

Tire Pressure Monitoring

DHBW Automotive Software Engineering – Stuttgart, © Kai Pinnow 2019

1. Introduction and Overview

The project work focuses on tire pressure monitoring. Detailed requirements below.

As depicted, the following equations govern the motion for a left turn:

$$(1) \quad R_{RR} = W + R_{RL}$$

$$(2) \quad R_{FR}^2 = B^2 + R_{RR}^2$$

$$(3) \quad R_{FL}^2 = B^2 + R_{RL}^2$$

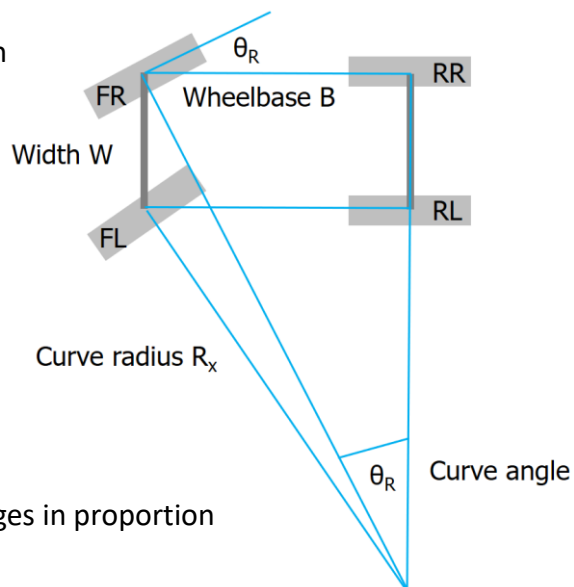
$$(4) \quad B = R_{RR} \tan \theta_R$$

$$(5) \quad B = R_{RL} \tan \theta_L$$

Please note that any pair of velocities changes in proportion to the related curve radiuses

$$(6) \quad V_x / V_y = R_x / R_y \quad \text{for any } x, y \text{ from } \{RR, RL, FR, FL\}$$

A drop in tire pressure of a single wheel results in an increase of the respective velocity of that wheel since it will be a little smaller then.



2. Requirements and Deliverables

Please refer to "ID" column in your documentation. A * denotes extra tasks / requirements.

ID	Check
R1	A tire pressure monitor allows observation of the four wheel speeds to detect an unexpected imbalance for a vehicle with sizes $B=1.53$ m and $W=2.65$ m.
R2	An imbalance of a wheel speed of 0.5 % of one of the wheels concerning the expected consistent wheel speeds will be regarded as an indication of a tire pressure drop.
R3	Detecting a tire pressure drop, some warning lamp shall be switched on and some "SOS" (three times short, three times long, three times short) sound shall appear (base rate 0.8 seconds).
R4*	The system shall allow "re-calibration" after inflation.
R5	Design the solution mainly with appropriate graphical modeling elements (i.e. block diagrams and/or state machines) or with scripts or ESDL and document all your decisions, reasoning, and results clearly with screenshots and text.

D1	Plan all necessary tasks based on three point estimates and monitor progress according to below requirements.
D2	Use the provided example data "curve.mat" to calculate, display, and analyze curve radiuses for selected situations. It contains the wheel speeds (vfl, vfr, vrl, vrr) in [km/h] and the steering wheel signal sw (without direction) in [degree] with time base tv in [s], plus the corresponding lateral acceleration q in [g] (with different time base tq again in [s]).
D3	Create a Simulink model that calculates the driving distance for each wheel and analyze the provided "curve.mat" data in this regard. Remember to analyze and document settings. Are there imbalances according to requirement R2?
D4	Set-up a simple tire pressure monitor in Simulink that detects a deviation according to requirement R1 and R2 by observing driving distances of the individual wheels for straight driving i.e. driving without curves.
D5	Code, configure, and apply a simple "linear congruential" random number generator like $X(i) = (a * X(i-1) + c) \bmod m$ with suitable parameters a, c, and m to test the tire pressure monitoring without the provided "curve.mat" data.
D6	Execute some system tests in Simulink with the number generator from D6 to check the tire pressure monitoring function feasibility.
D7	Transfer the tire pressure monitoring function to ASCET.
D8	Provide unit tests for all designed tire pressure monitoring components.
D9	Design a warning function according to requirement R3.
D10	Provide the random number generator designed in D5 in ASCET with unit tests.
D11	Create a system test with the aid of the number generator and some error model i.e. simulating some pressure drop over a certain time to demonstrate the tire pressure monitoring.
D12*	Think about the way to calibrate the system by means of requirement R4. Which parts of the implementation shall change in order to support such a feature? How long does one need to drive for calibration?
D13*	Shall the analysis incorporate curve driving or just analyze segments driving straight?
D14*	Reflect: Which other observations or comments are in place concerning the model, the requirements, the prescribed functions, or your solution, the testing, and the selected graphical approach.

1 Aufgabe D2

1.1 Berechnung der Radien

Im folgenden werden die in der Aufgabenstellung gegebenen Gleichungen verwendet, um die Kurvenradien der einzelnen Räder zu berechnen.

Mit (6) folgt: $\frac{R_{RR}}{R_{RL}} = \frac{V_{RR}}{V_{RL}}$

$$R_{RL} = R_{RR} * \frac{V_{RR}}{V_{RL}}$$

Mit (1) folgt: $R_{RL} = \frac{V_{RR}}{V_{RL}} * (W + R_{RL})$

$$R_{RL} - \frac{V_{RR}}{V_{RL}} * R_{RL} = \frac{V_{RR}}{V_{RL}} * W$$

$$R_{RL} * (1 - \frac{V_{RR}}{V_{RL}}) = \frac{V_{RR}}{V_{RL}} * W$$

$$R_{RL} = \frac{\frac{V_{RR}}{V_{RL}} * W}{1 - \frac{V_{RR}}{V_{RL}}}$$

$$R_{RL} = \frac{W}{\frac{V_{RR}}{V_{RL}} - 1}$$

Mit (1) folgt: $R_{RR} = W + R_{RL} = W + \frac{W}{\frac{V_{RR}}{V_{RL}} - 1}$

Mit (2) folgt: $R_{FR}^2 = B^2 + (W + \frac{W}{\frac{V_{RR}}{V_{RL}} - 1})^2$

$$R_{FR} = \sqrt{B^2 + (W + \frac{W}{\frac{V_{RR}}{V_{RL}} - 1})^2}$$

Mit (3) folgt: $R_{FR}^2 = B^2 + (\frac{W}{\frac{V_{RR}}{V_{RL}} - 1})^2$

$$R_{FR} = \sqrt{B^2 + (\frac{W}{\frac{V_{RR}}{V_{RL}} - 1})^2}$$

Die erstellten Gleichungen stellen die Grundlage für die folgenden Analysen dar.

1.2 Analyse der ausgewählten Situationen

Im Folgenden wird anhand einiger Graphiken das Verhalten bei Kurvenfahrten analysiert.

In der nachfolgenden [Abbildung 1.1](#) werden zuerst die Geschwindigkeiten der vier Räder des Fahrzeugs dargestellt. Es ist erkennbar, dass es Abweichungen zwischen den Geschwindigkeiten gibt. Diese werden nachfolgend genauer erläutern.

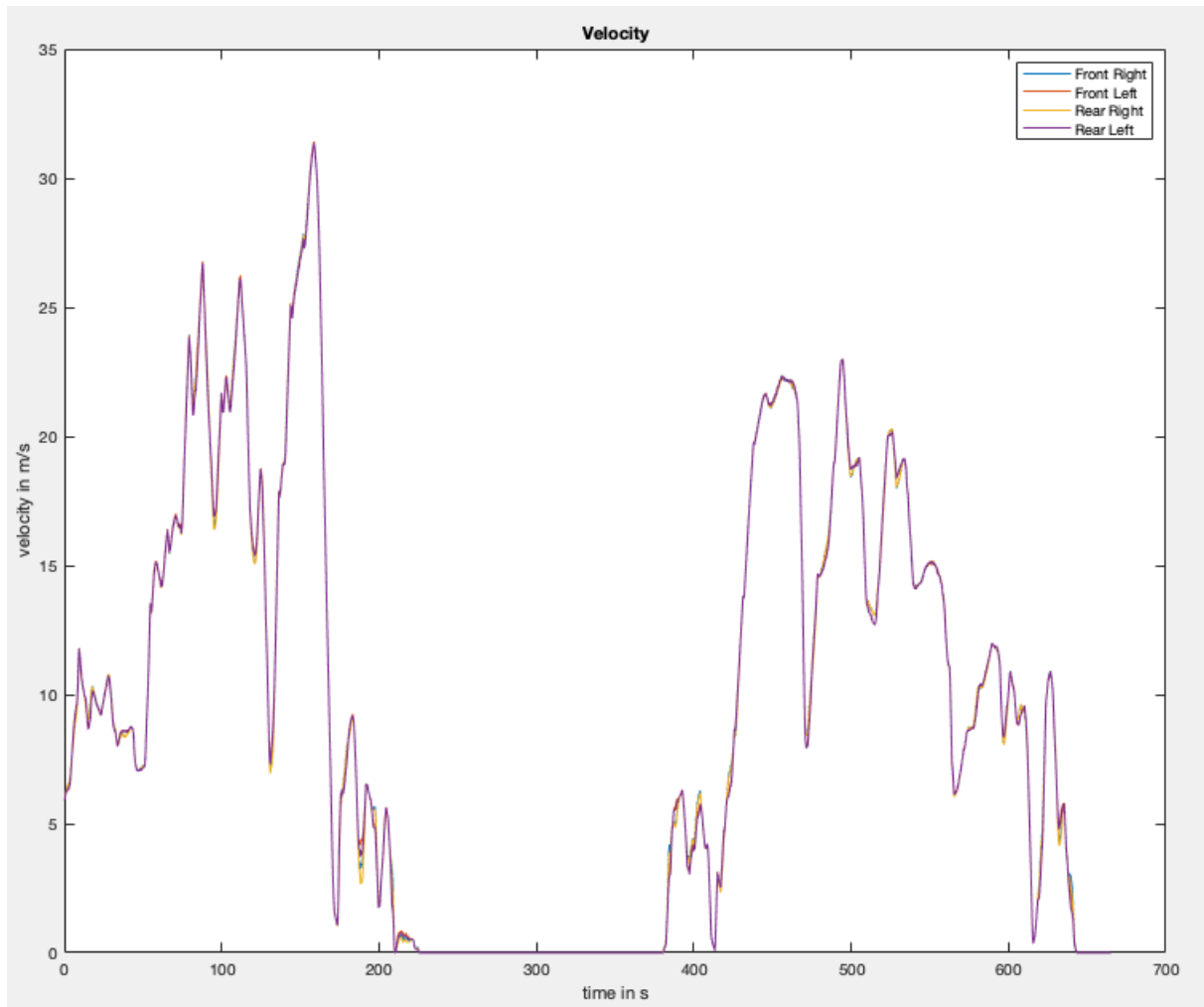


Abbildung 1.1: Übersicht der Geschwindigkeiten der vier Räder

Die [Abbildung 1.2](#) zeigt einen Ausschnitt zwischen 495s und 550s. Die Abbildung wurde in einzelne Abschnitte zur Analyse unterteilt. Im ersten Abschnitt, vor 500ms, ist der Lenkwinkel sehr klein. Man sieht dort kaum einen Unterschied zwischen den Geschwindigkeiten der linken und rechten Räder. Im zweiten Abschnitt ist der Lenkwinkel wesentlich höher. Das hat zur Folge, dass die Geschwindigkeit der rechten Räder über der Geschwindigkeit der linken Räder liegt. Beim Wechsel vom zweiten in den dritten Abschnitt ist der Winkel kurz Null und steigt dann wieder. Dort liegt die Geschwindigkeit der linken Räder über der Geschwindigkeit der rechten Räder. Daraus lässt sich schließen, dass sich die Fahrt von einer Links- in eine Rechtskurve ändert. Im vorletzten Abschnitt ist wieder eine Linkskurve erkennbar. Im letzten Abschnitt wird kaum gelenkt, aus diesem Grund weichen die Geschwindigkeiten nur sehr gering voneinander ab.

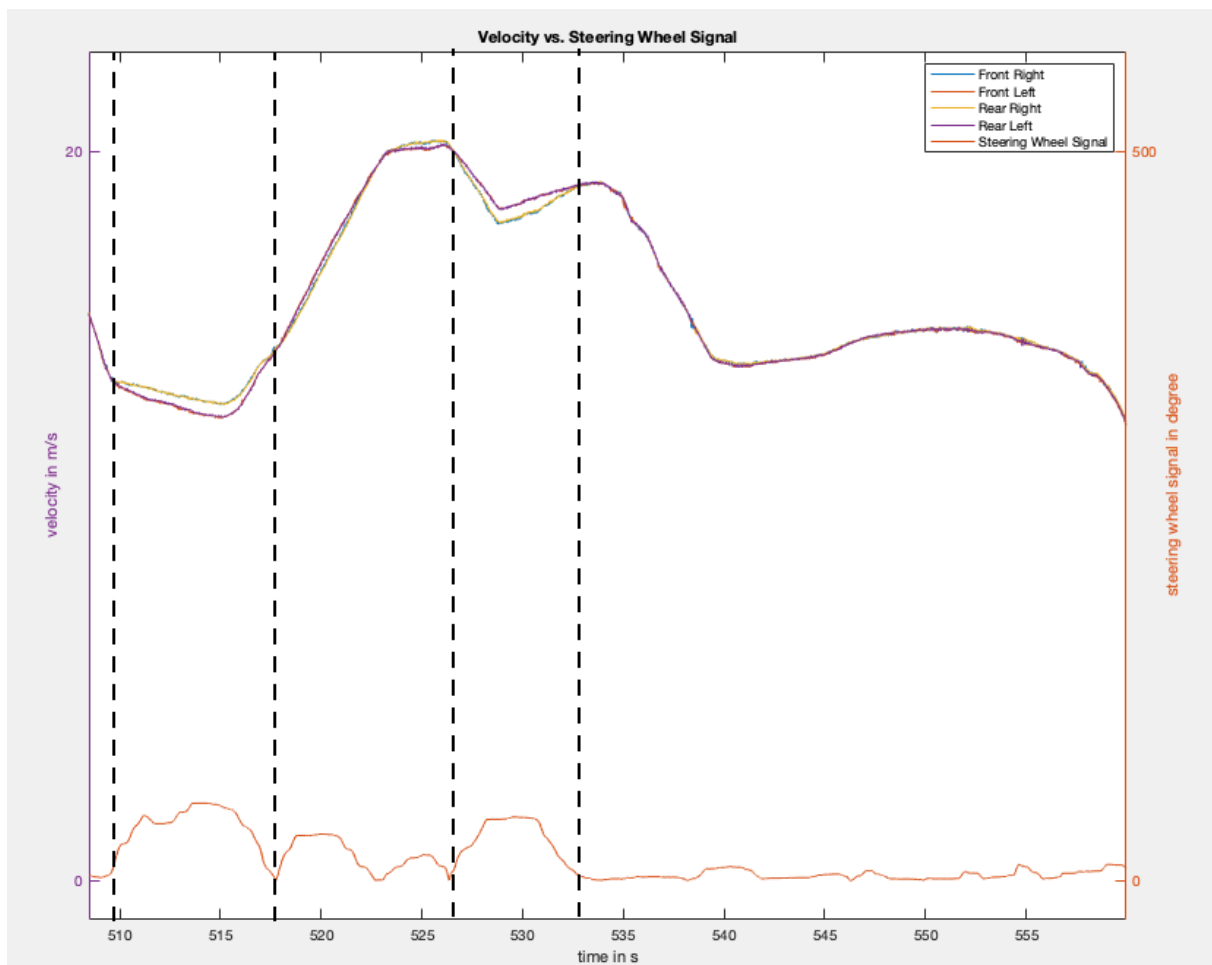


Abbildung 1.2: Übersicht der Geschwindigkeiten mit Drehwinkel des Lenkrads

Die [Abbildung 1.3](#) zeigt den Ausschnitt von ca. 510ms bis 517ms. Dieser entspricht dem dem zweiten Abschnitt der [Abbildung 1.2](#). Hier werden die Radien der beiden Vorderräder verglichen. Es ist erkennbar, dass der Radius des Rechten Vorderrads immer über dem Radius des Linken liegt. Dies zeigt nochmal, dass es sich hier um eine Linkskurve handelt.

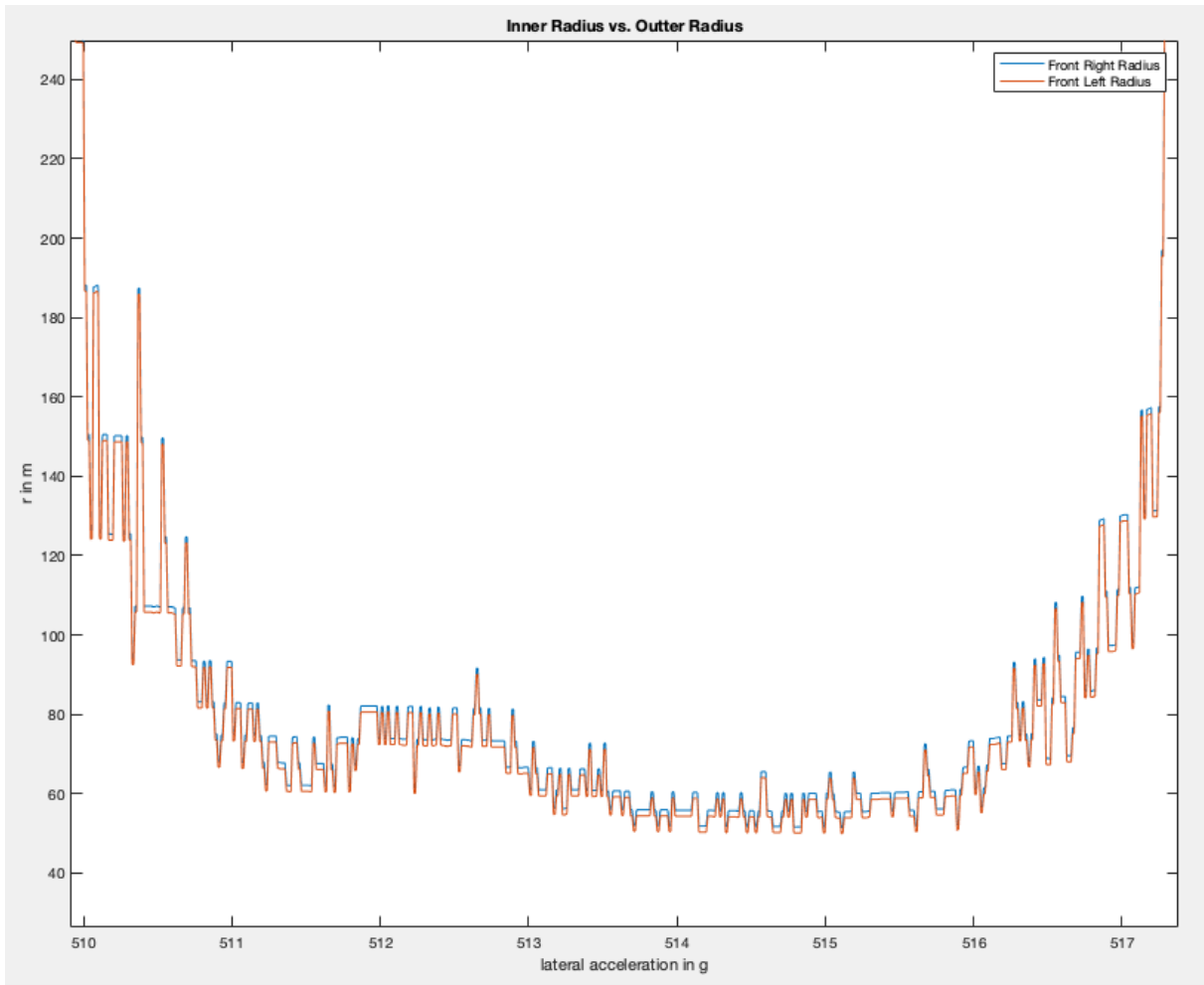


Abbildung 1.3: Vergleich des linken und rechten Radius bei Kurvenfahrt

In der nächsten [Abbildung 1.4](#) wird der Radius in Abhängigkeit des Drehwinkels abgebildet. Es zeigt, dass mit steigendem Drehwinkel der Radius immer kleiner wird. Stellen, an denen das Fahrzeug nicht fährt, können hierbei vernachlässigt werden. Allgemein gilt, dass bei Kurvenfahrten der Radradius abnimmt.

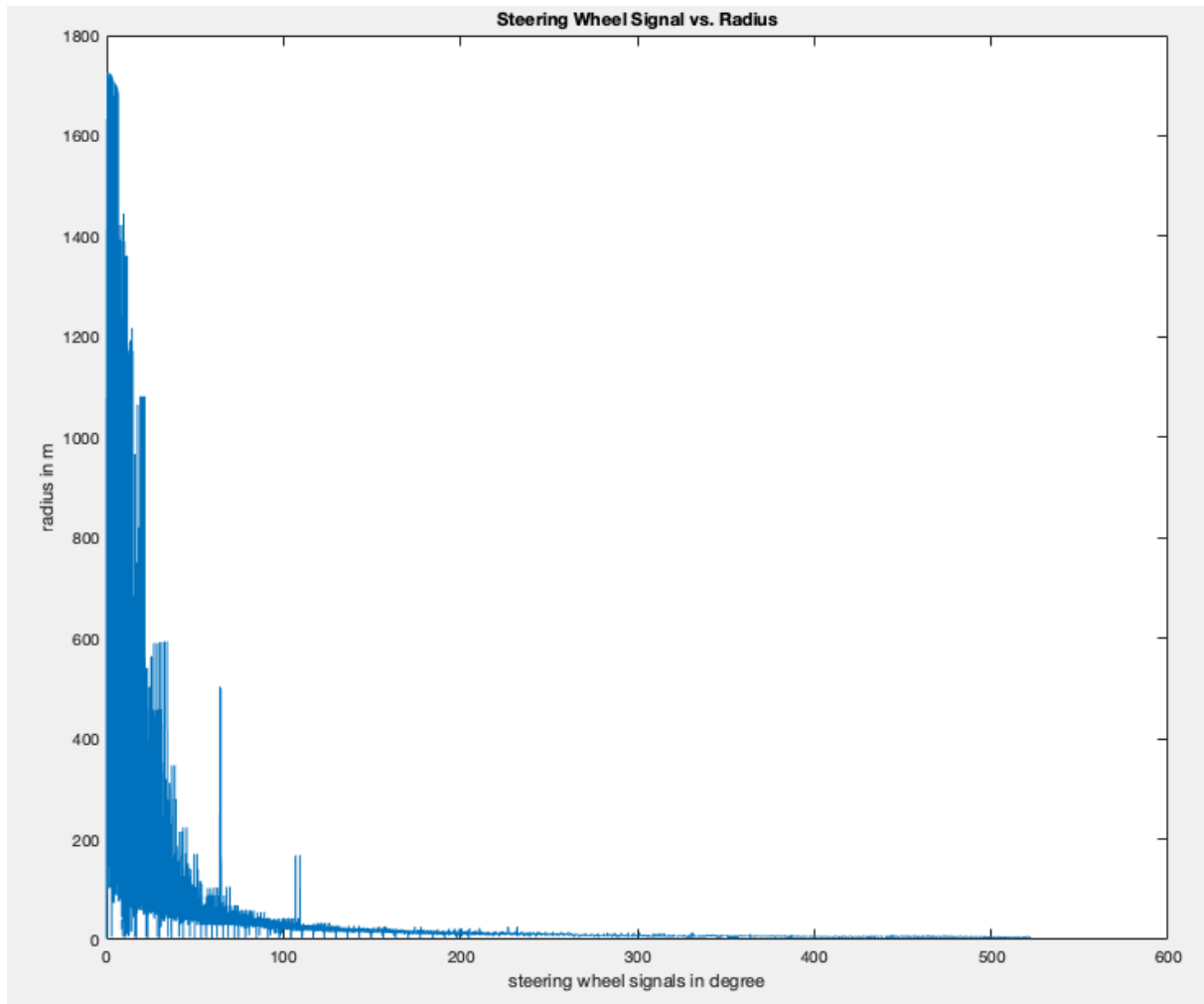


Abbildung 1.4: Lenkraddrehwinkel gegenüber Radius

In der folgenden [Abbildung 1.5](#) wird der Radius in Abhängigkeit der Geschwindigkeit dargestellt. Es zeigt, dass bei gleichem Drehwinkel des Lenkrads bei steigender Geschwindigkeit der Radius immer größer wird.

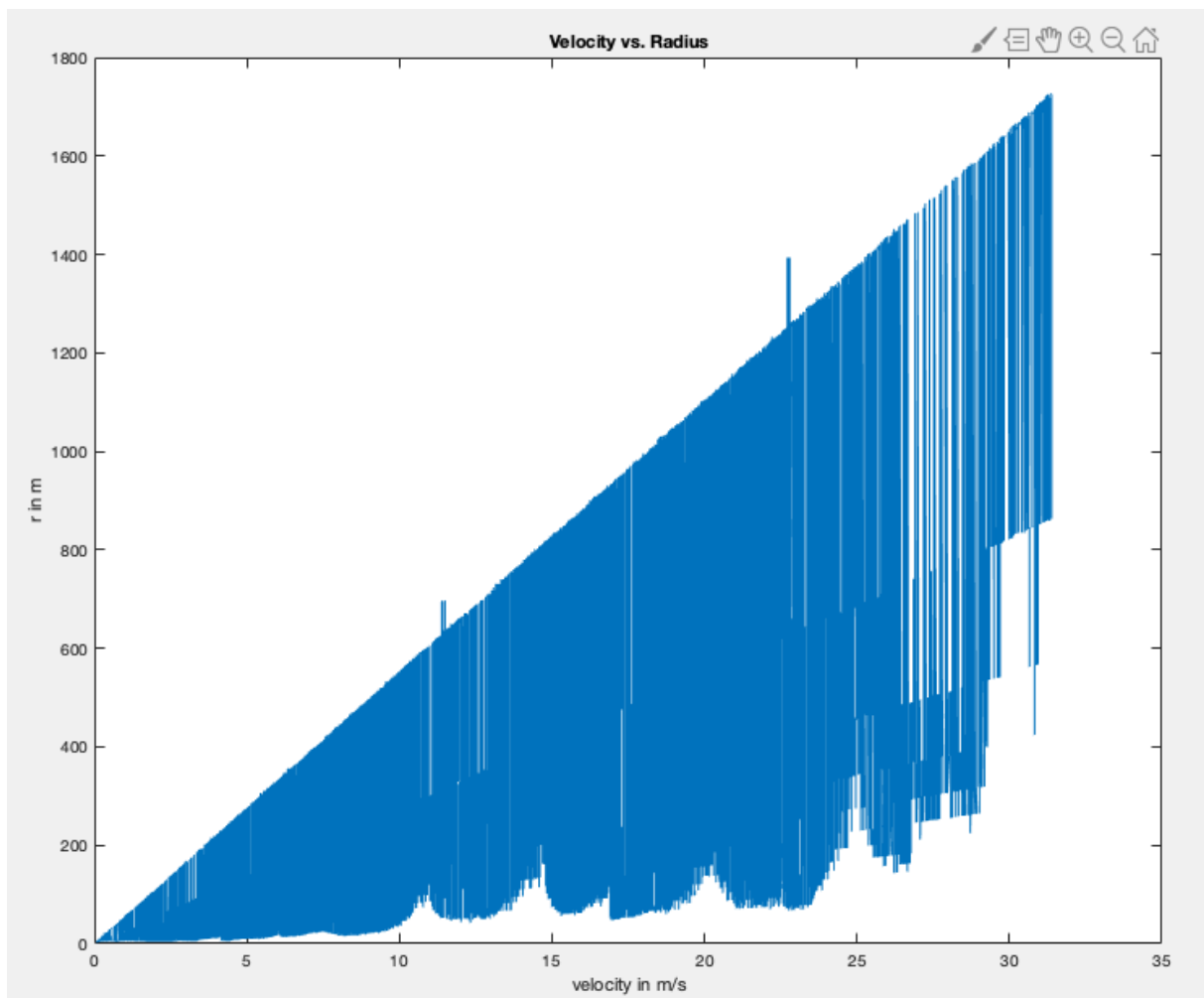


Abbildung 1.5: Geschwindigkeit gegenüber Radius

2 Aufgabe D3

2.1 Simulink-Modell

Die nachfolgenden Abbildungen zeigen das Simulink-Modell zur Berechnung der Distanzen der einzelnen Räder.

Das Modell in [Abbildung 2.1](#) hat die vier Geschwindigkeiten (in km/h) der Räder als Eingänge. Diese werden mit dem „From Workspace“ Block mit ihrer zugehörigen Referenzzeit t_v in das Model importiert.

Danach werden sie in m/s umgerechnet und integriert, um die Strecke zu erhalten.

Zur Simulation wird ein FixedStep von 0.01 mit dem ode3-Solver verwendet. Der ode-Solver wurde durch die Simulink „automatic solver selection“ ausgewählt. Das Ergebnis

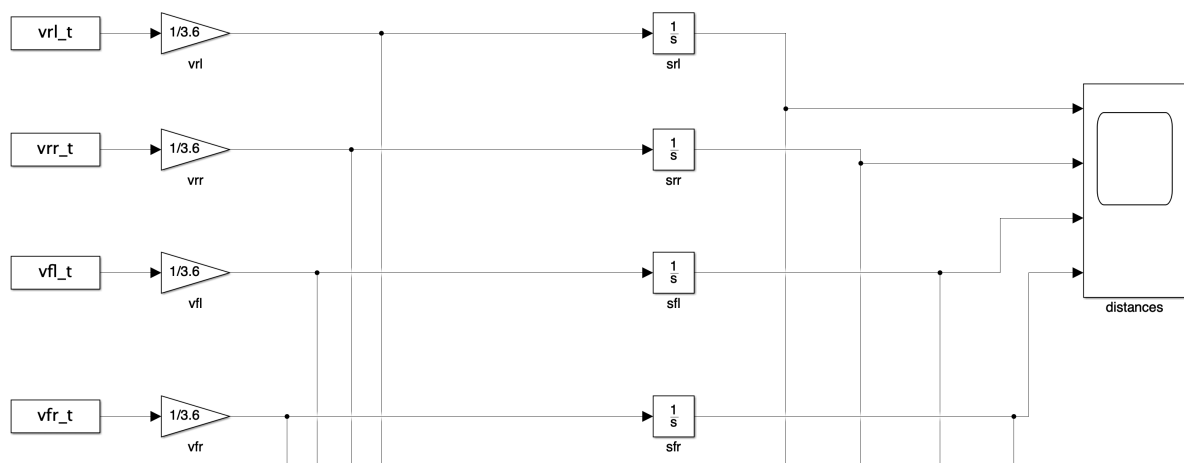


Abbildung 2.1: Tire Driving distances

der Integration sind die Distanzen der einzelnen Reifen, die gegenüber der Zeit aufgetragen sind.

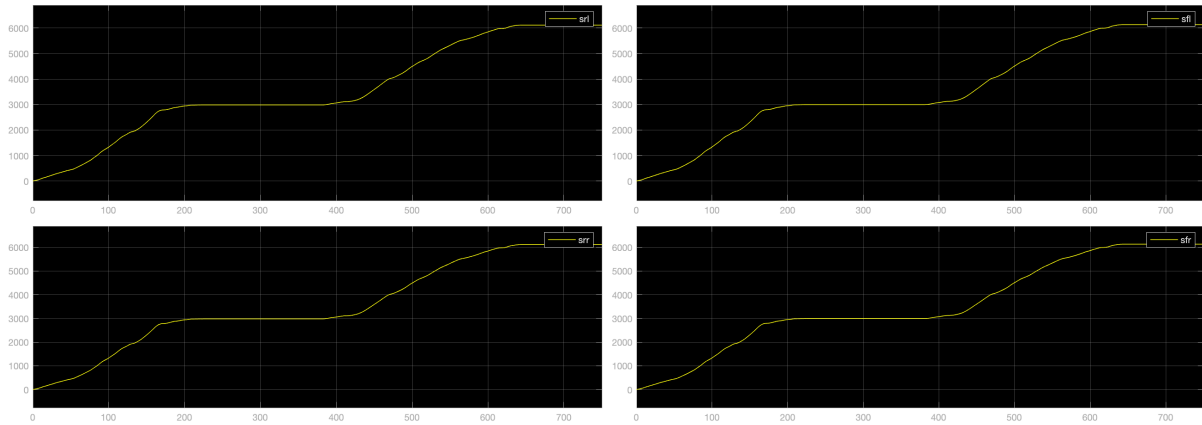


Abbildung 2.2: Tire Driving distances plot

Zur Überprüfung ob es Imbalancen in den Daten gibt wird das Modell wie in [Abbildung 2.4](#) zu sehen ist erweitert. Der Mittelwert der vier integrierten Distanzen gilt als Referenzwert, ob ein Reifendruckabfall vorliegt. Die Berechnung ist ebenfalls im erweiterten Modell [Abbildung 2.4](#) zusehen. Anschließend wird die prozentuale Abweichung vom Einzelsignal zum Mittelwert berechnet, weicht die Abweichung um mehr als 0.5% ab handelt es sich um einen Reifendruckabfall. Die Berechnung ist in [Abbildung 6.9](#) dargestellt.

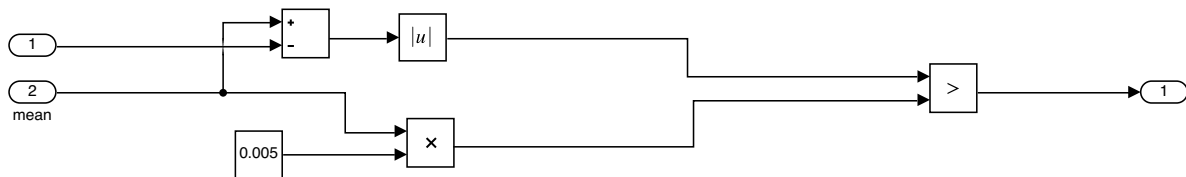


Abbildung 2.3: Prozentuale Abweichung nach R2

Damit die Messungen nicht durch Rauschen im Geschwindigkeitssignal verfälscht werden, wird das integrierte Signal zur Messung verwendet. Durch die Integration wird das kurzzeitiges Rauschen sozusagen gemittelt, sodass es die Messung nicht verfälscht. Damit jedoch ein Reifendruckabfall nicht untergeht, da das bereits integrierte Signal zu groß ist wird in Deltas von 10 Sekunden gemessen.

Dazu wird der „Transport Delay“ Block genutzt der das Signal um eine beliebige Zeit (hier 10 Sekunden) verzögert. Das Verzögerte Signal kann dann vom Originalsignal abgezogen werden, sodass ausschließlich das Integral der letzten 10 Sekunden betrachtet wird.¹

Die Berechnungen für die Abweichungen werden für jeden Reifen einzeln berechnet und

¹ In den ersten 10 Sekunden, ist der Output des Transport Delays 0, sodass in den 10 Sekunden Rauschen durchaus die Messungen verfälschen kann. Deswegen wird der Output des Transport Delays in den ersten 10 Sekunden manuell auf -1000 gesetzt, sodass die Messung in den ersten 10 Sekunden bewusst so manipuliert wird, dass es keine Imbalancen gibt

anschließend, wie in [Abbildung 2.4](#) zu sehen, verodert, um ein Überblick über das Gesamtsystem zu erhalten.

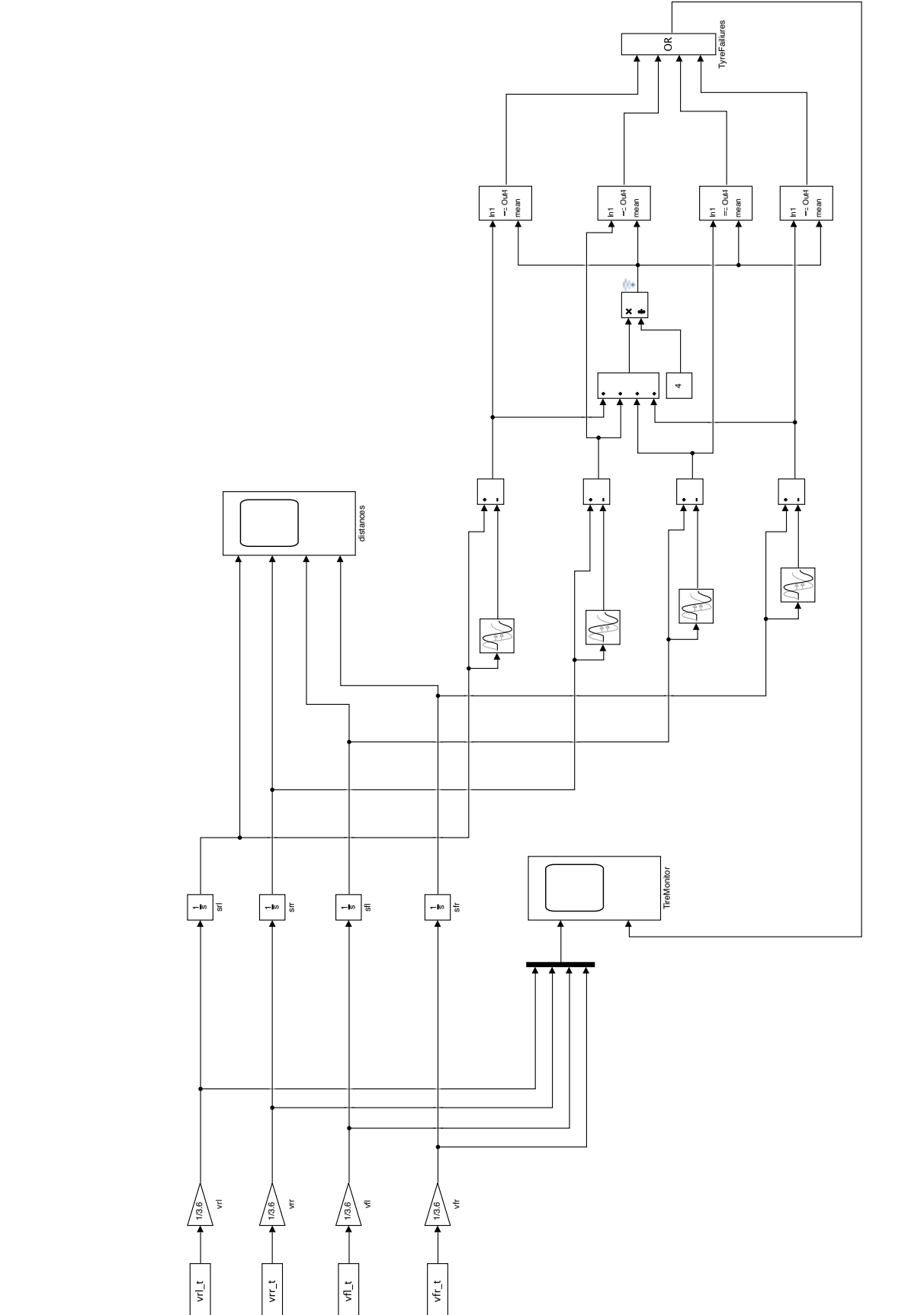


Abbildung 2.4: Simulink Tire Distance Model with Curves Data

Für die Kurvenfahrt entsteht die folgende Datenaufzeichnung über die Reifendruckabweichungen.

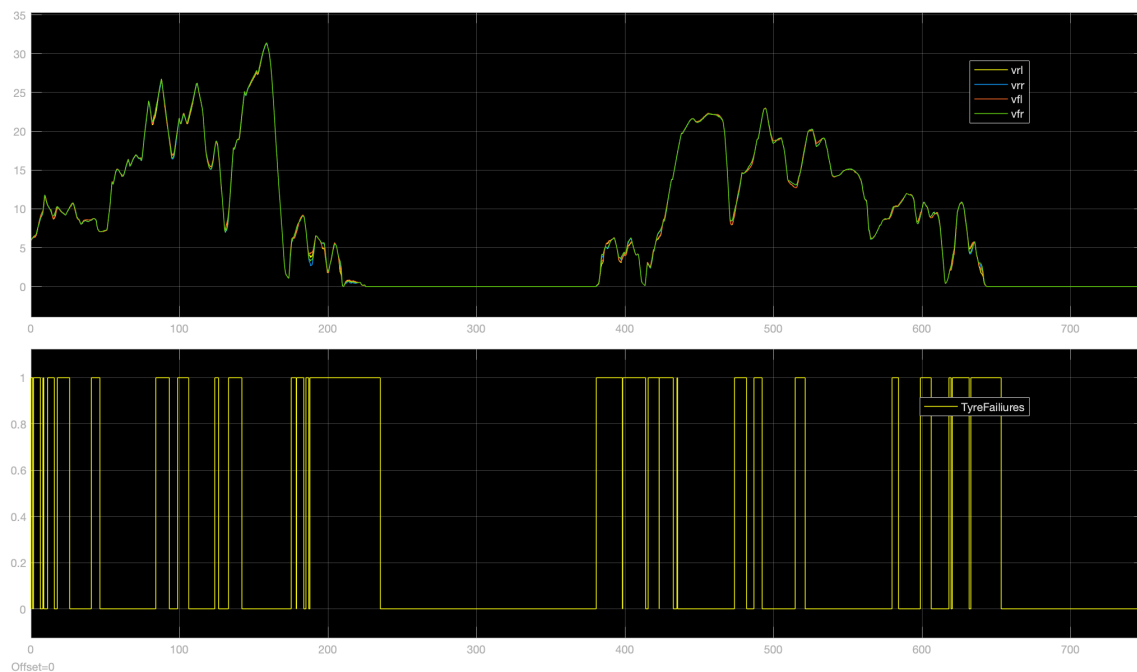


Abbildung 2.5: Tire Monitor für Kurvenfahrt

Ein Wert von 1 entspricht einer Beobachtung einer Imbalance und ein Wert von 0 entspricht keiner Anomalie.

3 Aufgabe D4

4 Aufgabe D5

Die untenstehende [Abbildung 4.1](#) zeigt den "Random Number Generator".

Die Daten werden nach der Vorschrift

$$X(i) = (a * X(i - 1) + c) \mod m$$

generiert.

Die Parameter a, c und m sowie $X(0)$ sind hierfür frei wählbar, um optimale Zufallszahlen zu erhalten.

Dafür sind einige Vorschriften zu beachten:

Modul $m \in \{2, 3, 4, \dots\}$

Faktoren $a_1, \dots, a_n \in \{0, \dots, m - 1\}$ mit $a_n > 0$

Inkrement $b \in \{0, \dots, m - 1\}$

Startwerte $y_1, \dots, y_n \in \{0, \dots, m - 1\}$ (nicht alle 0, wenn $b = 0$)

Nach diesen Vorschriften wurde $a = 253$ und $b = 89$ gewählt. Beide Zahlen sind Primzahlen und teilen somit keinen gemeinsamen Teiler mit $m = 2^{31}$. m als Zweierpotenz zu wählen ist besonders effizient, da so die Modulooperation durch das abscheiden der Zahl in Binärdarstellung berechnet werden kann.

Zur Generierung von 4 Datensätzen werden unterschiedliche Startwerte $X(0)$ gewählt. Die Rechenoperation wurde darüber hinaus noch ergänzt. Wie in [Abbildung 4.1](#) dargestellt wird die nach der Generierung der Zufallszahl nach oben genannter Vorschrift, noch die Hälfte von m subtrahiert, um auch negative Zufallswerte zu erhalten. Des Weiteren wird danach die Zufallszahl noch mit einem Wert multipliziert, um die Amplitude, um die die Zufallszahlen um 0 oszillieren zu setzen. Abschließend wird noch ein Zielwert addiert, um die Zufallszahlen um einen Zielwert oszillieren zu lassen.

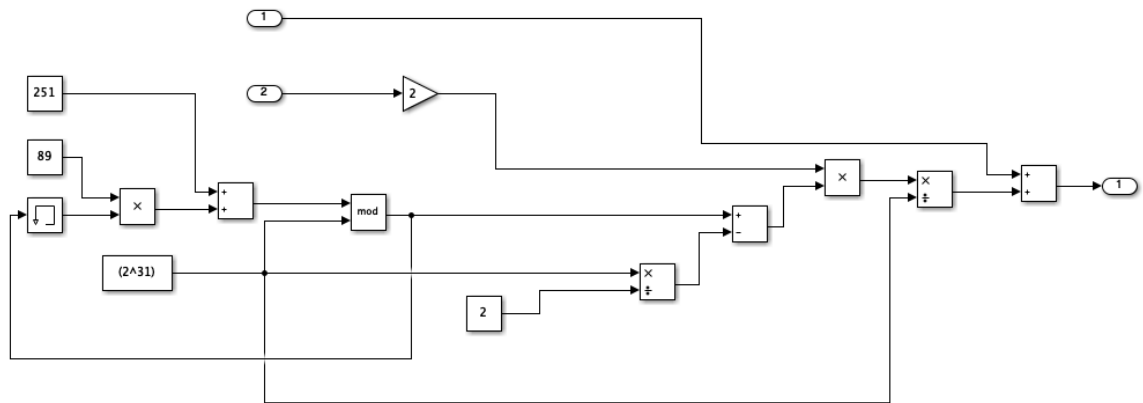


Abbildung 4.1: Random Number Generator

5 Aufgabe D6

Statt der aufgezeichneten Daten werden nun die Geschwindigkeiten aus dem Randomgenerator genutzt, um das Modell zu prüfen.

Eingestellt werden kann für den Randomgenerator zum einen die Basisgeschwindigkeit, um die durch den Randomgenerator Noise generiert werden kann. Sowie das Noiselevel/Amplitude, das das generierte Signal haben soll. Im Folgenden ist ein Ausschnitt zu sehen, wie der Randomgenerator in das bestehende Modell integriert wird, sowie die erzeugten Daten des Randomgenerators und ob es Imbalancen im System mit dem Randomgenerator gibt.

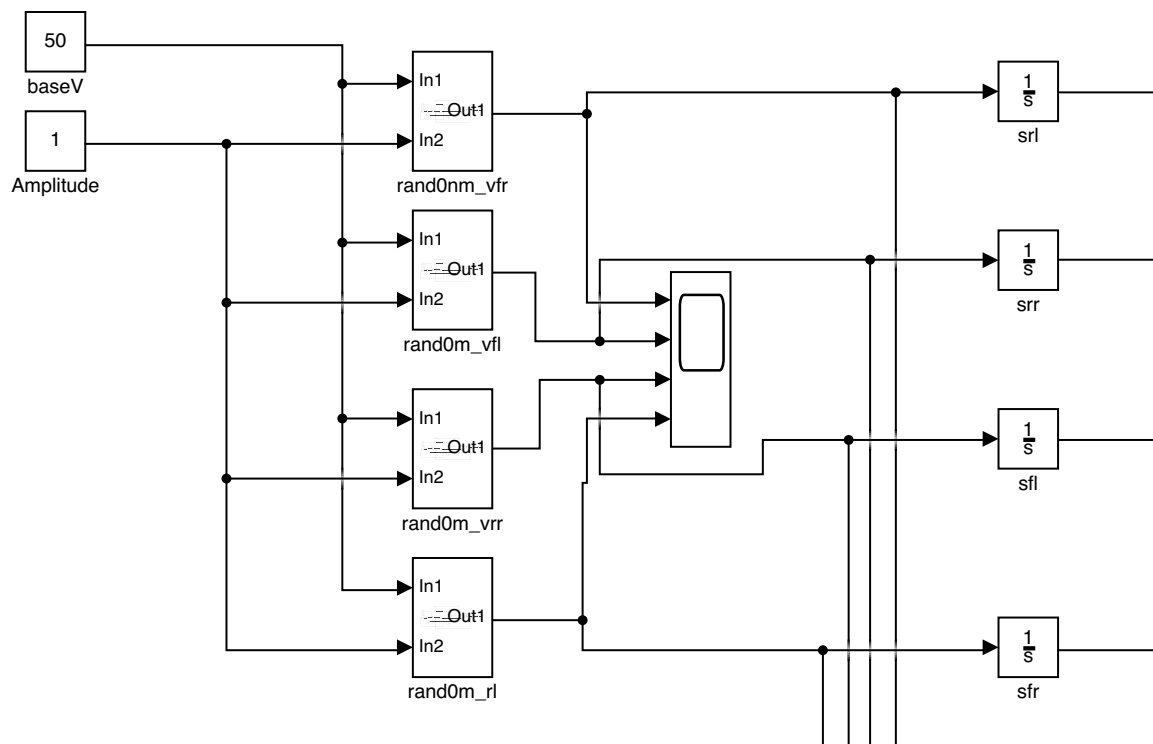


Abbildung 5.1: Include Random Generator

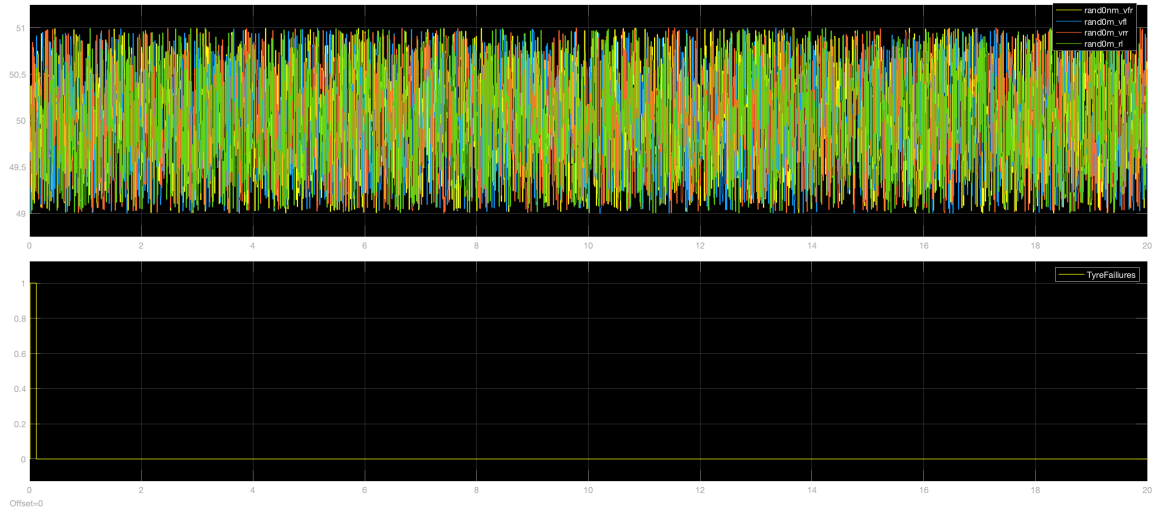


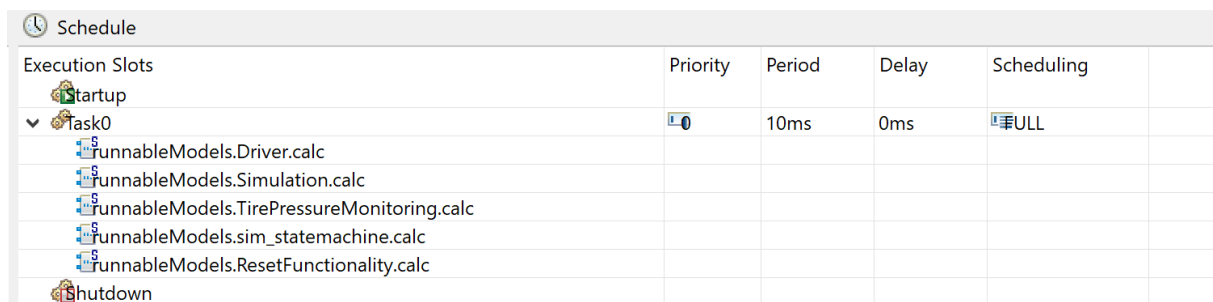
Abbildung 5.2: Generated Data

Es wird deutlich, dass es keine Imbalancen bei diesem Beispiel gibt, auch bei anderen Basisgeschwindigkeiten und anderen Noiseleveln, die einer realistischen Geradeausfahrt entsprechen treten keine Imbalancen auf.

Damit ist der konstruierte Randomgenerator gut geeignet, um Geradeausfahrten zu simulieren.

6 Aufgabe D7

Die folgenden Graphiken zeigen die Umsetzung der Simulink-Simulation in ASCET. Die



Execution Slots	Priority	Period	Delay	Scheduling
startup				
Task0	0	10ms	0ms	FULL
runnableModels.Driver.calc				
runnableModels.Simulation.calc				
runnableModels.TirePressureMonitoring.calc				
runnableModels.sim_statemachine.calc				
runnableModels.ResetFunctionality.calc				
shutdown				

Abbildung 6.1: ASCET Schedule

Simulink-Simulation ist in ASCET in drei Teile geteilt:

- Driver
- Simulation
- TirePressureMonitoring

Die weiteren Teile werden in folgenden Aufgaben weiter erläutert.

Driver

Der Driver ist ein Containermodell und enthält den Randomgenerator als Submodul.

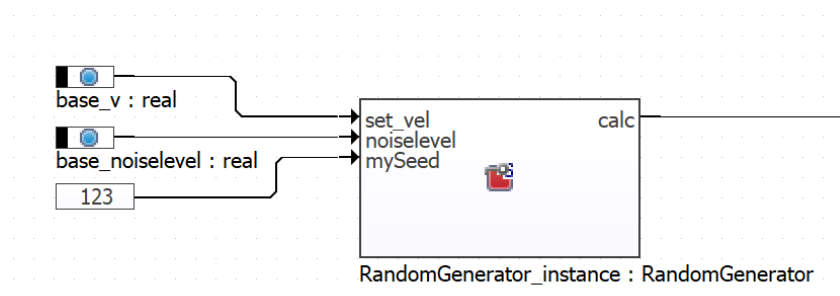


Abbildung 6.2: ASCET Randomgenerator

Dieser ist für jedes Rad einmal vorhanden. Eine Message sendet das jeweilige Signal weiter an andere Module.

Die Implementierung unterscheidet sich kaum von Simulink. Eine setSeed Funktionalität wurde ergänzt.

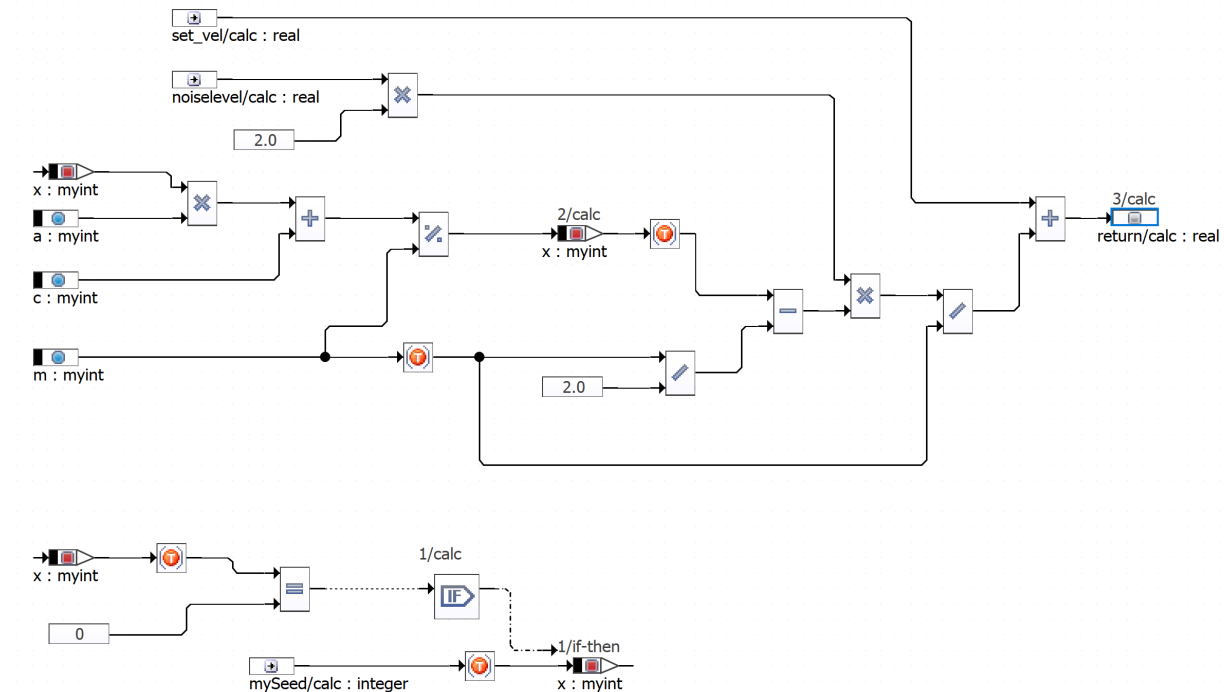


Abbildung 6.3: Random Number Generator

Es wird auch ein eigener Typ für die Modulooperation benötigt.

```
1 type myint is integer 0 .. 1024;
```

Die Parameter sind gleich implementiert. Bei der Codegenerierung mit $m = 2^{31}$ ist ein Fehler aufgetreten. Die Vermutung ist, dass dieser Wert zu groß ist. Deshalb wurde m auf 2^{10} gesetzt.

```
1 class RandomGenerator {
2   characteristic myint a = 89;
3   characteristic myint m = 1024;
4   characteristic myint c = 251;
5   myint seed;
6   myint x = 0;
7   @generated("blockdiagram")
8   public real calc(real in set_vel, real in noiselevel, integer in mySeed)
9   {
10   ...
11 }
```

Simulation

Die Simulation berechnet, die Distanzen und Deltas in einem Submodul und schickt diese dann per Message weiter.

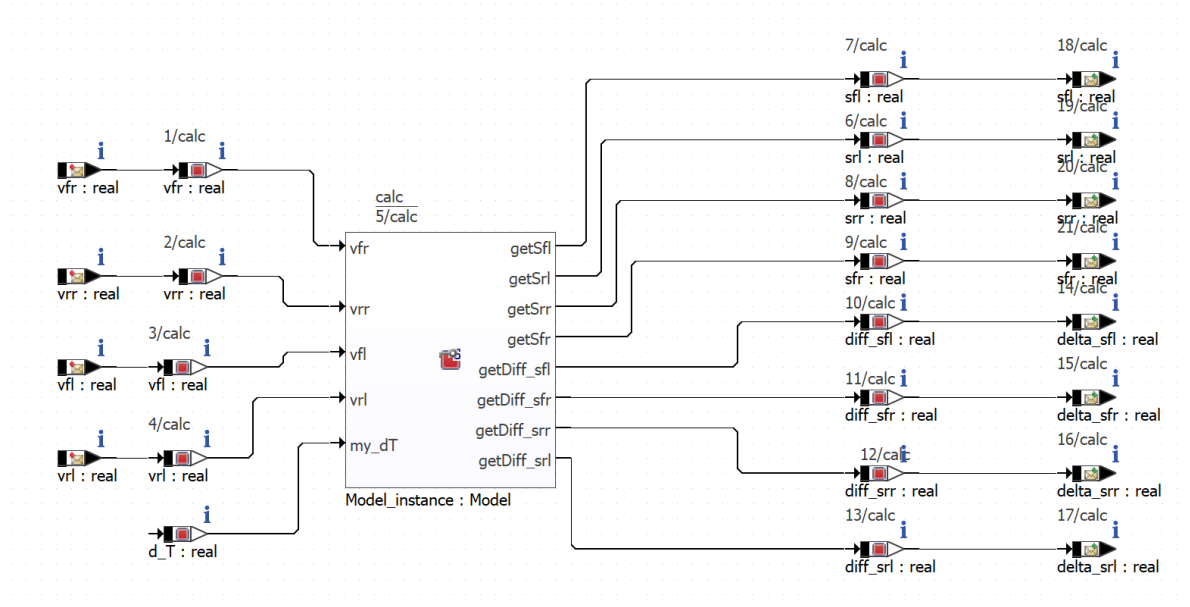


Abbildung 6.4: Simulation

Im Submodul wird die Strecke durch Integration mit dem dT-Element für alle vier Räder berechnet.

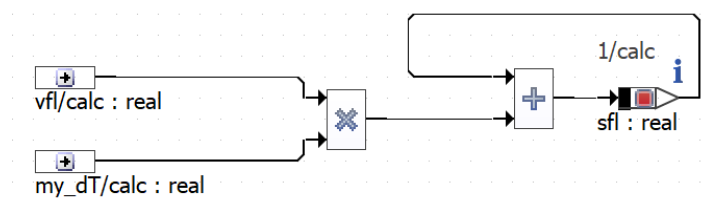


Abbildung 6.5: Streckenberechnung für Vornelinks

Um das 10 Sekundendelta zu erhalten wird der gespeicherte Wert von vor 10 Sekunden abgezogen (auch für alle vier Reifen).

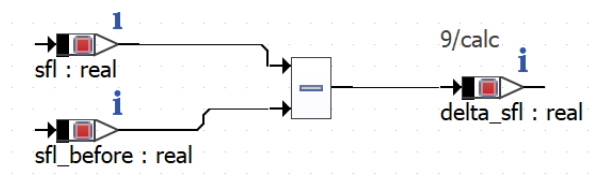


Abbildung 6.6: Deltaberechnung

Das Speichern vergangener Werte wird durch einen Buffer umgesetzt. Jeder Reifen besitzt einen eigenen Buffer.

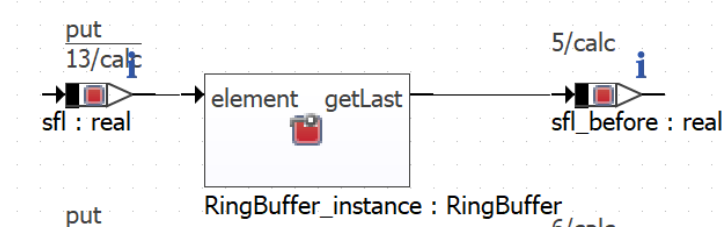


Abbildung 6.7: Buffer

Der Buffer wird als Code implementiert.

```

1 package components;
2 type s_array is array [] of real;
3
4 class RingBuffer{
5
6     s_array buffer[1000];
7     real c;
8     real swap;
9
10    public void put(real element){
11        swap = element;
12        for(i in 0 .. 999){
13            c = buffer[i];
14            buffer[i] = swap;
15            swap = c;
16        }
17    }
18
19    public real getLast(){
20        return buffer[999];
21    }
22
23    public real getIndex(integer i){
24        return buffer[i];
25    }
26 }

```

Die Codenotation ist einfacher zu implementieren und für diesen Fall auch übersichtlicher. Um die berechneten Größen im Topmodul zugänglich zu machen, werden für die Strecken und Deltas getter-Funktionen im Code implementiert.

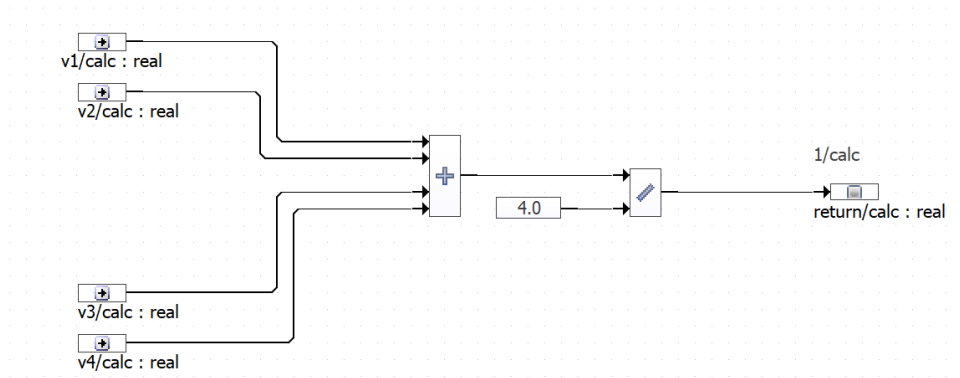


Abbildung 6.11: Mittelwertberechnung

Die Abweichungsberechnung unterscheidet sich ebenfalls nicht vom SIMulink Modell und ist ebenfalls wie in ?? in einem Submodul wie folgt umgesetzt.

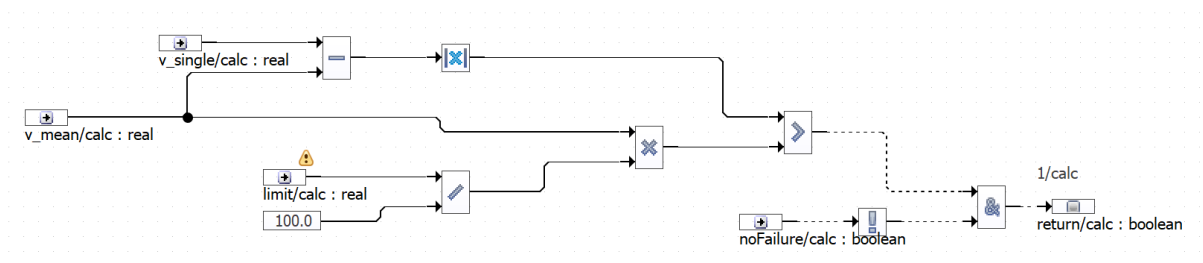


Abbildung 6.12: Mittelwertberechnung

7 Aufgabe D8

Hier werden die Codes der Unittests für die Funktionalitäten dargestellt.

7.1 Model

```
1 package components;
2 import assertLib.Assert;
3 import components.Globals;
4 import components.Model;
5 static class ModelTest {
6     real myDT = Globals.d_T;
7     real vfr;
8     real vrr;
9     real vfl;
10    real vrl;
11    Model m;
12
13
14    @Test
15    public void distanceIncreasing(){
16        myDT = 1.0;
17        vfr = 10.0;
18        vrr = 10.0;
19        vfl = 10.0;
20        vrl = 10.0;
21        m.calc(vfr, vrr, vfl, vrl, myDT);
22        Assert.assertTrue(m.getSfl() > 0.0);
23        Assert.assertTrue(m.getSrl() > 0.0);
24        Assert.assertTrue(m.getSfr() > 0.0);
25        Assert.assertTrue(m.getSrr() > 0.0);
26    }
27
28    @Test
29    public void predictDistance(){
30        myDT = 1.0;
31        vfr = 10.0;
```

```

32     vrr = 10.0;
33     vfl = 10.0;
34     vrl = 10.0;
35     m.calc(vfr, vrr, vfl, vrl, myDT);
36 }
37
38 @Test
39 public void deltasConsistentForSameVelocity(){
40     real s_return;
41     real buffer_time = 10.0;
42     myDT = 0.01;
43     vfr = 10.0;
44     vrr = 10.0;
45     vfl = 10.0;
46     vrl = 10.0;
47
48     for(i in 1 .. 50000){
49         m.calc(vfr, vrr, vfl, vrl, myDT);
50     }
51     Assert.assertNear(m.getDiff_sfr(), vfr*buffer_time, 0.1);
52     Assert.assertNear(m.getDiff_sfl(), vfl*buffer_time, 0.1);
53     Assert.assertNear(m.getDiff_srl(), vrl*buffer_time, 0.1);
54     Assert.assertNear(m.getDiff_srr(), vrr*buffer_time, 0.1);
55 }
56 }

```

7.2 Error Module

```

1 package components;
2 import assertLib.Assert;
3 import components.Globals;
4 import components.Model;
5 static class ErrorModuleTest {
6     ErrorModule cd;
7
8     @Test
9     public void deflation(){
10         real v = 10.0;
11         real error = 0.5;
12         real deflated_v = cd.calc(true, v, error);
13         Assert.assertNear(deflated_v, v*(1.0-(error/100.0)), 0.1);
14     }
15
16     @Test
17     public void noDeflation(){

```

```
18     real v = 10.0;
19     real error = 0.5;
20     real same_v = cd.calc(false, v, error);
21     Assert.assertNear(same_v, v, 0.1);
22 }
23 }
```

7.3 Ring Buffer

```
1 package components;
2 import assertLib.Assert;
3 import components.Globals;
4 import components.Model;
5 static class RingBufferTest {
6     RingBuffer rb;
7
8
9     @Test
10    public void start_with_empty_buffer(){
11        for(i in 0 .. 999){
12            Assert.assertNear(rb.getIndex(i), 0.0, 0.01);
13        }
14    }
15
16    @Test
17    public void fill_buffer_decending(){
18        real fill_element = 0.0;
19        integer buffer_maxIndex = 999;
20        for(i in 0 .. 999){
21            rb.put(fill_element);
22            fill_element = fill_element + 1.0;
23        }
24        for(n in 0 .. 999)
25        {
26            Assert.assertNear(rb.getIndex(n), real(buffer_maxIndex), 0.01);
27            buffer_maxIndex = buffer_maxIndex - 1;
28        }
29    }
30
31    @Test
32    public void read_last_element(){
33        real fill_element = 0.0;
34        for(i in 0 .. 999){
35            rb.put(fill_element);
36            fill_element = fill_element + 1.0;
```

```

37     }
38     Assert.assertNear(rb.getIndex(999),real(0),0.01);
39 }
40 }

```

7.4 SOS State

Für die Statemachine „SOS State“ wurden drei Unittests durchgeführt.

In „checkAllStatelocationsAndStatesActiveContinues“ wird der Parameter `active = true` gesetzt und bleibt über den gesamten Test auf `true`. Ist `active = true` so ist ein Fehler aufgetreten und der Off-State wird verlassen. In diesem Test werden alle Stati nacheinander durchlaufen, wie in R3 beschrieben. Nach jedem Statuswechsel wird anhand der Zeit überprüft, ob gerade der richtige Status aktiv ist. Nachdem der Zyklus einmal durchlaufen wurde, wird getestet, ob der Zyklus wieder von vorne beginnt, da `active` immer noch auf `true` ist.

In „checkAllStatelocationsActiveContinuesNot()“ wird der ganze Zyklus durchlaufen. Währenddessen wird `active` auf `false` gesetzt. Am Ende wird überprüft, ob man sich nach Abschluss des Zyklus im Off-Status befindet.

In „checkAllStatesDeactiv()“ ist der Parameter `active=false`. Es wird hier nach einer bestimmten Zeit überprüft, ob der Off-Status aktiv ist.

```

1 package components;
2 import assertLib.Assert;
3 import components.Globals;
4 import components.SOS_state;
5 static class Test_SOS_state {
6     integer counter = 0;
7     SOS_state sos;
8
9     //Der Wechsel eines States beansprucht 10ms zusätzlich
10    //wurde in die Berechnung der Zeit für die Stati aufgenommen
11    @Test
12    public void checkAllStatelocationsAndStatesActiveContinues(){
13        ...
14    }
15
16    @Test
17    public void checkAllStatelocationsActiveContinuesNot(){
18        ...
19    }
20
21    @Test

```

```

22 public void checkAllStatesDeactiv(){
23     ...
24 }
25 }

```

7.5 Tire Deviation

```

1 package components;
2 import assertLib.Assert;
3 import components.Globals;
4 import components.Model;
5 static class TireDeviationTest {
6     TireDeviation td;
7
8     @Test
9     public void noDeviationWhileCalibration(){
10         real v = 100.04;
11         real v_mean = 100.0;
12         real limit = 0.05;
13         boolean noFailure = true;
14         Assert.assertTrue(td.calc(v, v_mean, limit, noFailure) == false);
15     }
16
17     @Test
18     public void highDeviationWhileCalibration(){
19         real v = 100.06;
20         real v_mean = 100.0;
21         real limit = 0.05;
22         boolean noFailure = true;
23         Assert.assertTrue(td.calc(v, v_mean, limit, noFailure) == false);
24     }
25
26     @Test
27     public void lowDeviationWhileCalibration(){
28         real v = 99.94;
29         real v_mean = 100.0;
30         real limit = 0.05;
31         boolean noFailure = true;
32         Assert.assertTrue(td.calc(v, v_mean, limit, noFailure) == false);
33     }
34
35     @Test
36     public void noDeviation(){
37         real v = 100.04;
38         real v_mean = 100.0;

```

```

39     real limit = 0.05;
40     boolean noFailure = false;
41     Assert.assertTrue(td.calc(v, v_mean, limit, noFailure) == false);
42 }
43
44 @Test
45 public void highDeviation(){
46     real v = 100.06;
47     real v_mean = 100.0;
48     real limit = 0.05;
49     boolean noFailure = false;
50     Assert.assertTrue(td.calc(v, v_mean, limit, noFailure) == true);
51 }
52
53 @Test
54 public void lowDeviation(){
55     real v = 99.94;
56     real v_mean = 100.0;
57     real limit = 0.05;
58     boolean noFailure = false;
59     Assert.assertTrue(td.calc(v, v_mean, limit, noFailure) == true);
60 }
61 }

```

7.6 Tire Mean

```

1 package components;
2 import assertLib.Assert;
3 import components.Globals;
4 import components.Model;
5 static class TireMeanTest {
6     TireMean m;
7
8
9     @Test
10    public void calcMeanAllValuesTheSame(){
11        real v1 = 50.0;
12        real mean = m.calc(v1, v1, v1, v1);
13        Assert.assertNear(mean, v1, 0.1);
14    }
15
16    @Test
17    public void calcMean(){
18        real v1 = 50.0;
19        real v2 = 15.0;

```

```
20     real v3 = 75.0;
21     real v4 = 100.5;
22     real mean = m.calc(v1, v2, v3,v4);
23     Assert.assertNear(mean, 60.13, 0.1);
24 }
25
26 @Test
27 public void calcMeanNegativ(){
28     real v1 = -50.0;
29     real v2 = -15.0;
30     real v3 = -75.0;
31     real v4 = -100.5;
32     real mean = m.calc(v1, v2, v3,v4);
33     Assert.assertNear(mean,-60.13, 0.1);
34 }
35
36 @Test
37 public void calcMeanMixed(){
38     real v1 = -50.0;
39     real v2 = -15.5;
40     real v3 = 50.0;
41     real v4 = 15.5;
42     real mean = m.calc(v1, v2, v3,v4);
43     Assert.assertNear(mean,0.0, 0.1);
44 }
45 }
```

8 Aufgabe D9

Für die Umsetzung der Warnfunktion durch eine LED, die einen Druckabfall-/anstieg signalisiert, wurde eine Statemachine verwendet. Diese besteht aus insgesamt fünf Stati, vier die jeweils 0,8 bzw. 1,6s dauern und die LED ein- bzw. ausschalten und dem Off-Status.

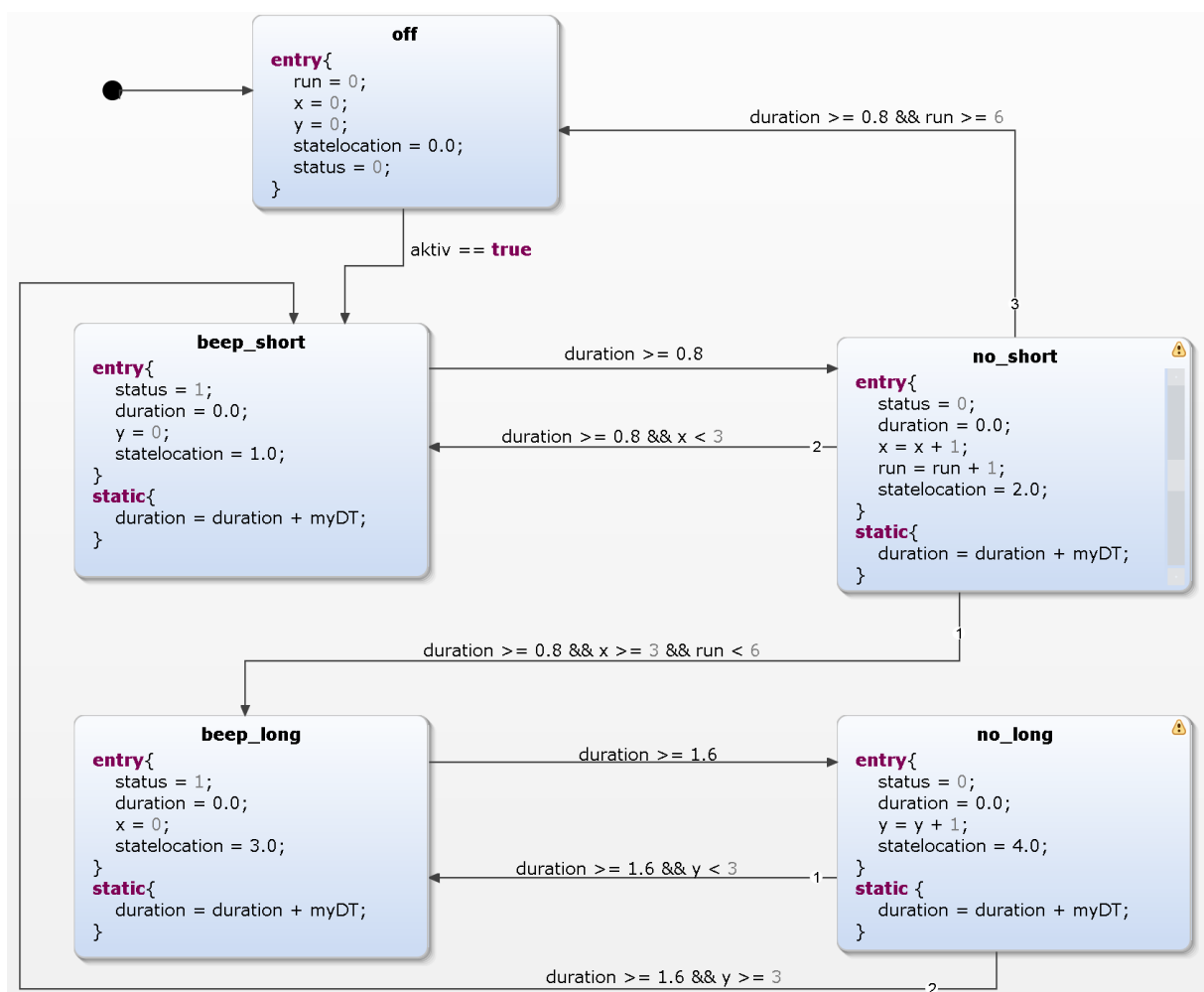


Abbildung 8.1: SOS Statemachine

9 Aufgabe D10

Der Code des Unittests für den „Random Number Generator“ wird im Folgenden dargestellt.

```
1 package components;
2 import assertLib.Assert;
3 static class RandomGeneratorTest {
4
5     RandomGenerator rg;
6     RingBuffer rb;
7
8     @Test
9     public void generateFirstDatastep() {
10         real set_vel = 100.0;
11         real noiselevel = 5.0;
12         integer mySeed = 10;
13         real erg = rg.calc(set_vel, noiselevel, mySeed);
14         Assert.assertNear(erg, 96.1425781, 0.01);
15     }
16
17     @Test
18     public void generateDifferentNumbers() {
19         real set_vel = 100.0;
20         real noiselevel = 5.0;
21         integer mySeed = 10;
22         for(i in 0 .. 99){
23             real erg = rg.calc(set_vel, noiselevel, mySeed);
24             rb.put(erg);
25             for(j in 1 .. 999){
26                 real act = rb.getIndex(0);
27                 Assert.assertNotEqual(act, rb.getIndex(j));
28             }
29         }
30     }
31 }
```

10 Aufgabe D11

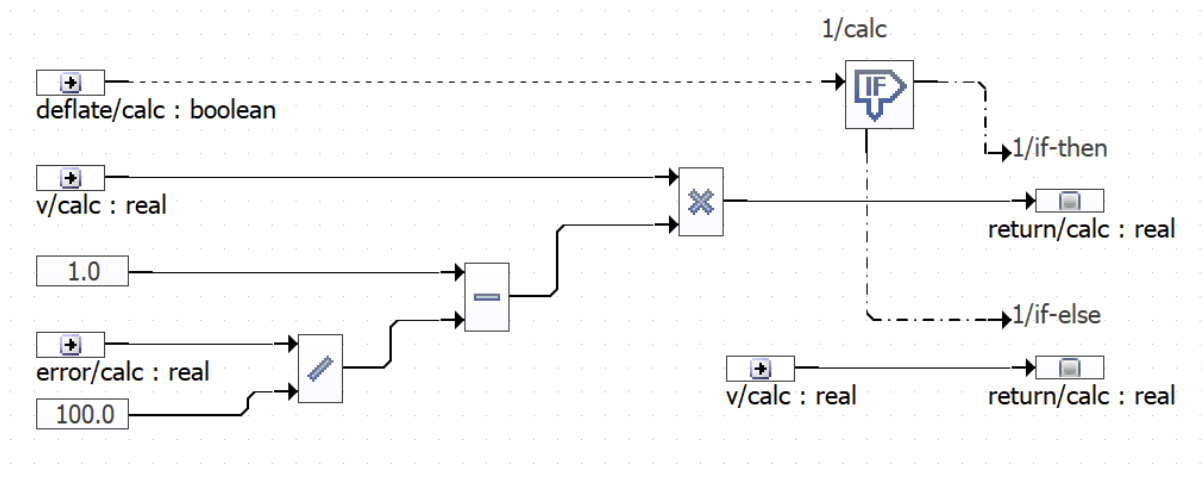


Abbildung 10.1: Error Module

11 Aufgabe D12

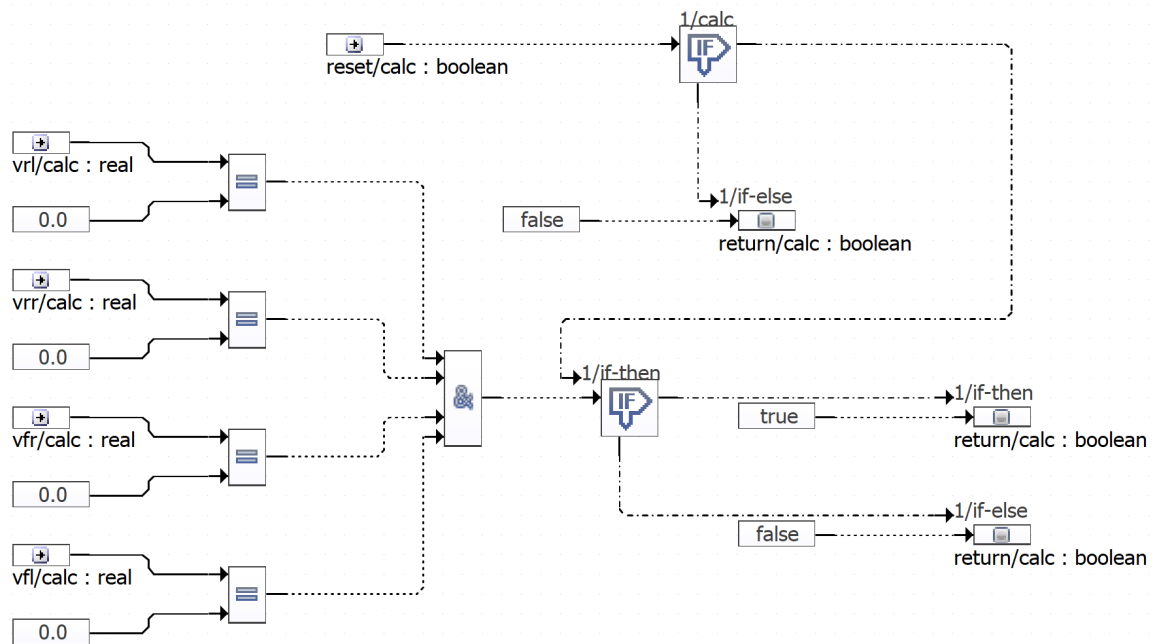


Abbildung 11.1: Reset

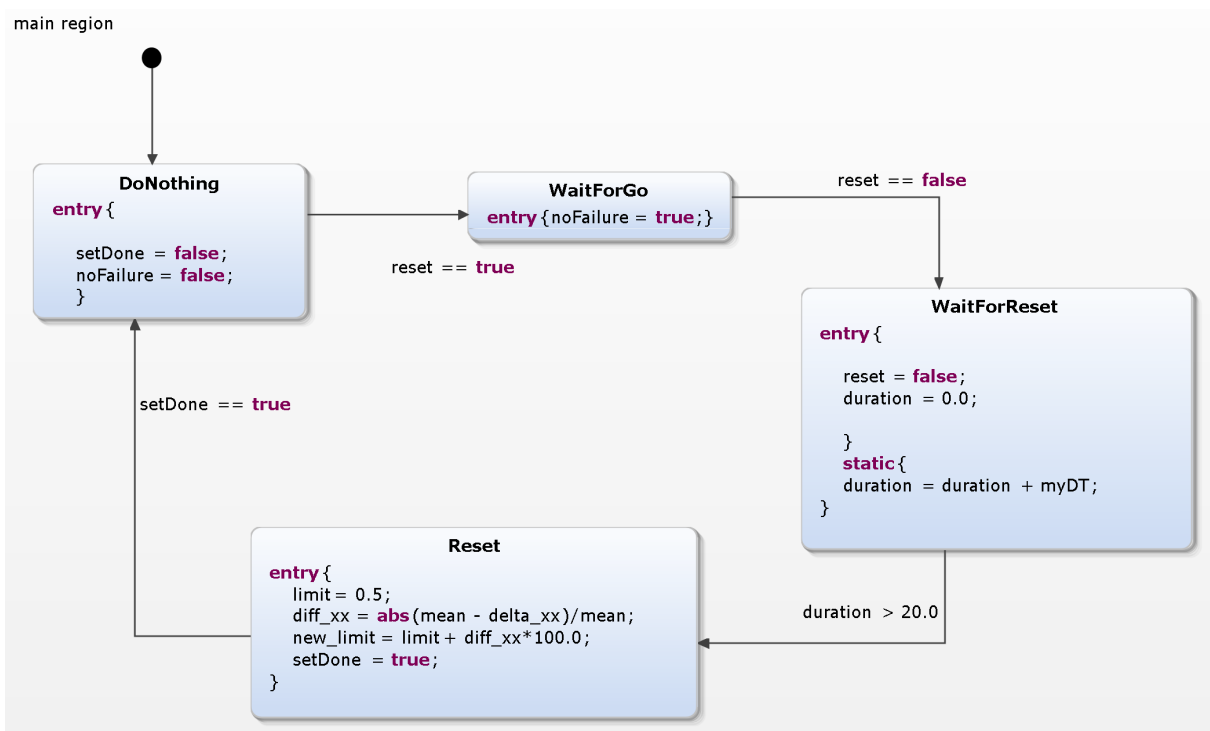


Abbildung 11.2: Reset Statemachine