-then
10.0
return/computeFrequency : real
1b
1/if-else
2a
1/if-then
return/computeFrequency : real
1/if-else
2b
0.0
return/computeFrequency : real
frequency_lookup : curve_cont

Figure 9.2: ASCET block diagram of class *FrequencyComputationClass*

After the frequency has been calculated, it is passed to generate the pulse signal. If the frequency equals 10, the desired output is a continuous pulse. This means the pulse is set to 1 (**1a** in Figure 9.3). If however the frequency equals 0, no pulse is desired resulting in the pulse being set to 0 (**2a** in Figure 9.3). If $0 < frequency < 10$, the frequency is integrated to generate the pulse signal. For integration a time component is required. Due to the ability to unit test the component, the time component is passed through a method argument. The frequency is integrated by multiplying frequency and time and adding the result to the previous results (**2b** in Figure 9.3). In Simulink it is possible to reset the integration block. Due to this not being possible in ASCET, the integrated frequency is reset as soon as the integrated frequency is greater or equal 1 because then one period is over and the new periodic time can be calculated (**2b** in Figure 9.3). While the integrated frequency is smaller or equal 0.5 the pulse signal is set to 1 (**3a** in Figure 9.3). Once the frequency is greater than 0.5 the pulse signal is set to 0 (**3b** in Figure 9.3). This means the signal is high for one half of a period and low for the other half of a period.

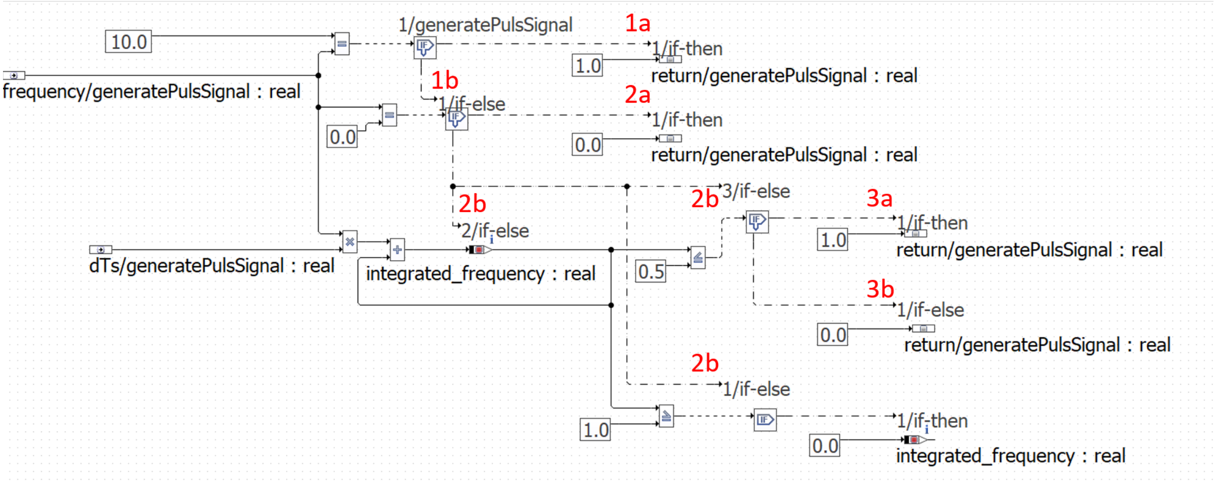
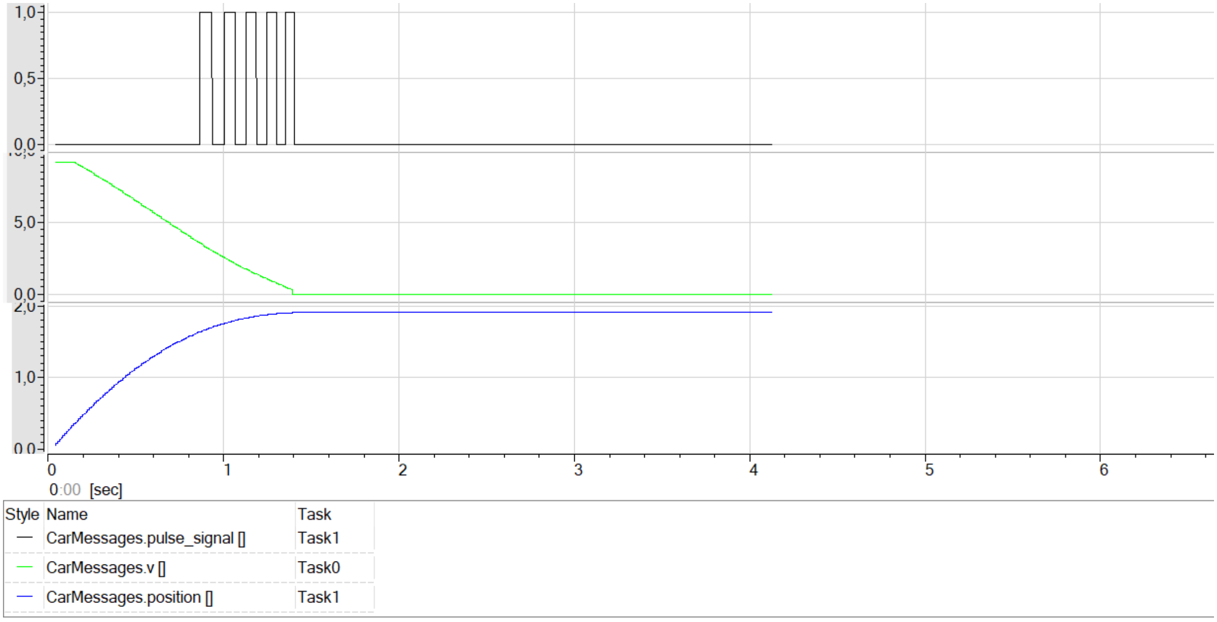

10.0
frequency/generatePulsSignal : real
1/generatePulsSignal
1a
1/if-then
1.0
return/generatePulsSignal : real
1b
1/if-else
2a
0.0
1/if-then
0.0
return/generatePulsSignal : real
2b
3/if-else
3a
2b
2/if-else
dTs/generatePulsSignal : real
integrated_frequency : real
0.5
1/if-then
1.0
return/generatePulsSignal : real
3b
1/if-else
0.0
return/generatePulsSignal : real
2b
1/if-else
1.0
1/if-then
0.0
integrated_frequency : real

Figure 9.3: ASCET block diagram of class *PulseSignalClass*

The resulting pulse signal is shown in Figure 9.4. It shows that the period of the pulse signal is getting shorter once the car approaches the target position. The plot also shows that the pulse signal starts once the car is slower than 3.6 km/h (≙ 1m/s) and ends once the car is standing.



| Style | Name | Task |
|---|---|---|
| — | CarMessages.pulse_signal [] | Task1 |
| — | CarMessages.v [] | Task0 |
| — | CarMessages.position [] | Task1 |

Figure 9.4: Output pulse signal

# 10 D9: Implementation of unit tests for ASCET model parts

This section covers the unit tests of all components of the ASCET model. The components are designed in a modular way to allow easy unit testing. Also wrapper classes, passing messages into the components, are used to make it possible to unit test the components. This is the only functionality the wrapper classes have. That is necessary to make sure that no new bugs are introduced by the wrapper classes, because they can not be unit tested.
The unit tests are bundled in a package called *test* in the ASCET project. The assertLib is imported for the assertions that are used in the unit tests. The unit tests are grouped by component with a static test class for each component. One test case corresponds to one method within the unit test classes.

Test cases are derived by analysing the requirements and finding equivalence classes. The goal of the test case coverage is to cover equivalence class and in addition to that select values that might result in errors based on experience.

## 10.1 Unit tests for DriveModel

*DriveModel* computes acceleration, velocity, position and ultrasonic distance based on a defined time and brake pedal position. Generally, there are two equivalence classes. One equivalence class covers all velocities greater than 0.29 km/h and the other all velocities less or equal 0.29 km/h. As only one time step is simulated while unit testing the position will become zero if the velocity is zero. By the time the velocity becomes zero in the system test environment, the position is greater than zero. Due to this inconsistency this equivalence class is not tested. Defined test cases can be seen in the table below.

Table 10.1: Test cases for unit testing driveModel

| **Brake pedal** | **Time constant [s]** | **Acceleration [m/s]** | **Velocity [km/h]** | **Position [m]** | **Ultrasonic distance [m]** |
|---|---|---|---|---|---|
| p = 0.0 | dTs = 1.0 | acc = -1.5 | v = 4.6 | s ≈ 1.278 | u ≈ 0.722 |
| p = 0.5 | dTs = 0.1 | acc = -6.5 | v = 7.66 | s ≈ 0.213 | u ≈ 1.787 |

## 10.2 Unit tests ParkAssistControllerTest

*ParkAssistControllClass* generates a brake pedal position based on a defined time using a look-up table. There are two equivalence classes defined for the *ParkAssistControll* class that cover the brake pedal position if the time is inside the time span of the look-up table and if the time is greater than the time span of the look-up table.

Table 10.2: Equivalence classes for unit testing ParkAssistControll

| **Time [s]** | **Output p(t)** |
|---|---|
| $0 \leq t \leq 1.2$ | $0 \leq p(t) \leq 0.078$ |
| t > 1.2 | p(t) = 0 |

Based on the equivalence classes three test cases have been derived. Two test cases correspond to the first equivalence class and the third test case corresponds to the second equivalence class.

Table 10.3: Unit test test cases pulse signal frequency generation

| **Test case name** | **Time [s]** | **Brake pedal** |
|---|---|---|
| time_0_0 | 0.0 | 0.0 |
| time_0_6 | 0.6 | 0.078 |
| time_2_0 | 2.0 | 0.0 |

## 10.3 Unit tests FrequencyComputationTest

*FrequencyComputationClass* generates a frequency based on a defined position and velocity using a look-up table. There are three equivalence classes defined for the *FrequencyComputationClass* class that cover three output scenarios: Zero frequency, variable frequency based on a defined function and Continuous frequency.

Table 10.4: Equivalence classes for unit testing pulse signal frequency generation

| **Position** | **Velocity** | **Output** |
|---|---|---|
| irrelevant | v = 0m/s or v > 1m/s | Zero |
| 1 m <= s <= 1.9 m | 0 m/s < v <= 1 m/s | $f(s) = -7.889\ Hz + 8.889 * s\ Hz$ |
| 1.9 m < s <= 2 m | 0 m/s < v <= 1 m/s | Continuous (10) |

Table 10.5 shows the test cases derived from the equivalence classes above.

Table 10.5: Unit test test cases pulse signal frequency generation

| Test case name | Velocity [m/s] | Position [m] | Expected frequency |
|---|---|---|---|
| continuousPulse | 0.5 | 1.91 | 10 |
| noFrequencyBecauseVelocityHigh | 1.5 | 1 | 1 |
| noFrequencyBecauseVelocityZero | 0 | 1.8 | 0 |
| noFrequencyBecausPosition | 0.9 | 0.9 | 0 |
| noFrequencyBecauseVelocityAndPosition | 1.1 | 0.9 | 0 |
| frequencyLow | 1.0 | 1.0 | 1 |
| frequencyHigh | 1.0 | 1.9 | 9 |
| frequencyMid | 1.0 | 1.5 | 5.44 < result <5.45 |

## 10.4 Unit tests PulseSignalClass

*PulseSignalClass* generates a pulse signal based on a defined frequency. There are three equivalence classes defined for the *PulseSignalClass* class that refer to the equivalence classes in section 10.3. The table below shows the equivalence classes for the pulse signal.

Table 10.6: Equivalence classes for unit testing pulse signal

| Position | Velocity | Pulse Signal |
|---|---|---|
| irrelevant | V = 0m/s oder V > 1m/s | Zero |
| 1m <= X <= 1.9m | 0m/s < V <= 1m/s | Alternating 0,1 |
| 1.9m < X <= 2m | 0m/s < V <= 1m/s | Continuous |

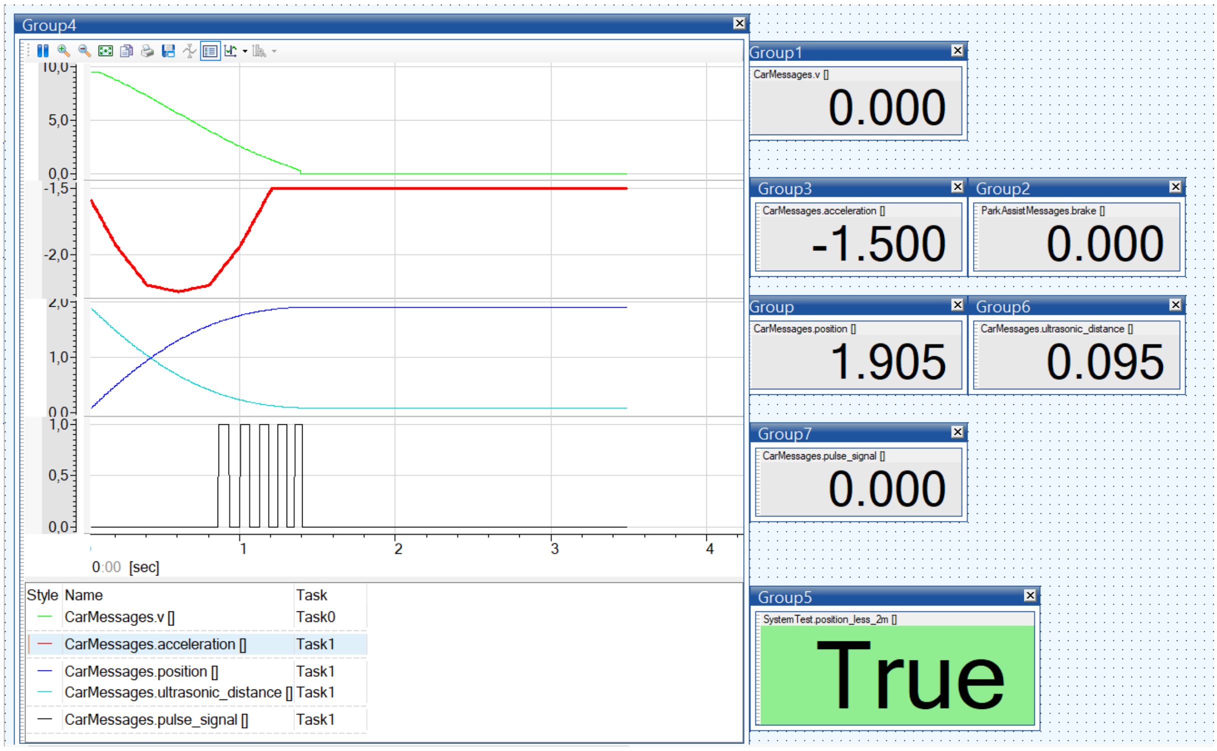
The derived test cases are shown below.

Table 10.7: Pulse signal test cases

| Frequency | dts | Expected result |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 0.1 | 1 |
| 1 | 0.6 | 0 |
| 10 | 1 | 1 |

# 11 D10: Development and implementation of a system test environment for ASCET simulation

- checkbox if position is less than 2m
- acceleration follows D3
- pulse signal periods become smaller



| Style | Name | Task |
|---|---|---|
| 🟩 | CarMessages.v [] | Task0 |
| 🟥 | CarMessages.acceleration [] | Task1 |
| 🟦 | CarMessages.position [] | Task1 |
| 🟦 | CarMessages.ultrasonic_distance [] | Task1 |
| ⬛ | CarMessages.pulse_signal [] | Task1 |

Group1
CarMessages.v []
0.000

Group3
CarMessages.acceleration []
-1.500

Group2
ParkAssistMessages.brake []
0.000

Group
CarMessages.position []
1.905

Group6
CarMessages.ultrasonic_distance []
0.095

Group7
CarMessages.pulse_signal []
0.000

Group5
SystemTest.position_less_2m []
True

Figure 11.1: ASCET Experiment Environment

# 12 D11*: Plausibility check comparing measured velocities and distances

The distance signal from the ultrasonic sensor (in case there is a non-moving object in front of it) and the measured car velocity are two measurements that provide independent measurement of travelled distance.
In the given scenario it is known that a non-moving object is 2 m in front of the car. So the ultrasonic sensor can be used to compute travelled distance by comparing distance measurements.
Integrating the velocity also provides a measurement of the travelled distance. These two measurements provide a redundancy in measuring travelled distance. If thes measurements would be significantly different, one of the measurements must be inaccurate.

The distance signal from the ultrasonic sensor and the measured velocity. -statistically independent ultrasonic distance and velocity -if distance to next object for ultrasonic sensor is known

# 13 D13*: Impact of inaccuracies

Ultrasonic sensor: Using one ultrasonic sensor has limits when measuring distance to objects when parking. Several scenarios can be imagined, in which a distance measurement from one ultrasonic sensor does not provide enough information to securely park without causing an accident. The scenario in 13.1 shows the limits of a ultrasound sensor. One ultrasonic sensor in the middle of the front of the car might not be able to detect the two poles. In this case parking based on the information of the ultrasonic sensor might result in an accident. Using multiple ultrasonic distributed over the front of the vehicle might be able to resolve such an issue.

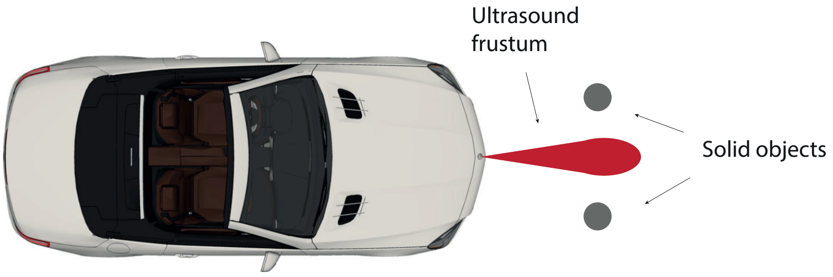


Figure 13.1: Ultrasonic sensor limits

With one ultrasonic sensor blind spot over ultrasonic sensor -can be seen in figure 13.2

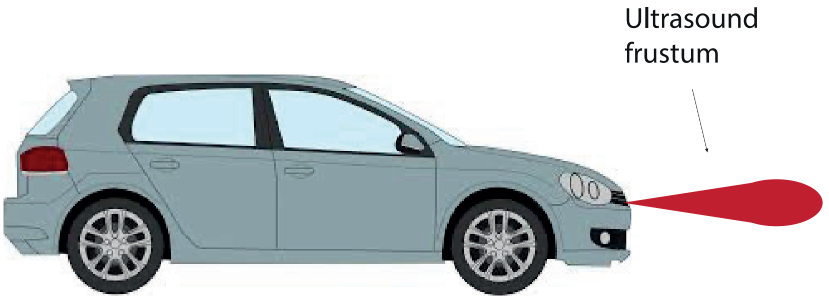


Figure 13.2: Ultrasonic sensor limits

pictures poles, round object.

a countermeasure for that could be to use multiple ultrasonic sensors -ultrasonic measurement: what if object is round? - -also above and below measurement

-velocity measurement -different wheel velocities

# 14 D14*: Reflection

-only 2 meter stop considered -no feedback -model not realistic