FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

FH Aachen

Fachbereich
Elektrotechnik und Informationstechnik

Masterarbeit

Der Titel der Arbeit ist zweizeilig

Vorname Nachname
Matr.-Nr.: 123456

Referent: **Prof. Dr-Ing. ...**

Korreferent: **Prof. Dr.-Ing. ...**

May 27, 2020

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Aachen, May 27, 2020

Danksagung

Danke.

Contents

1 Introduction **11**
1.1 Background 11
1.1.1 Railway Vehicle Operations 11
1.1.2 Train Protection Systems 11
1.1.3 Braking Curves 11
1.2 Problem 11
1.3 Solution 11

2 Fundamentals of Railway Vehicle Engineering **13**

3 Modeling of Train Operations **15**
3.1 Initial Model 15
3.2 Model Expansion 17
3.3 Further Expansion 21

4 Data Generation **23**
4.1 Matlab Code 23
4.2 Data Structure 28
4.3 Analysis of generated Data 29

5 Performance Analysis **31**

6 Conclusion **33**

Abbildungsverzeichnis **34**

Tabellenverzeichnis **36**

Anhang **37**

A Quellcode **39**

B Data visualization **41**

Abstract Modern day railway system operations require automated train control mechanisms, e.g. European Train Control System ETCS, to maximize efficiency, which is often times limited by outdated infrastructure, as well as safety of operations. One way to achieve this is by lowering the required distance between two trains on the same track, which in turn demands a reliable method of predicting the braking distance at any given moment.

While determination of the necessary braking curves is feasible for a limited number of train formations, the large diversity of vehicles in freight operations poses an issue. One approach for a solution would be using Big Data, which would be able to process the required amounts of data to calculate reliable braking curves even for freight operations.

The problem here is there is simply not enough data available since freight trains usually don't have the sensory equipment needed. To circumvent that obstacle, this work proposes generation of artificial data via white box modeling to be then used in further big data operations.

Introduction

Introduction This section describes the background and motivation of the research (Sect. 1.1), the problem to be addressed (Sect. 1.2) and the proposed solution (Sect. 1.3)

1.1 Background

1.1.1 Railway Vehicle Operations

1.1.2 Train Protection Systems

1.1.3 Braking Curves

1.2 Problem

As has been shown, to predict the braking behavior of trains, readings of wagons and locomotives are needed. Unfortunately, freight vehicles do not currently posses the sensory equipment that would be necessary to obtain such data in an adequate quantity and quality, especially in regards to *big data processing*. Although it has been proposed to equip freight wagons accordingly **<TODO: ref zu wagon4.0>**, it will be years before enough rolling stock has been retrofitted as to make it possible to obtain the desired data.

1.3 Solution

This work proposes to circumvent the problem described above by creation of an artificial data set. The set must replicate the actual distribution of braking behavior as close as possible. It is therefore necessary to first create a model encompassing the braking process of a freight train. This model will be discussed in depth in chapter 3. It can then, once finished, be also used to generate the data set by simulation. This process will be discussed in chapter 4.

As real life operations would yield very high quantities of data, the simulation output must be stored in a data structure which is suitable for big data processing. This structure will also be discussed in chapter 4.

Fundamentals of Railway Vehicle Engineering

Modeling of Train Operations

Introduction As has been noted in 1.3, it is necessary to model the braking process of freight trains. All modeling work has been performed with Matlab Simulink.

3.1 Initial Model

The initial model to be expanded upon describes a single braking process. It's sole input, apart from some constants, is pressure over time, meaning a distinct value ranging between 5 and 3.5 bar for every timestamp. For visualization, please refer to B. Let's take a look at the whole model first.

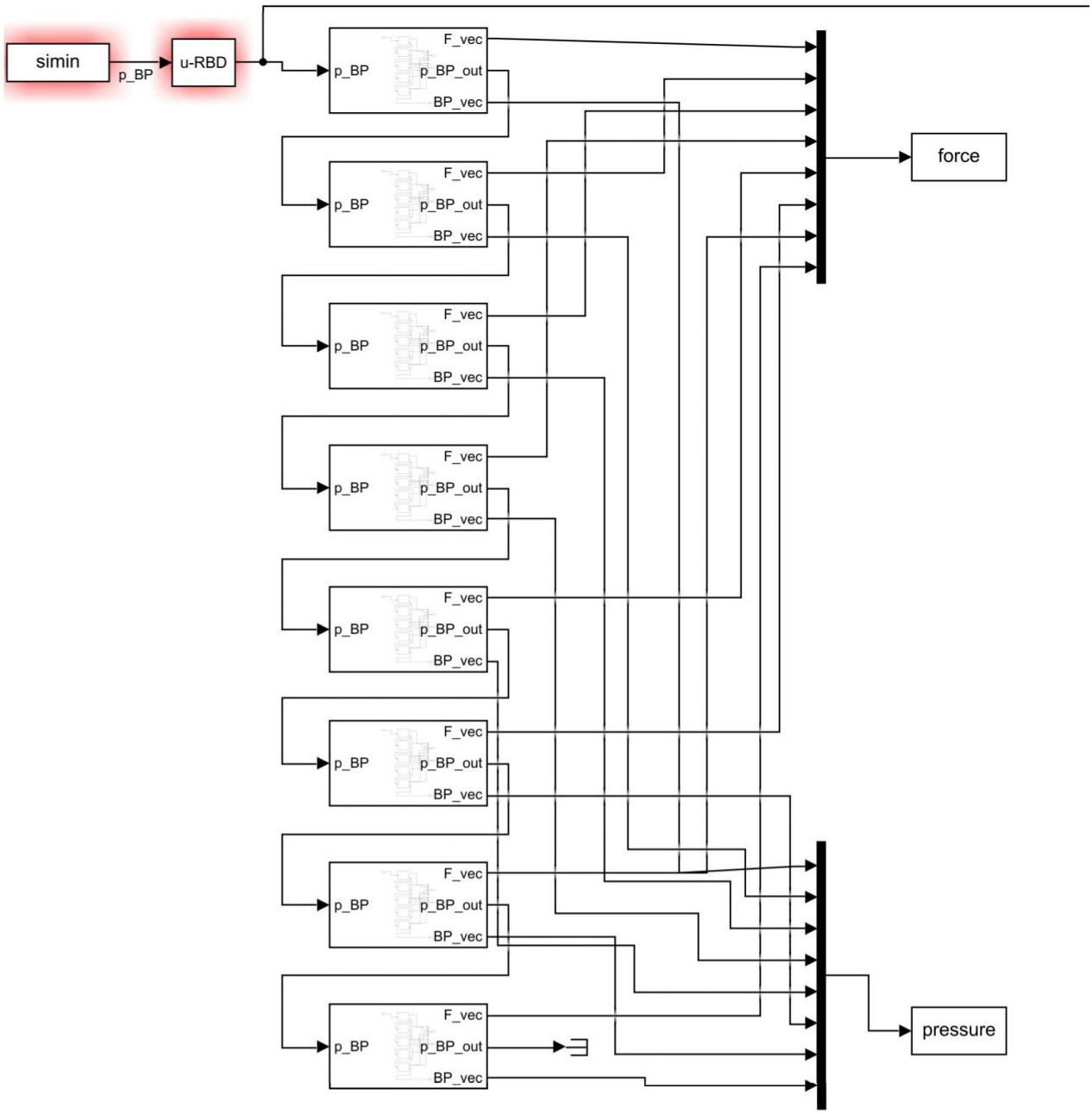

Figure 3.1: Initial Model

Here we see a model of a freight train of fixed length, consisting of 40 wagons, which are, for better readability, further condensed to subsystems of five wagons each, so there are eight of these subsystems. They are interconnected via braking pipe, which is also the sole input to each system. Outputs are braking pressure and braking force. We will take a look at the actual wagon model next.

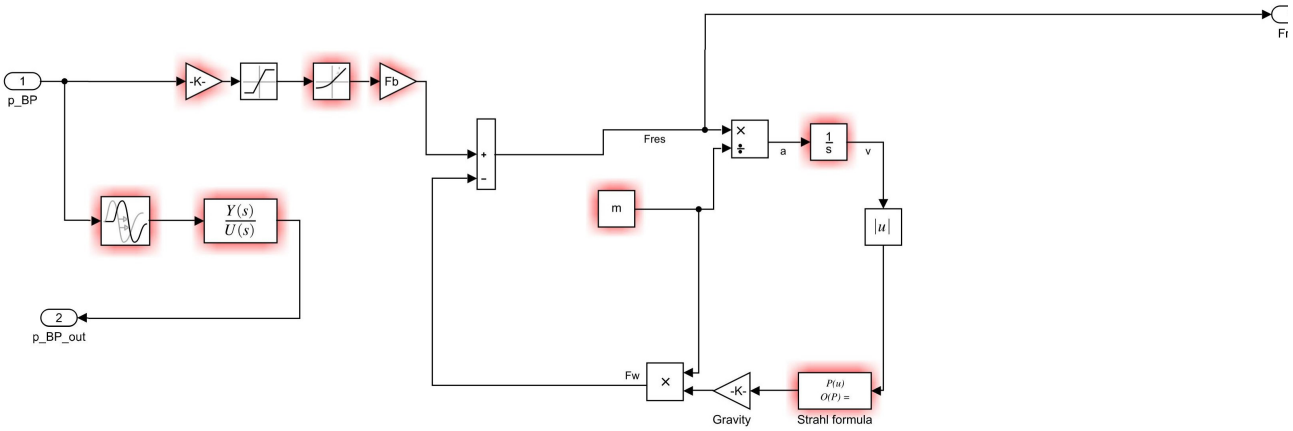

Figure 3.2: Initial Model - Wagon

Above is the initial wagon model. NOTE: All 40 wagon models are identical here. This will be addressed in section 3.2. It consists of three main components.

In the upper left corner is the input, which is the current pressure in the braking pipe. In the lower left corner, the propagation delay of the braking pipe is calculated. This is done by <**TODO:** >. Top center describes the calculation of the actual braking force, which is achieved by <**TODO:** >. Finally, the <**TODO: Formulieren: Fahrzeugwiderstand**>.

3.2 Model Expansion

This initial model is however not of sufficient detail. Where it merely describes one single braking process, we need to simulate a whole ride, with alternating phases of braking and accelerating. For that purpose, the simulation input has to be adjusted accordingly. Where previously it was only one braking process, using braking pressure as input was the obvious choice, whereas now the idea is to use a kind of track profile, which shall describe the maximum allowed velocity over time, of a notional track. For visualization, please refer to B. The simulation then only needs to brake or accelerate depending on train velocity versus maximum velocity at the current time.

Accordingly, the first expansion step is to create a mechanism to control the train so to speak. For this purpose, the system simply checks for each timestamp whether the current velocity of the train is greater than the maximum allowed velocity at the current time, according to simulation input. If this is the case, a braking pressure is applied to the pipe, scaling with the difference between v_{max} and v_{real}, v_{dif}. This means the higher v_{dif} is, the more braking pressure gets applied. This more or less covers the braking part of the system.

The model however also needs a component for acceleration. To simplify things, the logic here is that if the train is not braking, it is accelerating, which actually works out pretty well. To accelerate, a traction force is applied, which also scales with v_{dif}, so the higher v_{dif}, the higher the applied traction force.

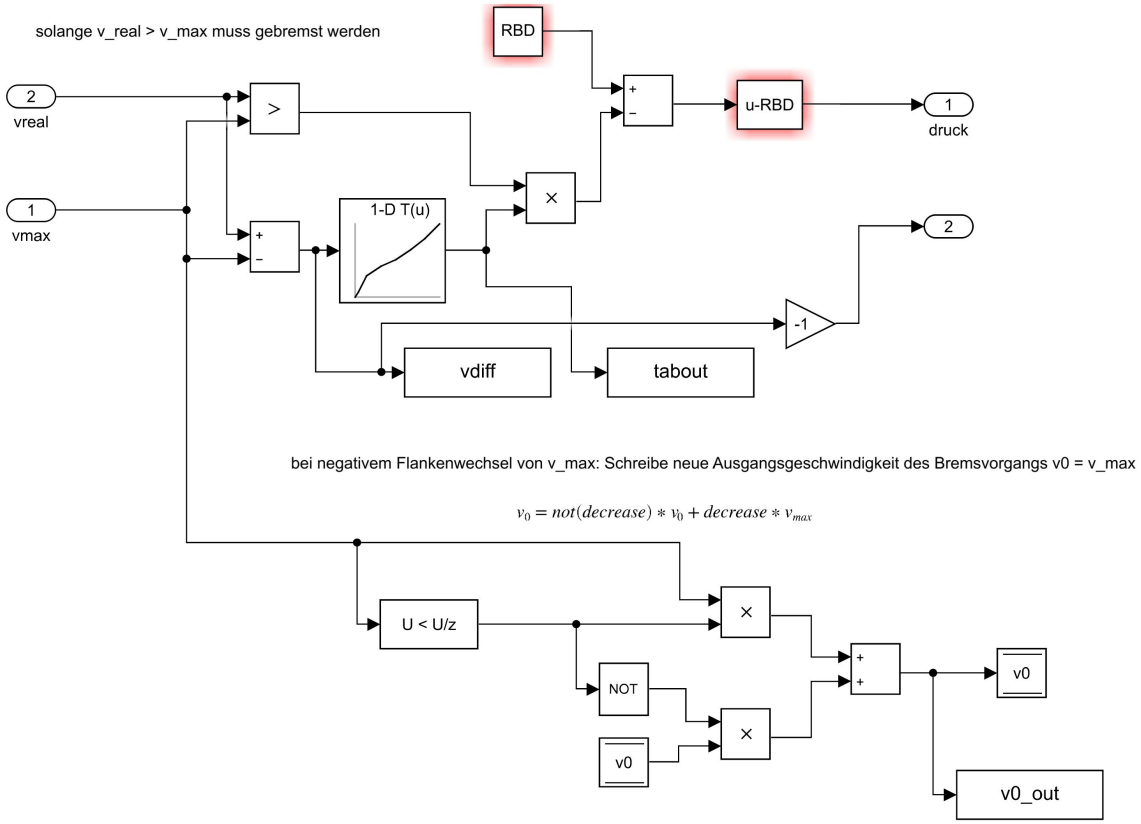

Figure 3.3: Expanded Model - Pressure Calculation

Depicted above is the system which determines braking pressure to apply. It calculates v_{dif} by subtracting v_{max} from v_{real}, which is then fed into a one-dimensional lookup table. The table is a sampled representation of a function with fixed breakpoints, mapping one function value to each breakpoint, like so

$$H(n) = \begin{cases} 0.1 & \text{if } n = 1 \\ 0.7 & \text{if } n = 15 \\ 0.8 & \text{if } n = 20 \\ .. \end{cases} \tag{3.1}$$

where n is the breakpoints of v_{dif}. Since the pressure should only be applied if v_{real} is greater than v_{max}, the ultimate result follows the logic of the following equation

$$P(n,t) = H(n) * (v_{real}(t) > v_{max}(t)) \tag{3.2}$$

where $v_{real}(t)$ is train velocity over time, $v_{max}(t)$ is maximum velocity over time, and $v_{real}(t) > v_{max}(t)$ is either 1 or 0.

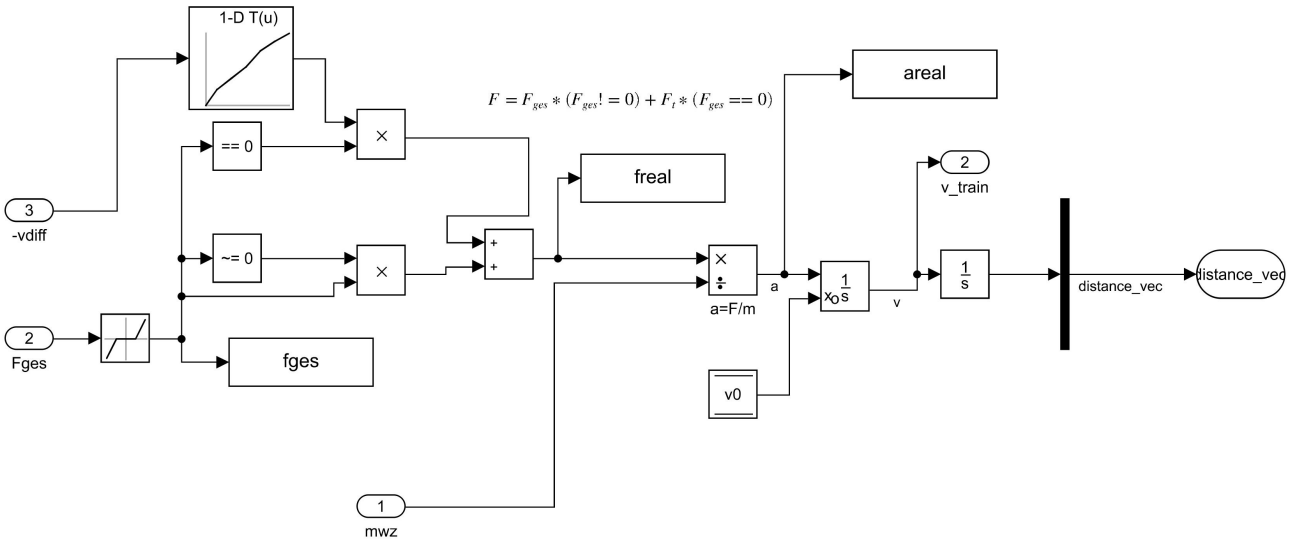

Figure 3.4: Expanded Model - Traction Force Calculation

The above system determines the traction force to apply. As has been discussed earlier, this works like a simple bang-bang controller. The design is very similar to the braking pressure system: v_{dif} is again fed into a one-dimensional lookup table, which outputs different values for traction force accordingly. The higher v_{dif} is, the higher the traction force to apply. It is then added to the current braking force, however only either traction or braking force is at any given time positive while the other is zero, which is achieved by the equations

$$f(n,t) = H(n) * (F_B(t) == 0) \tag{3.3}$$

where n is v_{dif}, $H(n)$ is the lookup table function (see equation 3.1), $F_B(t)$ is the braking force over time, and

$$g(t) = F_B(t) * (F_B(t) \neq 0) \tag{3.4}$$

where $F_B(t)$ is the braking force over time, so we have

$$F(n,t) = f(n,t) + g(t) \tag{3.5}$$

where F is the actual force over time, either braking or traction.

F is then used to calculate acceleration. According to Newton's Second Law,

$$F = m * a \tag{3.6}$$

Accordingly, acceleration is

$$a = F(n,t)/m \tag{3.7}$$

where m is the accumulated mass of all wagons and $F(n,t)$ relates to equation 3.5. The acceleration is then used to calculate the velocity by integrating a in relation to v_0, which is the initial velocity of the current braking or acceleration process **<TODO: überprüfen..>**. Integration of v in turn allows calculation of the traveled distance.

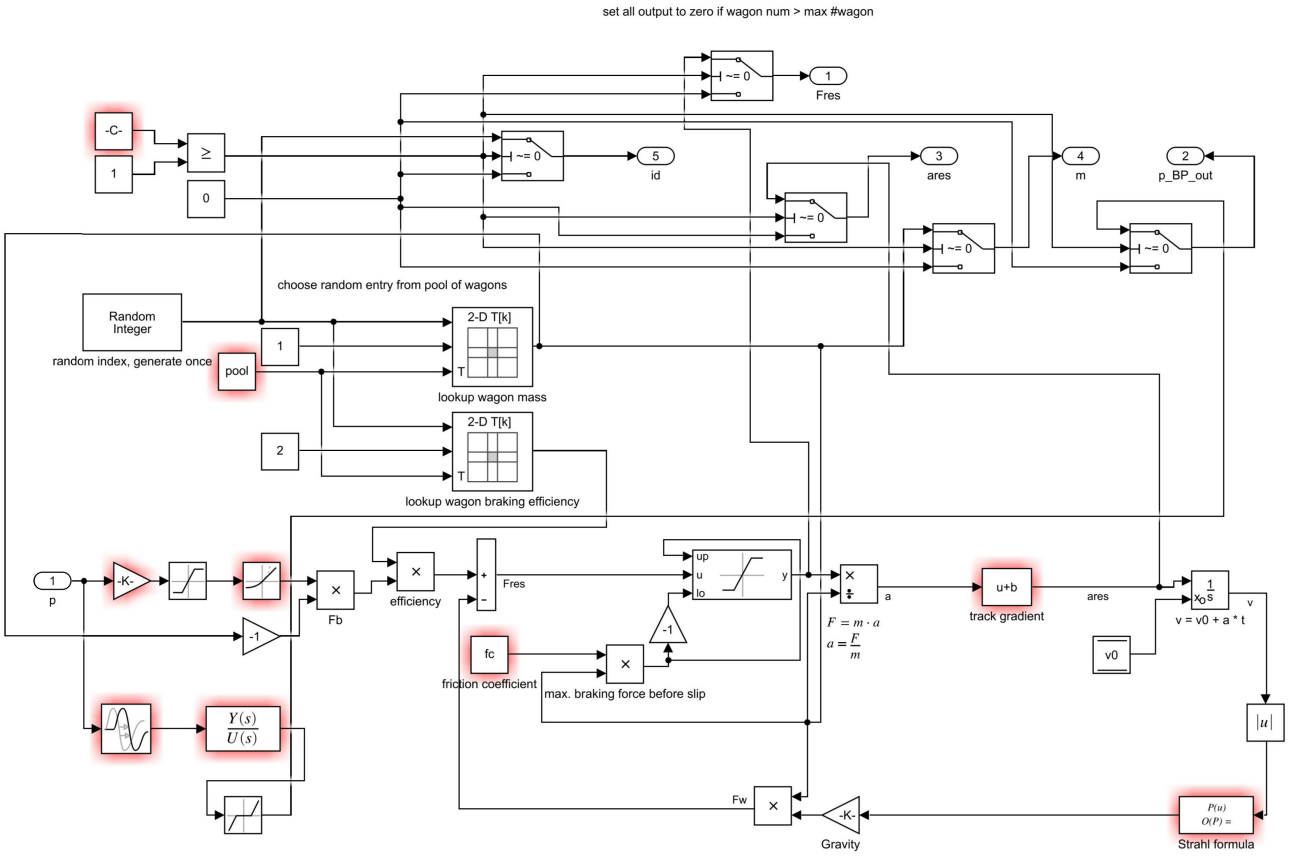

Figure 3.5: Expanded Model - Wagon

The last subsystem is the actual wagon. We will take a look at the largely unchanged elements first. The sole input is still the braking pressure on the brake pipe. Simulation of the propagation delay has also remained the same as before.

One new addition is a pool of different wagons. Whereas before all 40 were distinguishable only by their position, they are now assigned with different parameters. To that end, a pool of 500 wagons has been randomly generated via python script, where each wagon has a unique ID, as well as randomly generated mass and braking efficiency. In actual simulation, up to

40 of these 500 are, currently by generation of random indices, selected and their properties used accordingly. It would also be possible to determine the wagon ids to be used beforehand, instead of choosing randomly.

Another requirement was to make the number of wagons variable. In the initial model, the modeled train had a fixed number of 40 wagons, therefore also 40 wagon subsystems. Unfortunately, simulink offers no way to disable certain subsystems dynamically, but only by manually turning them off via model explorer, which would be unfeasible for such a large number of simulations. To circumvent this issue, output gets disabled for all unwanted wagons. For a simulation of a train of 20 wagons, the first 20 remain untouched, while the latter 20 produce no output and therefore also have no impact on the overall simulation. The turning off is achieved by simple switches; each wagon subsystem has a unique index from one to forty. If the index is greater than the specified number of wagons, all switches are turned to output zero.

3.3 Further Expansion

Data Generation

Introduction With the finished model, it is now possible to generate the actual data, via simulation. For this purpose, a matlab script initializes all relevant model parameters and feeds the simulation input, in this case the previously discussed track profile, into the model.

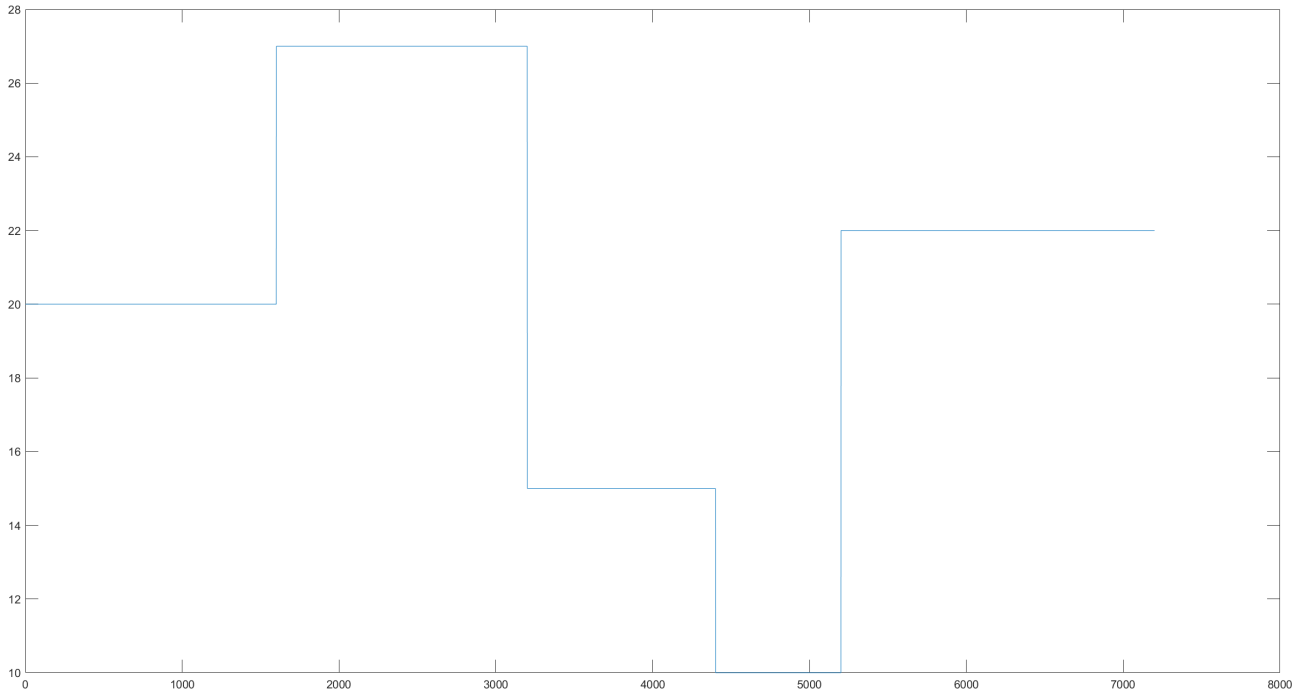

Figure 4.1: Simulation Input

The track profile, as depicted above, basically describes at which time which maximum velocity is allowed, and thus the train either brakes or accelerates to match that value at all times. What we can gather from this is that the simulation is time-based. Here we have 7200 data points as in input, which is 3600 seconds times two for better precision.

4.1 Matlab Code

We need to do some general configuration first, like clearing the workspace off all output which might still be present from previous simulations.

```
%% Initialise
```

```
clear all
clc
warning ('off','all');
numcores = 10; % configure number of cores to use in parallel processing
```

We then need to initialise all constants which values remain the same for all simulations. Please refer to the code snippet below for descpriptions:

```
%% Constants

RBD = 5; % regular operations pressure (bar)
VBD = 3.5; % full breaking pressure (bar)

l = 18; % wagon length (meters)
c = 250; % propagation velocity (km/h)

tf = 4; % brake cylinder filling time
tl = .1;

p0 = 0; % initial pressure
Pres = 0/1000*[5.7/771 0 1.6]; % strahl formula for m/s velocity

alpha = 0.9;

BPnum = [0.3 alpha];
BPden = [1 alpha];

%% Simulation input
tmax = 3600; % simulation time (seconds)
nmax = tmax * 2; % number of data points
t = linspace(0, tmax, nmax); % simulation time input
u = [20*ones(800*2,1); 27*ones(800*2,1); 15*ones(600*2,1); 10*ones(400*2,1);
   22*ones(900*2,1); 0*ones(100*2,1)]; % track profile
simin.time = t; % set simulation time input
simin.signals.values = u; % set simulation signal input

trackgradient = 0; % pitch angle
```

After having initialised all necessary constants, we now get to configuring variable simulation input.

```
pool = readmatrix('pool.csv'); % read wagon pool
```

```
mkdir output; % make output directory

wagons = 1:1:40; % create array for number of wagons to loop over
friction = 0.05:0.01:0.78; % create array for friction coefficient to loop over
tracforce = 200000:1000:400000; % create array for traction force to loop over

track = 1; % counter to index input array for one batch of simulations
allruns = 1; % counter to keep track of total number of simulations completed

for i = length(tracforce):-1:1   % all possible combinations of traction force,
   for j = length(friction):-1:1  % friction coefficient and
      for k = length(wagons):-1:1 % number of wagons
         % save all previously initialised constans and sim input into
         % one array in
         in(track) = Simulink.SimulationInput('Simulation_v2');
         in(track) = in(track).setVariable('num_wagons',wagons(k));
         in(track) = in(track).setVariable('fc',friction(j));
         in(track) = in(track).setVariable('Ft',tracforce(i));

         in(track) = in(track).setVariable('alpha',alpha);
         in(track) = in(track).setVariable('BPden',BPden);
         in(track) = in(track).setVariable('BPnum',BPnum);
         in(track) = in(track).setVariable('c',c);
         in(track) = in(track).setVariable('l',l);
         in(track) = in(track).setVariable('nmax',nmax);
         in(track) = in(track).setVariable('p0',p0);
         in(track) = in(track).setVariable('pool',pool);
         in(track) = in(track).setVariable('Pres',Pres);
         in(track) = in(track).setVariable('RBD',RBD);
         in(track) = in(track).setVariable('simin',simin);
         in(track) = in(track).setVariable('t',t);
         in(track) = in(track).setVariable('tf',tf);
         in(track) = in(track).setVariable('tl',tl);
         in(track) = in(track).setVariable('tmax',tmax);
         in(track) = in(track).setVariable('trackgradient',trackgradient);
         in(track) = in(track).setVariable('u',u);
         in(track) = in(track).setVariable('VBD',VBD);

         % also save the respective values for traction force and
         % friction coefficient into arrays
         Ft(track) = tracforce(i);
         fc(track) = friction(j);

         track = track + 1;
      end
   end
```

```
   fprintf('Starting␣pool␣for␣%d␣simulation␣runs.\n',length(in));
   parpool(numcores); % create a pool of #numcores workers for parallel processing
   out = parsim(in,'ShowProgress','on');
   % start parallel simulations for all
   % entries of in, save output to out

   matrix = []; % create an empty matrix

   parfor l = 1:1:length(out)
   % parallel loop over out, which is an array that
   % holds all simulation outputs

      tmp = Write(l + allruns,out(l).get('velocity'),out(l).get('force')
         ,out(l).get('pressure'),out(l).get('distance'),
         out(l).get('acceleration_neg'),out(l).get('ids'),t,u,
         trackgradient,Ft(l),fc(l)); % for each entry, pack all output into one single matrix
      matrix = [matrix;tmp]; % append to matrix
   end

   allruns = allruns + length(out); % update total number of simulations

   writematrix(matrix,'output/output.tsv','FileType','text','WriteMode','append',
      'Delimiter','tab');
   % write matrix, which holds all output matrices of the
   % current batch, to a tsv file

   track = 1; % reset batch index
   delete(gcp('nocreate')); % delete parallel pool
   fprintf('Run␣%d␣(of␣%d␣total)␣complete.\n',length(tracforce)-i,length(tracforce));
end
```

As our aim is to generate large quantities of data, it becomes feasible to use use parallel processing for our simulations. For this purpose, we need to create an array to hold input for all simulation runs. There are three variables which change for each run, they are traction force of the locomotive, wheel/rail friction coefficient and number of wagons, respectively. A nested loop is used to create all possible combinations of these variables, and each single combination gets stored, as a new entry, in the array which holds all input.

Variable ranges are from one to 40 for number of wagons, .05 to .78 for friction coefficient and 200000 to 400000 for traction force, with step sizes one, .01 and 1000 respectively. We therefore have a total number of $40 * 74 * 200 = 592000$ combinations. The obvious approach here would be to generate the full batch of 592000 input configurations, but performance and hardware limitations proved this to be unfeasible. Instead, we use rather small batche sizes of 2960 for parallel simulation.

Writing of the output is also done in smaller batches. The best approach to minimize the number of file accessions is of course writing everything into one big matrix, thus storing in

RAM, and then writing the matrix in one go. Unfortunately, growing an array by assignment or concatenation can be expensive. For large arrays, MATLAB must allocate a new block of memory and copy the older array contents to the new array as it makes each assignment. The other extreme would of course be to write every single output line directly, but the large number of file accessions required makes this even more unfeasible, so a middle ground is needed, and a batch size of 3000 works relatively well, although this could surely be optimized with a bit of runtime analysis.

The function below compresses all output of one single simulation into one matrix. Raw simulation output consists of a few time-series objects, like braking force and braking pressure, as well as some metadata. These objects are fed into the write function, which does as many loops as there are rows in the time-series objects. It then packs all rows from each time-series into one big row, adds the meta-data and appends the large row to the matrix, which gets returned at the end. Refer to section 4.2 for a more detailed look at the structure of the output matrix.

```
function ret = Write(id,velocity,force,pressure,distance,acceleration,wagon_ids,t,u,grad,ft,fc
   numrows = get(force,'Length');
   wagon_ids = double(wagon_ids);
   time = force.Time;
   matrix = [];

   for i = 1:1:numrows
      f = getdatasamples(force,i);
      p = getdatasamples(pressure,i);
      v = getdatasamples(velocity,i);
      a = getdatasamples(acceleration,i);
      d = getdatasamples(distance,i);

      row = [id,time(i),f,p,v,a,d,wagon_ids,grad,ft,fc];  % TODO: add simulation input aka trac
      %  print wagon ids as list delimited by colon (,)
      %  performance?
      matrix = [matrix;row];
   end
   fprintf('Created␣output␣matrix␣for␣simid␣%d\n',id);
   % writematrix(matrix,'output/output.tsv','FileType','text','WriteMode','append','Delimiter'
   % ret = 1;
   % fprintf('Create output matrix for Simulation ID %d\n', id);
   ret = matrix;
end
```

```
import csv
import random
```

```
with open('pool.csv', 'w', newline='') as pool:
   fieldnames = ['Wagon_ID', 'Mass', 'Braking_eff']
   writer = csv.DictWriter(pool, fieldnames=fieldnames)

   writer.writeheader()
   for i in range(0,500): # create 500 randomized wagons
      mass = random.randrange(12000, 90000, 100) # create random wagon mass between 12 t and 90
      braking_eff = round(random.uniform(0.75, 0.95),2) # create random braking efficiency as u
      writer.writerow({'Wagon_ID': i, 'Mass': mass, 'Braking_eff': braking_eff})
```

The above python script is used to create a randomized pool of 500 wagons to be used in simulation. Each wagon has a unique id, a randomized mass, ranging between 12000 and 90000 kilograms, and a braking efficiency, which is a uniform distribution between .75 and .95.

4.2 Data Structure

As has been denoted previously, generated data should be suitable for big data processing. The first approach was to create a new directory for each iteration, and also save the different parameters to different files. Since this is very inefficient in terms of number of file accessions and therefore negatively impacts performance, as well as being unsuitable for transformations to big data file systems like Apache Hadoop or Apache Hive, it has been proposed to write all output into one single file. Let's take a look at the structure first.

simID	Timestamp	F_{wagon0}	...	P_{wagon0}	...	v_{wagon0}	...	a_{wagon0}	...	Distance	Wagons	Tra

Some output objects are time-series, namely braking force, braking pressure, acceleration, distance and velocity. Force, pressure and acceleration, which get measured for every wagon, look like this

Timestamp	$Wagon_1$	$Wagon_2$	...	$Wagon_{40}$
0	0	0	0	0
10	x_1	x_2	...	x_{40}
...	...	...	...	...
3600	y_1	y_2	...	y_{40}

whereas distance and velocity only have two columns and thus look like this

Timestamp	Value
0	0
..	..
3600	x

Since they all posses the same number of lines, which is the number of timestamps at which measurements were taken, all columns from all these objects may be compressed into a single table. All columns containing meta-information, for example simulation id, which theoretically only needs to be printed once, get filled with the same value for all lines, and since the number of wagons may vary from one to 40, all possibly empty columns get filled with zeros, so that we do not get any columns containing no value.

4.3 Analysis of generated Data

Performance Analysis

Conclusion

List of Figures

3.1 Initial Model 16
3.2 Initial Model - Wagon 17
3.3 Expanded Model - Pressure Calculation 18
3.4 Expanded Model - Traction Force Calculation 19
3.5 Expanded Model - Wagon 20

4.1 Simulation Input 23

List of Tables

Quellcode

1. Source 1
2. Source 2

Data visualization

<TODO: Visualisierungen einfügen>

Initial model - simulation input Expanded model - simulation input