Chapter 7

Traction controller

In this Chapter a traction controller is developed for preventing slip while accelerating and decelerating. Firstly in section 7.1 "Basic idea" the basic idea of the traction controller will be explained in some more detail. Then in section 7.2 "Flowcharts" two flowcharts are created to show how this traction controller really works. In section 7.3 "Implementation" the implementation in the trajectory planner is shown and in section 7.4 "Simulations traction controller" some simulations are done to show what the traction controller does to the signals in the trajectory planner.

7.1 Basic idea

Like mentioned in previous Chapter is the traction controller developed in this report based on controlling the a_{max}. This control happens after the "trajectory preprocessor" in Figure 3.1 in Chapter 3. This because here a_{max} and v_{max} already get changed for dribbling. The a_{max} and v_{max} get used in the trajectory planner, so the traction controller must have been implemented before or in the trajectory planner. Since the signal has some strange behavior like shown in Chapter 4 "Trajectory Planner" and Chapter 5 "Global Signal Analysis" it would be helpful to use the cases in the traction controller. For that reason the traction controller is implemented ín the trajectory planner.

The a_{max} gets controlled by multiplying it with a so called safety factor. This safety factor is initial equal to 1 and never bigger than 1 or smaller than 0.1. When the Turtle detects slip it changes the safety factor to a value below 1 so the a_{max} is lower. By a feedback loop for the safety factor and the difference in previous safety factor, the ideal safety factor will be evaluated. Since slip behavior is very unstable and it is not desired to drive slowly unnecessary, the safety factor will be increased when slip is not detected anymore.

7.2 Flowcharts

In this section two flowcharts are developed to show how the traction controller works inside the trajectory planner. In the first subsection 7.2.1 "??" the implementation needed for the traction controller is explained. In subsection 7.2.2 "Traction_Controller" the subroutine traction_controller is discussed in detail.

7.2.1 Useful code in trajectory planner

In Figure 7.1 a flowchart is illustrated with the part of the trajectory planner where the traction controller is implemented in. In this flowchart no directions are shown because it holds for both, x- and y direction.

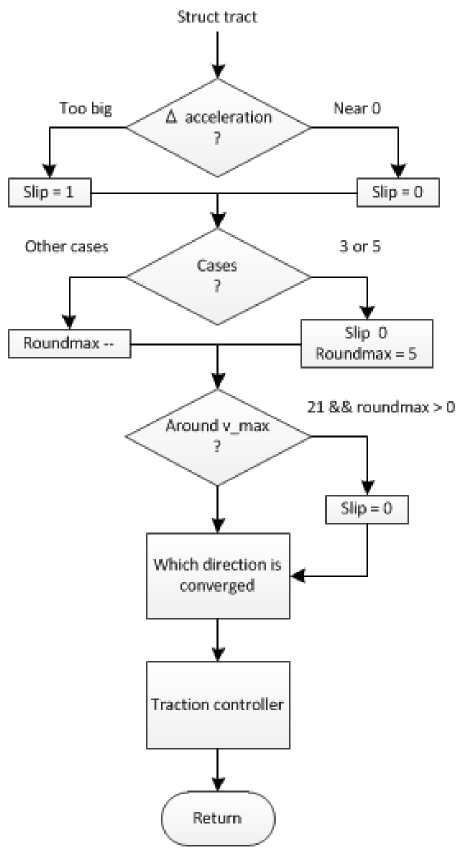

Figure 7.1: Flowchart of a piece of the trajectory planner with the traction controller

At first the Turtle checks if it detects slip. When the difference between the measured acceleration and the input acceleration is too big the variable *slip* is set to 1. The value used as threshold for the difference between the two accelerations should be based on such a diagram as Figure 1.2 in Chapter 1 "Traction and slip". For the accelerations also exists a stable area. This stable are should be found for the slip based on difference in acceleration.

Shown in section 5.2 "Results Turtle- movement" is that while the Turtle is driving around v_{max} there is some strange behavior in the signal input. This only occurs when the cases are 3 or 5. So when the cases of the previous time step (when the Turtle slipped) are 3 or 5, slip is ignored and the corresponding variable is set back to 0. Because sometimes when cases are 3 or 5 alternately it can happen that case 21 occurs. For this reason roundmax is set to 5 when case 3 or 5 is detected. When in a time step the case is not equal to 3 or 5 the roundmax is decreased. Now in short, when case 21 occurs and the Turtle was the previous 5 time steps never in case 3 or 5 the Turtle is not around v_{max} and slip cannot be ignored.

After that in the trajectory planner it is checked if the x- or y direction is converged or not. Depending on this the accelerations are calculated. Always before these get calculated the subroutine traction_controller gets called.

7.2.2 Traction_Controller

In Figure 7.2 a Flowchart is illustrated of the subroutine traction_controller. This is the block in the flowchart of the previous section 7.2.1 "Useful code in trajectory planner".

In the flowchart the safety factor is shorted with Sf and the difference in safety factor of previous time step with Δsf_last.

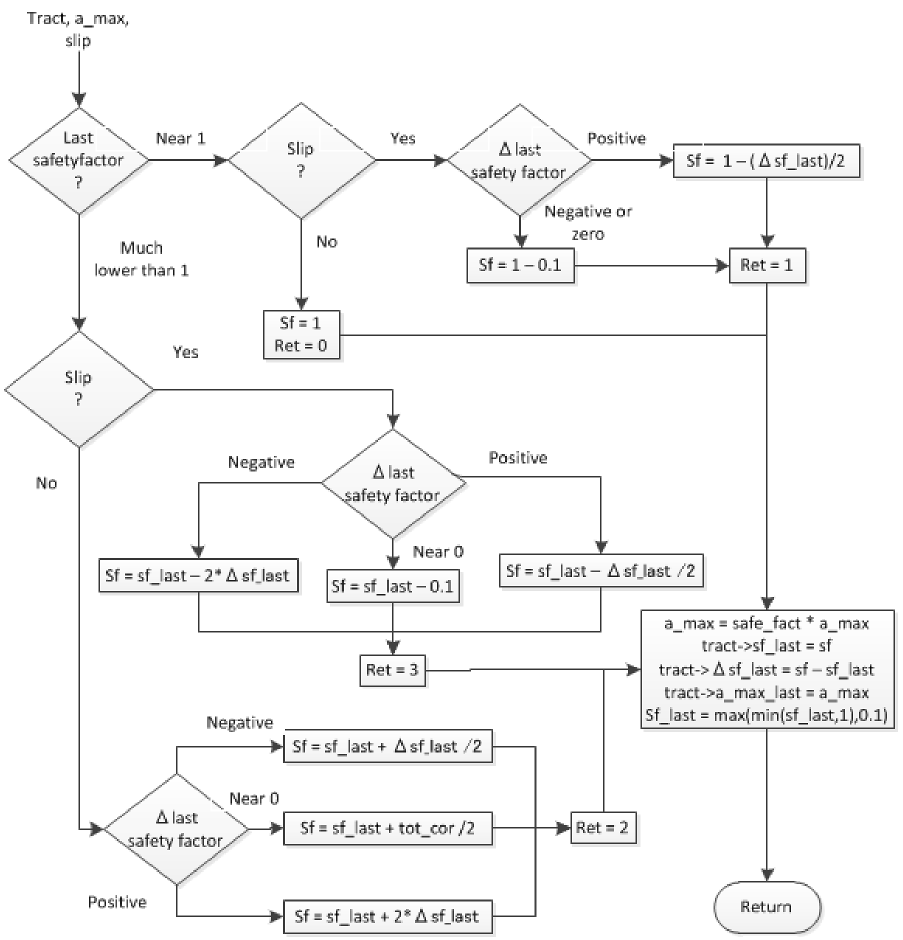

Figure 7.2: Flowchart of the actual traction controller

Before the flowchart will be explained, first a numeration of inputs and variables used in the traction controller:

- a_max_last; stored in a struct called tract. This variable contains the value of the a_{max} from the previous time step. For the traction controller this value will not be used. Because the safety factor is controlled the a_{max} does not have to be controlled.

- sf_last; also stored in tract and contains the safety factor of the previous time step.
- Δsf_last; stored in tract and contains the difference in safety factor in previous time stap.
- cases x respectively y; these contain the cases for respectively x- or y direction visited in subroutine Check Case (section 4.3 "Check Case" in Chapter 4 "Trajectory Planner" in the previous time step.
- overshootflag x respectively y; these contain whether an overshootflag is set for x respectively y or not. This value will also not be used in the traction controller. One could think that this should be considere for the reason that an overshootflag can cause a magnification of the accelerations. Well the decrease of the a_{max} while slipping also is magnified, so indirectly it ís considered.
- roundmax x respectively y; these contain if the Turtle was driving at a velocity near v_{max}.

Firstly the safety factor of the previous time step is evaluated. To keep it clear this is subdivided in two subsections 7.2.2 "Previous safetyfactor much lower than one" (follow arrow down "Much lower than 1" in the flowchart) and 7.2.2 "Previous safetyfactor near one" (follow arrow to the right "Near 1").

Previous safetyfactor much lower than one

If this safety factor is lower that the threshold it means that in the previous time step the a_{max} is controlled. After that the Turtle evaluates its slip variable.

1. If the Turtle slips in this time step the traction controller will decrease the safety factor. When the Turtle slips (follow the right arrow in the flowchart "Yes") the difference in safety factor can be:
    - **positive** (follow arrow down "Positive"). The last time slip was not detected so the safety factor was increased. Because the Turtle is slipping now the positive correction was too high assuming the ideal safety factor is constant. This is corrected by taking the previous time step again, but then multiplied with 0.5 and subtracted from the last safety factor. In this way the safety factor is converging to an ideal value.
    - **negative** (follow left arrow "Negative"). In the previous time step was slip detected and now again. This means that the correction of the previous step was not high enough. The same correction is used as the previous time but then multiplied with 2 and subtracted from previous safety factor. Note that the difference in safety factor is an absolute value.
    - **near 0** (follow right arrow "Near 0"). In the previous time step the difference in safety factor was very very small. In this case when slip is detected nothing will happen so when the difference in last safety factor is really too small a normal correction will be introduced.

2. If the Turtle does not slip the safety factor must be increased. When the Turtle does not slip (follow the arrow down in the flowchart "No") and the difference in safety factor can be:

    - **positive** (follow the middle arrow "Positive"). One time step ago also no slip was detected. This means that the correction back to 1 was not high enough. For this reason the same correction is used as the previous time but then multiplied with 2 and added to the last safety factor.
    - **negative** (follow the upper arrow "Negative"). The correction of the safety factor in the previous time step was negative. This means that the correction was too high. In trying to let the safety factor converge, the difference in safety factor of the previous time is divided by 2 and added to the safety factor of previous time. This in such a way that the safety factor converges to the ideal safety factor.
    - **near 0** (follow the arrow down "Near 0". If the previous difference in safety factor was 0 and no slip is detected the total correction is divided by 2 and added to the previous safety factor to make sure the maximal acceleration does not stay too low.

Previous safetyfactor near one

If the safety factor of the last time step was near 1 it means that there can be two possibilities. The Turtle is now

1. slipping (follow the arrow to the richt "Yes". The difference in the last safety factor can be:

    - **positive** (follow arrow to the right "Positive"). The last time step no slip was detected and the correction of the safety factor was too big. Now to let the safety factor converge the safety factor is now reduced with the difference of the previous safety factor divided by 2.
    - **negative or zero** (follow arrow down "Negative or zero"). In this case the traction controller will change the safety factor with a standard value.

2. not slipping (follow the arrow down "No"). Now there is no slip and previous time step there was no or negligible slip. Set the safety factor to 1.

After all this cases finally the values should be stored and send. In the new a_{max} the safety factor influences its value, the difference in safety factor gets calculated and the safety factor is minimized to 1 or maximized to 0.1 so no strange values will occur for a_{max}.

7.3 Implementation

In this section the actual implementation in the C - file: "trajectory planner" is presented. Sometimes pieces of the original trajectory planner are printed too but in the comments can be seen which pieces that are. Some added pieces especially for the traction controller are not displayed here like the code to implement extra ports for in- and output variables or lines that includes header files. In every case one line is added to define the case, this is also not added. The cases are explained in section 4.3 "Check Case".

This section is divide in several subsections. In the first subsection 7.3.1 "Struct traction" the struct called traction used in the trajectory planner is defined. In the second subsection 7.3.2 "Detecting slip" the code for detecting slip is defined. In subsection 7.3.3 "Converged directions" the code where the traction controller gets called is added and finally the subroutine traction_controller can be found in the fourth and last subsection 7.3.4 "Controlling slip".

7.3.1 Struct traction

Some variables used in the traction controller are stored in a struct. This struct is defined in this header file:

Listing 7.1: Struct traction

```
#ifndef structs_h
#define structs_h

struct traction {

    double a_max_last;  %a_max of previous time step
    double sf_last;     %safety factor of previous time step
    double dif_sf_last; %previous change in safety factor
    int casesx;         %case in x direction
    int casesy;         %case in y direction
    int overshootflagx; %overshootflag in x direction
    int overshootflagy; %overshootflag in y direction
    int roundmaxx;      %counter in x direction
    int roundmaxy;      %counter in y direction

};typedef struct traction tract;
#endif
```

7.3.2 Detecting slip

Here some calculations are done before subroutine traction_controller gets called. Firstly it tries to detect slip and ignores slip when the Turtle has nearly v_{max} as a velocity. Following the implementation for this:

Listing 7.2: Detecting slip

```
if ( dabs(a_measx - a_inx) >= EPSILON_SLIP ){// slip in x
    slip[0] = 1;
}
if ( dabs(a_measy - a_iny) >= EPSILON_SLIP ){// slip in y
    slip[1] = 1;
}
if( tract.casesx == 3 || tract.casesx == 5 ){//case 3 or 5
    slip[0] = 0;
    tract.roundmaxx = 5;
} else{
    tract.roundmaxx--;
}
if( tract.casesy == 3 || tract.casesy == 5 ){
    slip[1] = 0;
    tract.roundmaxy = 5;
} else{
    tract.roundmaxy--;
}
if( tract.casesx == 21 && tract.roundmaxx > 0 ){//v_max?
    slip[0] = 0;
}
if( tract.casesy == 21 && tract.roundmaxy > 0 ){
    slip[1] = 0;
}
```

7.3.3 Converged directions

In section 4.1 "Outputs" there are four possible ways illustrated where the Turtle can be in with. The following order is the same as in that section and the added lines are written in that part of the file.

Outputs trajectory planner

Every part has the same variables as output off course and this looks like:

Listing 7.3: Ouputs trajectory planner

```
y[0] = q_x;                    //from original file
y[1] = q_y;                    //from original file
t[0] = tract.sf_last;
t[1] = tract.dif_sf_last;
t[2] = tract.a_max_last;
c[0] = tract.casesx;
c[1] = tract.casesy;
c[2] = tract.overshootflagx;
c[3] = tract.overshootflagy;
c[4] = tract.roundmaxx;
c[5] = tract.roundmaxy;
```

x- and y direction converged

For the first possibility that both x- and y direction are converged. It is desired that traction control gets applied only once. Otherwise the a_{max} could get corrected twice. If x- and y direction are both converged there is no case or overshootflag determined so these are set to the default value of -1 respectively 0. Finally all values of the struct are outputted to be used as input for the next time step. Here is the implementation:

Listing 7.4: x- and y direction converged

```
controlled = traction_control(&a_max, slip[0],&tract);
q_x = dmin(dmax((xfdot−xodot)/SAMPLE_TIME,−a_max),a_max);//from original file
tract.casesx = −1;
tract.overshootflagx = 0;
if( !controlled ){
    controlled = traction_control(&a_max, slip[1],&tract);
}
q_y = dmin(dmax((yfdot−yodot)/SAMPLE_TIME,−a_max),a_max);//from original file
tract.casesy = −1;
tract.overshootflagy = 0;
```

Only x direction converged

In the below implementation only the x direction is converged. For this the same reasoning holds for the x direction as for the previous implementation. However for the y direction calculate _time gets called. this means that a case and an overshootflag will be defined.

Listing 7.5: Only x direction converged

```
controlled = traction_control(&a_max, slip[0],& tract);
q_x = dmin(dmax((xfdot-xodot)/SAMPLE_TIME,-a_max),a_max);//from original file
tract.casesx = -1;
tract.overshootflagx = -1;
if( !controlled ){
controlled = traction_control(&a_max, slip[1],& tract);
}
ETA_y = calculate_time(yf, yodot, yfdot, v_max, a_max, &q_y);//from original file
povershootflag[1]=overshootflag;//from original file

tract.overshootflagy = overshootflag;
```

Only y direction converged

For the following C- code hold the same reasoning as for previous implementation. However here it is vice versa for x- and y direction.

Listing 7.6: Only y direction converged

```
controlled = traction_control(&a_max, slip[0],& tract);
ETA_x = calculate_time(xf,xodot,xfdot,v_max,a_max,&q_x);\\from original file file
tract.overshootflagy = overshootflag;
if(controlled){
    overshootflag = 0;
}
povershootflag[0]=overshootflag; \\from original file
if ( !controlled ){
    controlled = traction_control(&a_max, slip[1],& tract);
}
q_y = dmin(dmax((yfdot-yodot)/SAMPLE_TIME,-a_max),a_max);
tract.casesy = -1;
tract.overshootflagy = 0;
```

No directions converged

For the fourth case the same idea holds. Following code is implemented before the loop where the ideal driving angle gets calculated (See section 4.1 "Outputs".

Listing 7.7: No direction converged

```
controlled = traction_control(&a_max, slip[0],& tract);
ETA_x = calculate_time(xf,xodot,xfdot,v_max_x,a_max_x,&q_x);//from original file
if(controlled){
    overshootflag = 0;
}
povershootflag[0]=overshootflag;//from original file
tract.overshootflagx = overshootflag;
if ( !controlled ){
        controlled = traction_control(&a_max, slip[1],& tract);
}
ETA_y = calculate_time(yf,yodot,yfdot,v_max_y,a_max_y,&q_y);//from original file
povershootflag[1]=overshootflag;//from original file

tract.overshootflagy = overshootflag;
```

7.3.4 Controlling slip

Finally the C- code of the traction controller traction_controller is given below. At first used variables are declared and the total correction is calculated. Secondly are the steps that are shown in the flowchart in previous section 7.2 "Flowcharts" exactly implemented.

Listing 7.8: Subroutine traction_control

```
int traction_control( double* a_max,int slip ,struct traction *tract){

double EPSILON_SF_LAST    = 0.005;//Value should be tuned
double EPSILON_dSF_LAST   = 0.05; //Value should be tuned
double EPSILON_STAND_VAL  = 0.1;  //Value should be tuned
double safe_fact          = 1;
double ret                = -1;

double sf_last            = tract->sf_last;
double dif_sf_last        = tract->dif_sf_last;
double a_max_last         = tract->a_max_last;

double tot_cor            = 1 - sf_last;

if ( sf_last > 1 - EPSILON_SF_LAST ){
    if ( slip ){                                // Decrease sf
        if ( dif_sf_last >= EPSILON_dSF_LAST ){ //Smaller +corr
            safe_fact = 1 - dabs(dif_sf_last/2);
        } else{                                 // Correction is nearly 0
            safe_fact = 1 - EPSILON_STAND_VAL;
        }
        ret = 1;
    } else{                                             // No correction , no slip
        safe_fact = 1;
        ret             = 0;
    }
} else{                                                 // There is a correction going on
    if ( slip ){                                        // Decrease sf
        if ( dif_sf_last >= EPSILON_dSF_LAST ){         //Smaller +corr
            safe_fact = sf_last - dabs(dif_sf_last/2);
        } else if( dif_sf_last <= -EPSILON_dSF_LAST ){  //Larger -corr
               safe_fact = sf_last - 2*dabs(dif_sf_last);
        } else {// Standard correction
            safe_fact = sf_last - EPSILON_STAND_VAL;
        }
        ret = 3;
    } else{// Increase sf
        if ( dif_sf_last >= EPSILON_dSF_LAST ){         //Larger +corr
            safe_fact = sf_last + 2*dabs(dif_sf_last);
        } else if( dif_sf_last <= -EPSILON_dSF_LAST ){  //Smaller -corr
            safe_fact = sf_last + dabs(dif_sf_last/2);
        } else{// Standard positive correction
            safe_fact = sf_last + dabs(tot_cor/2);
        }
        ret = 2;
    }
}
if ( safe_fact > 1 ){
    safe_fact = 1;
} else if{ safe_fact < 0.1 }
    safe_fact = 0.1;
}
*a_max              = safe_fact * (*a_max);
tract->sf_last          = safe_fact;
tract->dif_sf_last      = safe_fact - sf_last;
tract->a_max_last       = *a_max;

return ret;
}
```

7.4 Simulations traction controller

For the explained traction controller in previous sections the measurements done by an accelerometer are essential. It is not possible to test this traction controller on a Turtle because the accelerometer on the Turtles cannot be used properly. Because of this, only simulations on the computer are run for this traction controller.

In Figure E.1 in Appendix E the simulation setup is illustrated and explained.

With a signal builder an error is added to the simulation. Because the signal builder creates a signal that gets multiplied with the accelerationvalues its default value is 1. This is why in this simulation will be spoken of an error + 1. This error is applied in a referential signal, so whatever the traction controller does, the imposed error stays the same.

In the first two experiments just a movement in the x direction is modeled. In section 7.4.1 "Simulation with no error" a first simulation is performed with no error. In the next section 7.4.2 "Simulation with a small error" a small error is imposed. In section 7.4.3 "Simulation with an error and x- and y direction movement" an y movement is introduced to see what happens if two directions are not converged. Finally in section 7.4.4 "Simulation for slip at maximal velocity it is checked if the traction controller does not do anything while driving at maximal speed.

7.4.1 Simulation with no error

In this simulation no error is introduced to the system. Only one movement in x direction is simulated. This means that the accelerations, velocities and positions should be the same as in simulation 1 of Chapter 4 "Trajectory Planner. The safety factor should be equal to 1 and the variable roundmaxx should only have a value of 5 when the Turtle is in case 3 and drives on maximal speed.

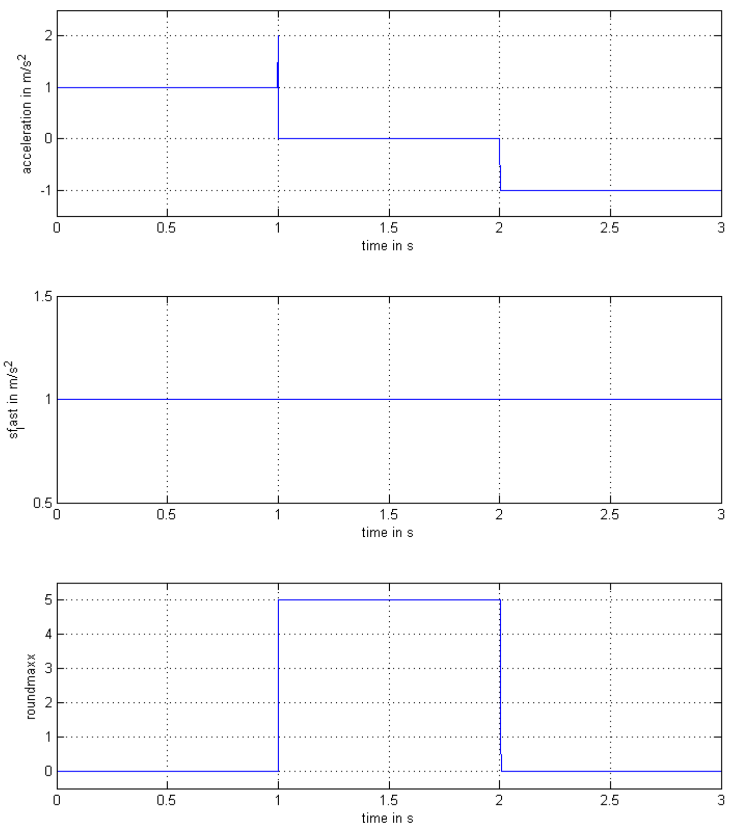

Figure 7.3: Acceleration in x direction, safety factor and roundmax in x direction

Results

The results are exactly the same as the expectations. In Figure 7.3 on the previous page the acceleration in x direction, the safety factor and the roundmax in x direction are shown.

As can be seen the same peek as in section 4.4.1 "Movement in x direction" is visible. Only when the acceleration is zero the roundmaxx is equal to 5. This shows that the traction controller does not do anything if no slip is detected.

7.4.2 Simulation with a small error

In this simulation a certain error is given. In reality this will not occur on this way, because whatever the traction controller does the error will not decrease. This means that the safety factor will get a minimal value defined by the traction controller, 0.1. Nevertheless with this simulation one can see what the traction controller does and how acceleration changes.

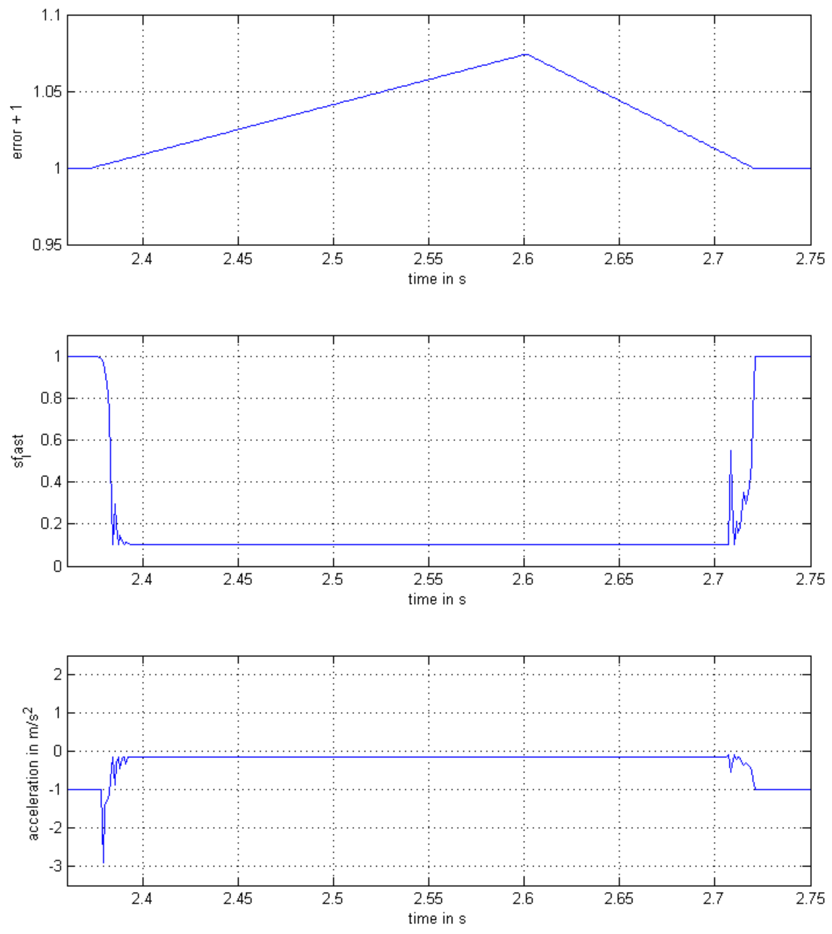

Figure 7.4: The piece where the error+1 is introduced, the safety factor in that same time frame and the acceleration value in x direction

Results

In the top figure of Figure 7.4 one can see what error is applied. These figures are zoomed in on a certain time frame where the error is applied. Outside that time frame the behavior of the signals is obviously exactly the same as in previous simulation.

It can be seen in the middle figure that when the error is bigger than a certain value the safety factor gets decreased. Because the error stays the same the Turtle constantly thinks it is slipping. When the safety factor reaches 0.1 it converges. When then the error becomes small again the safety factor gets increased again and becomes 1.

The maximal acceleration in the lowest figure follows this behavior since those two get multiplied. Right before the acceleration is decreased there is a peek in the signal. This peek is caused because of the overshootflag. Because the Turtle is decreasing its acceleration, but still has to drive some distance, the overshootflag is set. This error is not desired but it does not matter much.

Because the a_{max} is equal to 1, Equation 7.4.2 holds. This can be seen in Figure 7.5. The top figure shows the error+1.

$$a_{max} = safetyfactor \cdot a_{max} = safetyfactor \tag{7.1}$$

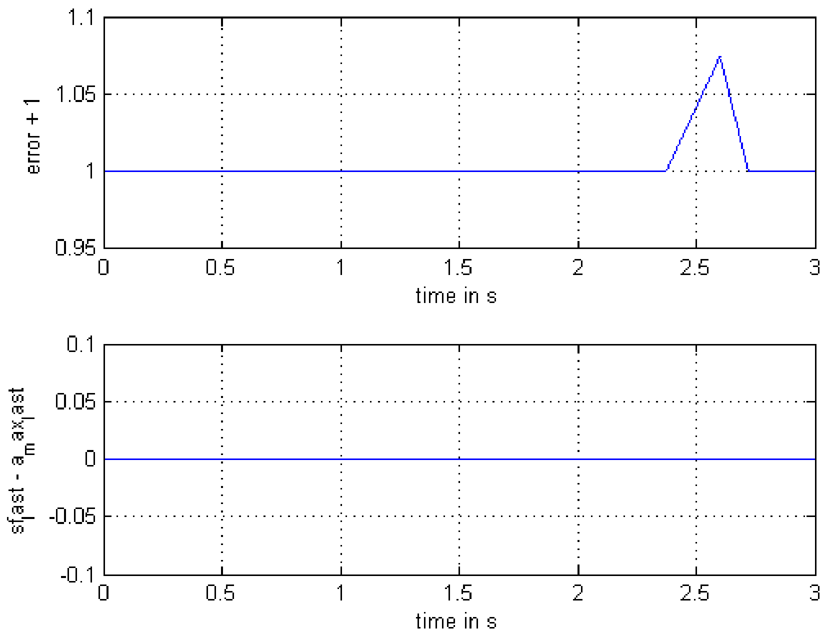

Figure 7.5: The error+1, the acceleration and the x- and y velocity

7.4.3 Simulation with an error and x- and y direction movement

This is a short experiment to show what the traction controller does with the speed of the Turtle. This simulation is somehow similar to that from section 4.4.2 "Movement in x direction and y direction" only here an error is introduced.

Results

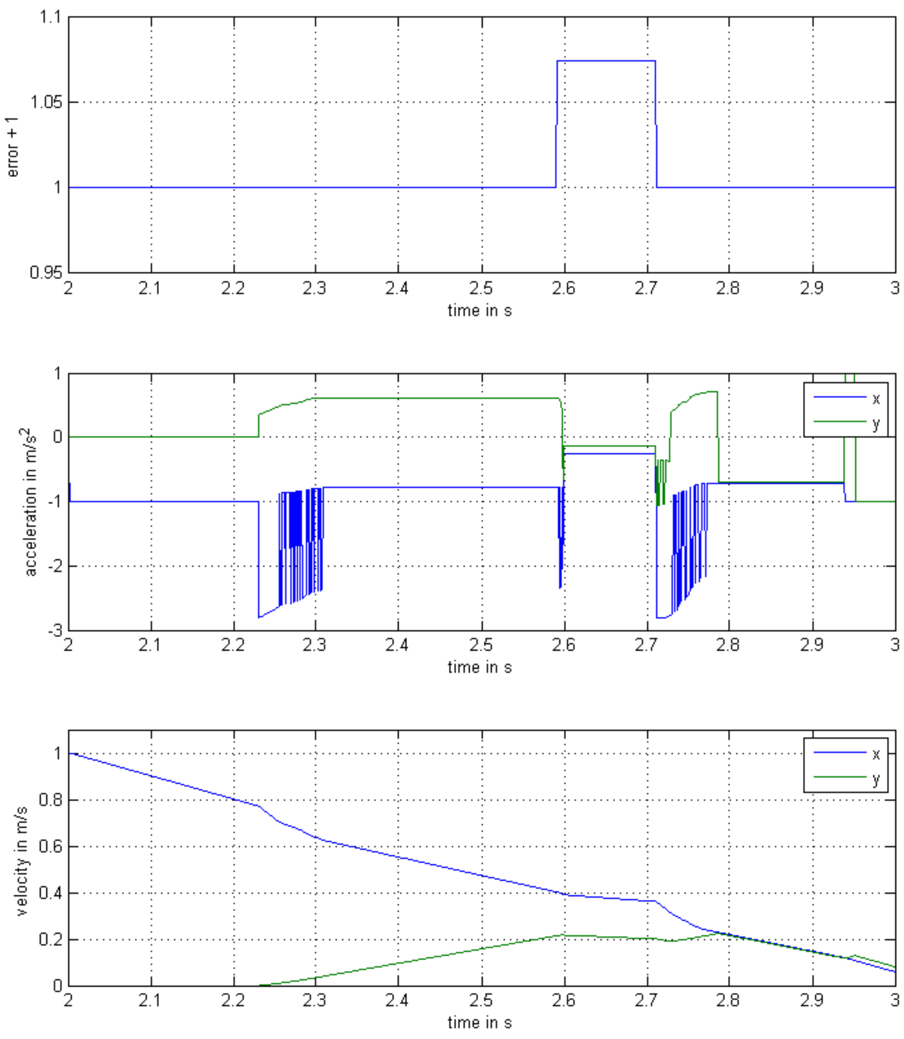

Figure 7.6: The error, acceleration in x- and y direction respectively velocity for movement in x- and y direction

In Figure 7.6 the error+1 imposed on the model is illustrated. In the second figure the acceleration is shown.

It is clear that the traction controller does its work nicely. The acceleration is purely constant around 0.1 and the direction with the overshootflag is 0.3. This is because the safety factor gets limited. It can be seen that from the moment no error is noticed an overshoot gets set. This does not really matter because in reality when the Turtle slips the a_{max} gets controlled. On that moment it does not matter whether an overshootflag is set or not.

On the moment the error + 1 is imposed, the velocity in the third figure has a much lower slope. This is very logic, because the acceleration is limited. The same a_{max} is used for both directions.

7.4.4 Simulation for slip at maximal velocity

In this section a small simulation is performed to show that slip at maximal velocity is ignored. In this case the v_{max} is set to 0.5 so the v_{max} is early reached.

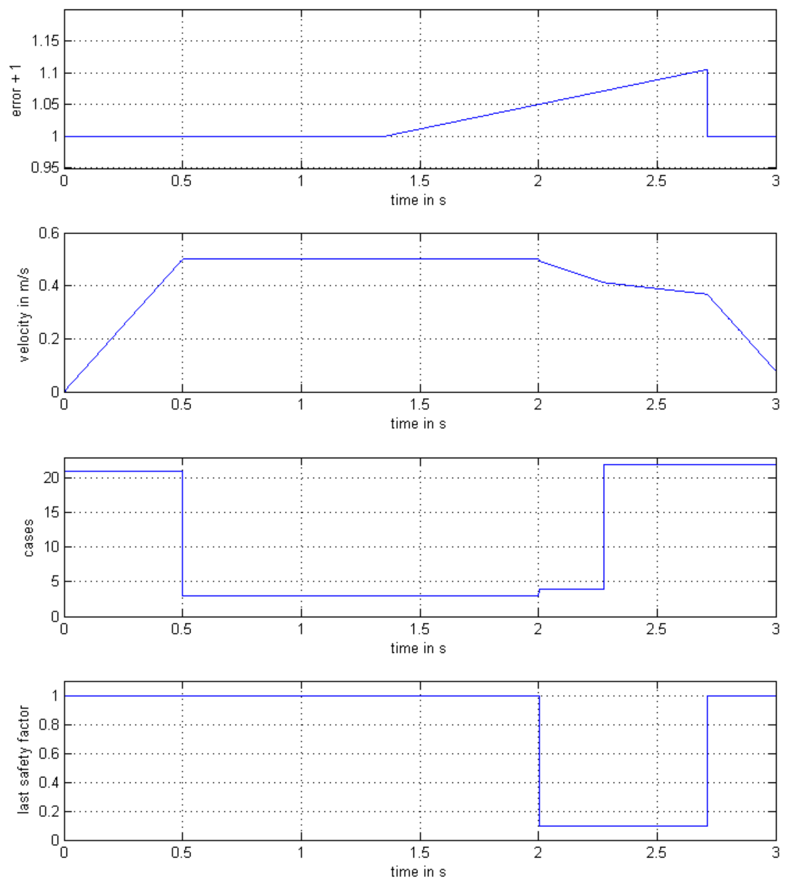

Figure 7.7: The error, velocity in x (driving) direction and the cases in the x direction

Results

In Figure 7.7 one could see that the error + 1 is imposed right before $time = 1.5sec$. In the lowest figure it can be seen that the case is 3 in the time frame [0.5 - 2]. While the error + 1 is imposed the case is 3 and the velocity is 0.5. Right when the Turtle is not in case 3 anymore the the a_{max} is controlled. This can be seen in the safety factor, drawn in the lowest figure.

7.5 Improvements of the traction controller

Because it was not possible to test the traction controller on the Turtles, it was also not possible to tune the parameters and build more features into it. Here some features that are not included in the tractioncontroller explained in this Chapter.

When in the struct tract a counter is included, it would be possible to make sure the safety factor does not variate too much. The idea of this counter is that it waits with the positive corrections and keeps the safety factor for some time steps on a constant value.

The traction controller does not react on an overshootflag that gets set. Even though an overshootflag amplifies the acceleration and could cause slip. It is true that when the safety factor changes a value, and that value gets amplified, the difference also gets amplified. So indirectly the overshootflag is taken into account. However this is probably not reliable. The overshootflag gets set while the traction controller is doing iets work. This should not matter because then the a_{max} is just made 3 times smaller, until the Turtle is not slipping anymore. When one could test the traction controller on the Turtle, one could investigate is this really does not matter.

Concluding Remarks

Slip can be prevented by driving very slowly. For a Turtle this would be a very bad strategy, because it wants to move over the soccer field as quick as possible. For that reason a fast adaptive traction control is desired for the Turtles.

Since the wheels of a Turtle are not in a rectilinear configuration it is not possible to use the same control systems as used in a car. For that reason is in this report chosen for another approach, namely comparing the intended acceleration with the actual acceleration measured by an accelerometer. For this it is important to clearly understand on which principles these intended accelerations are based on. So, this report contains an extensive research into the motion software of the Turtle, mostly in the trajectory planner. During this research some unwanted abnormalities are noticed and summarized in the next Chapter.

Basically the approach used in this report changes the maximal tolerable acceleration. The developed traction controller only works for accelerating and decelerating. A traction controller that also works when a Turtle is pushing against an opponent is shortly explained in the next Chapter, but this are only small basics.

The most trivial part of developing a controller was not possible in this report. It is very unfortnate that the accelerometer in the Turtle does not work properly because now the created traction controller is not validated, tuned and optimized. However in the simulations done on the computer one could see that the traction controller works like explained in the theory.

Future Work

The in this report developed traction controller is not tested yet on the Turtles. For this, one needs the accelerometer, or can use an observer to differentiate the encoders twice. When the traction controller is tested the parameters can be tuned. Parameters like thresholds and standard changes in safety factor.

One undesired property of the developed traction controller that is known for now is that the safety factor never reaches a constant value. This because when there is no slip anymore the safety factor should be 1 again. One could build in a counter that waits with positive corrections. This is not implemented in the in this report developed traction controller because it could not be tested properly.

As important thing in the development to a traction controller the accelerometer must be implemented. The accelerometer does not find itself in the middle of a Turtle, but somewhere on the side in a different coordinate system. This means the values obtained by the accelerometer are nonsense and must be transformed to the coordinate system of the Turtle. The rotating can be done by using a cosine direction matrix.[2] Although position is not important while discussing acceleration also the translation of the coordinate system is important. When the Turtle is only rotating in ϕ direction and the accelerometer is mounted on the side it will constantly detect an acceleration in a certain direction and this is not wanted.

While reading Chapter 4 "Trajectory Planner" one would have noticed some undesirable abnormalities are created by the trajectory planner. Following some examples.

- *In section 4.4.1 "Movement in x direction" a simple simulation is performed with the trajectory planner. The calculated acceleration can exceed the maximal acceleration which should not be possible. In the velocity and position the peek in the acceleration does not matter but nevertheless it is an irregularity in the signal.*
- *The behavior when case 3,5 or 21 (See section 4.3 "Check Case") gets visited, section 5.2 "Results Turtle- movement". Probably this behavior is caused because the threshold for the maximal velocity is too small. In the same section some vague peeks can be seen but then in another time frame, these are caused by a deviation in another direction and a alternate case (21 and 4)*

The traction controller described in Chapter 7 "Traction controller" is based on accelerating and decelerating of a robot on a path. However for the Turtles it is sometimes important to push against another robot. When a Turtle has a maximal acceleration without slip the pushing force against an opponent is maximal. The traction controller in previous section would lower the a_{max} until this is zero. This is not wanted and for that reason another idea should be performed. Following an example of such an idea.

- *Like mentioned in section 6.2 "Initial Ideas" the Turtle cannot compare its wheel speeds. However because in the motion scheme the signal out of the encoders gets decoupled from wheels to local coordinates the displacement of a Turtle is known in x, y and ϕ. With an observer can one trace the accelerations in x, y and ϕ direction the wheels think the Turtle drove. With the accelerometer one would be able to detect slip and correct the desired wheel speeds individually.*

Bibliography

[1] Rem- en slipgedrag.pdf, E. Gernaat (ISBN 987-90-808907-7-0) 02-09-2011

[2] Multibody Dynamics (4J400), N. van de Wouw; Technical University of Eindhoven (Department of Mechanical Engeneering) 2010

Appendix A

Decoupling local- to wheel coordinates

w_i is the wheel speed of the i^{th} wheel and V_r is the velocity vector of the robot. First the wheel speeds are defined like:

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} -|V_r| \cdot \sin(30+\phi) + R_1 \cdot \dot{\phi} \\ |V_r| \cdot \cos(\phi) + R_2 \cdot \dot{\phi} \\ |V_r| \cdot \sin(-30+\phi) + R_3 \cdot \dot{\phi} \end{bmatrix}$$

Then some conversion equations:

$$-\sin(30+\phi) = -\sin(30)\cdot\cos(\phi) - \cos(30)\sin(\phi) = -\frac{1}{2}\cos(\phi) - \frac{\sqrt{3}}{2}\sin)(\phi)$$

$$\sin(-30+\phi) = -\sin(30)\cdot\cos(\phi) + \cos(30)\sin(\phi) = -\frac{1}{2}\cos(\phi) + \frac{\sqrt{3}}{2}\sin)(\phi)$$

$$V_x = |V_r| \cdot \cos\phi$$

$$V_y = |V_r| \cdot \sin\phi$$

So the wheel speeds can be written as:

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} -\frac{1}{2} & -\frac{\sqrt{3}}{2} & R_1 \\ 1 & 0 & R_2 \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} & R_3 \end{bmatrix} \cdot \begin{bmatrix} V_x \\ V_y \\ \dot{phi} \end{bmatrix}$$

And the speeds in local robot coordinates become:

$$\begin{bmatrix} V_x \\ V_y \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} -\frac{1}{2} & -\frac{\sqrt{3}}{2} & R_1 \\ 1 & 0 & R_2 \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} & R_3 \end{bmatrix}^{-1} \cdot \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

$$\begin{bmatrix} V_x \\ V_y \\ \dot{\phi} \end{bmatrix} = \frac{1}{R_1+R_2+R_3} \cdot \begin{bmatrix} -R_2 & R_3+R_2 & -R_2 \\ -\frac{1}{\sqrt{3}}(2R_3+R_2) & \frac{1}{\sqrt{3}}(-R_3+R_1) & \frac{1}{\sqrt{3}}(R_2+2R_1) \\ 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

Appendix B

Simulations

In the simulation the Turtle follows the given signal perfectly. The idea is to let the Turtle drive 2 meters purely in x direction with a final velocity of zero. The set up is shown in Figure B.1 and the inputs for the trajectory planner in this simulation are in following order:

- desired movement in x direction
- desired final velocity in x direction
- desired movement in y direction
- desired final velocity in y direction
- instantaneous movement in x direction
- instantaneous movement in y direction
- instantaneous velocity in x direction
- instantaneous velocity in y direction
- maximal velocity
- maximal acceleration
- robotID; 2 for a normal Turtle.

The turn flag is set to zero. Outside this simulation the turn flag is set to one when the Turtle is dribbling. The control enable is set to one to enable the parking brake.

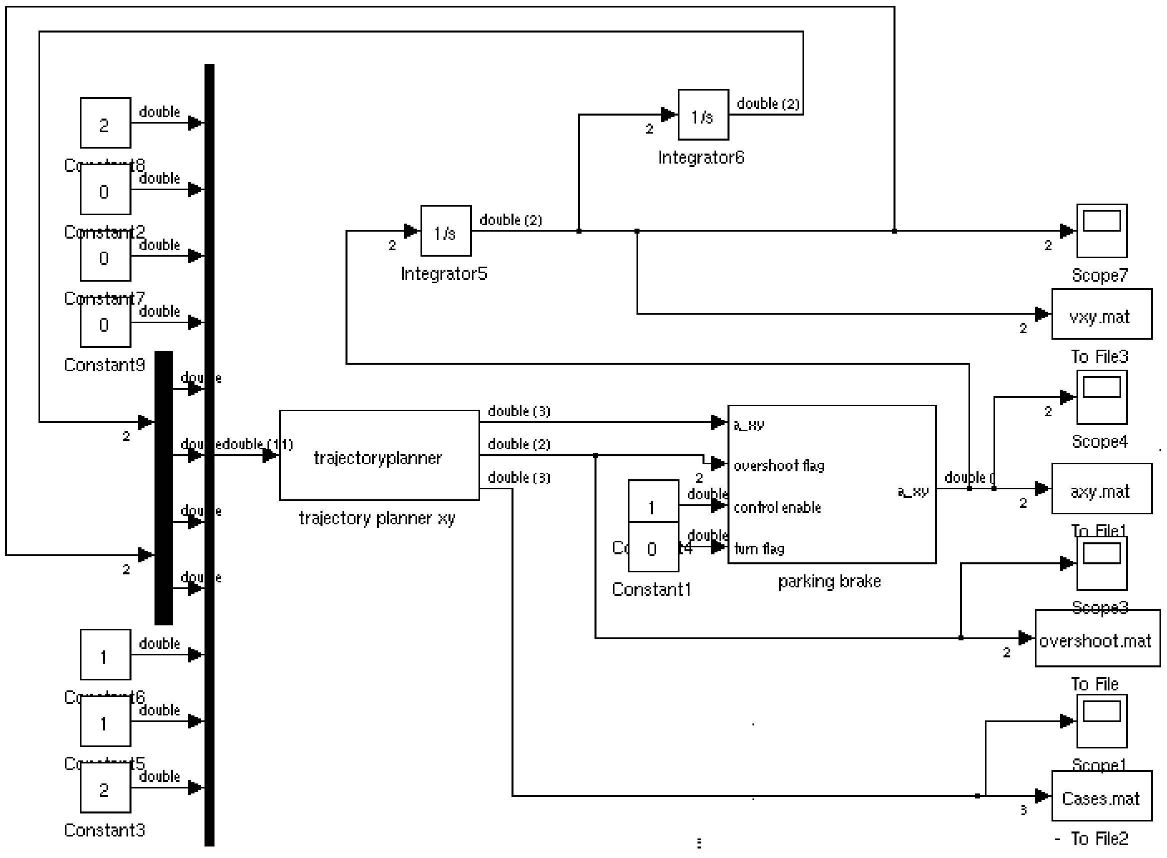

Figure B.1: Set up for simulation 1

For the simulation (simulation 2) where also a movement is introduced in y direction the same set up is used, but with a value for the desired movement in y direction.

For the simulation (simulation 3) where is attempt to simulate the real world a signal builder is used instead of a constant value for the y movement.

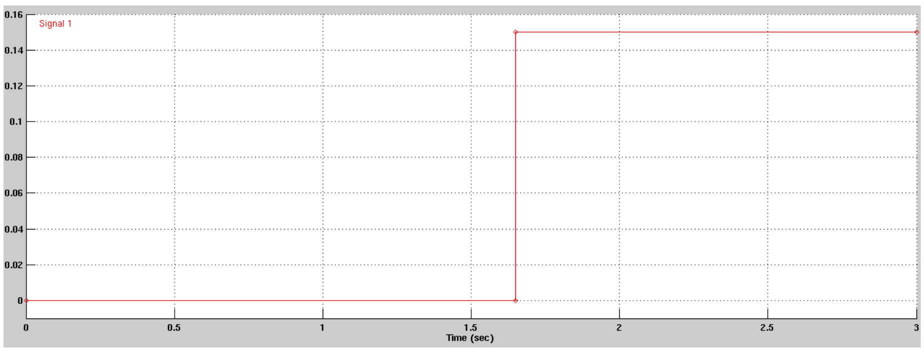

Figure B.2: Signal builder for simulation 3

Appendix C

C file trajectory planner

Listing C.1: C implementation trajectory planner

```
        /*   File     : traj.c
 *   Abstract: Trajectory generator
 *
 */

#define S_FUNCTION_NAME trajectoryplanner
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"
#include <math.h>
#include <stdio.h>
#include "global_par.h"
#include "bus.h"

#define dabs(A)  ((A)>=0?(A):-(A))
#define dmin(A,B)  ((A-B)>=0?(B):(A))
#define dmax(A,B)  ((A-B)>=0?(A):(B))
#define dsgn(A)  ((A)>=0?(1):(-1))

#define EPSILON_POSITION_CONVERGED 0.005 /* sample_time*v_max+0.5*a_max*sample_time*sample_time */
#define EPSILON_VELOCITY_CONVERGED 0.001*2.5 /* sample_time*a_max */
#define EPS 1e-16 /* Machine precision*/

/*====================*
 * S-function methods *
 *====================*/

/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    The sizes information is used by Simulink to determine the S-function
 *    block's characteristics (number of inputs, outputs, states, etc.).
 */

static int overshootflag;

static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 0);  /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return; /* Parameter mismatch will be reported by Simulink */
    }

    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, 11);
    ssSetInputPortDirectFeedThrough(S, 0, 1);
    if (!ssSetNumOutputPorts(S, 2)) return;
    ssSetOutputPortWidth(S, 0, 3);
    ssSetOutputPortWidth(S, 1, 2);

    ssSetNumSampleTimes(S, 1);
    ssSetNumRWork(S, 0);
    ssSetNumIWork(S, 0);
    ssSetNumPWork(S, 0);
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);

    /* Take care when specifying exception free code - see sfuntmpl_doc.c */
    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
}


/* Function: mdlInitializeSampleTimes =========================================
 * Abstract:
 *    Specifiy that we have a continuous sample time.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S,0,CONTINUOUS_SAMPLE_TIME);
    ssSetOffsetTime(S,0,0.0);
}

int check_case(double wf, double wodot, double wfdot, double v_max, double a_max, double* q_w,
double* t_accent, double* w_accent, double* wdot_accent){

    double w1dot = 0, t_I, t_II;
    double rev = 1;
    double SAMPLE_TIME = 0.001; /* global variable?! */

    if ( (wf<=0 && wfdot>=0) || (wf<0 && wfdot<0) ){
    /* Case 0: reverse negative cases */
        wf = -wf;
        wodot = -wodot;
        wfdot = -wfdot;
        rev = -rev;
    }

    if( dabs(wf)<=EPSILON_POSITION_CONVERGED && dabs(wfdot-wodot)<EPSILON_VELOCITY_CONVERGED ){
    /* No case: converged! */
        *q_w = dmin(dmax((wfdot-wodot)/SAMPLE_TIME,-a_max),a_max);
        *t_accent = 0;
        *w_accent = 0;
        *wdot_accent = wfdot;
        return 1;
    } else if (wodot>v_max){
    /* Case 5: too large velocity (after target switch, or noise); decelerate */
        *q_w = -a_max;
        *t_accent = (wodot-v_max)/a_max;
        *w_accent = 0.5*(wodot*wodot-v_max*v_max)/a_max;
        *wdot_accent = v_max;
    } else if (dabs(wodot-v_max)<=EPSILON_VELOCITY_CONVERGED &&
    wf>0.5*(wodot*wodot-wfdot*wfdot)/a_max){
    /* Case 3: driving at maximal velocity */
        *q_w = (v_max-wodot)/SAMPLE_TIME;
        *t_accent = wf/v_max+0.5*wfdot*wfdot/v_max/a_max-0.5*v_max/a_max;
        *w_accent = wf+0.5*(wfdot*wfdot-v_max*v_max)/a_max+EPS;
        *wdot_accent = v_max;
    } else if (v_max+EPSILON_VELOCITY_CONVERGED>=wodot && wodot>0 &&
    wf<=0.5*(wodot*wodot-wfdot*wfdot)/a_max){
    /* Case 4: need to decelerate to stop in time */
        *q_w = -a_max;
        *t_accent = (wodot-wfdot)/a_max;
        *w_accent = 0.5*(wodot*wodot-wfdot*wfdot)/a_max;
        *wdot_accent = wfdot;
    } else if (wf>0.5*(wfdot*wfdot-wodot*wodot)/a_max){
    /* Case 2: normal acceleration */
        t_I = (v_max-wodot)/a_max;
        w1dot = sqrt(wf*a_max+0.5*(wodot*wodot+wfdot*wfdot));
        t_II = (w1dot-wodot)/a_max;
        if (t_I<=t_II){
```

```
        /* Case 2.1: accelerate to maximum velocity */
            *q_w = a_max;
            *t_accent = t_I;
            *w_accent = 0.5*(v_max*v_max-wodot*wodot)/a_max;
            *wdot_accent = v_max;
        } else {
        /* Case 2.2: accelerate to w1dot<v_max */
            *q_w = a_max;
            *t_accent = t_II;
            *w_accent = 0.5*wf+0.25*(wfdot*wfdot-wodot*wodot)/a_max+EPS;
            *wdot_accent = w1dot;
        }
    } else if (wf>0.5*(wodot*wodot-wfdot*wfdot)/a_max){
    /* Case aanloopje: distance too small to reach desired final velocity */
        if (wfdot>0){
            w1dot = -1.1*sqrt(-wf*a_max+0.5*(wodot*wodot+wfdot*wfdot));
            *q_w = -a_max;
            *t_accent = (wodot-w1dot)/a_max;
            *w_accent = 0.5*(wodot*wodot-w1dot*w1dot)/a_max;
            *wdot_accent = w1dot;
        } else {
            w1dot = 1.1*sqrt(wf*a_max+0.5*(wodot*wodot+wfdot*wfdot));
            *q_w = a_max;
            *t_accent = (w1dot-wodot)/a_max;
            *w_accent = 0.5*(w1dot*w1dot-wodot*wodot)/a_max;
            *wdot_accent = w1dot;
        }
    } else if(wodot<0){
    /* Case 1: driving in negative (wrong) direction; accelerate */
        *q_w = a_max;
        *t_accent = -wodot/a_max;
        *w_accent = -0.5*wodot*wodot/a_max;
        *wdot_accent = 0;
    } else {
    /* printf("Error\n\n"); */
    }

    /* Compensate for Case 1, reverting */
    *w_accent = rev**w_accent;
    *wdot_accent = rev**wdot_accent;
    *q_w = rev**q_w;

    return 0;
}

double calculate_time(double wf, double wodot, double wfdot, double v_max_w, double a_max_w, double* q_impl){

    double              t = 0;      /* initial time */
    double            q_w = 0;      /* initial position */
    double       t_accent = 0;      /* update time */
    double       w_accent = 0;      /* update position */
    double    wdot_accent = 0;      /* update velocity */
    double            ETA = 0;      /* Estimated Time of Arrival */

    int                 N = 10;
    int                 i = 0;
    int         converged = 0;

    overshootflag = 0;

    for (i=0;i<N;i++){
        converged = check_case(wf,wodot,wfdot,v_max_w,a_max_w,&q_w,&t_accent,&w_accent,&wdot_accent);

        // if i==0 && q_w < 0.0 en je wil naar wf > 0.0 && (if i==1 && !converged)
        if(i==0 && q_w<0.0 && wf > 0.0)
        {
            overshootflag = 1;
        }
        if(i==1 && overshootflag==1 && !converged)
        {
            overshootflag = 3;
        }

        // if i==0 && q_w > 0.0 en je wil naar wf < 0.0 && (if i==1 && !converged)
        if(i==0 && q_w>0.0 && wf < 0.0)
        {
            overshootflag = 2;
        }
        if(i==1 && overshootflag==2 && !converged)
        {
            overshootflag = 3;
        }

        if (i==0){
            *q_impl = q_w;
        }

        /* Update time, position and velocity */
        t = t + t_accent;
        wf = wf - w_accent;
        wodot = wdot_accent;

        if (converged){
            ETA = t;
            break;
        }
    }
    return ETA;
}

/* Function: mdlOutputs =======================================================
 * Abstract:
 *      y = Cx + Du
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    double             *y     = ssGetOutputPortRealSignal(S,0);
    real_T             *povershootflag = (real_T *) ssGetOutputPortSignal(S,1);
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0); /* x_ref, xdot_ref, y_ref,
    ydot_ref, x, y, xodot, yodot, v_max, a_max, robotID */
    double             xf     = *uPtrs[0]-*uPtrs[4];
    double             xfdot = *uPtrs[1];     /* for the keeper (robot 1), it is assumed that
    xfdot and yfdot are zero! (velocities and accelerations are not adjusted for the keeper
    when end-velocities are considered) */
    double             yf     = *uPtrs[2]-*uPtrs[5];
    double             yfdot = *uPtrs[3];
    double             xodot = *uPtrs[6];
    double             yodot = *uPtrs[7];
    double             v_max = *uPtrs[8];     /* in case of robot 1 (goalkeeper), v_max and
    a_max represent the velocity and acceleration in y-direction */
    double             a_max = *uPtrs[9];
    int               robotID = (int) *uPtrs[10];

    double       SAMPLE_TIME = 0.001;     /* global variable?! */

    double               q_x = 0;
    double             ETA_x = 0;
    double         ETA_x_min = 0;
    double         ETA_x_max = 0;

    double               q_y = 0;
    double             ETA_y = 0;
    double         ETA_y_min = 0;
    double         ETA_y_max = 0;

    static double alpha = 0.25*M_PI;
    double         alpha_min = 0;
    double         alpha_max = 0.5*M_PI;

    double         v_max_x, v_max_y, a_max_x, a_max_y, v_max_d;
    double         SYNC_TIME_PRECISION1 = .001;
    double         SYNC_TIME_PRECISION2 = .001;
    double         EPSILON_VELOCITY_CONVERGED2 = 0.3;

    int                    j = 0;
    int            syncable = 0;

    /* check feasibility of final velocities */
/* this also prevents alpha_min>alpha_max (see below) */
v_max_d = sqrt(xfdot*xfdot+yfdot*yfdot);
if ( v_max_d>0.8*v_max ){
    xfdot = xfdot*(0.8*v_max)/v_max_d;
    yfdot = yfdot*(0.8*v_max)/v_max_d;
}
if (dabs(xfdot)<0.5*EPSILON_VELOCITY_CONVERGED2){
    xfdot = 0;
} else if (xfdot>0){
    xfdot = dmax(xfdot,EPSILON_VELOCITY_CONVERGED2);
} else {
    xfdot = dmin(xfdot,-EPSILON_VELOCITY_CONVERGED2);
}
if (dabs(yfdot)<0.5*EPSILON_VELOCITY_CONVERGED2){
    yfdot = 0;
} else if (yfdot>0){
    yfdot = dmax(yfdot,EPSILON_VELOCITY_CONVERGED2);
} else {
    yfdot = dmin(yfdot,-EPSILON_VELOCITY_CONVERGED2);
}

/* if final velocities are required, alpha_max and/or alpha_min are limited */
if ( dabs(xfdot)>0 ){
    v_max_x = dabs(xfdot);
    alpha_max = acos(v_max_x/v_max);
    v_max_y = sin(alpha_max)*v_max;
    a_max_y = sin(alpha_max)*a_max;
    a_max_x = cos(alpha_max)*a_max;

    ETA_x_max = calculate_time(xf,xodot,xfdot,v_max_x,a_max_x,&q_x);
    povershootflag[0]=overshootflag;
    ETA_y_min = calculate_time(yf,yodot,yfdot,v_max_y,a_max_y,&q_y);
    povershootflag[1]=overshootflag;
}
if ( dabs(yfdot)>0 ){
    v_max_y = dabs(yfdot);
    alpha_min = asin(v_max_y/v_max);
    v_max_x = cos(alpha_min)*v_max;
    a_max_x = cos(alpha_min)*a_max;
    a_max_y = sin(alpha_min)*a_max;

    ETA_x_min = calculate_time(xf,xodot,xfdot,v_max_x,a_max_x,&q_x);
    povershootflag[0]=overshootflag;
    ETA_y_max = calculate_time(yf,yodot,yfdot,v_max_y,a_max_y,&q_y);
    povershootflag[1]=overshootflag;
}
alpha = dmax(alpha,alpha_min);
alpha = dmin(alpha,alpha_max);

/* check whether either one or both directions are converged */
if( (dabs(xf)<=EPSILON_POSITION_CONVERGED && dabs(xfdot-xodot)<EPSILON_VELOCITY_CONVERGED2) &&
(dabs(yf)<EPSILON_POSITION_CONVERGED && dabs(yfdot-yodot)<EPSILON_VELOCITY_CONVERGED2) ){
    q_x = dmin(dmax((xfdot-xodot)/SAMPLE_TIME,-a_max),a_max);
    q_y = dmin(dmax((yfdot-yodot)/SAMPLE_TIME,-a_max),a_max);
    if (dabs(q_x)>0){
        ETA_x = (xfdot-xodot)/q_x;
    } else {
        ETA_x = 0;
    }
    if (dabs(q_y)>0){
        ETA_y = (yfdot-yodot)/q_y;
    } else {
        ETA_y = 0;
    }
    y[0] = q_x;
    y[1] = q_y;
    y[2] = dmax(ETA_x,ETA_y);

} else if ( dabs(xf)<=EPSILON_POSITION_CONVERGED && dabs(xfdot-xodot)<EPSILON_VELOCITY_CONVERGED2 ){
    q_x = dmin(dmax((xfdot-xodot)/SAMPLE_TIME,-a_max),a_max);
    ETA_y = calculate_time(yf,yodot,yfdot,v_max,a_max,&q_y);
    povershootflag[1]=overshootflag;
    y[0] = q_x;
    y[1] = q_y;
```

```
        y[2] = ETA_y;

    } else if ( dabs(yf)<=EPSILON_POSITION_CONVERGED && dabs(yfdot-yodot)<EPSILON_VELOCITY_CONVERGED2 ){
        q_y = dmin(dmax((yfdot-yodot)/SAMPLE_TIME,-a_max),a_max);
        ETA_x = calculate_time(xf,xodot,xfdot,v_max,a_max,&q_x);
        povershootflag[0]=overshootflag;
        y[0] = q_x;
        y[1] = q_y;
        y[2] = ETA_x;

    } else { /* no direction converged yet: synchronize if possible */

        /* check sync'ability of situation */
        if (robotID == 1){
            syncable = 0;           /* do not sync in case of robot 1*/

            ETA_x = calculate_time(xf,xodot,xfdot,v_max,a_max,&q_x);
            povershootflag[0]=overshootflag;
            ETA_y = calculate_time(yf,yodot,yfdot,v_max,a_max,&q_y);
            povershootflag[1]=overshootflag;
            y[0] = q_x;
            y[1] = q_y;
            y[2] = dmax(ETA_x,ETA_y);

        } else if ( dabs(xfdot)>0 && dabs(yfdot)>0 ){
            if ( ETA_y_min<=ETA_x_max && ETA_x_min<=ETA_y_max ){
                syncable = 1;       /* sync'able! */
            } else {
                syncable = 0;       /* non sync'able (yet) */
                if ( ETA_y_min>ETA_x_max ){
                    alpha = alpha_max;
                    v_max_y = sin(alpha)*v_max;
                    a_max_y = sin(alpha)*a_max;
                    ETA_y_min = calculate_time(yf,yodot,yfdot,v_max_y,a_max_y,&q_y);
                    povershootflag[1]=overshootflag;
                    y[0] = 0;
                    y[1] = q_y;
                    y[2] = ETA_y_min;
                } else if ( ETA_x_min>ETA_y_max ){
                    alpha = alpha_min;
                    v_max_x = cos(alpha_min)*v_max;
                    a_max_x = cos(alpha_min)*a_max;
                    ETA_x_min = calculate_time(xf,xodot,xfdot,v_max_x,a_max_x,&q_x);
                    povershootflag[0]=overshootflag;
                    y[0] = q_x;
                    y[1] = 0;
                    y[2] = ETA_x_min;
                } else {
                    /* printf("This should not be possible! ETA_y_min = %f, ETA_x_max = %f,
                     ETA_x_min = %f, ETA_y_max = %f\n",ETA_y_min,ETA_x_max,ETA_x_min,ETA_y_max); */
                }
            }
        } else if ( dabs(xfdot)>0 ){
            if (ETA_y_min<=ETA_x_max){
                syncable = 1;       /* sync'able! */
            } else {
                syncable = 0;       /* non sync'able (yet) */
                alpha = alpha_max;
                v_max_y = sin(alpha)*v_max;
                a_max_y = sin(alpha)*a_max;
                ETA_y_min = calculate_time(yf,yodot,yfdot,v_max_y,a_max_y,&q_y);
                povershootflag[1]=overshootflag;
                y[0] = 0;
                y[1] = q_y;
                y[2] = ETA_y_min;
            }
        } else if ( dabs(yfdot)>0 ){
            if (ETA_x_min<=ETA_y_max){
                syncable = 1;       /* sync'able! */
            } else {
                syncable = 0;       /* non sync'able (yet) */
                alpha = alpha_min;
                v_max_x = cos(alpha_min)*v_max;
                a_max_x = cos(alpha_min)*a_max;
                ETA_x_min = calculate_time(xf,xodot,xfdot,v_max_x,a_max_x,&q_x);
                povershootflag[0]=overshootflag;
                y[0] = q_x;
                y[1] = 0;
                y[2] = ETA_x_min;
            }
        } else {                        /* no final velocities */
            syncable = 1;               /* sync'able! */
        }

        /* if sync'able, check previously derived (static) alpha */
        if ( syncable ){
            v_max_x = cos(alpha)*v_max;
            a_max_x = cos(alpha)*a_max;
            v_max_y = sin(alpha)*v_max;
            a_max_y = sin(alpha)*a_max;

            ETA_x = calculate_time(xf,xodot,xfdot,v_max_x,a_max_x,&q_x);
            povershootflag[0]=overshootflag;
            ETA_y = calculate_time(yf,yodot,yfdot,v_max_y,a_max_y,&q_y);
            povershootflag[1]=overshootflag;

            if ( dabs(ETA_x-ETA_y)>SYNC_TIME_PRECISION2 ){
                /* (renew initial guess alpha omitted / not implemented) */
                while ( dabs(ETA_x-ETA_y)>SYNC_TIME_PRECISION1 ){
                    if (j==0){
                        /* printf("need to iterate! xf=%f, yf=%f, xodot=%f, yodot=%f, alpha_min=%f,
                        alpha_max=%f, alpha=%f, ETA_x=%f, ETA_y=%f\n",xf,yf,xodot,yodot,alpha_min,
                        alpha_max,alpha,ETA_x,ETA_y); */
                    }
                    if ( ETA_x<ETA_y ){
                        alpha_min = alpha;
                        alpha = 0.5*alpha+0.5*alpha_max;
                    } else{
                        alpha_max = alpha;
                        alpha = 0.5*alpha+0.5*alpha_min;
                    }

                    v_max_x = cos(alpha)*v_max;
                    a_max_x = cos(alpha)*a_max;
                    v_max_y = sin(alpha)*v_max;
                    a_max_y = sin(alpha)*a_max;

                    ETA_x = calculate_time(xf,xodot,xfdot,v_max_x,a_max_x,&q_x);
                    povershootflag[0]=overshootflag;
                    ETA_y = calculate_time(yf,yodot,yfdot,v_max_y,a_max_y,&q_y);
                    povershootflag[1]=overshootflag;

                    j++;
                    /* printf("   alpha = %f, ETA_x=%f, ETA_y=%f\n",alpha,ETA_x,ETA_y); */
                    if (j>50){
                        /* printf("   bi-section algorithm not converged!\n"); */
                        break;
                    }
                }
            /* } else { */
            /*     printf("previous alpha was ok\n"); */
            }

            y[0] = q_x;
            y[1] = q_y;
            y[2] = 0.5*(ETA_x+ETA_y);
        }
    }
}


/* Function: mdlTerminate =====================================================
 * Abstract:
 *    No termination needed, but we are required to have this routine.
 */
static void mdlTerminate(SimStruct *S)
{
    UNUSED_ARG(S); /* unused input argument */
```

```
}

#ifdef  MATLAB_MEX_FILE     /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

Appendix D

Acceleration x direction Chapter 5

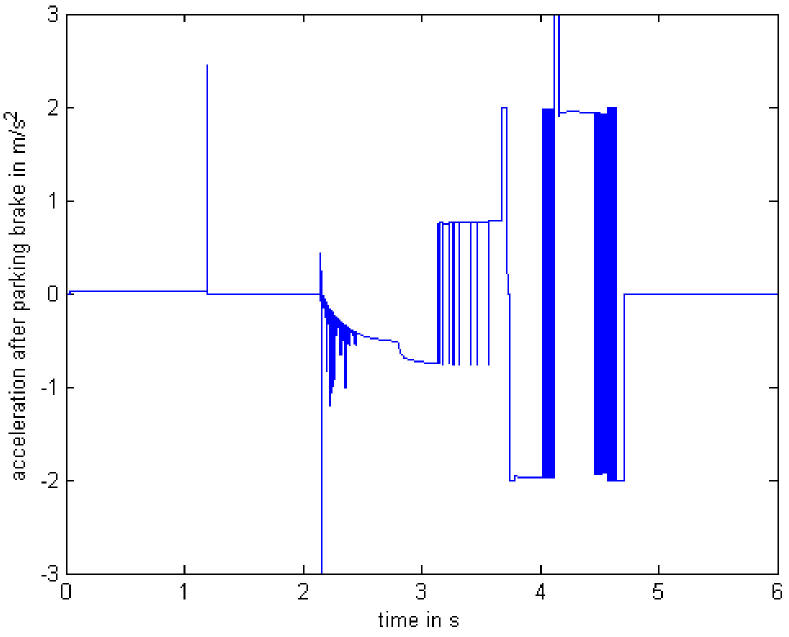

Figure D.1: Acceleration after the parking brake in x direction

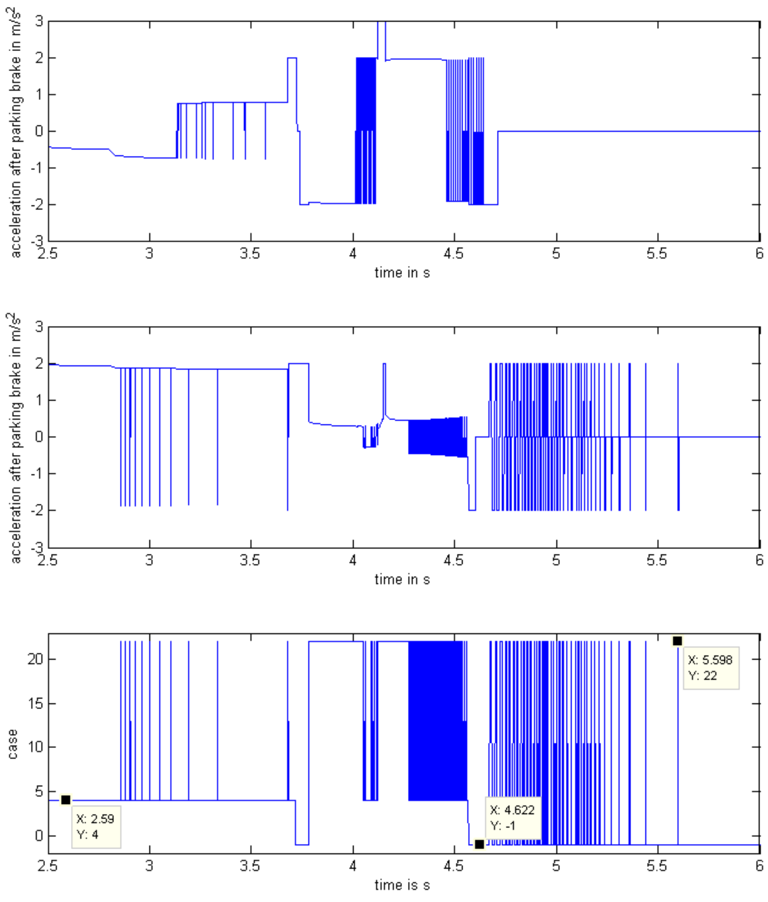

Figure D.2: A zoom of the x- and y acceleration and the case for the y direction on time frame[2.5 - 6].

Appendix E

Set up for simulations traction controller

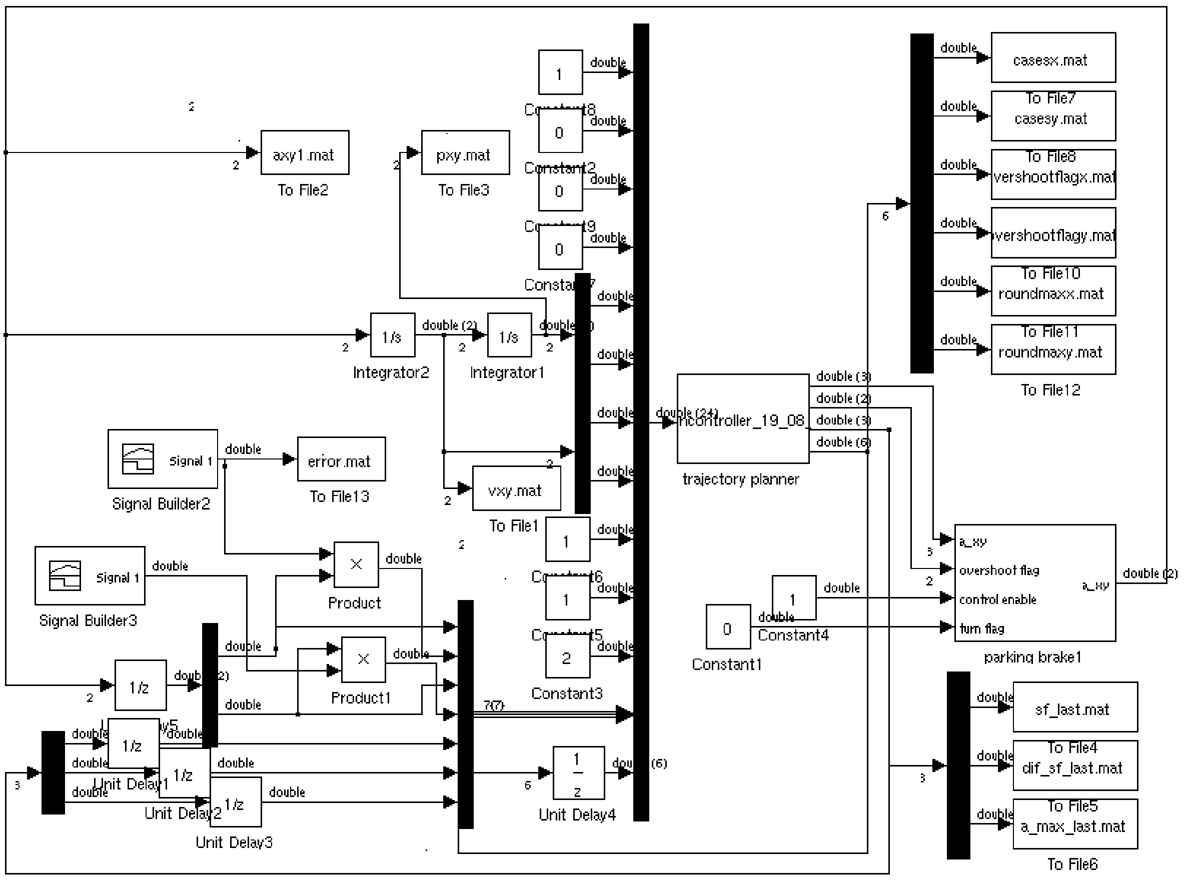

Figure E.1: The set up for the simulation of the traction controller.

Now the inputs are the same as in Appendix A, but there are 13 inputs more:

- The instantaneous input acceleration in x direction.
- The measured acceleration in x position
- The instantaneous input acceleration in y direction.
- The measured acceleration in y position
- The safety factor of previous time step
- The difference in safety factor of previous time step
- The maximal acceleration of previous time step
- the cases in x direction
- the cases in y direction
- the overshootflag in x direction
- the overshootflag in y direction
- the roundmax in x direction
- the roundmax in y direction

With the signal builders there is an error imposed in the signal. The signal builders are initial 1, but for some time frames they have a value above 1.