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

?? matlab version Matlba R2019b ASCET ... ascet version

todo describe block diagram vs script for every necessary requirement

2 D1: Time estimate based on three point estimation

Table 2.1: Three point estimation of effort for meeting requirements

Requirement	Optimistic	Likely	Pessimistic	<T>	sigmahoch2	Actual
D1	.	.	.	.	.	

3 D2: Feasibility study

The aim of the feasibility study is to analyse whether it is possible to realise a smooth stop 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 a Simulink model is created representing the model above in the next section. 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, corresponding to the scenario, that should be tested. Since the output of the 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 and the velocity would become negative eventually. This is realised by 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.

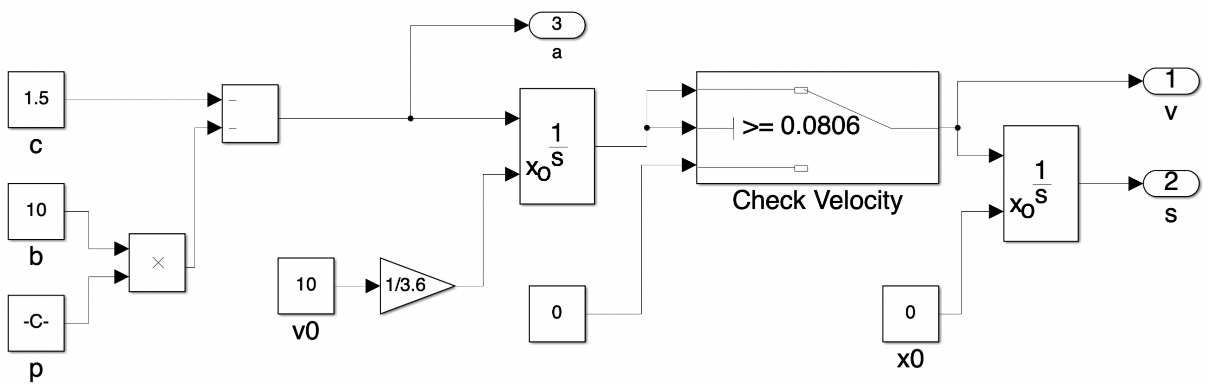


Figure 3.1: Simulink model of the differential equations

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 position < 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 realise a human-like smooth stop, but to show that that 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 position < 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.
The execution of this scenario is automated using a matlab script, which parametrizes the model and displays the outputs. Listing ?? 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
8 set_param('D2/p','value',sprintf('%f',0.05));
9
```

```

%% 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. (todo acceleration remains the same but velocity does not increase) 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 plot 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.
These simulation results demonstrate show that it should be possible, with variation of the brake pressure over time, to realise a smooth stop with a human-like brake profile. todo vllt mehr begründen

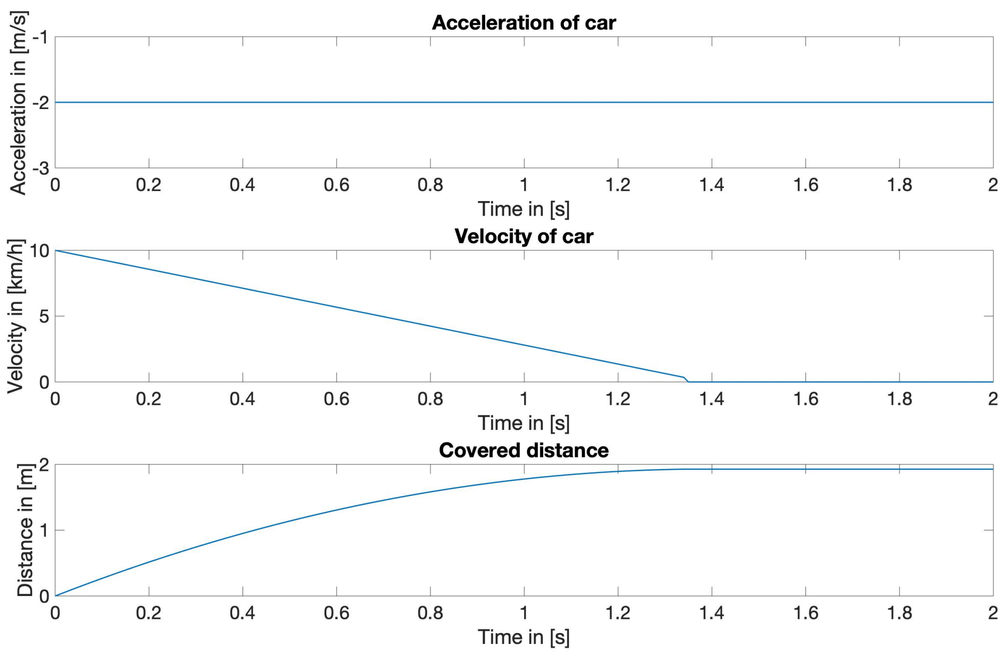


Figure 3.2: Simulation results of feasibility study

4 D3: Analysis of human velocity profile

In this section the provided human velocity profile is analysed in order to find a pattern that shows how a human is breaking a car. This pattern will be used in the following sections and will be adapted to the braking function of the ParkAssist. The analysis is divided into four steps.

-um auf bremse rückzuschließen, beschleunigung berechnen, weil bremse geht in beschleunigung ein

4.1 Import the measurement data

The measurement data is imported in Matlab.

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

Listing 4.1: Import measurement data in Matlab

4.2 Data preparation

Before analysing the data it is necessary to preprocess it. First the four velocities of each wheel are extracted from the measurement data and a mean velocity is calculated (see line 1,2).

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

The mean velocity is converted to [m/s] for further calculations. For analysing the braking behaviour the acceleration is calculated. This can be done by differentiating the velocity (see line 11). As can be seen in figure 4.1 in the upper plot the calculated acceleration is ?? as cause of minor variations in the velocity. Due to the ?? it would be hard to analyse the braking behaviour. Therefore, we used a moving average filter to smoothen the data (line 8). A moving average filter is calculating a mean value of neighbourhood elements within a sliding window of a predefined size. Applying a moving average filter results in a smoother graph as can bee seen in the lower plot in figure 4.1.

-linear dependance between acceleration and brake pedal

```
%compute mean velocity of all 4 wheels
velocity_per_wheel = velocity_data(:,2:5);
mean_velocity = mean(velocity_per_wheel,2);
mean_velocity = mean_velocity/3.6;      %convert velocity from km/h to m/s
raw_mean_velocity = mean_velocity;      %for demonstration purposes

%apply moving average filter to smoothen the data
mean_velocity = movmean(mean_velocity, 200);

%differentiate velocity to get acceleration
acceleration = diff(mean_velocity);
raw_acceleration = diff(raw_mean_velocity);     %for demonstration purposes
```

Listing 4.2: Preprocessing measurement data

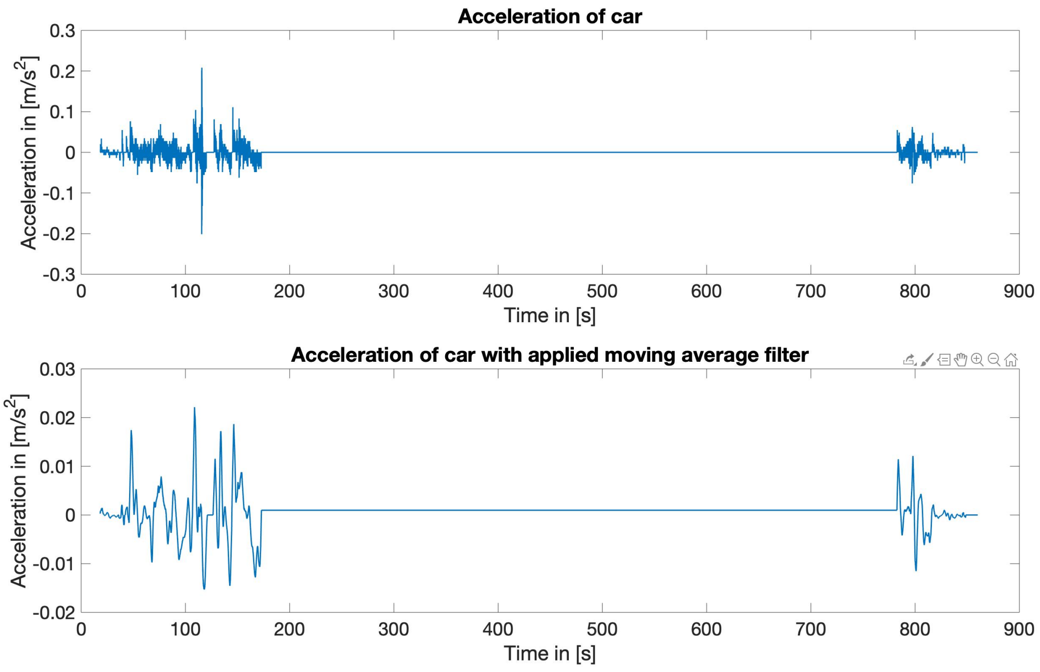


Figure 4.1: Human velocity profile car acceleration

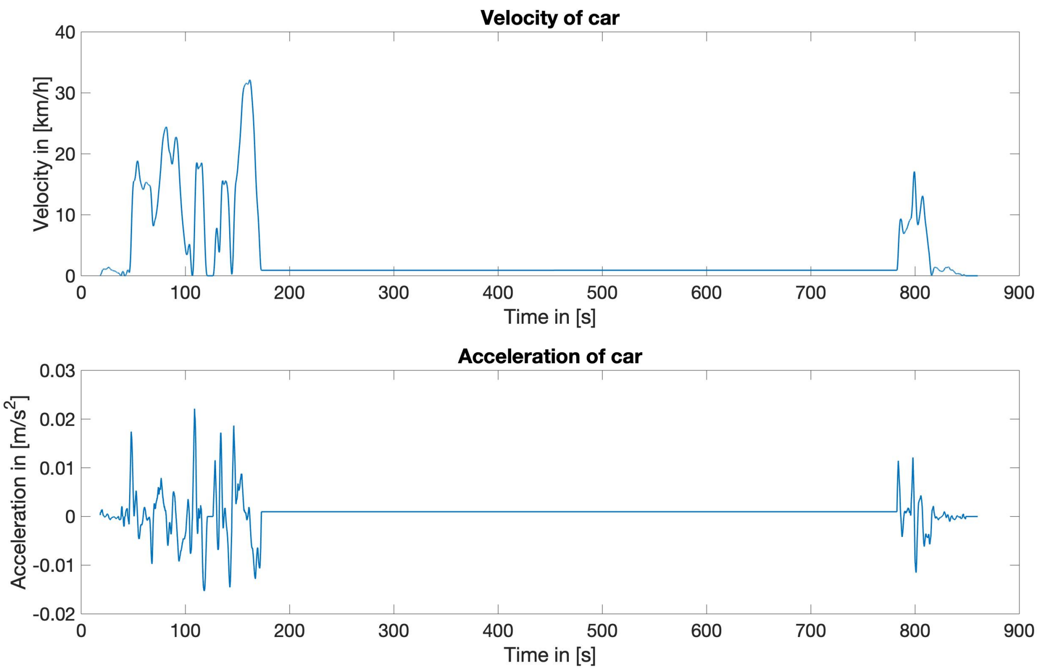


Figure 4.2: Smoothend data from human velocity profile

4.3 Extracting negative acceleration

As the goal is to analyse the braking behaviour only the negative acceleration is relevant and is thus being extracted from the overall acceleration (line 5). Also, all velocities that are based on a positive acceleration are not considered anymore (line 6). 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 plot REF. todo create plot

```
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

4.4 Separating breaking sequences

For improved visualisation the individual breaking sequences that can be seen in plot 4.3 are stored separately. One breaking sequence consists of multiple consequent velocities.

This means that the breaking sequences can be separated by searching for a gap in the velocities. Therefore, the indices of all velocities that are set not NaN are extracted (line 2) and differentiated (line 3). Considered the differentiation, every differentiation that is greater than 1 marks a gap between velocities because indices of one braking sequence are consequent (differentiation equals 1). Furthermore, small breaking sequences are not considered because they are most likely not relevant for recognizing a breaking pattern (line 13). Two individual breaking sequences can be seen in plot 4.4 and plot 4.5.

```
 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

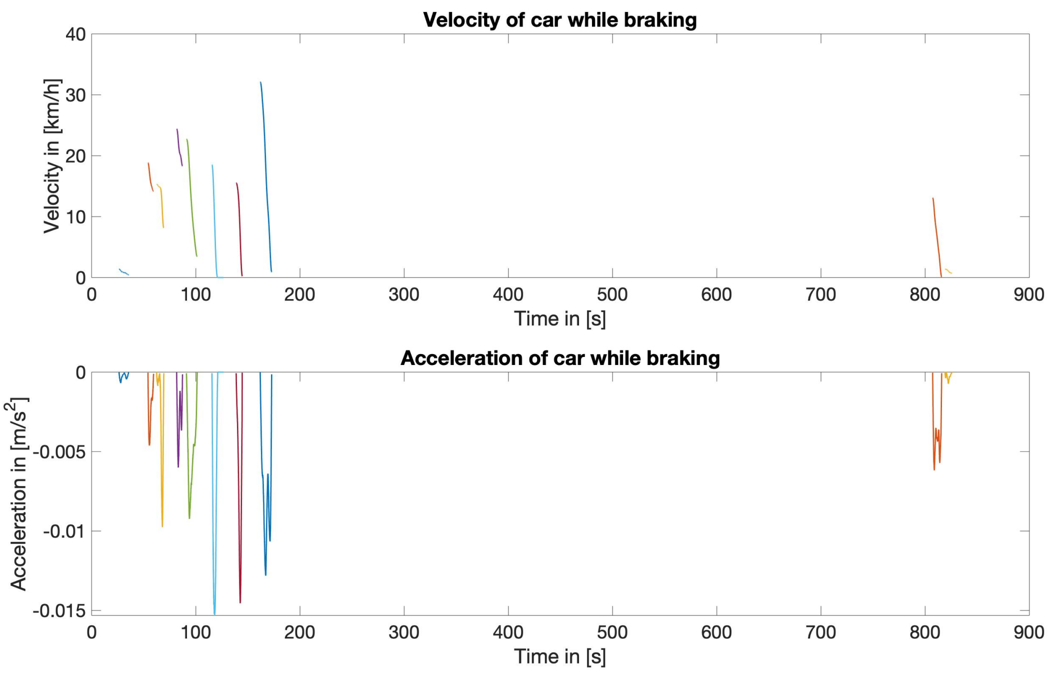


Figure 4.3: Human velocity profile extracted individual braking manoeuvres

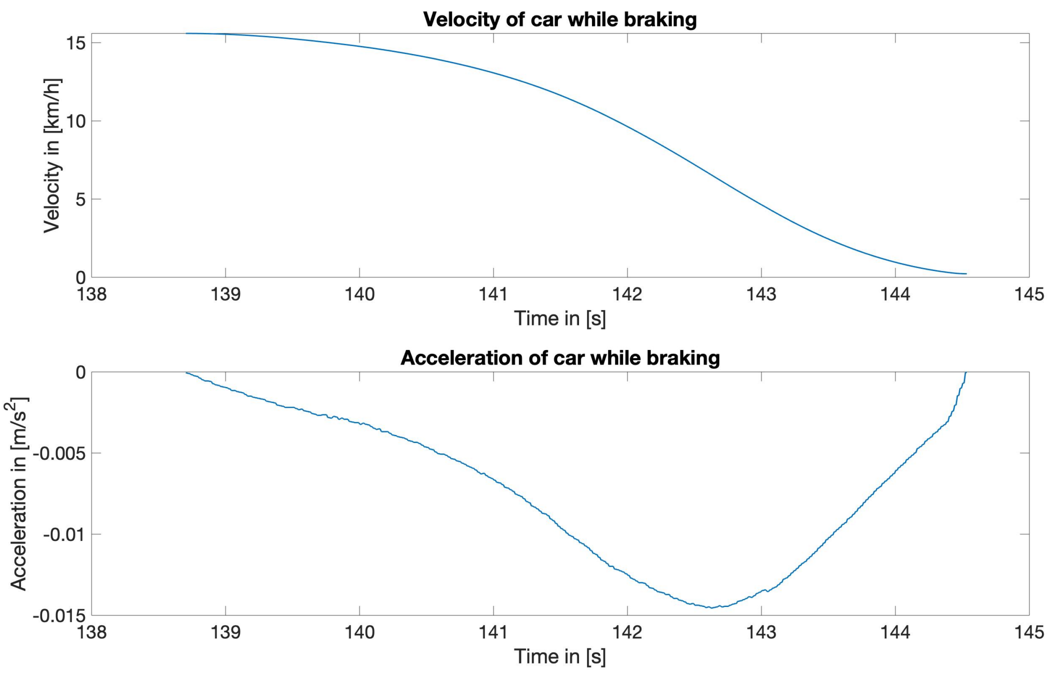


Figure 4.4: Human velocity profile extracted braking manoeuvre 1

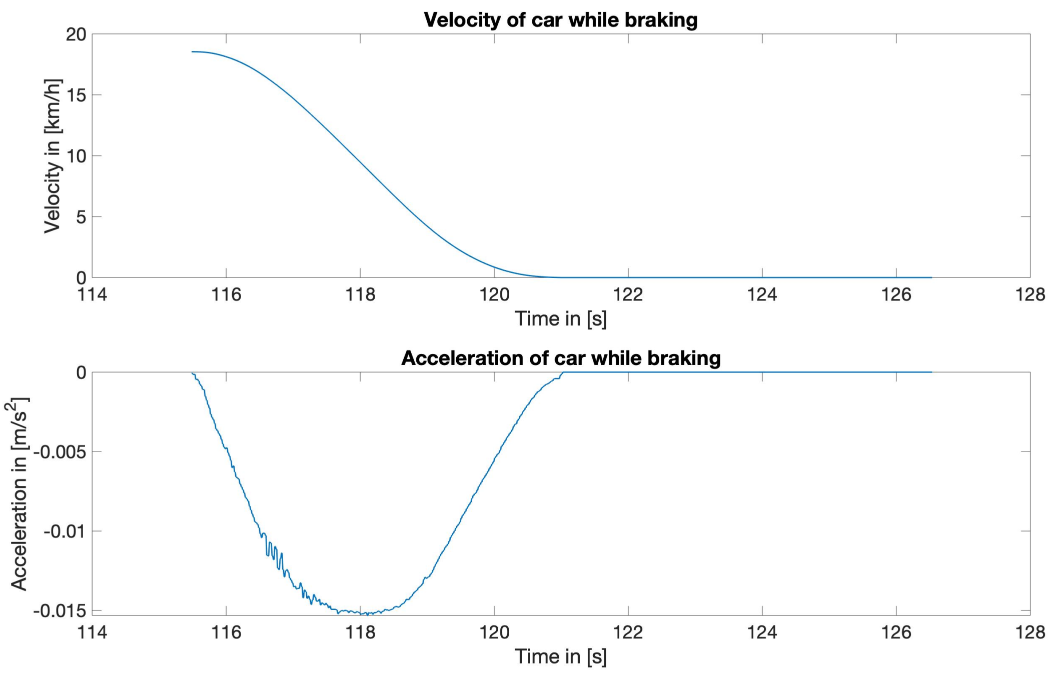


Figure 4.5: Human velocity profile extracted braking manoeuvre 2

4.5 Result

Considering Figures REF, REF and REF, it can be seen that the acceleration is approximately in the shape of a parable with the minimal turning point at XX. For further implementation of the ParkAssist, we will start the breaking process with an acceleration and increase it until we reach approximately REF and will then decrease it.

5 D4*: Consideration of uneven parking spaces

-diagram

-forces need to be considered (hangabtriebskraft)

-negative slope would increase stopping time and position

-positive slope would make stopping time shorter

-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. 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

-inaccuracy of velocity +- 0.1 km/h -also minimal velocity is 0.29 km/h which is not realistic, because velocity does not drop from 0.29 km/h to zero -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

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.

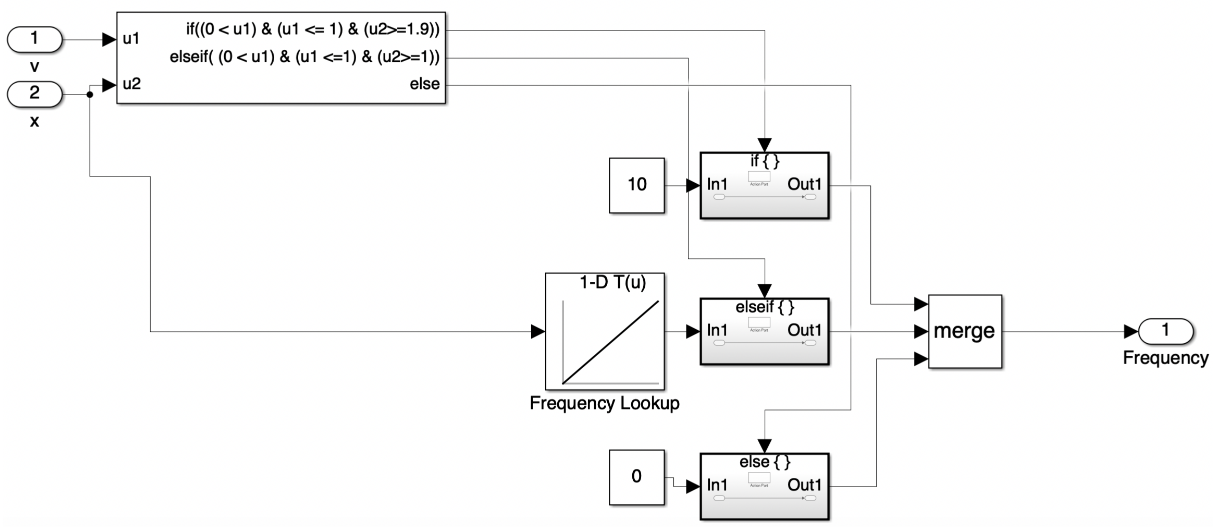


Figure 7.1: Simulink model of the frequency computation subsystem

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 that is not the case 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.

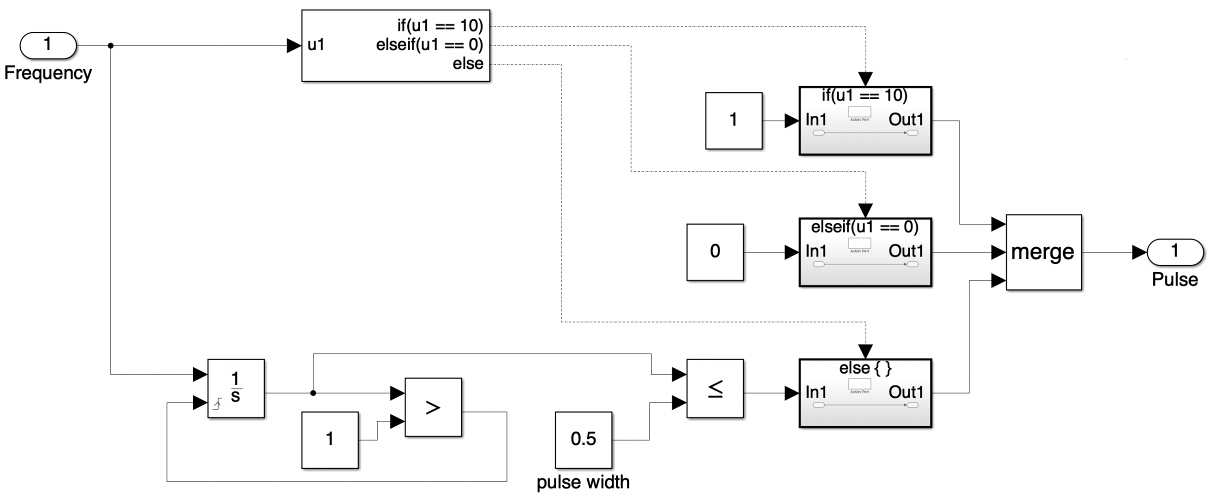


Figure 7.2: Simulink model of the pulse generation subsystem

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.

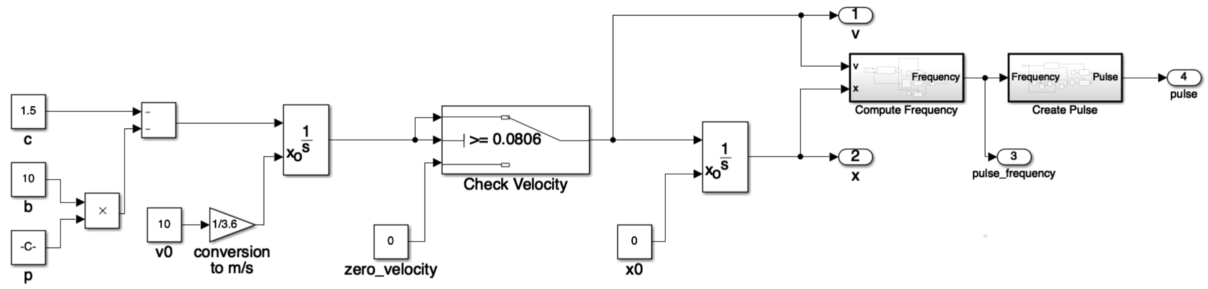


Figure 7.3: Simulink model of the car including the pulse signal

7.4 Signal demonstration

todo neu

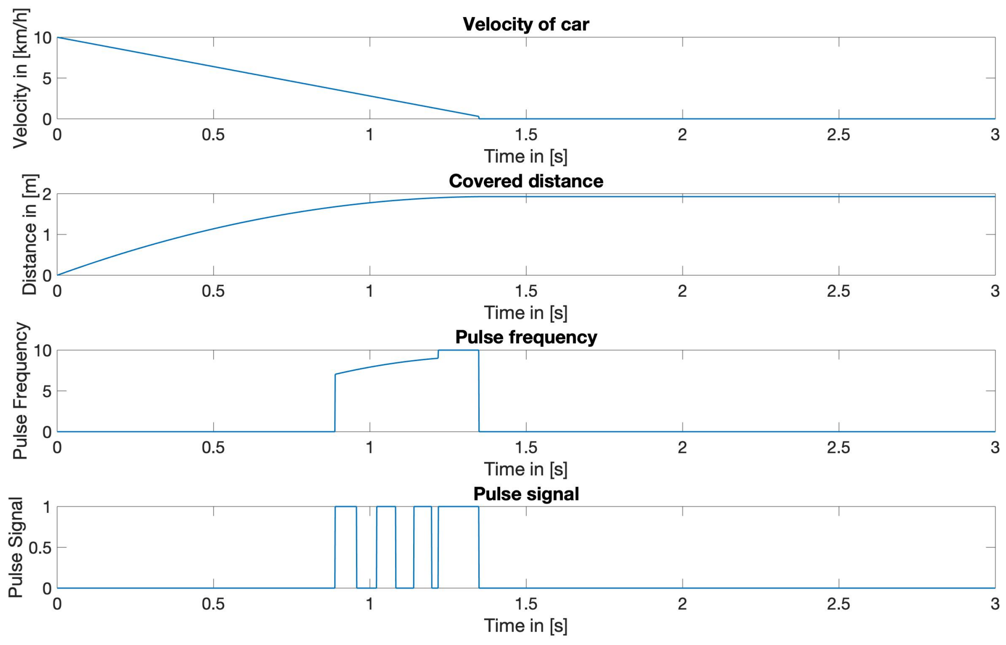


Figure 7.4: Demonstration of pulse signal

8 D7: Transfer of Simulink model to ASCET

- integration block as combination of multiplication and addition

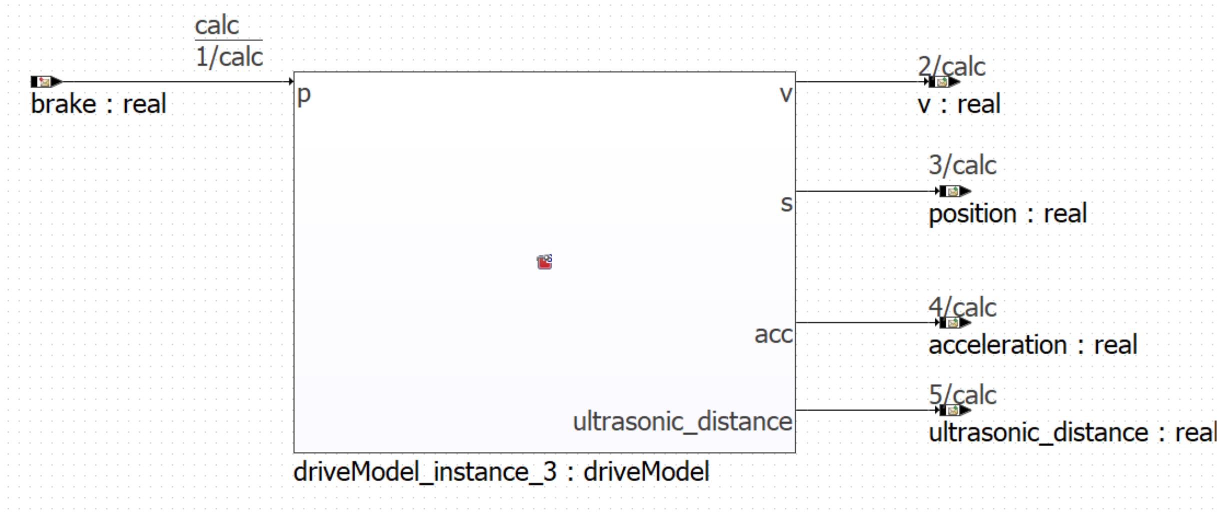


Figure 8.1: ASCET Blockdiagramm of car

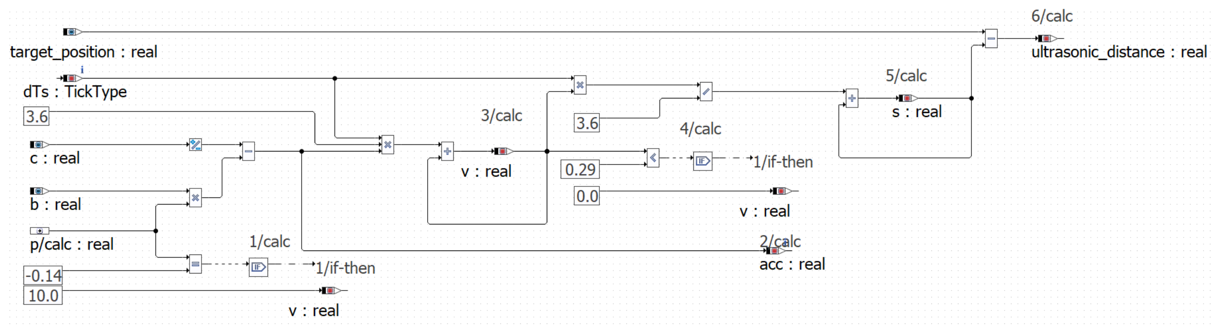


Figure 8.2: ASCET Blockdiagramm of driving model

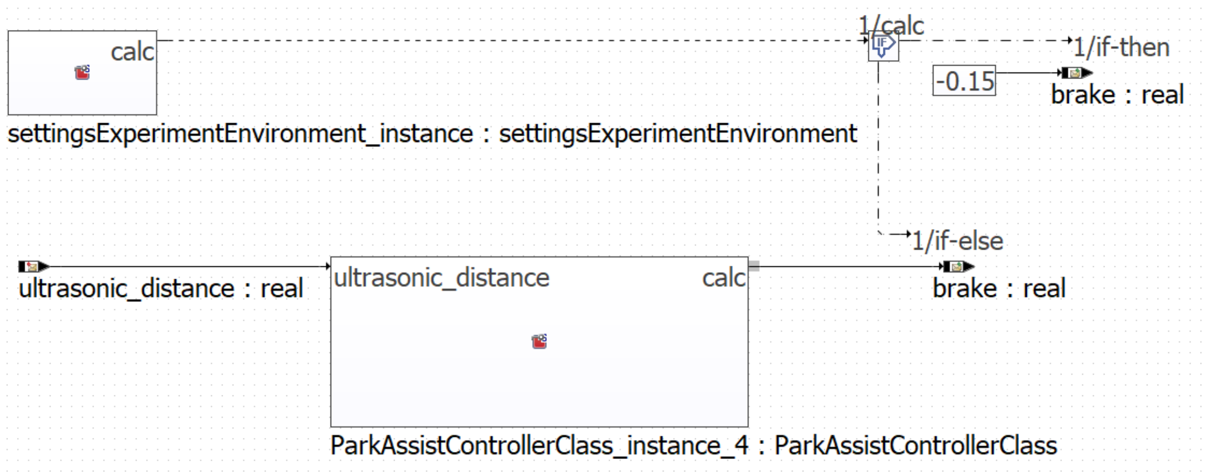

calc
settingsExperimentEnvironment_instance : settingsExperimentEnvironment
1/calc
1/if-then
-0.15
brake : real
1/if-else
ultrasonic_distance : real
ultrasonic_distance
calc
brake : real
ParkAssistControllerClass_instance_4 : ParkAssistControllerClass

Figure 8.3: ASCET Blockdiagramm of park assist controller

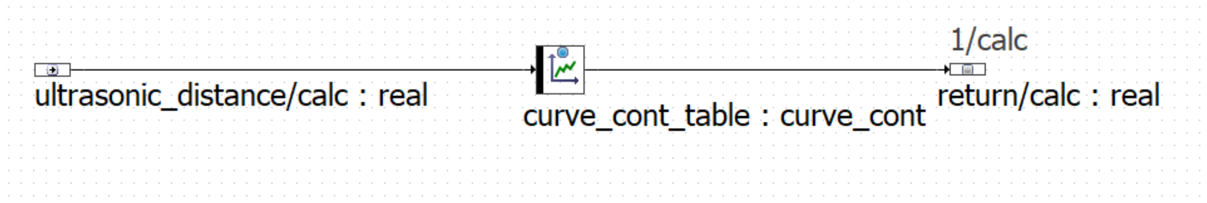

ultrasonic_distance/calc : real
curve_cont_table : curve_cont
1/calc
return/calc : real

Figure 8.4: ASCET Blockdiagramm of park assist controller class

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.

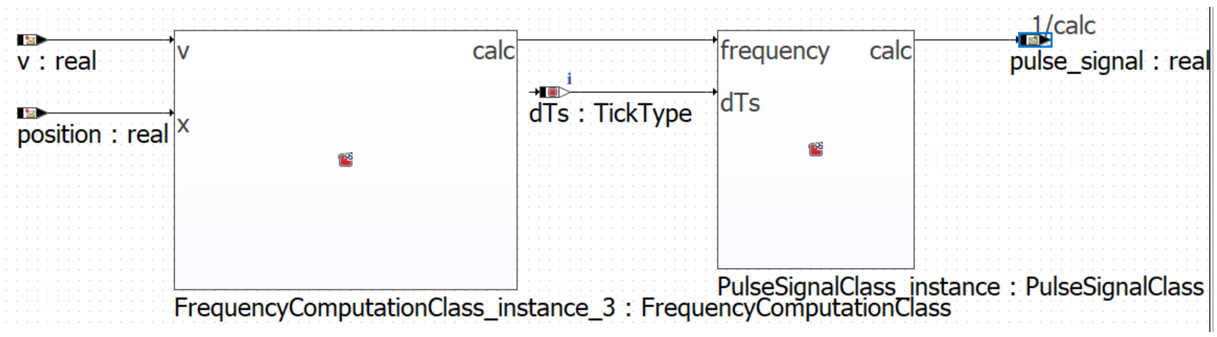

v : real
position : real
v
x
calc
FrequencyComputationClass_instance_3 : FrequencyComputationClass
i
dTs : TickType
frequency
dTs
calc
PulseSignalClass_instance : PulseSignalClass
1/calc
pulse_signal : real

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

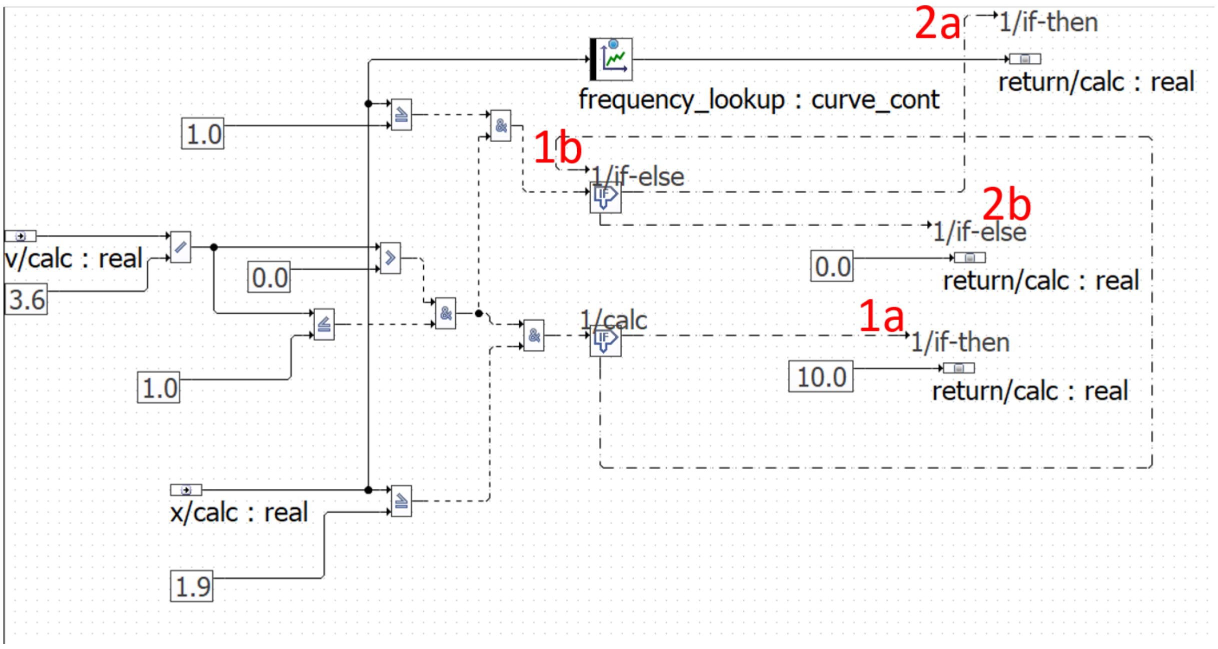


Figure 9.2: ASCET block diagram of class *FrequencyComputationClass*

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.

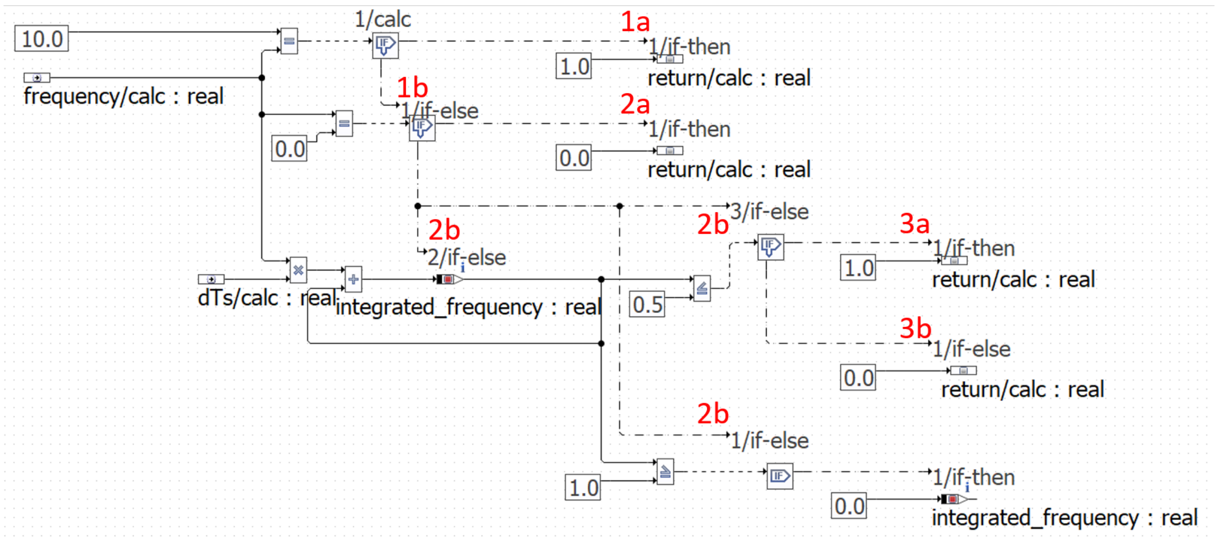
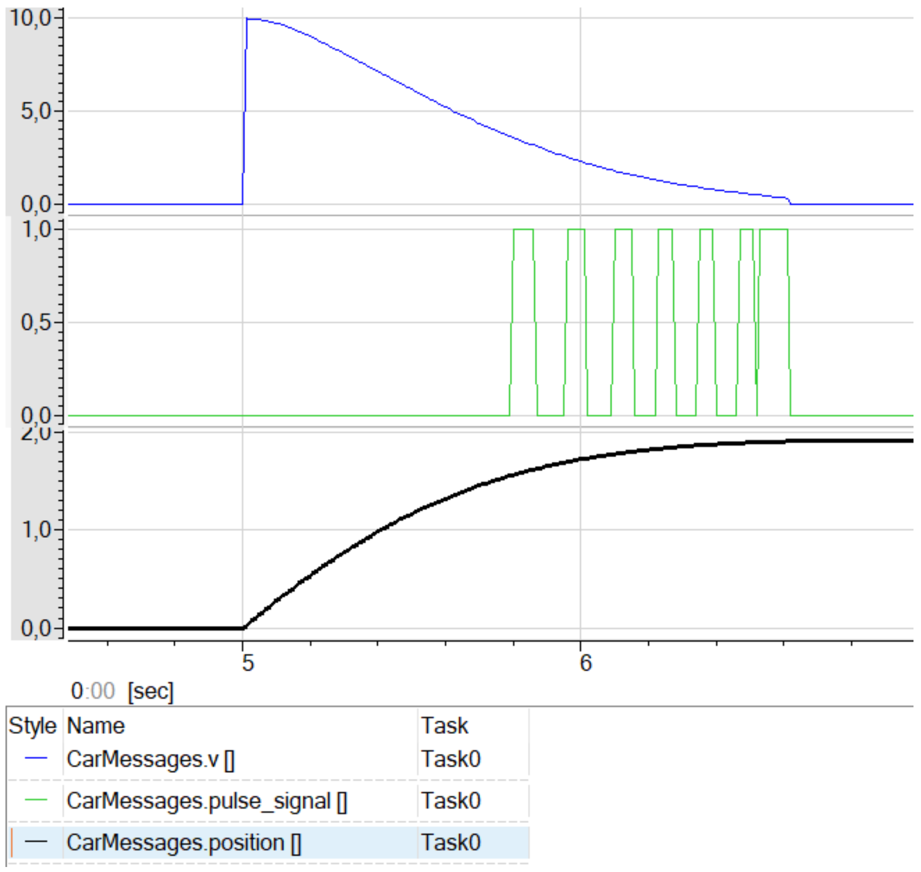


Figure 9.3: ASCET block diagram of class *PulseSignalClass*

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.v []	Task0
—	CarMessages.pulse_signal []	Task0
—	CarMessages.position []	Task0

Figure 9.4: ASCET 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 were 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 include. 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.

- pulse signal, drive model not possible to unit test because of time component - Messages erklären

10.1 Unit tests for ParkAssistControl

10.2 Unit tests for pulse signal frequency generation

Table 10.1: Equivalence classes for unit testing pulse signal frequency generation

Position	Velocity	Pulse Signal
0, 1, 1.5, 1.9, 2	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

Table 10.2: 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

10.3 Unit tests for pulse signal

Table 10.3: Equivalence classes for unit testing pulse signal

Position	**Velocity**	**Pulse Signal**
0, 1, 1.5, 1.9, 2	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

-test cases

Table 10.4: Pulse signal test cases

Frequency	**dts**	**Expected result**
0	1	0
10	1	1

- Alternating pulse signal not possible to test because dependant of time

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

- wait 5 seconds until starting because ascet experiment environment does not display the first seconds
- in first 5 seconds brake = -0.15 to counter c and b, v = 0
- after five seconds set v to 10 km/h and start experiment

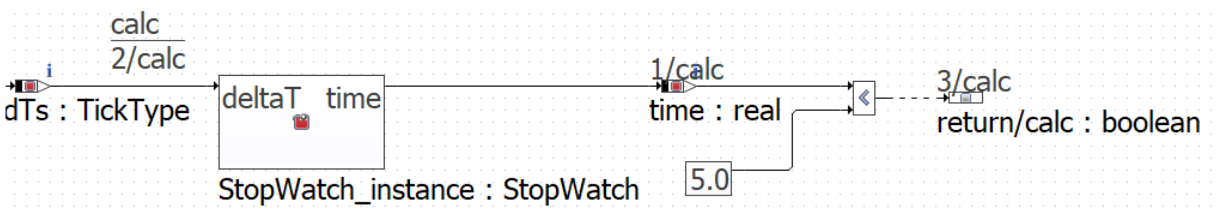


Figure 11.1: ASCET Blockdiagramm of settings

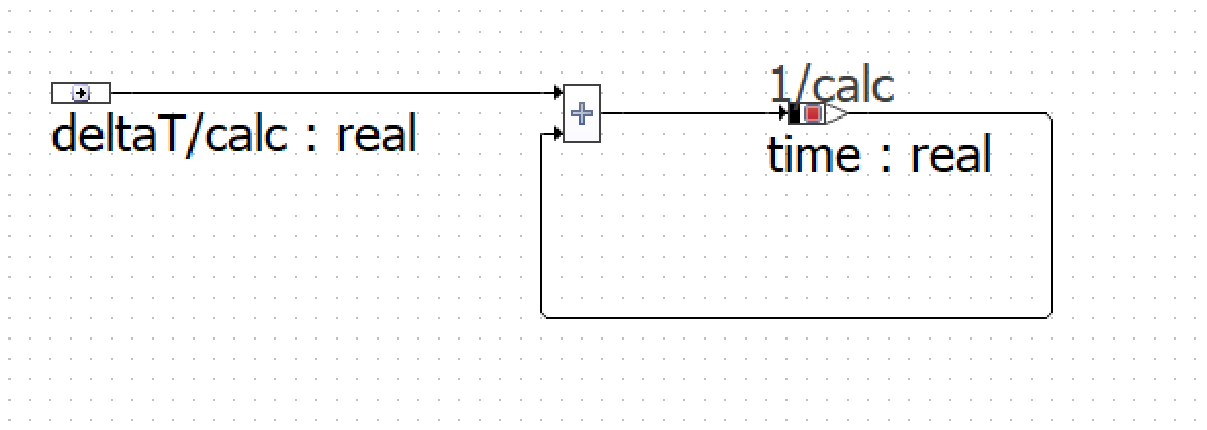


Figure 11.2: ASCET Blockdiagramm of stopwatch

12 D11*: Plausibility check comparing measured velocities and distances

-statistically independent ultrasonic distance and velocity -if distance to next object for ultrasonic sensor is known

-al

13 D13*: Impact of inaccuracies

-ultrasonic measurement: what if object is round? -

14 D14*: Reflection

-only 2 meter stop considered -model not realistic