Karlsruher Institut für Technologie

Hybrid Electric Vehicles
Elektrotechnisches Institut (ETI)
Hybridelektrische Fahrzeuge

Karlsruhe, 13.04.2015

Diploma Thesis

Map-Based Driving Cycle Generation

cand. el. Andreas Wolf
Matr.-Nr: 1359966

Prof. Dr.-Ing. Martin Doppelbauer
Advisor: M.Sc. Miriam Boxriker
Co-Advisor: M.Sc. Jürgen Römer

Delivery date: 12.10.2015

Revision: print@c814d7e (2015-10-22)

DOI: 10.5445/IR/1000049932

Erklärung:
Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

cand. el. Andreas Wolf

Contents

Acronyms

API Application Programming Interface 26, 30

CSV comma-separated values 63

DSL domain-specific language 85

EPSG European Petroleum Survey Group 7, 8, 97

GCRS Geodetic Coordinate Reference System 7

JSON JavaScript Object Notation 62, 63, 84, 99

JVM Java Virtual Machine 20, 21, 27, 43

NEDC New European Driving Cycle 1, 3, 95

OOP object-oriented programming 22

OSM OpenStreetMap 3, 12, 13, 15, 26, 27, 46, 48, 49, 58, 82, 83, 85

OSRM Open Source Routing Machine 27

PCRS Projected Coordinate Reference System 7, 8

SRTM Shuttle Radar Topography Mission 15, 47

TMC Traffic Message Channel 86

URI Uniform Resource Identifier 58

URL Uniform Resource Locator 58, 63

UTM Universal Transverse Mercator 8, 9

VHDL Very High Speed Integrated Circuit Hardware Description Language 15, 42, 85

WGS84 World Geodetic System 1984 7–9, 11, 12, 37, 97

WLTP Worldwide harmonized Light vehicles Test Procedures 3, 95

List of Symbols

a the semi-major axis of the earth ellipsoid / m

b the semi-minor axis of the earth ellipsoid / m

f a) the relative road length error due to ignored grade, b) the earth ellipsoid flattening / *dimensionless*

f_r the rolling friction coefficient / *dimensionless*

γ the grade (steepness) of a road / °

i transmission coefficient / *dimensionless*

λ a (geodetic) longitude / °

n wheel rotational speed / $\mathrm{rad\,s^{-1}}$

p the air pressure / mbar

φ a (geodetic) latitude / °

r_E the mean earth radius / m

ρ the air density / $\mathrm{kg\,m^{-3}}$

List of Signals

This list contains all signals internally used in RoadHopper which are mentioned in this thesis.

For signals with a greek letter or an index, two names are specified. The plain text version is the one used in program code, the formatted version is used throughout this thesis.

For every signal, the data type is specified after the slash.

a the current vehicle acceleration / `Double`

alpha/α the amplified gas pedal input signal / `Double`

alpha*/α^* the delayed gas pedal signal (engine input) / `Integer`

alpha_in/α_{in} the gas/brake pedal input signal (= driver output) / `Double`

beta/β the amplified brake pedal output signal / `Double`

beta*/β^* the delayed brake input signal / `Integer`

M the engine torque / `Double`

M* the delayed engine torque applied to the wheels / `Double`

pos the current vehicle position / `GHPoint3D`

s the distance already travelled by the vehicle / `Double`

time the current time / `Integer`

v/v_{curr} the current vehicle speed / `Double`

v_diff/v_{diff} the difference of current and allowed speed / `Double`

v_limit/v_{limit} the maximum speed allowed within the current lookahead distance / `Double`

v_target/v_{target} the current target speed for the driver / `Double`

1 Introduction

This thesis strives to contribute a new approach for generating driving cycles not by test drives and calculation from the measured data (backward calculation), but instead forward calculate a cycle from a simulation with models for the driver, vehicle and road. The road model is derived from detailed map data.

1.1 Motivation

Over the last decades, driving cycles have evolved into a standard tool for various purposes. The most prominent might be emissions measurement, where cycles like the New European Driving Cycle (NEDC) in Europe and those developed by California's Air Resources Board have found wide usage.

Another area of broad usage for driving cycles is component design, where standardized measurement programs can be used to verify the fitness of a component e.g. for a given load profile.

Gathering a driving cycle for a given configuration (vehicle and driver) is still a costly task: At least one, but usually a lot more measurements need to be performed. Doing such tests for a number of setups can quickly become prohibitively expensive: [Est+01] gives an estimate of around £10 000 for a single vehicle emissions test.

Therefore, being able to quickly gather a driving cycle without the need for a full test setup and test drive brings a number of advantages:

- cost savings potentially in the order of tens of thousands
- much quicker iterations, as a simulation can be performed a lot faster
- the potential to test competing approaches easily with parameter sweeps in a simulation

This thesis presents a concept and implementation for a versatile and extensible driving cycle simulator.

1.2 Overview

This thesis is divided into seven chapters:

- Chapter 2 explains basic concepts necessary for understanding the following chapters.
- Chapter 3 gives an overview of the concept for generating a driving cycle based on a route gathered from a map.
- Chapter 4 presents GraphHopper, the routing software that was chosen as the base for this thesis, and the extensions made to it.
- Chapter 5 introduces RoadHopper, the solution built on top of GraphHopper for simulating a driving cycle.
- Chapter 6 discusses the results gathered from simulations and compares them to some real-world measurements from an earlier thesis.
- Chapter 7 summarizes the thesis and gives an outlook on the topic, including a broader vision for RoadHopper.

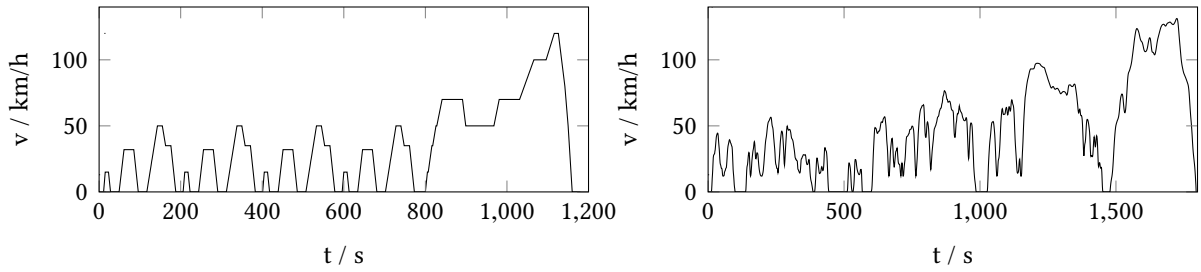
2 Basics

This chapter presents a number of topics which are relevant for understanding both the driving cycle generation concept and the software RoadHopper which was developed to realize it.

After a quick introduction to driving cycles, sections 2.2 and 2.3 lay the necessary foundation for understanding both maps and mapping applications, with a focus on OpenStreetMap. This part is more detailed than necessary to provide a solid foundation also for understanding the calculations on map data.

Section 2.4 gives a quick overview of the most important concepts in VHDL, which was chosen as the reference for a part of the simulation model. Further information can be found in the listed references. Section 2.5 presents a generic form of the driving equation, which is the basis of the vehicle dynamics implemented in the simulation models.

In the last part (sections 2.6 and 2.7), concepts necessary for understanding the simulation engine implementation are introduced. Naturally, only a small part of these topics can be covered here. For further information, the cited reference documentation and books on the topics should be consulted.



(a) NEDC (b) WLTP Class 3

Figure 2.1: Comparison of a modal (NEDC) and a real-world approximation cycle (WLTP)

2.1 Driving cycles

A driving cycle is a standardized, fixed schedule of vehicle operation. Usually, they are defined as functions of velocity (and optionally gear selection) over time [Bar+09].

Driving cycles are designed for a variety of purposes, with the major ones being emissions testing and component design. A huge number of cycles were developed over time; [Bar+09] lists and compares over 200 of them, with a focus on emission testing cycles. The test procedures employed in such tests are also described there.

To create a driving cycle, four different general methodologies are described in [DNE08]:

1. analyzing measurement data and chaining representative small extracts with zero start and end speed (called *micro-trips* in [DNE08] and *driving pulse* in [Lia06])

2. distinguishing different road types and segmenting the measurements by these types; unlike the first method, the segments can have any start and end speed

3. pattern classification on sequences of driving pulses, plus a stochastic model for succession probabilites; the sequences are randomly reconnected to a cycle of the desired length afterwards

4. modal cycle construction based on driving modes (steady speed, acceleration, deceleration) with defined parameters (speed, gear, duration); the driving modes are chained to create a cycle

The first three of these methods create fuzzy, real-looking driving cycles, while the modal cycles often look very artifical (see figure 2.1 for an example of both). The exact construction of a cycle from real-world data is done in different ways; [DNE08] lists a number of sources which detail the cycle creation.

The list of driving cycles in [Bar+09] lists a variety of parameters for each cycle, which were calculated using a tool called Art.Kinema.

In this thesis, no standardized driving cycles are used. Instead a custom driving cycle based on a given map track is to be created. For that, a vehicle and driver behaviour are assumed and the drive along the track is simulated. Those driving cycles are comparable to those generated by the first three patterns listed above; the modal cycles are not comparable as they show much more uniform velocity distributions.

2.2 Maps

Maps in the sense discussed here are two-dimensional depictions of a three-dimensional object: the earth's surface. To create them, a concept called projections—well-known from mathematics in general—is used.

Depending on the projection used, a map will be distorted in one or another aspect, potentially making it unusable for certain purposes. It is therefore important to know the basics of mapping when working with maps. This is why the topic is discussed here in broader sense than strictly required for this thesis.

2.2.1 Representing the earth

The earth is no perfect shape like a sphere or an ellipsoid. Instead, the height of the land surface varies along the surface as a result of past continental movements which formed mountains and valleys. Even the sea level is not uniform, but varies depending on the local gravity field and other effects.

Approximations of the earth for geodetic usage consist of two parts, namely [Lu14, ch. 4]

1. a geoid—the physical shape of the earth—and
2. a reference ellipsoid, which is the mathematical shape of the earth that matches the geoid as closely as possible.

The surface of the geoid is an equipotential of the earth's gravity field, which means that the gravitational force is constant along its surface [Mey10]. It is often defined along the idealized sea level ignoring air pressure variations, water currents etc. and is then continued on the same gravity potential beneath the landmass. The vertical distance between the geoid height and the ellipsoid height for a point is called *geoid undulation*.

Calculations are always performed using the ellipsoid, as the geoid is not usable due to its complicated shape. The ellipsoid is defined by the length of its axes, a (major) and b (minor).

Of both the geoid and the ellipsoid, different variants have been defined, e.g. the Bessel and Krassowski ellipsoids or the EGM96 geoid, which are used in different contexts.

2.2.2 Creating maps

Mapping a part of the earth's surface involves a multi-step process to get from a point on the earth to a point on a map. During that process, the point on the earth is assigned a geodetic

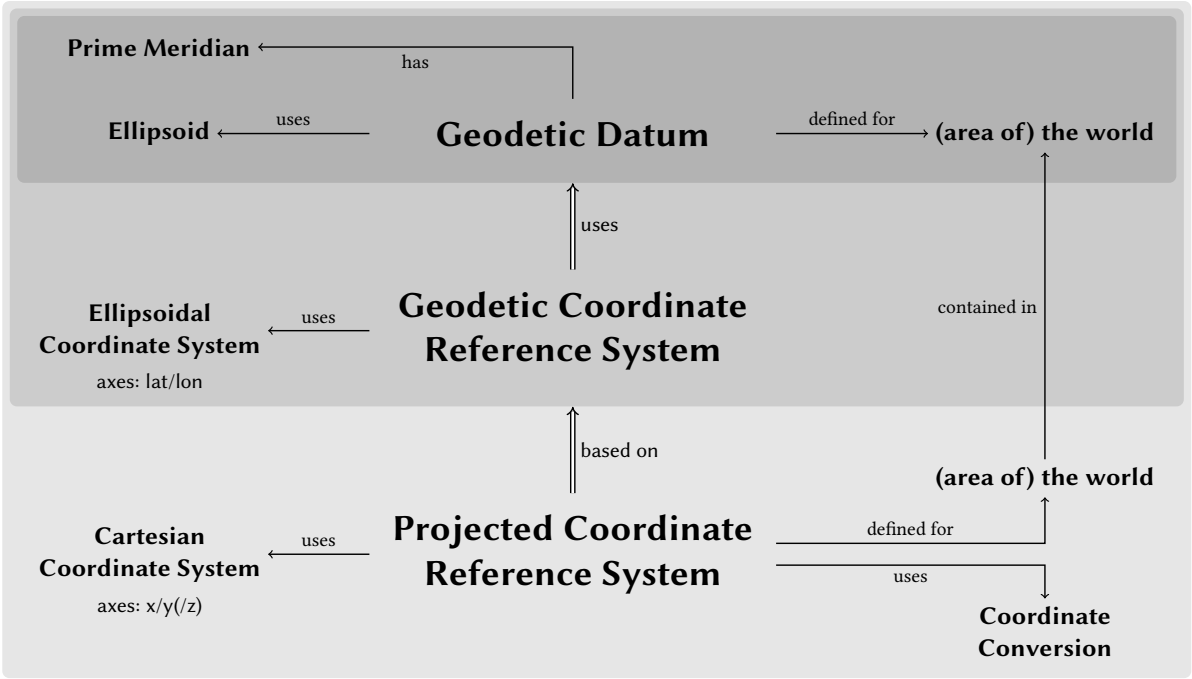


Figure 2.2: Datum, Geodetic/Projected Coordinate Reference System and their relations

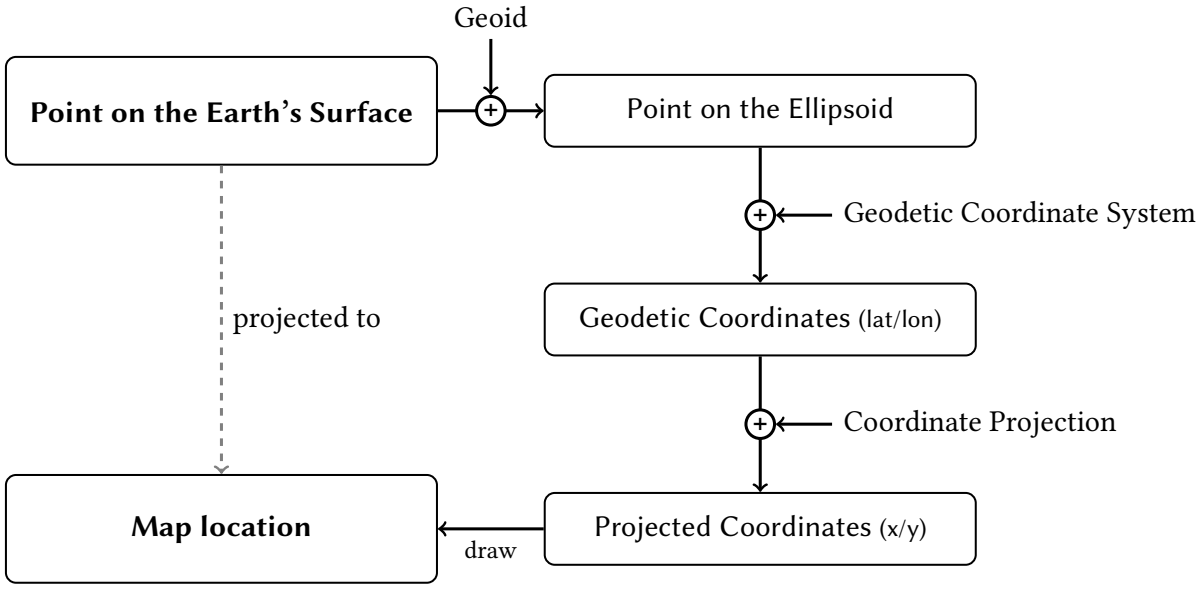


Figure 2.3: The Mapping process

coordinate, which is then *transformed* into the map's coordinate system, resulting in the location on the map.

Several concepts are needed for creating a map, which should be discussed here:

Geodetic Datum is a reference used for geodetic surveying. It connects an ellipsoid and a prime meridian (defining the coordinate system root) with one or more reference points on the earth's surface.

The datum may be valid for the whole world or only a defined area, depending on the way it is constructed. Many geodetic data—e.g. the Postdam Datum as the base of geodetic survey in Germany—are just usable in a single country or or for parts of a country.

Geodetic Coordinate Reference System (GCRS) consists of a geodetic datum and an ellipsoidal coordinate system, the GCRS allows for assigning a pair of coordinates (usually latitude/longitude) to every point of the datum's area.

Projected Coordinate Reference System (PCRS) projects a GCRS into a different coordinate system, e.g. a Cartesian system for 2D maps.

The hierarchy of these concepts and their components can also be seen in figure 2.2.

The mapping process effectively uses three steps to get to a map location for any given point on the earth:

1. map the real point to the datum's ellipsoid surface, using the geoid and the datum reference(s)
2. map the ellipsoid point to an ellipsoid coordinate, using the GCRS
3. map the ellipsoid coordinate to a map location, using the PCRS

The process is depicted in figure 2.3.

The mapping applications discussed in this thesis use the World Geodetic System 1984 (WGS84) as their reference system [NIMA00]. WGS84 uses the aforementioned EGM96 geoid and defines its own geodetic datum and Geodetic Coordinate Reference System. Based on WGS84, different projections can be used to create a map.

The different projections (and their components) are defined by their parameters. To get reproducible results, a database of standardized projection components was created by the European Petroleum Survey Group (EPSG) [EPSGReg]. It is available at `http://www.epsg-registry.org/`; a set of important definitions for this thesis is listed in table 2.1.

entity	EPSG code
WGS84 Geodetic Datum	6326
WGS84 Ellipsoid	7030
WGS84 GCRS	4326
WGS84/UTM zone 31N PCRS	32 631
UTM zone 31N coordinate conversion	16 031
Ellipsoidal Coordinate System	6422
Cartesian 2D Coordinate System	4400
Greenwhich Prime Meridian	8901

Source: [EPSGReg], retrieved 2015/10/01

Table 2.1: EPSG codes for various components important i.a. in WGS84

name	symbol	value	unit
semi-major axis	a	6 378 137.0	m
semi-minor axis	b	6 356 752.3	m
flattening	f	1/298.257 223 563	—
mean radius	r_E	6 371 008	m

Source: [NIMA00], own calculation (r_E)

Table 2.2: WGS84 earth ellipsoid parameters

Projections

As said above, different types of projections are used in creating a map. This section revolves only around those used in a PCRS, which map a point from the reference ellipsoid onto a map, shaping the final look of the map. A variety of such projections exists, defined for various purposes. Each of them has their own advantages and drawbacks, which makes them fit for certain tasks and unusable for others.

In this section, only one of the most popular ones, the Mercator projection, is discussed, as it is required for understanding the projections used for maps discussed in this thesis. Based on Mercator are its specializations Transverse Mercator and Web Mercator and the Universal Transverse Mercator (UTM) system of projections, which are widely used in mapping.

Mercator The Mercator projection [Mey10] maps the earth's surface onto a cylinder wrapped around the earth. Usually, the cylinder's axis equals the rotational axis of the earth and it touches it along the equator. Mapping points of the surface onto the cylinder is done by drawing a line from the center through the earth's surface. The point is then mapped to the intersection of the line and the cylinder's surface.

In the Mercator projection, all longitudes (or meridians) are equidistant, while the distance between latitudes increases with the distance from the equator. The mapping is pretty accurate near the equator, but distortions increase rapidly when approaching the poles. The Mercator projection cannot map the poles, and areas towards the poles are mapped with a multitude of their real size. One notable example of this is Greenland, which on Mercator-projected maps often looks bigger than the USA, while in reality it is only around a fifth of the size of the US (2 166 086 km^2 vs. 9 826 675 km^2).

A detailed discussion of the disadvantages of the Mercator projection is also given in [Bat+14].

Transverse Mercator Transverse Mercator [Mey10] projections are a specialization of the Mercator projection. They are used by the UTM, which in turn is the basis for many mapping applications.

In a Transverse Mercator projection, the cylinder is tilted so that its axis lies in the equatorial plane. It also has a slightly smaller diameter, so it does not only touch, but cut the surface with two circles (or ellipses for an ellipsoid). These lines are called *standards*. The center between the two cuts is the central meridian, which is the defininig feature of any given TM projection.

A small band around this central meridian can be mapped to a plane with low distortions, which is why the Transverse Mercator projection has found wide usage in mapping applications worldwide.

Universal Transverse Mercator The Universal Transverse Mercator Projection [Mey10] is a system of 60 Transverse Mercator projections, each covering a width of 6°. These zones are numbered starting with 1 at 180° W to 174° W to 60 for 174° E to 180° E. Germany is located in zones 31 and 32 (6° E to 18° E).

As the zones would become too small near the poles, it only covers a range from 80° S to 84° N. The areas around the poles are mapped by the universal polar stereographic (UPS) system.

The UTM maps to a cartesian coordinate system for drawing. This coordinate system uses a fixed reference with a few specialties (false easting/northing) to avoid using negative values for coordinates. This should not be discussed here, instead refer to e.g. [Mey10].

Web Mercator Although the usage of Mercator projections for mapping was declining in printed cartography, an adapted version became popular again with the rise of web-based map services [Bat+14]. This version is called *Web Mercator* or *Spherical Mercator*, as it projects WGS84 coordinates using a sphere (and not an ellipsoid as would normally be done) [Bat+14].

Because of this, it incurs an even higher error than the mercator projection—[NGIA14] describes the potential errors of coordinates compared to an ellipsoid-based projection as "over 40 km".

The Web Mercator projection was standardized as EPSG:3857.

Coordinate representations

Degrees can be written in different formats:

- divided into degrees, minutes, seconds: 123°12′34″; minutes and seconds can be left out
- decimal: 123.4567°

The decimal format will be the most widely used throughout this thesis, as coordinates are represented this way in computers.

An alternative unit for angles used in Geodesy is Gradian [Mey10], which divides the full circle into 400 units. It is also known as gon, and denoted with ^{g} (e.g. 123^{g}).

The ellipsoidal coordinate systems in figure 2.2 usually have their positive semi-axes towards north and east. The origin of this system is at the intersection of the equator and the prime meridian.

Ellipsoidal coordinates can be annotated with ± prefixes or N/S and W/E suffixes, e.g.:

- +49.010 89°/+8.412 75°
- 49.010 89° N/8.412 75° E
- 49°0′39″ N/8°24′46″ E

2.2.3 Calculations

Depending on the used projection and earth shape approximations, different types of distances can be distinguished [Mey10, sec. 6.1]. Of those, for this thesis only great circle distances will be relevant, as the distances are usually in the range of a few hundred meters and the errors incurred are therefore negligible. On the other hand, spherical calculations are very fast compared to more exact formulae, which makes them better suited for usage in a simulation.

Two types of calculations are mainly used [Mey10]:

1. forward (or direct) calculations to get an end point based on a start point, distance and initial bearing

2. backward (or inverse) calculations to get the distance and bearings[1] for a given pair of points

Forward

A forward calculation gets a point, a distance and an initial bearing (direction of travel) as the input values. The output is the resulting position after travelling the given distance.

The formula presented here is taken from the *Geosphere* package[2] of the R statistical computing software (https://www.r-project.org/). It is, as stated above, only valid for a spherical approximation of the earth, which is accurate enough for our purpose, given that most distances are only up to a few hundred meters and the coordinates used for calculations are gathered from a more accurate approximation (WGS84).

Given a point A with coordinates φ_A/λ_A, a length l and an initial bearing (direction clockwise from geographic north) α, the resulting position φ_B/λ_B can be calculated to

$$\varphi_B = \arcsin\left(\sin(\varphi_A) \cdot \sin\left(\frac{l}{r_E}\right) + \cos(\varphi_A) \cdot \sin\left(\frac{l}{r_E}\right)\right) \cdot \cos(\alpha) \tag{2.1}$$

$$\lambda_B = \lambda_A + \arctan\left(\frac{\cos\left(\frac{l}{r_E}\right) - \sin(\varphi_A) \cdot \sin(\varphi_B)}{\sin(\alpha) \cdot \sin\left(\frac{l}{r_E}\right) \cdot \cos(\varphi_A)}\right) \tag{2.2}$$

Inverse

For a spherical approximation of the earth, a great arc can be calculated like this: Given the latitude/longitude pairs φ_A/λ_A and φ_B/λ_B, the angle between the two points is

$$l = \zeta \cdot r_E \tag{2.3}$$

$$\zeta = \arccos(\sin(\varphi_A) \cdot \sin(\varphi_B) + \cos(\varphi_A) \cdot \sin(\varphi_B) \cdot \cos(\lambda_B - \lambda_A)) \tag{2.4}$$

This formula is e.g. required for determining the length of roads in GraphHopper's road graph, as the edges are only marked by their coordinates.

[1] When travelling along a great arc, the bearing changes over the distance, unless travelling directly towards one of the poles (in which case it is fixed to 0° or 180°.

[2] see https://cran.r-project.org/web/packages/geosphere/; visited 2015/10/08

For short distances, the argument's cosine approaches 1. For lengths in the range of a few tens of meters, it might yield incorrect results depending on the floating point accuracy. [Mey10] therefore suggests using the haversine formula instead:

$$l = 2r \cdot \arcsin\left(\sin^2\left(\frac{\varphi_B - \varphi_A}{2}\right) + \cos\varphi_A \cdot \cos\varphi_B \cdot \sin^2\left(\frac{\lambda_B - \lambda_A}{2}\right)\right) \tag{2.5}$$

For the earth's radius, the mean of the WGS84 ellipsoid's radius can be assumed, see table 2.2.

2.3 OpenStreetMap

OpenStreetMap (OSM) is an international project with the aim to create an as complete as possible map of the whole world. It was started in 2004 and to date has almost three billion points mapped[3].

Internally, OSM uses WGS84-based coordinates. Most rendered maps are Web Mercator projections, as it is the de-facto standard for web mapping applications.

2.3.1 Data model

OSM is a depiction of the real world. As such, it must be able to represent every possibly interesting item. Still, its data model consists of only three main entity types[4]:

Nodes mark a position on the map. In their most simple form, they just have a pair of coordinates. The meaning of a node is further described with tags (see below). Nodes may be included in any number of ways.

Ways are a generalized concept to connect consecutive edges—they do not only represent streets or footways, but also the walls of buildings or area boundaries. Ways have a defined direction, making the OSM graph a directed graph. This is important e.g. for mapping oneway streets.

Not all ways are connected, i.e. it is not always possible to travel from one node to the other only via edge traversal. The OSM graph thus consists of many independent subgraphs. However, by using the node coordinates, a connection between these subgraphs can still be constructed using their spatial proximity (e.g. to connect houses to the street next to them).

[3] see https://wiki.openstreetmap.org/wiki/Stats; visited 2015/09/21
[4] see https://wiki.openstreetmap.org/wiki/Elements; visited 2015/09/06

Relations link two or more elements of the graph (e.g. as parts of a building or buildings to a street or stations to a public transit line). Political borders, from states down to the level of urban districts, are also mapped as relations.

Additionally, all entities have a number that explicitly identifies them in the OSM database [RT09].

Tags, simple key-value pairs, can be used for further describing all entities. The most often used key for ways is *highway*[5], which is used to denote all kinds of streets and street-related information (e.g. traffic lights and signs).

The possible values and intended usage of tags are not prescribed by the OpenStreetMap project, but instead stem from a continuously changing consensus among the mappers [RT09]. For some tag keys, users also define new values where they seem fit (mostly general-purpose key like *highway*, see above).

An overview of tag usage (frequency, combinations, geographic distribution etc.) is available at `https://taginfo.openstreetmap.org/`.

2.3.2 Data Export

As the data of OpenStreetMap is freely available, it can be accessed both in the form of ready-made web services (e.g. online maps) and as raw data packages for local usage, e.g. in geoinformation utilities or statistical analysis tools.

OpenStreetMap data can be exported as either XML or in a binary format called PBF. An export includes the whole world[6], but can also be stripped down to only a part of the world. One tool used for processing OSM data dumps is Osmosis [RT09] (`https://wiki.openstreetmap.org/wiki/Osmosis`).

Free downloads of OpenStreetMap data dumps are offered e.g. by the German company *Geofabrik*, whose download server is located at `https://download.geofabrik.de/`. This server is the source of the data used in this thesis.

2.3.3 Routing

The general problem solved by routing algorithms is finding the shortest path between two points A and B. *Shortest path* can be further generalized to *path with the lowest edge weight*—different

[5] see `https://wiki.openstreetmap.org/wiki/Key:highway`; visited 2015/09/23
[6] see `https://wiki.openstreetmap.org/wiki/Planet.osm`; visited 2015/09/17

edge weights then allow optimizing for different use cases, e.g. avoiding high traffic roads or preferring highways over other roads.

Many routing algorithms, like the widely used Dijkstra family, require a huge effort for finding a single path, but require little precomputation. Newer approaches like Contraction Hierarchies exist that trade extensive precalculations for speed advantages during search [Gei08].

Of the many algorithms developed over time, this section focusses on the few that are relevant in the context of this thesis, i.e. those that are implemented in GraphHopper.

Routing Algorithms

Dijkstra's algorithm is a shortest-path algorithm that iteratively searches from a start node s to a target note t [Ski08]. From a currently visited node v, all neighbouring, unvisited nodes get a weight set based on the edge weights. Each node encountered gets a weight w assigned which denotes the distance from s to this node via the current node v. If a node already has a weight, it is only changed if the new weight would be lower, ensuring an optimal solution.

The algorithm forms a circle around the start node, with a radius increasing in each step. The search can be stopped once t has been visited. The resulting path is the shortest path from s to t.

Bidirectional Dijkstra The bidirectional version of Dijkstra's algorithm starts at both the start and target nodes. The two "search circles" will have only half the radius and thus cover a much smaller area with less nodes. The bidirectional version therefore is usually faster.

A* or A-star is a variant of Dijkstra's algorithm which uses a heuristic to estimate the cost to completion for a given path. This limits the algorithm's applicability, as such a function must first of all exist, but it also makes it much more efficient [ZC09]. One cost estimation function often used in road network routing is the air line between the current path end and the destination. For paths pointing away from the target, costs will therefore rise, making them less attractive than the (more probable) paths already heading into roughly the right direction. The algorithm can then focus on those more attractive paths, which improves space usage and—for some applications—also processing time [ZC09].

Contraction Hierarchies

The concept of Contraction Hierarchies [Gei08] is to insert shortcuts into a graph to avoid repeated traversal of multiple edges. These shortcuts are inserted on a higher level than the original nodes, thereby creating the hierarchy.

More recent developments include time-dependent edge weights to include e.g. traffic load [Bat+09], and customizing Contraction Hierarchies in general [DSW14].

Contraction Hierarchies are implemented in GraphHopper and enabled by default.

2.3.4 Elevation data

The OpenStreetMap database only contains latitude and longitude coordinates, but no elevation (height information). As this information is required for e.g. calculating the road grade, it must be obtained from an external data source.

GraphHopper already contains an interface to fetch and include elevation data from the Shuttle Radar Topography Mission (SRTM). A high precision version of this data (SRTMGL1; resolution 1″ or one point ca. every 30 m) was published in 2014 and 2015[7]. An interface to this high-resolution version was added to GraphHopper during this thesis.

2.4 VHDL

Very High Speed Integrated Circuit Hardware Description Language (VHDL) is a general-purpose hardware description language, standardized by IEEE in 1987. It separates the descriptions of interfaces (entities), architecture and configuration (implementation), allowing different implementations to be used. VHDL inherently supports parallelism in execution [LWS94].

For this thesis, VHDL is not so much of interest because of its hardware description capabilities, but more because of its execution model and the simulator engine built on top of it: These are both a good fit for the task solved in this thesis. Of the numerous concepts in the language standard [VHDL], four are therefore relevant for this thesis and should be described here in further detail: signals, variables, processes and delta cycles.

Processes describe one block of sequential data processing [VHDL]. Multiple processes can run in parallel. Each process has a so-called sensitivity list of signals (declared with `WAIT ON`) it listens to; when one of the signals changes, the process is executed.

[7] see https://lpdaac.usgs.gov/nasa_shuttle_radar_topography_mission_srtm_global_1_arc_second_data_released_over_middle_east; visited 2015/09/18

Signals and variables both hold values used in the code. Their main difference is a behavioural one: assignments to variables are always effective immediately, while signal values only change after all processes have run [VHDL, ch. 12.6]. Variables are also local to one process, while signals are used for transmitting values between processes.

Delta cycles are tightly coupled to the concept of delayed signal updates. To explain them, we must first have a look at what delta cycles actually are: Each time step is divided into one or more delta cycles. The number of required delta cycles is determined at run time: If a process updates a signal, the update is delayed until all processes have finished. Afterwards, the update is performed and all processes sensitive to the signal are executed in a further delta cycle [VHDL, ch. 12.6.4].

2.5 Vehicle kinematics

Like every object in motion, a vehicle has a lot of forces which influence its movement. Only some of these can directly be controlled by the driver, while others are indirect consequences of the driver's actions.

These forces are usually a part of the so-called driving equation [Bra13]:

F_{eng} the engine force

F_{brake} the brake force

F_{climb} the climbing force—the force required to overcome a height difference along the road

F_{acc} the acceleration force—the force induced by vehicle inertia against an acceleration

F_{roll} the rolling resistance

F_{air} the air drag

A simplified version of the driving equation with all its parts is depicted in figure 2.4. As a result of the simplification, several aspects are not taken into account, namely

1. the different pressures of and load on the single wheels,
2. multiple powered axles,
3. power train losses, and

$$F_{\text{res}} = F_{\text{eng}} - \left(F_{\text{brake}} + F_{\text{acc}} + F_{\text{climb}} + F_{\text{roll}} + F_{\text{air}}\right) \tag{2.6}$$

$$= \frac{1}{2} c_{\text{W}} \cdot A \cdot \rho \cdot v^2$$

$$= f_{\text{r}} \cdot m \cdot g \cdot \cos\gamma$$

$$= m \cdot g \cdot \sin\gamma$$

$$= m(1+e) \cdot a$$

$$= \frac{i}{r_{\text{w}}} \cdot M_{\text{eng}}$$

- $e > 0$ is a factor for the rotational inertia of the power train. It increases the effective mass of the vehicle relevant for the acceleration.
- γ is the grade of the road in degrees.
- i is the (dimensionless) transmission factor.
- r_w is the wheel radius.

Figure 2.4: The driving equation

4. changes in the air resistance due to wind.

Also several factors like the friction coefficient f_r are assumed to be steady, while in reality they vary depending on the road surface and weather conditions (wet or icy road). [Bra13] delivers a more detailed discussion of each part, with many of the details left out here for brevity.

For steady driving, F_{acc} equals zero, similarly F_{climb} for even roads.

2.6 Concurrent programming

Running a simulation necessitates a lot of computations, many of which do not depend on each other and thus can be parallelized to speed up the simulation. Therefore, parallel or concurrent programming is an important topic for simulations.

When creating sequentially executed code, the order of operations is pretty clear—it can be directly derived from the order of lines in the source code. The main advantage of this is the obviously simple reasoning about the program's behaviour. The main disadvantage of

sequential execution quickly becomes apparent when external systems like a database or a hard drive come into play: if an operation blocks, the whole code is not executed further until this result is ready.

Additionally, if the code is executed in a multiprocessor environment, only one of the many processors will be used, while all others are idle (unless they execute different programs). Especially given the recent trend towards massive horizontal scaling of processors (multiple cores) instead of vertical scaling (faster clock speed), using this potential becomes more and more desirable.

So parallelized actions may speed up computation, if enough execution units are available to really run the parts at the same time. It may also speed up the software if it runs on a single core, as code that needs to wait for external, slow resources like the hard drive or the network can be put to the background while the execution unit continues with another sequence of operations; this technique is called time slicing or multitasking.

Threads are nowadays the most widely used abstraction for parallel programming, even though their model leads to many problems seen today with concurrent programming [Lee06]. Most of these problems stem from operations on the same data. Threads are a good model for data architectures where each single thread operates on its own data (shared-nothing architecture [Sto86]). In the case of shared data special precautions must be taken to ensure that the data is always consistent and no deadlocks can occur which let the program grind to a halt [MS07].

Several approaches to mitigate these negative effects have been developed. One of them, which tries to get rid of shared mutable data altogether, is the actor model. The actors in this model are small, independent collections of sequential operations. They keep all mutable data to themselves and only pass immutable data (messages) on to other actors. This way, most of the effort required for synchronizing data access in regular programming environments becomes unnecessary and can be omitted.

2.6.1 Promises and futures

Before delving deeper into the actor model, another part of concurrent programming needs to be introduced: *promises* and *futures*.

They are one method of synchronising code segments: If two pieces of code run on two different threads, it must be possible to only continue operation as soon as both are finished. As such, futures and promises help mitigating the chaotic behaviour of threads described in [Lee06].

Promises and futures are closely coupled, but cover different concerns:

- A **promise** is an object that acts as a placeholder for the result of a running computation. It can be *completed* with either a value or an error, if the computation failed for any reason. The promise becomes immutable once it was completed, i.e. no second assignment of a value is possible.

- A **future** is a read-only view of a promise's value, and as such a means for separating the concerns of the computation (user of the promise) and waiting for the computation to finish (user of the future).

2.6.2 Actors

The actor model, first defined in the early 1970s, tries to separate the sequential aspects of computations from the communication aspects [Cli81]. Fundamental work on the model has been done in [Cli81; Agh85].

Actors are defined as independent agents that do not share any mutable state. They can send each other messages, which are by definition immutable. These messages are delivered to an actor's mailbox, from which the actor can take them as soon as he is ready to process messages. The message reception and queueing itself is not implemented by the actor. This way, the message queue handling and the actual actor implementation can be more easily separated.

While processing a message, an actor may send messages to any number of other actors, including the sender of the current message. A message must only contain immutable data: Closing over the actor's local, mutable state to other actors would reintroduce the locking and synchronisation problems known from other parallel computing approaches.

Messages passed to an actor are usually processed in the order they arrived [Cli81]. Implementations might also define high-priority messages that destroy this natural order (e.g. the Akka toolkit discussed below introduces different kinds of prioritisation mailboxes).

A system of actors can to some extent be compared to any traditional office, where clerks sit and wait for incoming messages. Every clerk has its personal area of responsibility, for which they receive messages. They process these messages sequentially, using the information in each message to change the data they manage. Meanwhile, they may send messages back to the sender or to other business units (e.g. to inform them about changes or confirm that the message was successfully processed).

Notable implementations of the actor model include the Erlang language, which is widely used in telecommunications applications, and Akka, an actor toolkit for Java and Scala, which is discussed below.

Akka

Akka is a toolkit for implementing the actor model based on the JVM. The initial actor model implementation was part of the Scala programming language and was later on extracted to Akka as an independent project.

On top of the generic actor model, Akka defines a tree-shaped structure, called supervision hierarchy. This structure is embedded into the so-called *actor system*, which takes care of allocating the necessary resources, e.g. threads, for the execution of the actors.

The levels below the root form a hierarchy of actors, with actors being a parent to the actors they use for sub-tasks of their computations. An actor automatically is the supervisor of the actors it creates and is responsible for handling their failures. [Akka, section 2.4] shows four different patterns for dealing with errors.

This error locality eases development of resilient applications that can still work despite failed parts. Another feature of Akka that fosters this resilience is the ability to span an actor system across multiple instances of the JVM, which can possibly also reside on different computers [Akka, section 2.2.1].

Messaging Messaging between Actors in Akka is implemented in two flavors:

- *tell* messages without a receipt (*fire-and-forget*), sent with the `!` operator, and
- *ask* messages where a result is explicitly expected by the sender, sent with `?`.

This thesis mainly uses the latter message style, as all messages during a time step in simulation need to be processed before the time step can be finished.

2.7 Scala

Scala is an object-functional programming language that has been developed at the EPFL in Switzerland since 2001. The name is an abbreviation of *Sca*lable *la*nguage [OSV08, ch. 1], which points out one emphasis of its design: extensibility. Like every modern programming language, Scala is a too diverse language to be discussed in all details in this thesis. Therefore, a few

certain aspects will be highlighted in which Scala differs from other modern languages. Some of these aspects are also especially different compared to Java, the language whose ecosystem Scala largely shares.

Though designed from the ground up, Scala is intended to coexist and cooperate with the Java programming language [OSV08, ch. 29]. It is internally compiled to the same bytecode format as Java [Lin+15] and thus can run in the same execution environment, the Java Virtual Machine (JVM). The JVM is an intermediate layer between the compiled code and the actual execution engine (hardware + operating system), making it possible to run the same compilation on different hardware architectures, like x86 or PowerPC; see [Lin+15] for the detailed specification of this environment.

Scala constructs can also be used from Java code, and vice versa. [OSV08, ch. 29.1] presents an overview of the ways Scala's specialties are mapped to Java constructs.

2.7.1 Functional programming

Functional programming has two main ideas [OSV08, ch. 1]: functions as first-class members of the language and using input–output mappings in functions instead of mutating a global state. Scala supports both these concepts.

To understand what makes functional programming special, it will help to look at the usual style of programming first: Classical imperative programming revolves around changing some global state using statements. In contrast, functional programming uses expressions that return a value based on their input parameters without changing these or any global state (i.e. the expressions are side-effect free).

When functions are treated as first-class members of the language, they can be used instead of a value. They can be passed as an argument or returned as a result of a function and can also be saved in variables. This allows for totally new approaches to composing programs, e.g. composing a collection library with filter functions passed into a method [Sue12, ch. 1.1].

Scala also implements functions of higher order, which take functions as their argument. One pretty good example is the aforementioned `filter` function of a collection that implements the necessary boilerplate for filtering: looping over all elements, evaluating a given predicate and discarding the element or not. The predicate itself is supplied as a function:

In the example, only the elements that are < 5 are kept in the collection. The predicate was supplied as an inline expression without declaring a function body. This is one example of Scala's concise syntax that removes much of the boilerplate code required in other languages.

```
scala> val someElements = List(1,2,3,4,5,6,7,8,9,10)
someElements: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> someElements.filter(_ < 5)
res0: List[Int] = List(1, 2, 3, 4)
```

2.7.2 Read-only data structures

Being a functional programming language, Scala has a strong focus on immutability [Bra11]. Therefore, variables can be defined as immutable by declaring them with the `val` keyword instead of `var`, which is used for mutable variables.

Immutable values can also be marked as *lazily evaluated* by using the `lazy val` keyword. This can be handy for situations where a value might not be used, so a costly calculation can be deferred to the first usage (or skipped if the value is not used).

2.7.3 Traits

Scala, like most object-oriented programming languages, only supports single inheritance, that is, every class can have only one parent class it extends. In traditional object-oriented programming (OOP), interfaces can be defined which add method signatures to a class.

Unlike other languages, Scala does not support interfaces, but only traits. Traits can be seen as *rich interfaces* [Bra11], in that they supply both the method signatures and their implementation. A class can extend a number of traits, e.g. to combine different behaviours. This is widely used in the collections library; refer to the Scala API documentation for more information.

An example class definition from RoadHopper's source code:

```
class SignalsJourneyActor(val timer: ActorRef, val signalBus: ActorRef, val route: Route)
  extends Process(signalBus) with ActorLogging {

  // class implementation
}
```

Here, the class `SignalsJourneyActor` is derived from the class `Process` and enriched with the `ActorLogging` trait. Such a trait is a typical example: it adds one specific aspect, logging in this case, to a class, so that the implementation does not have to be repeated over and over again.

The parameter `signalBus` is passed from the class to its parent's constructor.

2.7.4 Pattern matching

Pattern matching [OSV08] is a concept not known from many other languages. It is comparable to a `switch-case` statement, but more powerful in so far that it returns a value, which can then be assigned to a variable. This makes it possible to assign a variable with a single, complex pattern matching instead of deeply nested `if-then-else` structures:

```
val normalizedOrientation = orientation match {
  case o if o < -Math.PI => o + (Math.PI * 2)
  // ensure that the interval is open at the right end
  case o if o % (Math.PI * 2) == Math.PI => -Math.PI
  case o if o >= Math.PI * 2 => (o - Math.PI * 2) % (Math.PI * 2)
  case o if o >= Math.PI => o - (Math.PI * 2)
  case o => o
}
```

2.7.5 Case classes

Case classes are classes in Scala declared with the `case class` keyword. The compiler automatically adds a number of methods to them which e.g. make them directly usable in pattern matching [Bra11]. The parameters of a case class are marked as immutable by default, i.e. the `val` keyword can be left out.

A case class can be used in a pattern matching like this:

```
case class Message(type: String, text: String)

// msg is a variable containing a Message case class instance.
msg match {
  case Message("stop-sign", _) => println("Stop sign ahead")
  case Message("traffic-light", "red") => println("Red traffic light ahead")
  case Message("speed-limit", "50") => println("Speed limit set to 50 km/h")
  case _ => println("Unknown message")
}
```

By using _, the value of the corresponding variable is ignored for matching, i.e. the text could have any value if the type was “stop-sign” above. Likewise, a message of type “traffic-light” with value “green” would be matched by the last clause and result in “Unknown message” being printed.

3 Driving cycle generation concept

This chapter gives an overview of the driving cycle generation concept created for this thesis. The first section explains the developed three-step driving cycle process, followed by sections that further detail the concepts for each of the steps and the necessary implementation work. The implementation is not covered here, but in the following two chapters.

3.1 The driving cycle generation process

Given the requirement of generating a driving cycle from a map, the following two steps must be performed:

1. get a track based on the map
2. get a time–velocity profile for the track

Step 1 is already possible with existing routing software, though not yielding a result as detailed as required for the purpose of this thesis. Hence, the chosen solution needed to be adjusted and extended.

To realise the second step, some model for creating the velocity profile for a given road (part) is required. A simple, generic mathematical model could not be found in existing literature. Instead, an approach to simulate a tour along a given track was used. The second step can thus be split into two parts:

2. simulate a trip along the track, and
3. calculate the time–velocity profile from the simulation data.

The full process is also depicted in figure 3.1.

Figure 3.1: The three-step driving cycle process

3.2 Map processing and enrichment

To power map-based applications, a number of data sources can be considered, depending on the level of detail required for the particular use case. For normal map display and routing with human-readable instructions, most free-to-use map services will already suffice. They do however not provide enough detail to realize more complex use cases, like the task discussed in this thesis.

3.2.1 Mapping service choice

In the context of this thesis, as much data as possible must be extracted from a map, especially an as exact as possible course of the road, including speed limits, traffic signs etc. In order to fulfil these requirements, several mapping services were considered.

The most popular free-to-use map services include Google Maps and Bing Maps (by Microsoft). Popular commercial vendors (in no particular order) are Mapbox[8] and HERE[9]. All these map services also include a routing service, also called "directions API" by some vendors. The output of these routing services is mostly coarse-grained, consisting of the track on the map and instructions for a human driver[10]. They are therefore not sufficient for the base of a simulation road.

The mentioned commercial vendors, Mapbox and HERE, also offer a free-to-use version of their map service, but these versions also do not offer the data depth required for the basis of a driving simulation.

An alternative source of freely usable map data is the OpenStreetMap project (see also section 2.3). This data is usable under its license terms, see [ODbL]. Additionally, as the raw

[8] http://www.mapbox.com/

[9] https://www.here.com/, recently sold by mobile phone manufacturer Nokia to a consortium of German car manufacturers.

[10] See e.g. the documentation for the *Google Maps Directions API* at https://developers.google.com/maps/documentation/directions/intro; retrieved 2015/09/11.

graph data is available, all information necessary for the cycle generation can be extracted; the graph can even be enriched with data from additional sources (see section 4.1.2).

3.2.2 Routing

The input of a routing task are 2…n waypoints, between which a route is then searched. To find the best route, a metric is applied to all ways. Examples of such a metric are "shortest path" or "fastest path". The result is a list of instructions to get from one waypoint to the next, or more detailed data.

Routing based on OpenStreetMap data is implemented in various existing solutions for different platforms. The most prominent ones are Open Source Routing Machine (OSRM), GraphHopper and MapQuest, all three featured on the project's official platform openstreetmap.org. The former two are also open source and free software, which makes them a good choice for the basis of this thesis: Both softwares already provide the required general feature set to get maps at a detail level required for this thesis. They are also flexible enough (as their source code is available and may be modified) for the adaptations necessary to get all needed data.

As the technology was deemed a better fit, GraphHopper was chosen for this thesis over OSRM. Its internals and the necessary adaptations to its data model and import process are described in chapter 4.

3.3 Driving cycle simulation

As stated above, no mathematical model exists to directly get a (realistic) travelled speed for an arbitrary given road segment at the accuracy needed for a driving cycle. Additionally, such an abstract model would require adaptation for every new set of parameters (driver behaviour, vehicle parameters, etc.)

Therefore, a different approach was chosen for this thesis: Simulating a driving cycle with models for the vehicle and the driver's behaviour. These models should be derived from literature and embedded into a simulation framework. From the simulation results, a data basis for calculating velocity profiles can be gathered. The software for this was created specifically for this thesis and is called *RoadHopper*.

An overview of the system architecture of RoadHopper is given in figure 3.2. RoadHopper was implemented on top of GraphHopper, using of the same software stack, of which the most important components (the JVM and the embedded Jetty web server) are included in the diagram.

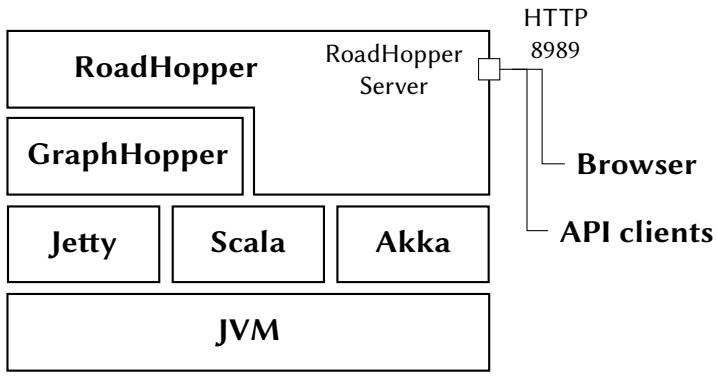


Figure 3.2: RoadHopper's system architecture

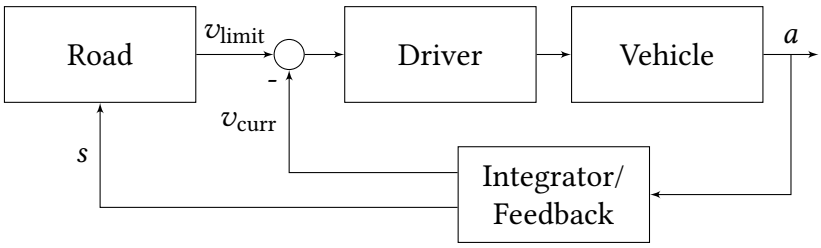


Figure 3.3: The system model for simulating a driving cycle

Newly introduced into the stack are mainly Akka and Scala, both of which were presented in chapter 2.

Given the three components road, driver and vehicle, it immediately becomes clear that driver and vehicle form a controller–plant system, with the driver as the controller steering the plant. The driver gathers its input from the road and the vehicle state. The resulting system model is depicted in figure 3.3.

3.3.1 Simulation models

Creating a complete model of a human driver and a vehicle is a huge task that must be split into smaller parts to be manageable. Also some of these parts are negligible depending on the desired properties and the application of the model, so they can be left out or replaced by simpler approaches.

The usual driving tasks can be grouped into different categories based on their characteristics like the involved sensory channels, as e.g. done in [Mac03]. The major tasks related to vehicle steering are *lateral* and *longitudinal* control. While the former will have some influence on the spatiotemporal behaviour of the vehicle and thus the resulting driving cycle, this behaviour

mainly results from the longitudinal control decisions, that is accelerating and decelerating (as the direct consequence of the driver's impact on the respective pedals).

As a result, the simulator was designed to ignore lateral control for now. To have a safeguard against irrational and unrealistic behaviour—like travelling a road bend with a speed that would let the vehicle derail—some measures were taken, which are described in greater detail in chapter 5.

The vehicle model was also designed to be as sophisticated as necessary for first simulation results, but not overly complicated. Therefore, a lot of physical effects that affect the vehicle performance, like the suspension or power train efficiencies, were not taken into account. Instead, a proof-of-concept model was developed that can be extended as it is required for the next steps. The detailed design and the rationale behind it is shown in section 5.4.4. Future development possibilites of the model and simulation are discussed in sections 7.2 and 7.3.

3.3.2 Simulator concept

For the simulation, a time-discrete implementation using fixed-frequency scheduling was chosen. Fixed intervals are necessary to get a deterministic behaviour of mathematical operations, e.g. integrations or differentiations of the input over time.

For the timing frequency, the aim is to have a good compromise between short enough time slots so that small changes don't get lost and an efficient computation of the simulation, which leads to the chosen frequency of 100 Hz. Compared to the human vision frequency of 25 Hz, this would also mean an oversampling factor of four, without even taking into account processing times. [Mac03] cites various sources that show human response times between 140 and 180 ms for near-ideal conditions. The 10 ms between two simulation steps should therefore have enough safety margin to model any possible future applications.

As the final scope of the thesis with respect to the models was not known in advance, special emphasis was put into keeping the model extensible and the existing components easily replaceable with new implementations. The components are therefore loosely coupled and should follow the single responsibility principle [Mar12], that is, they should only have one task to do. This is the reason why e.g. the simulation timer and the signal bus are two separate components, though both of them are vital for the simulation runtime as a whole.

Another focus was scalability: As the possible range of applications for a driving simulation is very wide, the system should be as flexible as possible to cope with upcoming future requirements. Though not everything can be taken into account from the very beginning—as

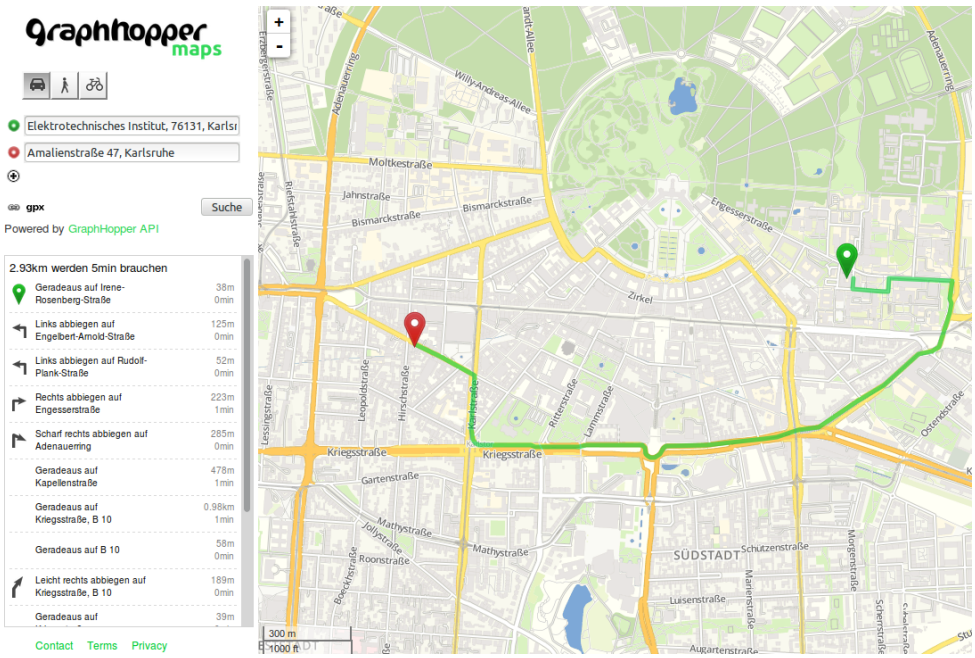


Figure 3.4: The default GraphHopper user interface with a route selected

more and more requirements will be discovered and/or understood more deeply throughout development—, the architecture should still be able to cope with future growth.

Because the final goal is to get a driving cycle from the simulation results, data must be recorded throughout the simulation. For this application, only observing the velocity as an external value would already be enough. For a more detailed analysis of simulation behaviour, it is however desirable to save also internal states of the system. Therefore, a generic concept to store all signal values in short time intervals was implemented instead of an external velocity observer. This was combined with a module that extracts the data and reports it back to a client for further processing. Analysis scripts to plot graphs from this data were also created, but they are not part of RoadHopper itself.

3.3.3 User interface

GraphHopper already exposes a web interface for input (coordinates) and output (the routing results). A screenshot of this interface can be seen in figure 3.4. This interface was extended to support starting a simulation and playing back its results, see section 5.6.1.

Although not in the strict sense of *human–machine interface*, Application Programming Interfaces (APIs) are also an interface from the system to the outside world. In addition to the existing API endpoints of GraphHopper, new ones were created to power both the fine-grained

route display and the simulation control (starting, status reporting, results view). The signal values recorded during a simulation can also be fetched by using a dedicated endpoint, see section 5.7.

3.4 Driving cycle computation

As explained in the simulator concept above, telemetric data is constantly stored during the simulation. This data can be used afterwards to calculate the driving cycle (i.e. a t–v diagram) and perform other statistical, spatial or temporal analyses.

As the recorded data is already time-indexed, the computation of a t–v driving cycle is a mere matter of extracting the data into a plotting-compatible format. Interpolation is not required, as the data can be recorded with a frequency of up to 100 Hz.

If multiple simulation runs on the same road should be compared, it might be more sensible to plot the velocity as a function of the travelled distance `s` instead of using the time `t`. This way, the model's reactions to certain road conditions like traffic signs, slopes or road bends can be better assessed compared to a time-indexed representation.

4 Map processing with GraphHopper

Based on the concept given in the previous chapter, this chapter details the first step of the driving cycle process—extracting road data from a map. The software used for this is GraphHopper, which was also already introduced in chapter 3.

This chapter consists of three main sections, which detail

1. the data model and import process, and the changes made to them,
2. the routing algorithms used, and
3. the basic processing of the routing results.

The construction of the road model is detailed in the next chapter, as it rather belongs to the simulation models than to GraphHopper.

4.1 Data model

To make GraphHopper routing work, it needs data to perform its algorithms on. For routing, only a subset of the information available in OpenStreetMap is actually required. Therefore, a two-step import process is used to (1) reduce the data set and (2) convert it to an optimal format for routing.

The reduction step actually removes all OSM ways that are not tagged as roads, e.g. walls of buildings or footways. Subsequently, all nodes are ignored that are not part of one of the interesting ways. The remaining nodes are grouped into two parts:

- tower nodes—intersections of two or more ways, which makes them interesting for routing—, and
- pillar nodes which only belong to one way in the graph and are only relevant for drawing the route in the map.

Tower nodes also have a unique ID within GraphHopper by which they can be directly addressed. In contrast, the pillar nodes are stored as an edge property and can only be retrieved if the edge is known.

Only tower nodes can have additional properties, so the import process was changed to convert pillar nodes to towers if a traffic light or other relevant info is attached to them. The relevant code part is located in GraphHopper's `OSMReader` class. This change does not affect other parts, e.g. the routing, even though these converted tower nodes only have one or two edges they are connected to, in contrast to the three+ normal tower nodes have.

4.1.1 Import process

GraphHopper can import an OpenStreetMap data dump in both the XML and PBF formats. The dump can also contain only a small part of the world[11].

To tell tower and pillar nodes (and the other, irrelevant nodes) apart, GraphHopper actually further divides the first import step into two parts, so the total process consists of steps 1a, 1b and 2[12].

In step **1a**, all ways are traversed and all nodes are marked; these markings are simple counters incremented each time the node is used. The nodes can then be grouped by their counter value n:

- $n = 0$: not part of a road $\Rightarrow$ ignore
- $n = 1$: pillar node
- $n \geq 2$: tower node

During step **1b**, all nodes, ways and relations which are relevant for the routing graph are processed. For tower nodes, an entry in the node map is created, together with a set of node flags (bits stored in a large integer).

OSM ways are split at tower nodes and the pillar nodes in between are transformed into a list of points for the edge. This list is stored as a property of the edge, making the pillar nodes only reachable through their edge; unlike tower nodes, they don't have a unique ID which makes them directly accessible.

Endstanding pillar nodes are also automatically converted to tower nodes (e.g. for blind alleys).

[11] To speed up development, a large part of the work for this thesis was actually conducted with a dump of only the Karlsruhe City area. Larger data sets were only used for validating finished results.

[12] This numbering is introduced in this thesis and not part of the original GraphHopper source code.

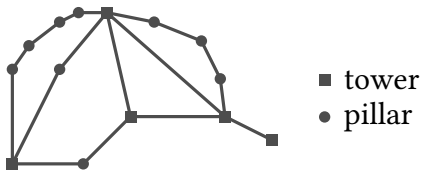


Figure 4.1: An example road network with tower and pillar nodes

Step **2** is dedicated to optimising the routing graph: The contraction hierarchies are introduced to speed up routing, and small separate subnetworks are removed. Such networks can e.g. exist because of road barriers that isolate a road network (e.g. in military facilities).

4.1.2 Storage Extension

The GraphHopper storage implementation already has an integrated mechanism for adding information to the routing graph: the so-called *storage extensions*. One of these extensions can be added per graph, and it can contain additional flags for nodes and edges.

The flags are simple bit sets embedded in a large integer. By using a custom helper class (called *encoder* in GraphHopper), these details can be abstracted away and arbitrary numerical values can be stored. GraphHopper itself also uses this pattern e.g. for encoding the allowed speed: The speed is divided by a fixed factor and only the (rounded) result is stored. For the speed, this saves three bits per edge: The speed limit is capped at $155\,\mathrm{km\,h^{-1}}$, it would thus need $\lceil \mathrm{ld}(155) \rceil = 8$ bit. When using a scaling factor of five, the values to store are effectively shrinked to 0 to 31 ($^{155}/_{5}$), thus needing only $\lceil \mathrm{ld}(32) \rceil = 5$ bit.

For usage in this thesis, a custom storage extension (`RoadSignExtension`) was developed to store additional information. Over the course of developing RoadHopper, information about road signs and traffic lights was added. As we only need information if either of them is present[13], they can be encoded in the same block. In general, information that is mutually exclusive can be encoded using the same "place", like it is done here.

There are (currently) three possible values of the road sign flag:

0: no road sign

1: traffic light

2: stop sign

[13] Road signs and traffic lights both are mutually exclusive properties of a node in OSM; cf. documentation on the `highway` tag in OSM.

Using the formula $\lceil \mathrm{ld}(n) \rceil = \lceil \mathrm{ld}(3) \rceil = 2$, we see that this piece of information will use 2 bits. The fourth state available with two bits is currently unused. There are at least two possible usages for this:

1. "give way" signs (as an alternative to the stop signs, requiring a slowdown, but no halt for one second as stop signs), or
2. a distinction between regular traffic lights and button-operated pedestrian traffic lights.

If a traffic light has a subsidiary stop or "give way" sign attached to it, this is currently not respected at all; the effect of such a sign on the cycle would likely be negligible, as the signs would only be valid if the traffic light is turned off, which will not happen during the simulation. What might be more interesting is modelling the *green arrows* placed at traffic lights in Germany which during red phases effectively turn them into a stop sign for right turning vehicles. As the behaviour would then be different for right turns and straight driving, the green arrow might have a partly higher influence than the aforementioned subsidiary signs.

4.2 Routing algorithms

GraphHopper supports two shortest path routing algorithms in different variants:

- A* in uni- and bidirectional variants
- Dijkstra in the variants uni-, bidirectional and one to many

By default, the bidirectional Dijkstra variant is used[14].

In addition, contraction hierarchies are supported, with different possible weights: "fastest", "shortest" or a user-defined weight[15].

4.3 Route processing

For routing, GraphHopper is fed with a list of n points, $n \geq 2$. All points between the start and end are called *intermediate* points. The output of the GraphHopper routing run is a list of edges that, when traversed, lead from the start point to the end point, going over the intermediate points in their given order.

[14] see https://github.com/graphhopper/graphhopper/blob/bb5de58/core/src/main/java/com/graphhopper/routing/AlgorithmOptions.java#L58; visited 2015/10/01

[15] see class `com.graphhopper.GraphHopper`, method `setCHWeighting()`

Every edge has a base and an adjacent node, both of which are tower nodes, and pillar nodes in between which are only relevant for drawing the route. For every edge, all nodes (tower and pillar) must be taken into account, as we are interested in all coordinates along the route. To get a road as the basis for a simulation, this node list must be transformed into a format suitable for simulation. From this step on, everything described here has been newly implemented for this thesis.

For every set of two consecutive nodes in the node list, a straight road segment is constructed, with the two nodes' coordinates as start and end points. The final road is a sequence of straight road segments directly attached to each other, without any bends in between. Why such curved segments don't need to be fully implemented for the simulation will be detailed later.

In addition to the coordinates, the road is enriched with the road sign (if any) and its allowed maximum speed according to the information tagged in OpenStreetMap. As road signs belong to a node, they are added to the road segment that ends at that node.

4.3.1 Calculating the road segments

A lot of information is required for the simulation, which can only be partly read from the GraphHopper database (like the allowed maximum speed).

The length and grade are not directly stored, but can be derived from the coordinates. This operation is not only done in preparation of the simulation, but also during it. It should therefore be fast to not slow down the simulation. This is why the accurate formulae given e.g. in [Mey10] are not usable: they would require a lot of costly trigonometric operations.

To speed up calculations, the earth is assumed to be a sphere. Therefore the haversine formula can be used [Mey10], which makes calculating the distance between two points possible with little effort. The formula can also be found in section 2.2.3. For the earth's radius r_E, the medium ellipsoid radius of 6371 km from WGS84 is used; see table 2.2.

The grade of the route is calculated by dividing the height difference by the length:

$$g = \frac{\Delta h}{l} = \tan(\alpha) \tag{4.1}$$

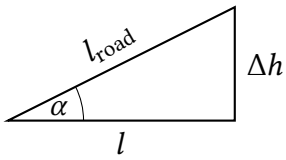


Figure 4.2: Road length vs. base length

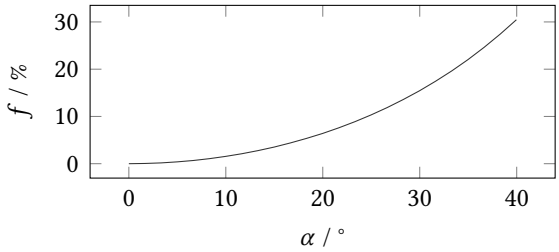


Figure 4.3: Relative error between base length and actual road length

A graded road also means a slight difference between the length calculated above and the actual length travelled along the road. The actual length can be calculated using the well-known trigonometric relations:

$$l_{\text{base}} = l \tag{4.2}$$

$$l_{\text{road}} = \frac{\Delta h}{\sin(\alpha)} = \frac{l_{\text{base}}}{\cos(\alpha)} \tag{4.3}$$

As can easily be seen from figure 4.2, the relative error between the two lengths is

$$f = \frac{l_{\text{road}} - l_{\text{base}}}{l_{\text{base}}} = \frac{l_{\text{road}}}{l_{\text{base}}} - 1 = \frac{1}{\cos(\alpha)} - 1. \tag{4.4}$$

The relative error is plotted in figure 4.3. For grades from 0° to 20°, it is less than 6 %. For higher grades, the error increases (10 % for 25°, 15 % for 30°), but such steeply graded roads are relatively rare. In fact, for rural roads in Germany, grades higher than 8.0 % (≈ 5°) are already considered exceptional [RAL, ch. 5.3], so the difference between base and actual road lengths should be very small, even if accumulated over longer distances.

As the errors are quite small, they are currently ignored in the simulation.

4.4 Road postprocessing

After the road has been created from the list of edges, it is further processed to insert more information required for the simulation. This includes adjusting the speed limits for some parts, e.g. the section right before a turn, or curves in the road that cannot be travelled with the allowed speed.

The postprocessing is tightly coupled to the simulation model and therefore not described in detail in this chapter. Refer to section 5.4.2 instead.

5 Driving cycle simulation with RoadHopper

This chapter describes the architecture, design and implementation of RoadHopper, the driving cycle simulation software created for this thesis. In section 5.4, the genesis of the simulation models is also presented.

The later sections also give a short overview on how to use RoadHopper and on postprocessing the gathered data.

5.1 Architectural overview

Categorizing the components of RoadHopper is possible in different ways. With the simulation as the central part of the software, the most logical approach to describing the system is a division into two categories, simulation and infrastructure, with two parts each:

Simulator engine The simulation runtime with the core and utility components (results logging etc.)

Simulation models The control blocks that implement the actual system model.

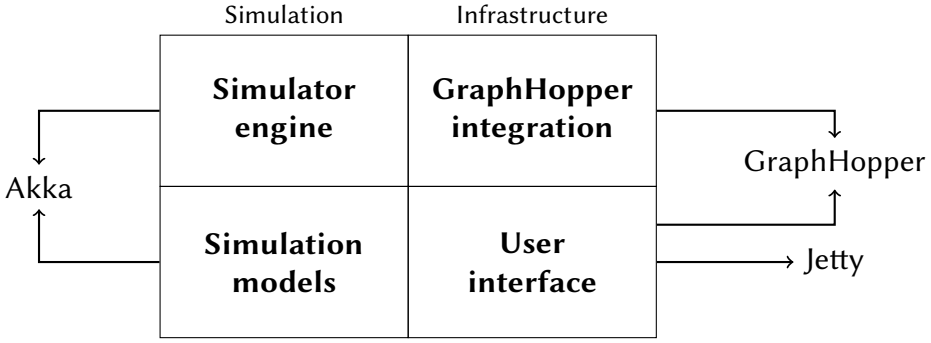


Figure 5.1: RoadHopper: architectural overview

GraphHopper integration The code that extracts data from GraphHopper as a basis for the simulation, and extensions to GraphHopper for storing simulation-relevant data.

User interface The client-side (HTML/JavaScript) parts for controlling RoadHopper and their server-side servlet pendants.

The parts with their most important external dependencies are also depicted in Figure 5.1.

The simulator engine consists of the main components *timer* and *signal bus*, which are discussed in the next section. All other simulation components are connected to either the timer or bus, depending on their purpose.

The infrastructural components are covered in sections 5.5 to 5.7.

5.2 Simulator model

Being a—hugely simplified—model of the real world, the simulation consists of a number of components which represent physical or biological systems or parts thereof. These models are naturally an incomplete depiction of their real counterparts. Instead, they serve as a means to abstract a complicated system to a level where it is possible to comprehend its role in the simulation. In addition, these simplifications make the simulation possible in the first place: running a fully-fledged model of a road–vehicle–driver system would probably require a lot more computing power than usually available, without necessarily leading to better results.

All components of the simulation are intended to function independently of each other to the largest possible extent. This especially means that all data should be kept locally and only be modified by the component responsible for it (i.e. the (sub)system that influences the real-world property). This approach helps avoiding the problem of overlapping changes to the same data by different components.

In addition, extensibility is a primary goal of the whole simulation model. Therefore, the models currently used for the vehicle and the driver can easily be exchanged for more accurate counterparts. Also, additional components can be introduced to make the simulation more realistic, e.g. other vehicles in traffic or a more realistic physics model (varying road surfaces, a wind generator etc.)

Given these requirements, it was pretty obvious that the basis for the simulator model needs to support highly parallel processing and easy decoupling of components. The actor model seems a natural fit here and was therefore chosen as the implementation model for the simulation.

5.2.1 Timer

The central component of the simulation is the timer, which serves as the internal system clock. As such, all components that need to have a notion of time need to be registered with the timer before the simulation starts.

The timer controls the progress of the simulation and makes sure that all components are called at the right time using an internal schedule of future invocations. All invocations of a component need to be registered with the timer. If multiple components are scheduled for one time slot, they are called in an arbitrary order[16].

Simulation start and stop

The simulation is started and stopped by explicit messages sent to the timer. The start is done from outside the simulator system, the final message is sent from within it.

At the start of the simulation run, the timer calls each component for a first time to let them initialize their internal state and schedule the first actual invocation. Afterwards, a loop is executed over and over again until the timer is instructed to stop the simulation.

The stop message is sent by the road watcher which keeps track of the already travelled and remaining route (and updates signals like `s` for the travelled distance and `pos` for the vehicle's current position). The simulation must be ended by the timer as most components do not know enough about the system state to decide on their own if they should schedule another invocation or not. Hence, the simulation would continue endlessly without the explicit stopping procedure.

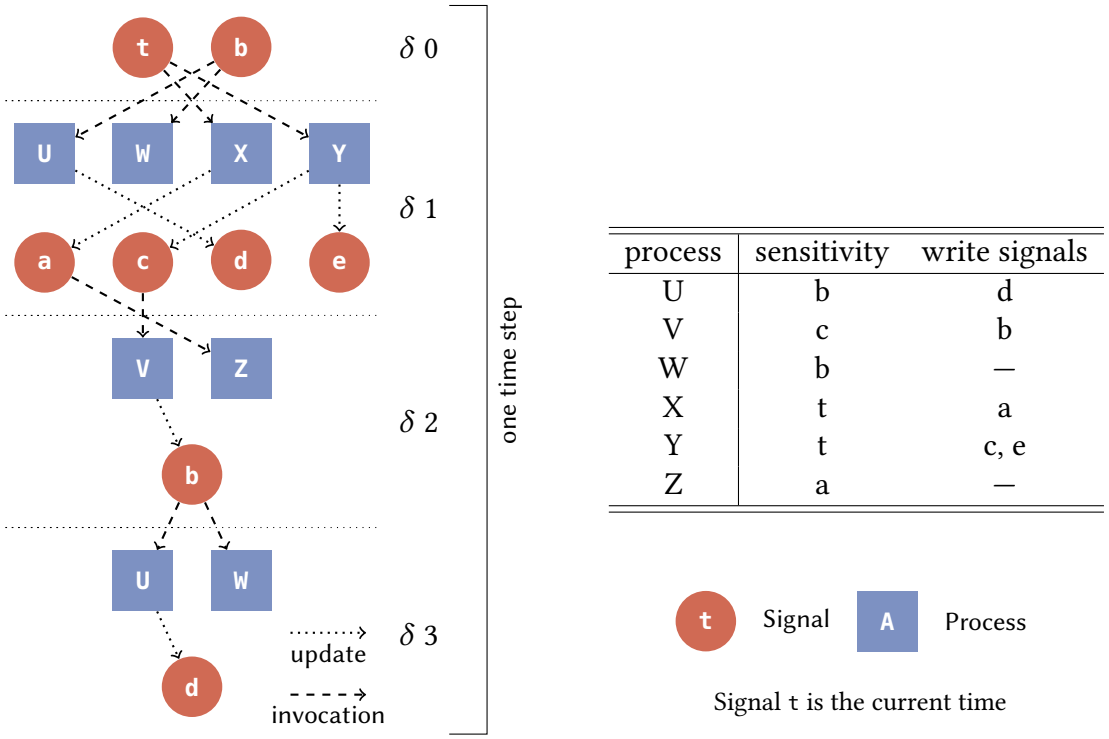
After receiving the stop message, the timer will not advance the time any more. Also no further scheduled invocations will be performed, so components cannot rely on scheduled calls to be really made.

Time steps

For every time step, three consecutive steps are executed by the timer:

1. tell time (all components)
2. update step (only scheduled components)
3. act step (only scheduled components)

[16] In fact, multiple components are called in parallel by the timer.



process	sensitivity	write signals
U	b	d
V	c	b
W	b	—
X	t	a
Y	t	c, e
Z	a	—

Figure 5.2: Example of signal updates throughout the delta cycles

While the time step is only used for synchronization, *update* and *act* are two distinct, but connected phases of the clock step: During the update step, all components can update their data, which can then be used in the act phase for reacting (e.g. by adjusting the commands as a reaction to a change in velocity or the travelled distance).

5.2.2 Signals

The separation into update and act implemented in the timer proved to not suffice for complex cause-and-effect chains, such as the driver–vehicle system. Therefore, other simulation models were investigated to find a better solution. One such simulator model, which was chosen as the basis for RoadHopper's model, is part of VHDL (Very High Speed Integrated Circuit Hardware Description Language).

The signal model of RoadHopper uses three concepts defined in VHDL:

Signals The central part of the signal model: A named container which can hold an arbitrary value (numbers, strings, objects). Signals are written to by one component and can be read by an arbitrary number of components. The signals are managed by the signal bus, which also keeps track of all components that can read or write signals.

Processes A unit of computation which can be invoked by the signal bus. Processes can listen to a number of signals, including the time, and update any number of signals.

Delta cycles One process execution inside the signal bus. Multiple delta cycles might be run inside one time step. An example of one such time step with three delta cycles is depicted in figure 5.2.

The signal-based scheduling model is implemented in addition to the timer described above, which still provides the central clock with its discrete time steps. Inside every time step, the signal bus now performs one delta cycle. Additional delta cycles will be appended depending on the signal value updates done during the cycle.

Besides being a cleaner approach than mixing the two responsibilities of time and data synchronization, the separation also opens a possibility to have multiple centrally coordinated signal busses, e.g. for different vehicles or to separate components within one vehicle.

5.3 Simulator implementation

As mentioned above, the actor model was chosen for the simulation framework: The simulator and all its components (including those based on signals) are implemented as separate *actors*. The underlying actor engine is Akka, a software created by the Swedish company Typesafe. Their creators describe it as

> a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM[17].

Some other parts, which will be described later, are implemented without the actor model, as it is not necessary for their purpose.

All parts of RoadHopper run inside an instance of the JVM (Java Virtual Machine). The inner workings of the JVM and Akka will not be described in detail here; instead, their documentation should be consulted.

5.3.1 Actor system setup

The base of each actor-driven application is an actor system, which can be created as part of a regular Java-based application. The simulation is set up by a few lines of code, which create an actor system and all the components. See listing 5.1 for an example with two components.

[17] http://akka.io; visited 2015/10/01

```
package info.andreaswolf.roadhopper.simulation

import akka.actor.{ActorSystem, Props}
import info.andreaswolf.roadhopper.simulation.signals.SignalBus
import info.andreaswolf.roadhopper.simulation.vehicle.VehicleFactory

val actorSystem = ActorSystem.create("signals")

val timer = actorSystem.actorOf(Props(new TwoStepSimulationTimer), "timer")
val signalBus = actorSystem.actorOf(Props(new SignalBus(timer)), "signalBus")

// vehicleParameters and route are supplied from the outside
val vehicle = new VehicleFactory(actorSystem, timer, signalBus).createVehicle(vehicleParameters)
val journey = actorSystem.actorOf(Props(new SignalJourneyActor(timer, signalBus, route)), "journey")
```

Listing 5.1: Simulation actor system setup

As described above, the central simulation component is the timer. Before handing over control, the components must be registered with the timer, using `RegisterActor` messages. The answer to these messages must be awaited to confirm that the component has been registered.

Once the surrounding setup code has handed over control to the timer, it will automatically run the simulation until the end. An example code for this is printed in listing 5.2.

```
Future.sequence(List(
  timer ? RegisterActor(signalBus),
  timer ? RegisterActor(vehicle),
  signalBus ? SubscribeToSignal("s", journey)
  // further subscriptions would go here
)) onSuccess {
  case _ =>
    println("Starting")
    timer ! StartSimulation()
}
```

Listing 5.2: Component registration and simulation start

5.3.2 Messaging

Most of the messages in RoadHopper are implemented using the ask pattern, to allow controlling time flow: When a component is called via ask, it returns a future that is completed after the answer to the message has arrived. Therefore, the timer can use the futures to control if all components have finished the current time step.

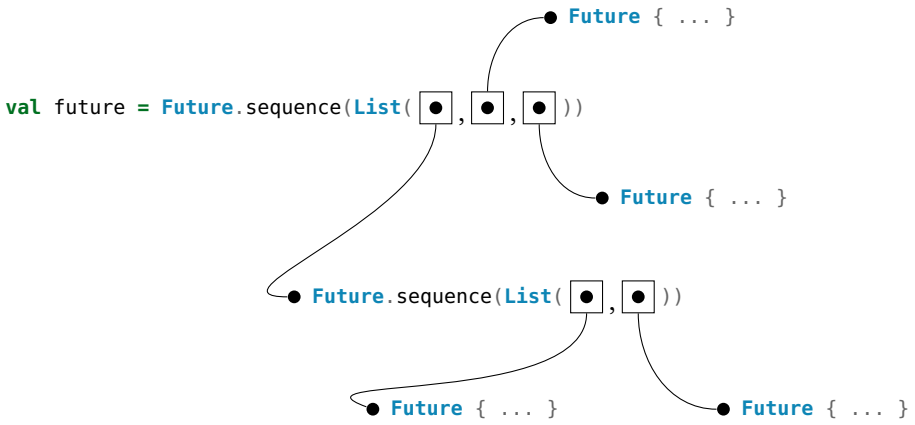


Figure 5.3: Example of chaining futures

As it returns a future, sending a message with `ask` does not block the application. A future can have code applied that will be executed as soon as the future is completed. Multiple such futures can be merged with `Future.sequence()`. An example can be seen in listing 5.2 and figure 5.3.

A component can in turn use the same pattern to call other components and wait for all their results. It will then wrap the futures returned by the messages to its subordinate components into one single future. As soon as this future has finished, it is safe to return an answer to the timer. The different call levels during one time step form a tree-like hierarchy, with the timer being the root. Each leaf needs to finish before its parent node can finish, and their parent nodes in turn need to wait for their children. In other words: each parent–child relation is a "waits for" relation. These relations are both transitive, so a parent also waits for its grandchildren. The timer, being the parent to all other components, waits for all of them to finish.

5.3.3 Signal bus implementation

The call-and-wait pattern described above is also used in the signal bus. Apart from that, the processes are completely different: The timer can finish each time step after one round of update and act messages, while the signal bus does neither know in advance how many rounds (delta cycles) there will be nor which components to call for each step. An example of a time step with three delta cycles is depicted in figure 5.2.

Each process can update every signal by sending an `UpdateSignalValue` message to the signal bus. The update is not executed immediately, but delayed until all running processes have finished, i.e. the end of the current delta cycle. Effectively, the signal values during one delta

cycle are frozen. If necessary, signal updates can also be further delayed to a time slot in the future.

This value freezing is the main advantage of using delta cycles: There is one point in time where a list of all called processes is created (the start of a delta cycle) and until these processes are finished, no further processes will be invoked. This also means that a process must not invoke other processes by itself. Nevertheless, a process can call other components, if they are not a process (but a regular actor or normal object). Such a call should not be named an "invocation".

When sending messages or calling other actors, processes must make sure that they only complete their invocation when all subsequent processing has been finished. For that, they must always use the ask pattern for sending messages (`?` operator) and collect all futures returned from sending messages. When all these futures have completed, a response must be sent to the signal bus, telling it that the component has finished processing.

5.4 Simulation models

In this section, the detailed design and the rationales behind the three main simulation models

1. road,
2. driver, and
3. vehicle

are discussed. The general system model has already been shown in figure 3.3.

5.4.1 Designs considerations

The road model naturally derives from the representation in OpenStreetMap and GraphHopper—there, a road is already deconstructed into single parts. These parts can be directly mapped into a format suitable for the simulation. Given the considerations in the last chapter, no changes to the road shape are required, which simplifies the mapping. In contrast, the driver and vehicle models deserve more attention.

The question of modelling vehicles and a human driver has been covered in great depth already. Various models with different intents have been proposed.

A good general overview of driver design considerations is given in [Mac03]. This paper also features an interesting approach to driver–vehicle system modelling in general:

> To sum up, we see that the combination of human parameters and of mechanical parameters enter into the process of driving in a manner which does not permit their clear-cut separation. The car and the driver form, in a sense, an individuum.

This approach could not be followed for this thesis: not enough data and experience was available to create such an integrated model. Also the intent of the simulation is to parameterize the driver and use different vehicles. Therefore, separate models were created instead, based on examples from literature.

5.4.2 Road model

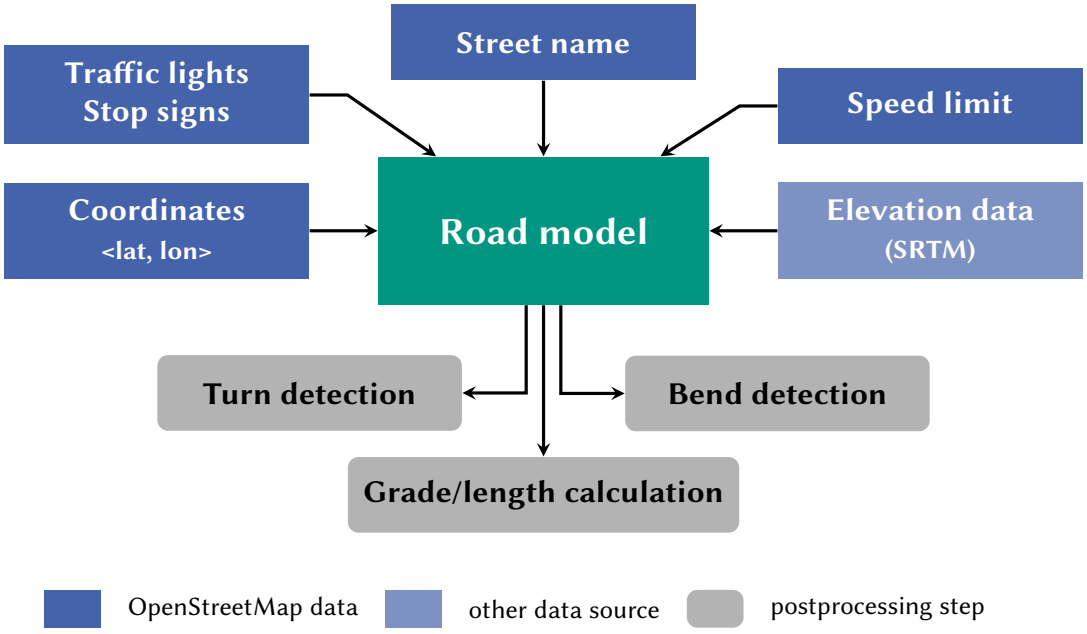


Figure 5.4: The road model components

The road model contains the route to travel in the simulation, delivering parts of the stimuli necessary for the driver to decide on its actions. To deliver its value, it must incorporate information from various sources. The different kinds of information and the necessary postprocessing steps are depicted in figure 5.4.

The road consists of a list of connected segments the vehicle drives along. Each segment can optionally have a road sign or traffic light at its end. The single segments are constructed from the routing coordinates, using the information from the GraphHopper database and elevation data from the Shuttle Radar Topography Mission (SRTM). The length and grade of the road are calculated using the formulae discussed in section 2.2.3.

Compared to its real-world counterpart, the road model is simplified in a few aspects:

- the road has only one lane
- there are only straight segments with abrupt changes in orientation at the transition
- the road category (highway, residential etc.) is not evaluated
- different kinds of road surfaces (concrete, bitumen etc.) are not incorporated[18]

As the simulator model does not include a lateral control, i.e. it only controls the longitudinal speed of the vehicle, straight road segments are sufficient for simulating; it is not necessary to create bended roads that can be followed by the vehicle (or simulate such a movement based on the straight road).

The other simplifications can be removed once the vehicle could use this information (road surface) or the driver is more sophisticated (multiple lanes, road category). The road category could also be used for various other simulation purposes, which are detailed in chapter 7.

Speed limit adjustments

In general, the driver can directly take the target velocity from the road, as detailed below in the driver model. At some occasions, a reduction of this velocity is however required. It was for now decided to incorporate these changes already in the road model, to keep the driver implementation simpler. This could as well be changed in favor of a more complex target speed derivation model in the driver (which would be preferable in case multiple vehicles with different driver characteristics are to be simulated).

As the vehicle–driver model discussed below does not account for lateral steering, the direction is not changed by the driver. Instead, the vehicle orientation is abruptly changed once the border between the two road segments was passed (i.o.w. the vehicle always points in the direction of the current road segment). Changing the direction of a vehicle induces an acceleration in lateral direction (centripetal force), for which certain limits need to be kept in order to keep the driving experience comfortable. The lateral acceleration can be calculated to

$$a_{\text{lat}} = \frac{v^2}{2 \cdot r}, \tag{5.1}$$

with r being the radius of the curve.

[18] This information is encoded in OpenStreetMap with the key “surface”, see https://wiki.openstreetmap.org/wiki/Key:surface; visited 2015/10/01

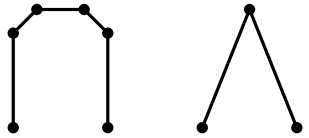


Figure 5.5: U-turn road: two road examples

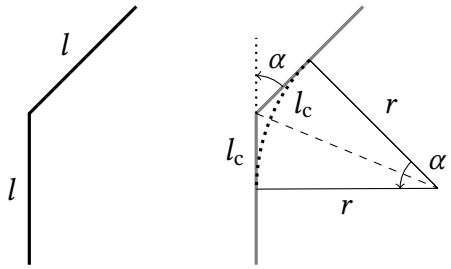


Figure 5.6: Curve radius approximation

The following situations must be considered separately:

1. turns—a sharp bend between two road segments, usually to change from one street to another, that can only be travelled with a very low speed
2. road bends—a curvature in the road, going over several segments, which can be travelled with a relatively high speed compared to a turn
3. road signs, traffic lights—these are already covered in the road model construction process

The general approach to both road bends and turns is to check if the regular speed defined for the segments before and after is feasible. If this speed would result in a too high lateral acceleration, the speed is reduced accordingly.

This behaviour relies on the underlying data containing only well-formed curvatures. This means that e.g. u-turns must be modelled as multiple segments with an increasing change in direction vs. the direction before the turn. One single change is not possible, see figure 5.5. For most of the data, this can be taken for granted; for extremely rare edge cases special precautions should be put in place (e.g. throwing a warning during road construction if a turn angle is greater than ca. 135°).

Road bends Curves are modelled as sequences of straight segments in OpenStreetMap. A real curve can be approximated by laying a virtual circle through the straight segments. The circle radius is determined by viewing the two segments as an arc. With the full length being $l = 2\pi r$, the radius can be calculated to

$$r = l_{\text{road}} \frac{\alpha}{2\pi} \tag{5.2}$$

To better approximate real road curvatures, only the halves of the segments next to the arc should be used, as shown in figure 5.6. The other halves are either part of the preceding/succeeding arc, or belong to the straight road before/after the road bend.

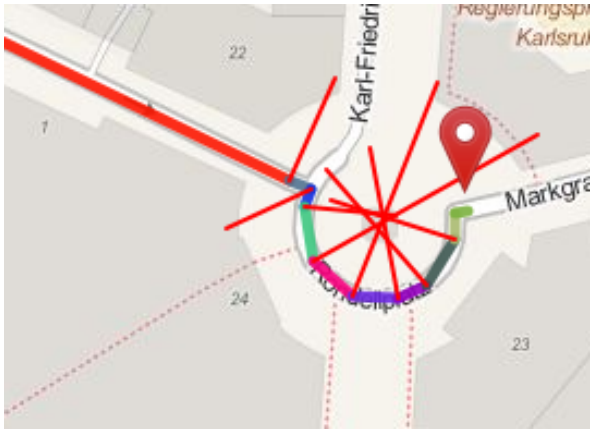


Figure 5.7: A roundabout with road bend help lines

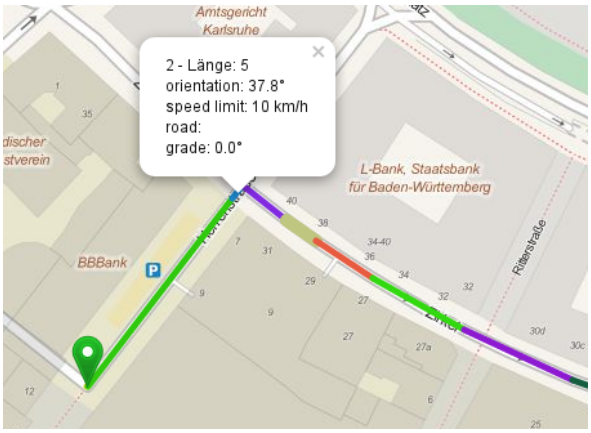


Figure 5.8: A pre-turn road segment inserted by the road postprocessing

turn angle	speed limit
> 110°	7 km/h
> 80°	10 km/h
> 45°	15 km/h
> 30°	25 km/h
≤ 30°	*unchanged*

Table 5.1: Pre-turn road segment speed limits

This implementation should be treated as preliminary, as the length and shape of the curves differs greatly e.g. between cities and highways. On highways, the segments are often relatively long, proportional to the allowed speed, as the roads are built for an optimal flow of traffic. Most highway curves can thus be assumed to not need a speed reduction; [Zie10] mentions a curve radius of 500 m at which no change in driver behaviour is noticeable anymore. Problems have been observed however with junctions, where often the allowed speed of $> 100\,\mathrm{km\,h^{-1}}$ of the main road is also applied to the junction parts.

In cities, the segments can be very short and require some speed reduction. There are certain situations—most notably smaller roundabouts—where the allowed speed is physically impossible to travel.

Turns Turns require a huge decrease in speed compared to a straight road, as the direction is changed with a very small radius.

The radius approximation used for bends would not work here: the segments before and after a turn are often so long (see e.g. in figure 5.8) that the virtual curve arc would directly go

coefficient	proportional	integrator	differentiator
value	−0.0069	$-2.59 \times 10^{-6}\,\mathrm{s}^{-1}$	$5.35 \times 10^{-5}\,\mathrm{s}$

Note: The software implementation of the controller uses controller coefficients converted to ms.

Table 5.2: PID controller coefficients from [LM09]

through the surrounding buildings. Therefore, a fixed radius is assumed instead, depending on the turn angle.

To realise the necessary speed change, a short segment with a very low speed limit right before the turn is introduced. The length of this segment is currently fixed to five meters, and the speed limit depends solely on the angle. An example of such a segment can be seen in figure 5.8. The speed limits in table 5.1 are currently used for the pre-turn segment. They were calculated based on an assumed comfortable acceleration of $5\,\mathrm{m\,s^{-2}}$, which was derived from literature such as [FN98], where a range of 0.35 g to 0.40 g is mentioned as the comfortable acceleration range.

5.4.3 Driver model

For validation of the general simulator approach, some experiments were conducted using a strongly simplified model of the driver. This first implementation was discarded in favor of a model proposed in literature and should not be further discussed here.

A huge number of different approaches for modelling a human driver has been published. [Mac03] gives a good general overview of the topic and provides insight into various modelling ideas. Besides the finally implemented approach from [LM09], a few others have influenced the system design and should therefore also be presented here.

An early and very detailed model of the interaction of driver and vehicle is presented in [Don78], with a special focus on lateral steering. [HM90] discusses a model containing various blocks for the body parts involved in steering, like the neuromuscular system around the arm, and the inherent time delays of human signal processing. These two have influenced the general design of the system and should also play a role in further refining the model at hand.

The model proposed in [HM90] is further detailed in [MH93]. The latter also gives a detailed explanation why a single-loop system is too simple for lateral steering. As we focus on longitudinal steering, this should not be further detailed here.

A particularly interesting take is presented in [LM09], where longitudinal and lateral control tasks of a driver are separately implemented using controllers. The controller coefficients were

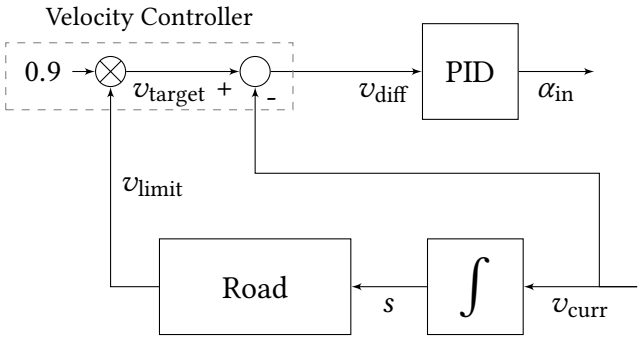


Figure 5.9: Velocity controller and driver PID controller

derived from driving simulator experiments and a following linear regression analysis. They are printed in table 5.2.

The controllers used by the model is one single PID controller each, which leads to a very simple structure of the whole driver implementation. This made the solution a good fit for validating the general simulator model.

Velocity control

For the longitudinal control part of the model, the delta between target and actual velocity is fed as the input into the PID controller.

The target velocity is fetched from a velocity-dependent part of the road ahead. This so-called *lookahead distance* is computed to

$$s_{\text{lookahead}} = \frac{v^2}{2 \cdot b} \tag{5.3}$$

with b being the *comfortable braking deceleration* in $\mathrm{m\,s^{-2}}$. [LM09] recommends $b = 8.0\,\mathrm{m\,s^{-2}}$, but this was found to be very high, resulting in a too short lookahead distance. The value was at first lowered to 4.0 for the tests conducted for this thesis and later reduced again down to 1.0. A detailed discussion of the reasoning for this is included in section 6.3.

The system part deriving the target velocity is depicted in figure 5.9. The estimator takes all road segments within the lookahead distance into account and selects the minimum allowed velocity as the target velocity

$$v_{\text{limit}} = \min \left\{ v_{\text{limit,seg}} \right\} \qquad \forall\ \text{seg} \in s_{\text{lookahead}} \tag{5.4}$$

with seg being a road segment.

Directly using the allowed speed on the current road segment would lead to a too optimistic result for most traffic conditions, as it assumes a negligible traffic load. To compensate for that, a general factor of 0.9 is applied to the speed limit, reducing the allowed velocity by 10 %. This also helps compensating an observable velocity overshoot when accelerating from standstill.

The resulting velocity difference v_{diff} between the target and actual velocity is fed to the PID controller. The controller was implemented using coefficients from [LM09]. They are also printed in table 5.2 for reference.

Traffic sign behaviour

The velocity control task of the driver includes reactions to everything that might affect the allowed speed of the vehicle. Besides the road bend/turn behaviour discussed in the road model, this especially includes traffic lights and signs like stop signs.

Traffic lights are already part of the road model, but currently have no functionality implemented behind them. They are therefore always treated as if they were green.

Stop signs must trigger an always-stop behaviour with the driver—according to official regulation e.g. in Germany, a vehicle must always come to a halt [StVO, Anlage 2]. Therefore, when encountering a stop sign within the lookahead distance, the driver switches to a different operating mode. In that mode, it stops the vehicle, waits for a second and then accelerates again to the target velocity it used before encountering the stop sign.

5.4.4 Vehicle model

For the vehicle model, special emphasis was put on modelling the power train, as it is the most relevant component for simulating a driving cycle. The model is similar to the ones discussed in [SHB10, Fig. 8.14] and [GME07, Fig. 14]. In addition, the driving resistances were modelled as part of the wheels. As no lateral control is required, the steering part was completely left out.

The main devices within the power train are the engine (influenced via the gas pedal) and the brakes (controlled with the brake pedal). The torque released by the engine is transmitted to the wheels through the transmission, normally involving a complex physical model for the coupling, which is left out for the sake of simplicity. The vehicles used as a basis also have a fixed transmission factor, so no gear changing needed to be modelled.

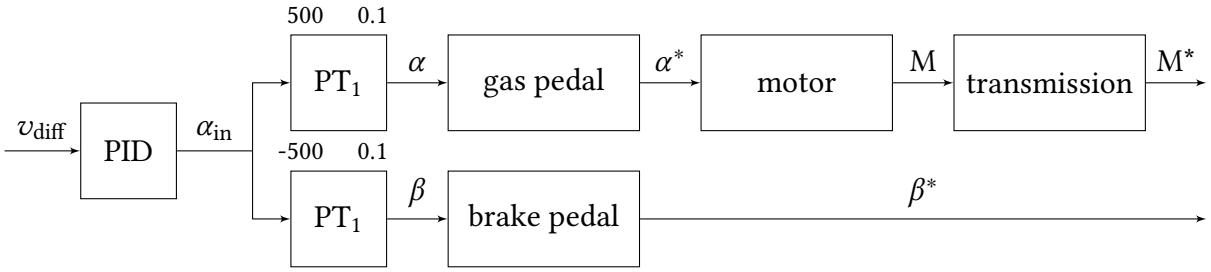


Figure 5.10: Model of the vehicle input train

Power train input

Figure 5.10 shows the input/output model from the power train. α_{in} is the output of the driver, which serves as the input for both gas ($\alpha_{in} > 0$) and brake pedal ($\alpha_{in} < 0$).

Both the engine and the brakes do not directly react to changed driver input, but show some delay e.g. due to inertia of the various components. This is modelled using PT_1 controllers directly after the driver input. Their time constants were estimated based on literature, but should be given further attention when refining the model. Also the vehicle exhibits some resistance to acceleration due to inertia, which is modelled as part of the wheels.

Gas and brake pedal outputs both are modelled as linear components, with a value range of 0 to 100. In reality, these components are however nonlinear [SHB10] [Mit14]; this is discussed again in section 6.2.

To achieve the pedal value range, the huge amplification factors for the two PT_1 controllers are necessary. The initial values of ±500 were educated guesses based on the observed output of the PID controller. They were later reduced when it had become apparent that the driver model was too aggressive.

Transmission

The transmission is modelled as a simple PT_1 controller with a fixed transmission ratio as included in table 5.3. The time constant was estimated to 0.1 s. The usage of a PT_1 controller for modelling the transmission was suggested in [SHB10, ch. 8.3].

[SHB10] also includes an additional dead time element right after the PT_1 controller, which was not included in the model, as the cited source contained no indication of a realistic value for its time constant. When refining the model, such a component should be added as soon as the time constant of the existing PT_1 controller has been validated.

part/system	parameter	Opel Ampera	*small car*	unit
vehicle	mass	1732	1325	kg
	c_w	0.27	0.29	—
	drag area	2.57	2.4	m^2
engine	maximum torque	370	200	N m
	maximum power	111	84	kW
	maximum rot. speed	12000	12000	min^{-1}
transmission	translation factor	9.4	10	–
wheels	rolling friction coefficient	0.012	0.012	—
	radius	0.334	0.32	m
	maximum braking force	500	500	N

The maximum braking force was estimated based on [Mit14, fig. 9.34].

Table 5.3: Vehicle parameters; source: MATLAB driving cycle computation model used at HEV

Engine

The engine was modelled with the parameters given in table 5.3, which were taken from the existing Matlab model for reverse driving cycle calculation. Although present in the Matlab model, no efficiencies are currently applied to the engine or the mechanical power train.

```
val loadFactor = signals.signalValue("alpha*", 0.0).round.min(100.0 toLong).max(0.0 toLong)
val wheelAngularVelocity: Double =
  // make sure the vehicle won't roll backwards; even if it is, the engine will only move it forward
  Math.max(0.0, signals.signalValue("v", 0.0)) /
    (2.0 * Math.PI * vehicleParameters.wheelRadius / 100.0)

// the engine's rotational speed in [1/s]; if the engine reaches the velocity limit, rotation is set
// to infinity to make the torque very small so the wheel/engine velocity does not exceed the limit
val rotation = wheelAngularVelocity match {
  case x if x == 0 => 0.00001
  case x if x * vehicleParameters.transmissionRatio > (vehicleParameters.maximumEngineRpm / 60) =>
    Double.PositiveInfinity
  case x => wheelAngularVelocity * vehicleParameters.transmissionRatio
}
val M = Math.min(
  vehicleParameters.maximumEngineTorque,
  loadFactor / 100.0 * vehicleParameters.maximumEnginePower / (2.0 * Math.PI * rotation)
)
```

Listing 5.3: Engine torque calculation

The (delayed and amplified) gas pedal input α^* is used as a linear *load factor*, which directly influences the torque released by the engine. The torque calculation source code is printed in listing 5.3. The other required parameter is the current engine rotation speed. The engine

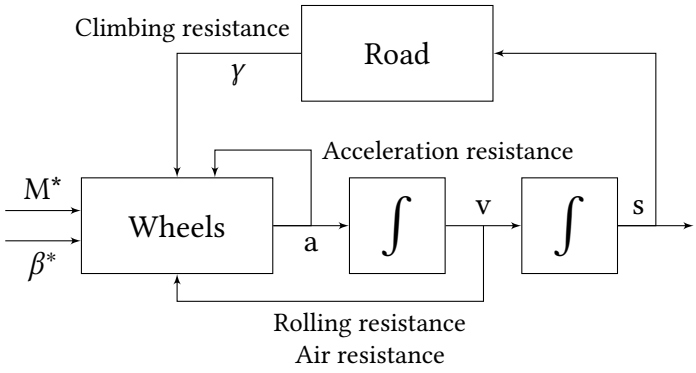


Figure 5.11: The driving resistances

$$F_{\text{res}} = F_{\text{eng}} - \left(F_{\text{brake}} + F_{\text{acc}} + F_{\text{climb}} + F_{\text{roll}} + F_{\text{air}}\right) \tag{5.5}$$

$$= \frac{1}{2} c_{\text{W}} \cdot A \cdot \rho \cdot v^2$$

$$= f_{\text{r}} \cdot m \cdot g \cdot \cos\gamma$$

$$= m \cdot g \cdot \sin\gamma$$

$$= m(1+\varepsilon) \cdot a$$

$$= \frac{i}{r_{\text{w}}} \cdot M_{\text{eng}}$$

Figure 5.12: The driving equation as implemented in RoadHopper's vehicle model

and wheels are coupled back-to-back by the transmission. Therefore, the wheel rotation speed directly translates to the engine (multiplied by the translation factor); small loss effects due to mechanical distortion by the applied momentum are assumed to be negligible. If a power train efficiency is introduced, these mechanical losses should be incorporated.

Driving resistances

Like the braking force, driving resistances contribute a decelerating moment to the driving equation. In the current model, all these moments are applied directly to the wheels; see figure 5.11 for a block diagram of the parts involved.

Figure 5.12 shows the resulting forces applied at the wheel [Bra13]. As can be seen, most components of the driving equation as shown in section 2.5 were implemented. The only thing left out is the mass factor for the acceleration resistance.

Completely missing in both equations is the wheel slip, i.e. the losses occuring at the wheel–road contact. These should also be integrated in a higher-fidelity model.

Additionally, the wheel friction coefficient is constant (effectively assuming an ever constant road surface), despite different road construction materials and weather conditions. Some other factors also play a role for the rolling friction, but these are negligible [Mit14, ch. 2.1] and can be left out.

The model also assumes the air density ρ to have a constant value of $1.2\,\mathrm{kg\,m^{-3}}$, which is approximately correct for an environment of $T = 20\,^{\circ}\mathrm{C}$ and $p = 1013.25\,\mathrm{mbar}$. However, as these conditions change, the air density also quickly changes ($\rho \approx 1.14\,\mathrm{kg\,m^{-3}}$ for $T = 35\,^{\circ}\mathrm{C}$ and $\approx 1.34\,\mathrm{kg\,m^{-3}}$ for $T = -10\,^{\circ}\mathrm{C}$). As the air resistance is proportional to v^2 and thus one of the bigger factors in the equation for higher speeds, a change in the air density also affects the total driving resistance relatively strong.

Accelerating force The accelerating force F_{acc} applied to the vehicle can directly be derived from the equation in figure 5.12: The resulting force F_{res} in the equation must be 0 according to the Third Newtonian Law of Motion. Therefore, the accelerating force is

$$F_{\mathrm{acc}} = F_{\mathrm{eng}} - F_{\mathrm{resistance}} = m \cdot a \tag{5.6}$$

$$= F_{\mathrm{eng}} - (F_{\mathrm{brake}} + F_{\mathrm{climb}} + F_{\mathrm{roll}} + F_{\mathrm{air}}) \tag{5.7}$$

From this, we can directly derive the current vehicle acceleration a:

$$a = \frac{1}{m}\left(F_{\mathrm{eng}} - F_{\mathrm{resistance}}\right) \tag{5.8}$$

5.5 Integration into GraphHopper

RoadHopper relies on GraphHopper for all map-related tasks. Therefore, a tight integration between the two software products is required. GraphHopper itself offers different methods of integration: An interface based on the HTTP protocol with an accompanying web interface, and direct access to the internal data structures and services via the `com.graphhopper` Java package.

Extending the web interface is easily possible by plugging in additional end points that provide their services under a distinctive URI. These end points are implemented as Java Servlets. New servlets were created for these purposes:

- retrieve more extensive information on the road
- controlling the simulation (start, get status, get results)
- retrieve measurement data (see section 6.1.2)

For all these end points, accompanying frontend functionality was integrated into the existing JavaScript code of GraphHopper's web UI.

Additionally, the server integration class `GraphHopperServer` was replaced with a custom class to integrate the new servlets.

5.6 Running a simulation

A simulation in RoadHopper can be run via one of two default ways:

1. using the RoadHopper web UI
2. with a call to the API endpoint `/roadhopper/simulate`

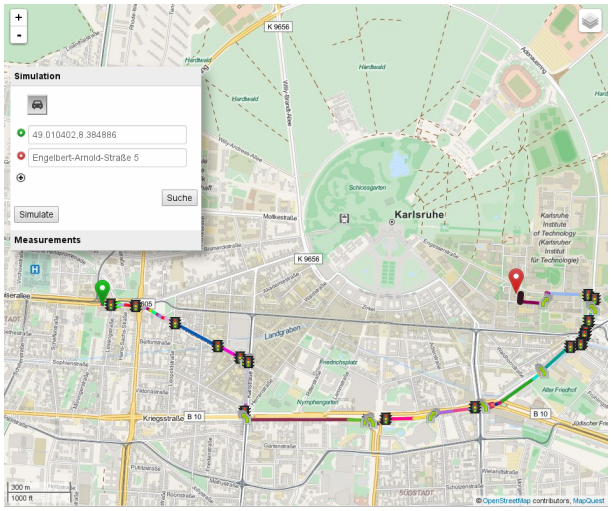
This section only describes the web UI, which internally uses the API endpoint.

After startup, RoadHopper exposes its user interface via HTTP. The default port used is 8989, so the address is `http://localhost:8989/` if RoadHopper runs on the same machine as the browser. As the port is not the standard HTTP port 80, it must explicitly be specified when entering the URL.

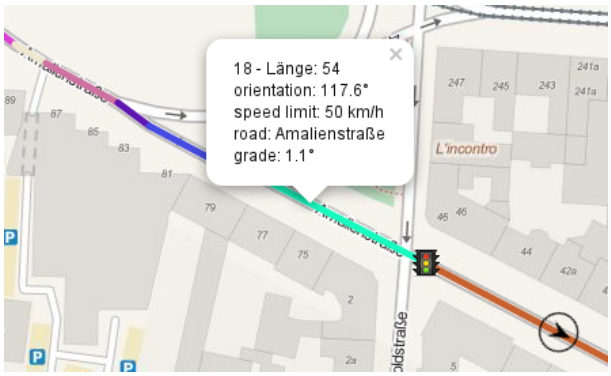
The basic user interaction concept of RoadHopper has been inspired by GraphHopper (and almost every other popular mapping service): After selecting two points on the map (right click → "Set as start/intermediate/end"), the route is automatically calculated and highlighted on the map, see figure 5.13a. Instead of selecting the points on the map, they can also be entered as either an address or coordinates. The addresses are internally resolved using a service called *OpenStreetMap Nominatim*[19].

The route visualization in GraphHopper displays the whole route as one long segment. To support debugging, this was changed for RoadHopper: now each single road segment is

[19] See `https://wiki.openstreetmap.org/wiki/Nominatim` for more information; visited 2015/09/18



(a) With a route selected



(b) Road segment detail information

Figure 5.13: RoadHopper user interface

drawn with a random color to get a contrast between them. The visualization in both cases is implemented using Leaflet[20].

When clicking on a segment, further information about it is displayed in a popup box, again to support debugging (see figure 5.13b).

The simulation is started with a click on the “Simulate” button beneath the input fields. A command is issued to the server, to which it responds with a confirmation and the simulation’s identifier (figure 5.14). This identifier is used for tracking the simulation status.

By using the API endpoints directly, multiple simulations can be run without manual user interaction. This was used to gather the results discussed in chapter 6.

5.6.1 Simulation data display in the browser

Simulation data can currently only be displayed in retrospective, after the simulation run was completed.

Live status

As a status indicator during the simulation run, the current simulation time is shown next to the “Simulate” button.

[20] A JavaScript library for interactive map display, see `https://www.leafletjs.com`; visited 2015/09/18

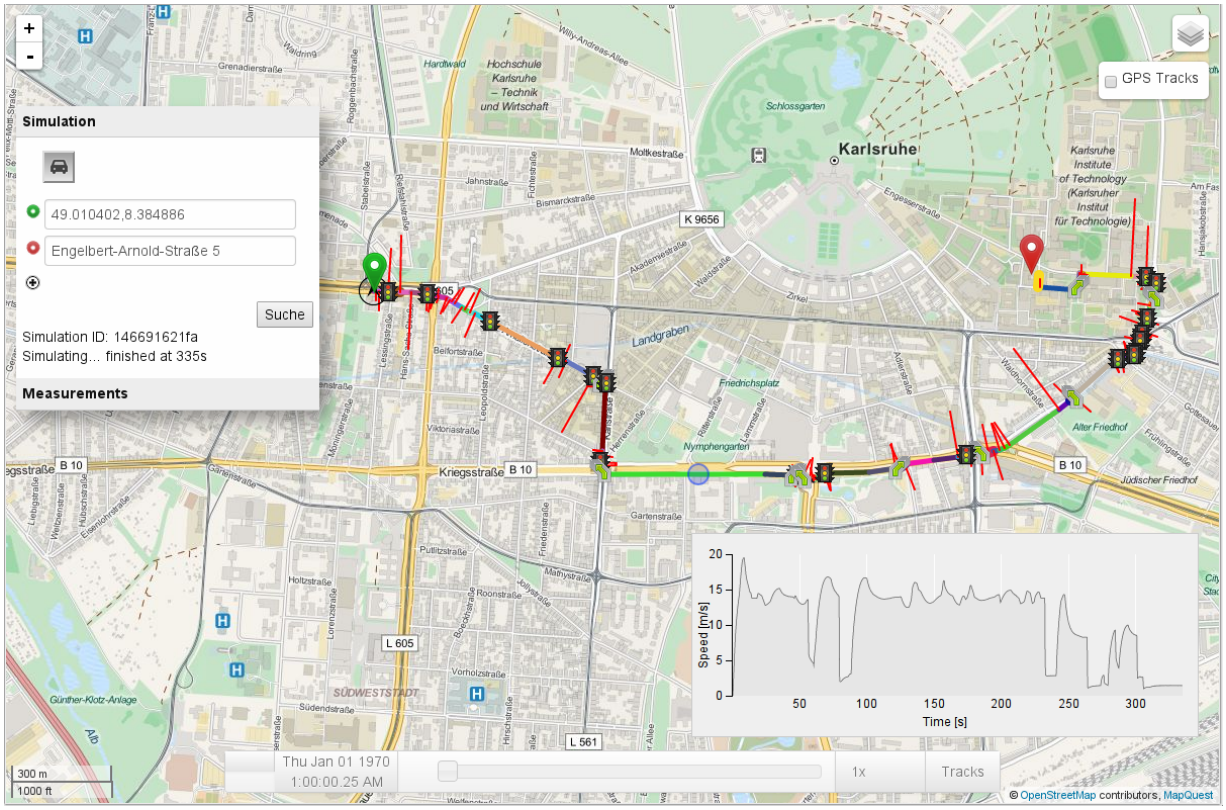


Figure 5.14: Simulation results displayed in the map

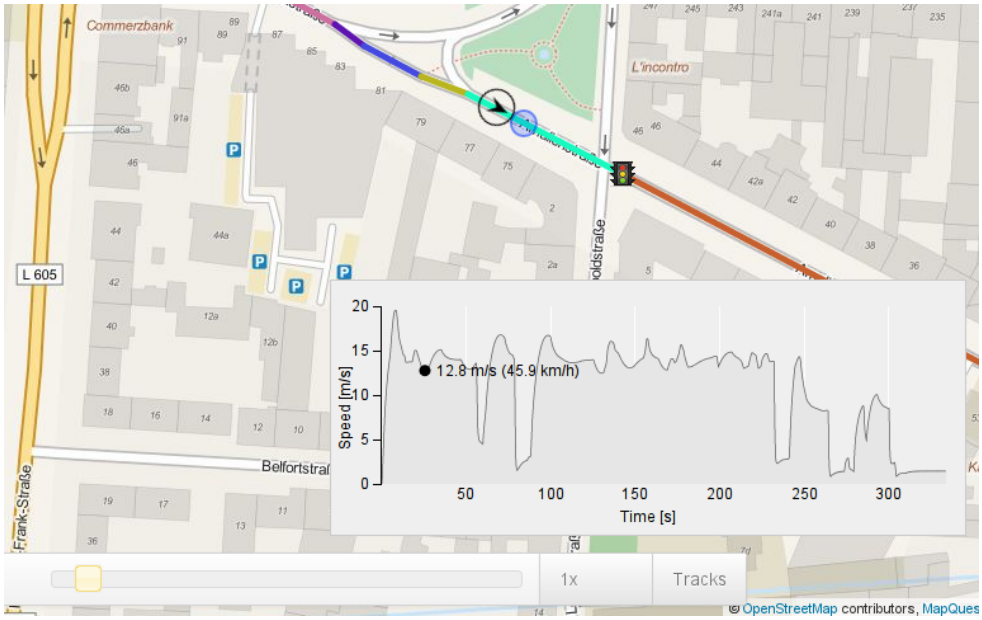
An extended live progress view could e.g. include the current vehicle position, current speed and the remaining distance to travel. Additionally, if the driver is extended to support different action modes, this could also be added to the simulation progress view, at least in a special debugging mode.

Playback

After the simulation has finished, all data that is necessary for a simulation playback in the browser is transferred to the client. The data can then be played back by clicking the button next to the time slider. The slider can be used to navigate to an arbitrary position in the tracked time. The playback is based on an Leaflet.Playback[21], an existing plugin for Leaflet, the mapping library used for displaying the map.

The simulation playback will show a marker at the current position, pointing into the direction the vehicle is heading (see figure 5.15). Also visible on screen (but outside the screenshot area)

[21] available at https://github.com/hallahan/LeafletPlayback; visited 2015/09/29



The blue circle shows the position currently selected in the graph

Figure 5.15: Simulation playback with position marker

is a box with the vehicle's current velocity. This could also hold more telemetric data like the current vehicle grade.

5.6.2 Data storage

Simulations can possibly create huge amounts of data. RoadHopper makes no exception here, though currently to a smaller degree due to its rather simple simulation models. Also much data is discarded directly after the simulation has run. For the tests performed for this thesis, a sample of all signal values was taken every 50 milliseconds ($f = 20\,\text{Hz}$). This simulation data is stored in memory at runtime. Also all road data is stored in memory.

To keep data between server runs for further analysis, it should be stored in a persistent data storage, e.g. a database. The most widely used type of databases are relational databases, which store data in a row-based format using a fixed schema (very much like normal tables). The cells of such rows can store either simple values (like strings and numbers) or a reference to another record.

Relational databases have been around for multiple decades and were the go-to data storage for a long time. They are very convenient for flat data structures, but are not very well suited for complex object structures (the so-called object–relational impedance mismatch [Ire+09]).

```
{
  result: {
    250: {
      direction: 1.5292135708293981
      position: {
        elevation: 122,
        lon: 8.38488440398377,
        lat: 49.010427161155334
      },
      speed: 0
    },
    ...
  },
  simulation: "146fef12bd0",
  status: "finished",
  time: 334000
}
```

Listing 5.4: Simulation status in JSON

```
time,v,s,M
50,0.0,0.0,0.0
100,0.0,0.0,0.0
150,0.0,0.0,0.0
200,0.0,0.0,0.0
250,0.0,0.0,0.0
300,0.0,0.0,0.0
350,0.0,0.0,0.0
400,0.0,0.0,0.0
450,0.0,0.0,0.0
500,0.0,0.0,0.0
550,0.0,0.0,0.0
600,0.0424394011339587,3.373559376469073E-4,200.0
650,0.2015608261734605,0.005341521861560992,200.0
700,0.40646010840509667,0.0194018026941671,200.0
750,0.629322632419306,0.04413605695006531,200.0
...
```

Listing 5.5: CSV simulation data export

As the signal values in RoadHopper can also be complex values like a coordinate or a complete road segment, the relational storage model is not a good choice. To put a nested object structure into the database, the data would need to be either serialized to a format like JSON, defeating the purpose of the relational model (easy searchability), or stored in various tables and linked via relations. Instead, OrientDB—a database that directly supports storing complex documents—was chosen for a proof-of-concept implementation.

For validating the simulation results of RoadHopper, a comparison with measurements performed in [Rap13] was done. This data was converted into a format similar to the signal values format and is also stored in the database. The exact process and the results are discussed in section 6.1.2.

5.7 Simulation data export

Simulation data can be represented in various ways for different purposes:

1. For simple usages like the browser playback mentioned above, only a small—externally observable—subset of telemetric data is necesssary, with a low time resolution (one to five data points per second; the default of the used visualization library is four).

2. To get an insight into the simulator's behaviour, in contrast, internal state of the simulation engine should be viewable, with a rather high time resolution. This export should be flexible to allow different usecases depending on the analysis goal.

To account for both these demands, two data exports are integrated into RoadHopper. The first, simple use case is fulfilled by an export to JavaScript Object Notation (JSON) that is delivered to the browser after the simulation has finished. This import includes the following data for each time step:

- timestamp (in ms)
- position (lat/lon; in °) and elevation (in m)
- speed (in $\mathrm{m\,s^{-1}}$)
- direction of travel (in rad)

An example of the JSON data is shown in listing 5.4. The default time resolution is one step every 250 ms. Judging from the experiments performed during development, this is sufficient for a smooth visualization.

The second, more sophisticated export is based on comma-separated values (CSV). Listing 5.5 lists an example of the resulting data. The export can include values of arbitrary signals, depending on the configuration. Exports can be performed during and after a simulation, but there is no indication during a simulation that it is still running (the status can be obtained by querying the SimulationStatus servlet).

The signals to export are handed to the servlet via the (multi-valued) `signal` parameter. An example URL to export the velocity and acceleration would look like this: `http://localhost:8989/roadhopper/signalvalues?simulation=<simulationid>&signal=v&signal=a`. Further signals can be added by appending `&signal=<name>` to the URL. The simulation ID can be obtained from the response sent when starting a simulation.

6 Results

To prove that the assumptions made during the design of a model hold, it must be tested and validated afterwards. Therefore, tests were performed on the implementations of simulator and simulation models. Section 6.1 describes the test methodologies, the test data used and the results that were observed. From the interpretation of these results in section 6.1.1, recommendations for improving the simulator models are derived in section 6.3.

In addition to synthetic tests with custom scenarios, measurement data from [Rap13] was used for comparing the models to real-world behaviour. This is discussed in detail in section 6.1.2.

6.1 Simulation behaviour tests

For simulated driving cycles, a lot of possible scenarios could be tested. In fact, the space of feasible inputs (waypoints to travel between) is nearly endless, given the total length of worldwide road networks of about 35 million kilometres [WFB, Country comparison: roadways].

Two types of tests can be performed while developing a simulator like the one discussed here. The one type are short ad-hoc tests that test one small aspect on a specific data set, to validate a new feature or bugfix. To test the simulation behaviour during development, lots of such small-scale experiments were conducted, using a variety of different road situations. Examples are testing the slowdown behaviour around a turn or the acceleration after a traffic calmed area. They can easily be repeated using any fitting road segment, therefore no scenarios were standardized here. Such scenarios could however be valuable for increasing the coverage of RoadHopper with (automated) tests, see next chapter.

The other type are longer tests with predefined scenarios that validate the whole model or a significant part of it. The scenarios that were developed are detailed further below in this section, the results are discussed in the following sections.

Several driving cycles, e.g. the CADC, divide driving situations by scenarios, depending on the (primarily) used road category. CADC uses the categories urban, rural and motorway; see e.g. [Bar+09] for details on the parameters. [Lia06] uses the additional categories “stop and go”

scenario	from	to	length
urban1	49.010 557°N/8.412 732°E	49.006 257°N/8.371 421°E	4.4 km
urban2	48.998 409°N/8.390 551°E	49.046 476°N/8.377 419°E	6.5 km
rural1	49.010 557°N/8.412 732°E	49.036 208°N/8.707 223°E	26.9 km
rural2	47.997 097°N/7.847 558°E	47.920 599°N/7.787 933°E	11.5 km
highway1	49.010 557°N/8.412 732°E	48.893 425°N/8.702 835°E	33 km
highway2	48.783 088°N/9.181 459°E	48.594 989°N/8.869 599°E	36.1 km

Table 6.1: Test scenario routes

and "suburban"; the boundaries between all five categories are defined with fuzzy logic rules there. For these rules, expected travel distances and velocity distribution plays a role.

The categorization by road and (expected) velocity distribution also seems a good fit for testing a model, as each of the categories highlights different aspects. For the tests conducted for this thesis, three types of scenarios were used:

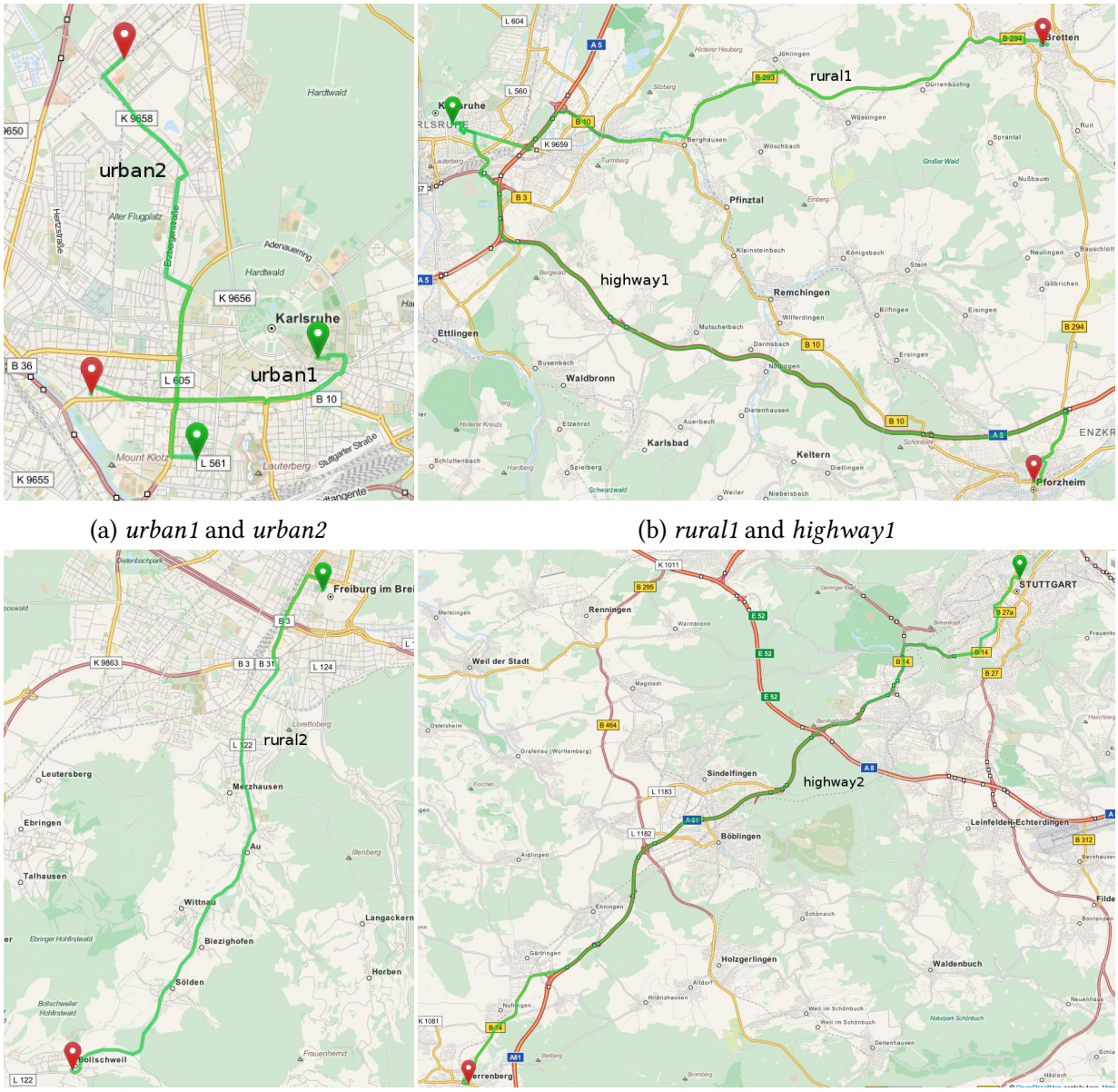
- In *urban* scenarios, the observed velocities are relatively low, while vehicles often accelerate and slow down. Traffic will strengthen this effect. Additionally, road bends and turns are observed relatively often.
- *Rural* scenarios have higher velocities and less traffic interruptions by e.g. road bends and turns than urban roads.
- For *highway* scenarios, the highest speeds can be observed as there are no disturbances by interfering traffic from other directions, leading to relatively uniform velocity distributions for low traffic conditions on the road. Also the roads are usually shaped to allow high travel velocities, e.g. with high bend radii.

The three categories were chosen because they can be distinguished relatively easily. Stop and go traffic is not possible with the current, traffic-less simulation model, and suburban and rural areas are likely harder to distinguish in Germany than the USA, where the test data for [Lia06] was gathered.

To get a common ground for testing the simulation, two scenarios for each of the three categories were defined, which are listed in table 6.1. For maps of all routes see figure 6.1. The scenarios were run in a batch, to quickly get reproducible results.

The basis for all tests was the OpenStreetMap dump `baden-wuerttemberg-latest.osm.pbf` downloaded from `http://download.geofabrik.de/` on 2015/09/06.

The tests were conducted using the Opel Ampera model as specified in table 5.3.



(a) *urban1* and *urban2* (b) *rural1* and *highway1*

(c) *rural2* (d) *highway2*

Figure 6.1: Test scenario routes: map view

6.1.1 Test results

The results from the simulation are depicted in figure 6.2. The graphs show a comparison of the actual velocity v and the target velocity v_{target} as applied to the velocity controller.

Some observations from the graphs should be discussed here:

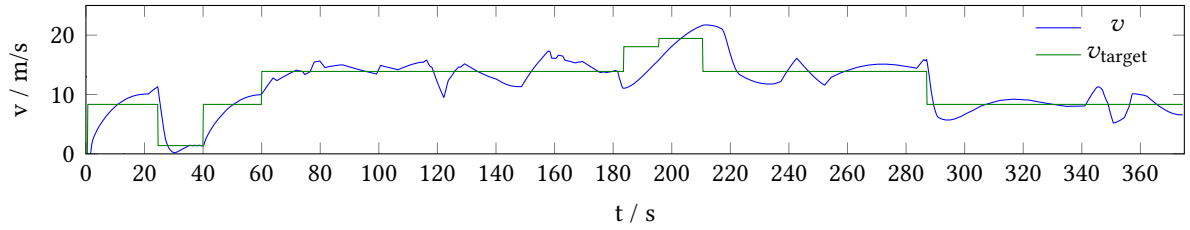
- The velocity control overshoots in many situations. This is especially visible in figure 6.2f at around 420 s, while figures 6.2a and 6.2c show much smaller, but longer lasting overshoots at around 220 s and 750 s, respectively.
- The different plots all show some oscillating movement around the target velocity even for stable conditions. Noticeable are the huge differences in figure 6.2e at around 750 s, which can likely be accounted to a comparingly steep grade in the road.
- One striking example of odd driver behaviour is the *increase* in velocity in figure 6.2b at 200 s, while v_{target} *decreases* only a few seconds later. The reason for this errant behaviour is yet unknown; further analysis of the exact driver behaviour and road conditions at this very location would need to be performed.
- A similar error appears for the brake to halt in figure 6.2a at 25 s. The driver should normally not get the vehicle to halt where only a decrease in velocity (in this case due to a traffic calmed area on the KIT campus) is required[22]. The same behaviour could be observed for turns, where the velocity also needs to be reduced to a comparatively low level (7 km h^{-1} to 15 km h^{-1}). The behaviour is similar to the observed overshooting.

6.1.2 Conformance with real-world measurements

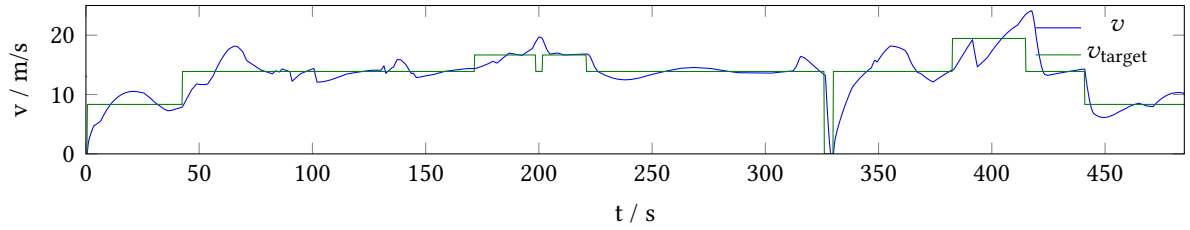
With the first simulation runs described above, the simulator was proven to work in general. To validate the model itself, the simulation results were additionally compared to real measurements.

Comparing measurements performed under actual traffic conditions is a difficult topic. The data is often distorted by long periods of standing, especially in cities during rush hour. A general impact on driver steering behaviour will also be noticeable, depending on the traffic density. A comparison of measurement data thus should only take times into account when the vehicle was actually moving. Such small excerpts from a large data set are called *driving pulses* in [Lia06].

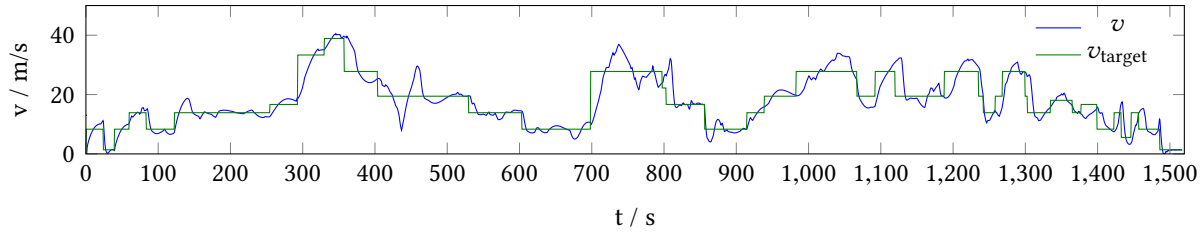
[22] In another test on the same route with the *small car* model described in table 5.3, the effect was even larger—the vehicle stopped for a few seconds at this position before continuing the journey.



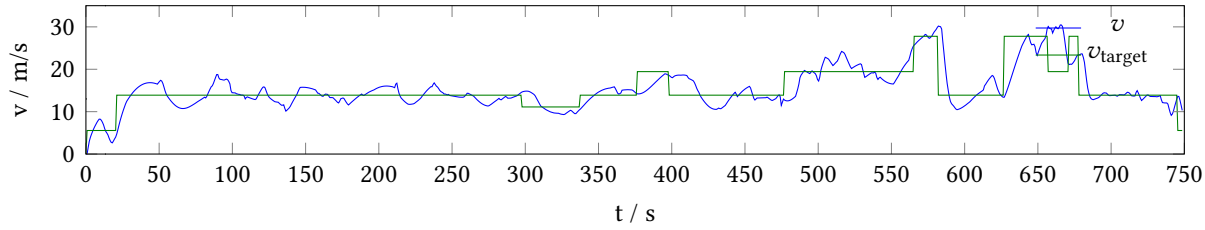
(a) Scenario *urban1*



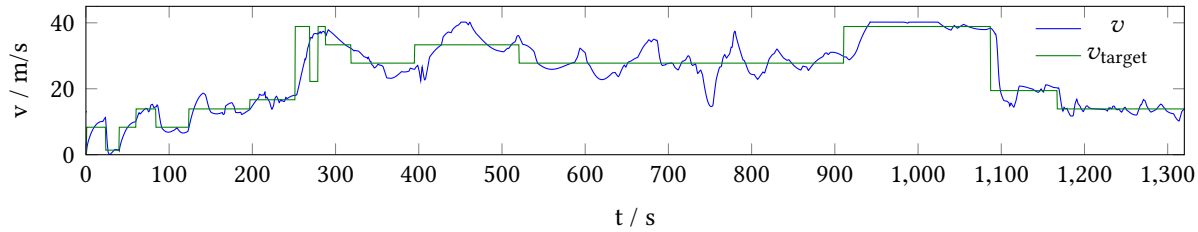
(b) Scenario *urban2*



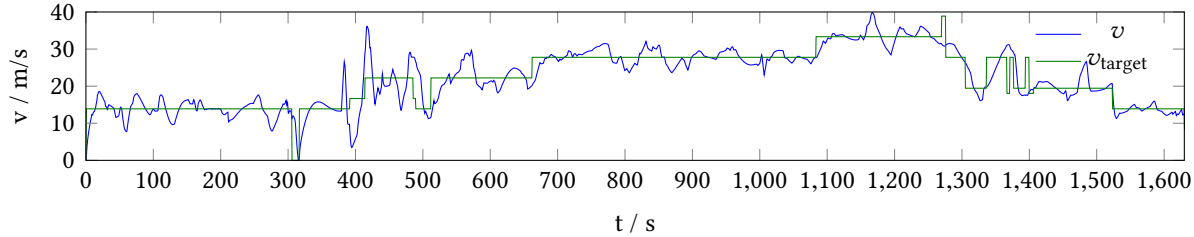
(c) Scenario *rural1*



(d) Scenario *rural2*



(e) Scenario *highway1*



(f) Scenario *highway2*

Figure 6.2: Simulation results before optimization

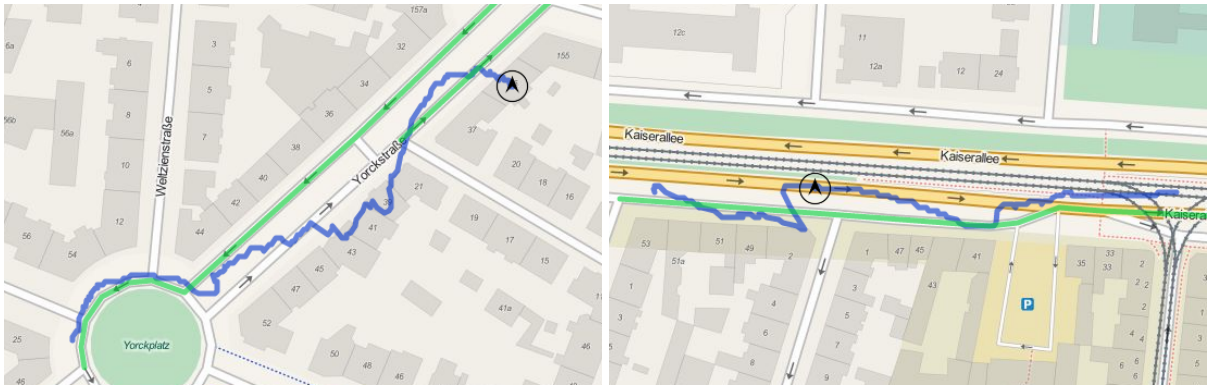


Figure 6.3: Noise in the GPS measurements, examples taken from measurement *Wednesday 2*

cat.	day of week	pulse	from	to
urban	Wednesday	2/72	49.008 945°N/8.387 321°E	49.005 926°N/8.386 946°E
urban	Wednesday	2/74	49.005 866°N/8.386 908°E	49.000 184°N/8.385 916°E
urban	Wednesday	2/91	48.994 580°N/8.393 265°E	49.002 164°N/8.394 418°E
h'way	Friday	4/86	48.971 654°N/8.454 237°E	49.015 694°N/8.466 511°E

Table 6.2: Measurement scenario data

To prepare the comparison, the data set from [Rap13] was separated into driving pulses using the algorithm listed in appendix A. The velocity present in the data was calculated from the position change in the GPS data [Rap13] and not fetched from the vehicle electronics.

Upon examination, the measurement data was found to contain a lot of noise, especially in densely populated areas (city centres) where the GPS signal might be weakened by buildings; see figure 6.3 for two examples. The single velocities might therefore be inaccurate, but all in all, the data should still be reasonably good, as long as the noise in the position data is not too high (e.g. positioning the vehicle within a block of buildings). Over a full driving pulse, the medium velocity should still fit, as the start and end positions and the distance between them should be pretty accurate.

Additionally, the measured GPS positions were matched to their nearest road segment using an algorithm published by the GraphHopper developers[23]. This algorithm returns the edges found next to the points. The start/end markers are therefore sometimes offset by a few dozen meters and need to be manually adjusted; the positions listed in table 6.2 are already changed to conform to the measured positions.

The simulation was run for every data set listed in table 6.2, followed by an examination of the results. Based on the findings, model parameters were adjusted to change the behaviour to

[23] Available at `https://github.com/karussell/map-matching/`.

match the measurements more closely. Three iterations were performed with the following model properties:

A) the unchanged model as presented in chapter 5

B) v_{target} set to 60 % of its original value

C) B plus the measured current velocity v offset by a random value $v_{\text{off}} \in \{-4.0 \ldots 3.0\}$ (to emulate errors in reading the speed indicator)

All adjustments in the model are related to the PID controller input value v_{diff}, to keep a close link between changes performed and the new simulator behaviour. The PID controller input is adjusted every 500 ms, to try to emulate the frequency of human steering [MH93].

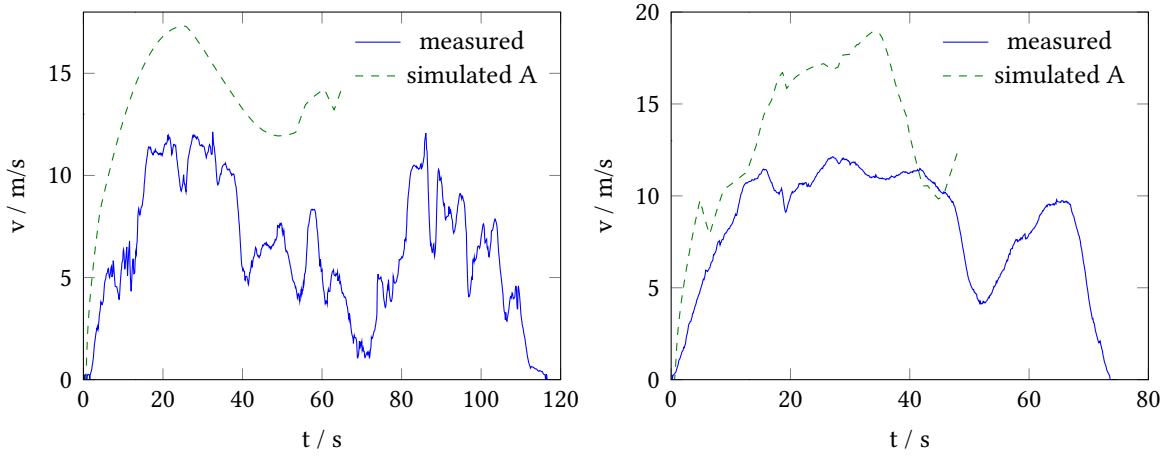
Similar experiments could be conducted with the vehicle parameters, although the PID controller might show an unexpected reaction to such changes to the vehicle.

In contrast to the driving pulses in the measurements, the vehicle does not slow down to a halt at the end of the road. Instead, the driver assumes to continue driving, but the simulation ends because the simulation road was finished.

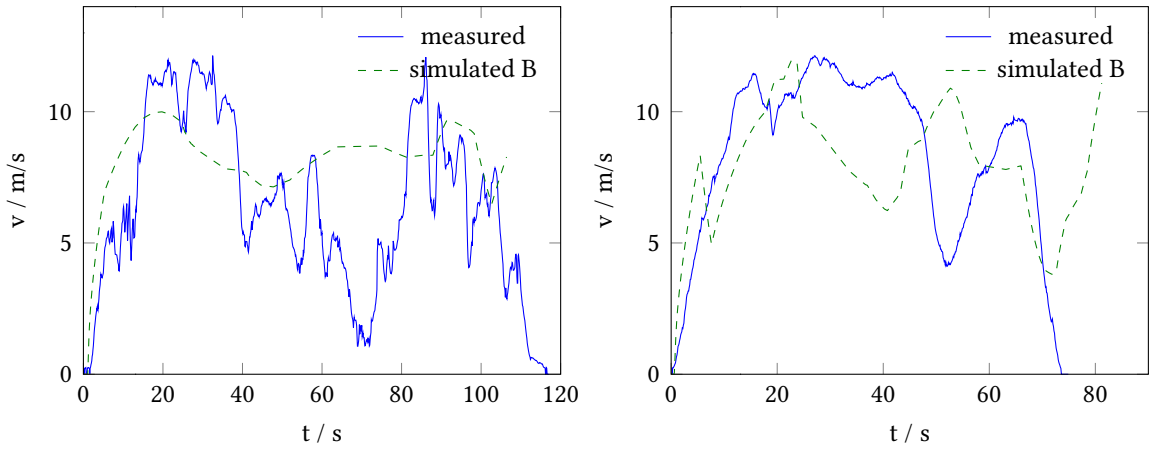
Despite several attempts to change this behaviour, no satisfactory parameter set for the driver model could be found that would work under all conditions. The general idea that was tried is to insert a short (1 m to 5 m) segment with a speed limit of $0.01\,\mathrm{km\,h^{-1}}$ at the end of the road.[24] This results in the vehicle slowing down, but either it stopped long (10 m to 30 m) before the end or it did not slow down fast enough to come to a halt at the desired position. The likely cause of this was identified to be the lookahead distance, which was either too high (leading to a too early slow down) or too low (= braking too late, thus not stopping in time; the vehicle then still had a leftover velocity when the simulation ended).

A similar behaviour was observed for stop signs, which the vehicle missed under certain conditions. As the driver switches to a different operating mode when encountering a stop sign, it will always come to a halt, but it might already be past the stop sign at this position. Using parameters that allowed stopping in time for one sign, other signs were still missed, so more research in this area (and a standardization of the tests so they can be conducted repeatedly and reliably) is clearly necessary.

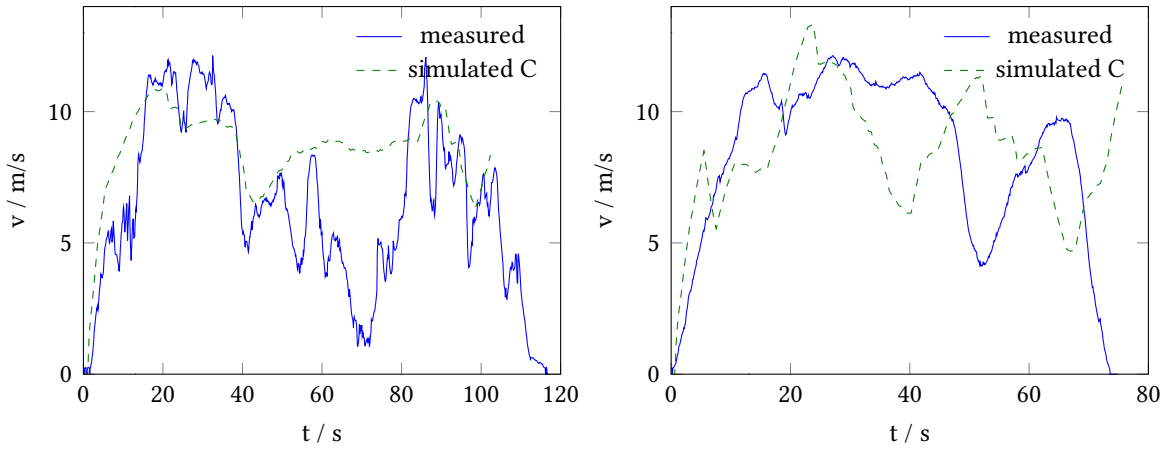
[24] A limit of $0\,\mathrm{km\,h^{-1}}$ will be filtered by the velocity controller.



(a) Simulation A



(b) Simulation B



(c) Simulation C

Figure 6.4: Measurement *Wednesday 2/91* (left) and *Wednesday 2/74* (right) compared to different simulation parameter sets

6.2 Interpretation of results

Judging from the plots in figure 6.2, the target velocity is already kept quite well. The performance concerning the limit velocity (*are the limits always kept?*) is not included in these graphs and a detailed discussion should be omitted here for the sake of brevity. In case the target velocity is not kept, this might be a result of a too short look ahead distance, thus the driver would be informed too late about the new velocity. In this case, tuning $s_{\text{lookahead}}$ (see eq. (5.3)) might help.

b, the comfortable braking deceleration, influences the lookahead distance from which the driver gathers the target velocity. It was therefore identified as the variable of choice to change for tests. Several values were tested ($1\,\mathrm{m\,s^{-2}}$, $2\,\mathrm{m\,s^{-2}}$ and $4\,\mathrm{m\,s^{-2}}$), but with each, the driver only reacted correctly for some simulation scenarios, but failed in others.

Striking are some strong overreactions to changes in the target velocity, the source for which has yet to be found. A number of possible reasons come to mind:

1. Real-world measurement data was used to calculate the coefficients of the driver's PID controller. As these were gathered from an analysis of a single driver with an unknown number and type of vehicles [LM09], the whole model is probably not applicable here.

2. v_{diff}, the difference of target and actual velocity, is the only stimulus for the driver. Thus, it cannot react to bent segments or changes e.g. in the road grade that would be obvious to a human driver. Some effort was made to mitigate these effects for road bends by limiting the target velocity. Graded roads are however not respected at all by the current driver.

3. The driver model was calculated from real data, which is likely not directly applicable to an all-electric power train with e.g. its better torque characteristic at low speeds. This would explain the overshooting in initial acceleration compared to measured acceleration data. Additionally, the brake and gas pedals are modelled as linear components, which does not eaxctly reflect their real-word equivalent [Mit14, fig. 9.34].

4. Various model parameters, like the maximum braking force or the gains of the PT_1 controllers in the power train, were estimated based on literature data. This data might very well be wrong for the particular vehicle model, therefore leading to erroneous simulation behaviour.

5. No traffic is simulated, which would be required for a realistic simulation behaviour, especially in cities. As an approximation, the global reduction of the target velocity was

introduced. The comparison of the measurement's overall velocity distribution with the one from the simulation in figure 6.4c leads to the conclusion that the global reduction of the target velocity is already pretty similar in both cases.

As can be clearly seen in the plots in figure 6.4a, the unchanged simulator achieves much greater velocities than observed in the measurements. This is most likely a result of the lack of traffic in the simulation, which causes the slow down in the real world. Also the speed limit is not always kept, which might be related to difficulties of the driver model with the vehicle characteristics (higher acceleration/quicker response to changes than expected).

Comparing the plots to some driving cycles derived from real world measurement data already shows a pretty good result for the adjusted models B and C. The time needed to travel the road is similar to the measured values when taking into account the different behaviour towards the end of the road. Further analyses on a larger base of measurement roads would be necessary to really judge the model in detail.

6.3 Improving the simulation models

The goal of a driving cycle simulation is to gather data that matches actual measurements as exactly as possible. As shown above, this could partly be achieved, though more intensive testing would be required to actually validate the models.

To get meaningful data, the vehicle model should accurately represent its real-world pendant. Some effort in this direction has been made and the first results look promising. To create a really accurate model, however, more data will be necessary. This includes a detailed performance characteristic of the engine, power management including a battery (in case of an electric/hybrid vehicle) and a transmission model, all these with proper loss characteristics.

The vehicle largely influences driver behaviour (how much force is applied to the pedals etc.), so optimizing the driver is of little use as long as the vehicle model is not validated at least to a certain degree or a driver model is used that can adjust itself to unknown vehicle characteristics. Therefore, further work should put emphasis on improving and validating the vehicle model before adjusting the driver.

The existing PID driver model should be validated against other sources and probably be compared to other driver models that use different control approaches. When sticking with the existing model, the methods used in [LM09] could be used to gather new controller coefficients from measurement data, as those listed in table 5.2 seem to not fit the current model. Also the

input should be extended to include more data than just the difference of actual and target velocity.

As stated earlier, for the adaptive lookahead distance of the driver no satisfactory value could be found. The varied parameter was the comfortable braking deceleration b, which is defined in [LM09] with a value of $8\,\mathrm{m\,s^{-2}}$. Judging from the results when changing the value of b, the lookahead distance must probably be calculated with a totally different formula, e.g. not depending on v^2 but v or $\log(v^2)$, which would lead to a more shallow increase of $s_{\mathrm{lookahead}}$ over v.

7 Summary and outlook

This chapter summarises the preceding chapters with a special focus on RoadHopper's implementation status and gives an outlook of the software's future.

The summary discusses the driving cycle process and explains how far it was implemented, concluding with a look on the implementation status of the different models required for the driving cycle simulation.

The outlook is divided into section 7.2, with a list of short-time implementable features for improving the simulator, and section 7.3, where a more general vision of the topic and the role RoadHopper could play is shaped.

7.1 Summary

The goal of this thesis was to get a process and the necessary software to generate driving cycles based on a route taken from a map. The driving cycles should be created in a format suitable for processing with third-party tools.

The foundation of this thesis is the three-step driving cycle generation process as described in chapter 3. The three steps were successfully implemented. A special emphasis was put on the second step, the driving simulation, as it is the most complex part of the process.

The first step of the driving cycle process is built on the already existing, open sourced routing solution *GraphHopper*. The software was extended to add to the routing graph information which is necessary for performing a simulation in the second step. Additionally, the existing user interface of GraphHopper was extended to support displaying additional information about the road and the simulation status.

To execute the second and the third step of the process, a new software called *RoadHopper* was created. It is largely independent of GraphHopper, while still using some of the interfaces offered by it, mainly to perform routing and extract information from the road network graph. From this information, a digital road model is derived, on which a simulation is performed.

The simulation is run by a time-discrete simulation engine. The simulator models are implemented as independent blocks, also called processes, which are connected by signals. The signals are wired to a shared signal bus, which also takes care of invoking a process if any of the signals it listens to changes.

To avoid problems with the concurrent execution of blocks and signal updates, they use a concept called delta cycles. Before a delta cycle, signal updates that were previously held back are executed. Using their sensitivity lists, processes are then selected for execution if they listen to any of the updated signals.

The simulation model status is discussed in its own section below.

The third step consists of calculating a driving cycle, i.e. a t–v diagram, from the data gathered in the previous step. The diagram itself is only generated in a very rough preview version in the frontend, which is not suitable for a detailed analysis. Instead, the raw values can be fetched from the simulation engine. They can then be further processed using software like MATLAB or Python's Matplotlib.

7.1.1 Model implementation status

The simulation models were designed based on different sources found in literature, among them [LM09; SHB10].

Some parameters had to be guessed, as the existing literature just gave schematic overviews of the parts without detailed information on the actual implementation. The most prominent examples for this are the various PT_1 blocks in the power train, where the coefficients were approximated based on the observed behaviour of the driver PID controller and educated guesses on the delays occuring in the power train (based on other literature on power train modelling).

Clearly, due to these inaccuracies, the current state of the models is unsatisfactory to get driving cycles that closely match real-world car behaviour. As the vehicle and driver are strongly coupled, they should be viewed as one system, as proposed in [Mac03] already:

> To sum up, we see that the combination of human parameters and of mechanical parameters enter into the process of driving in a manner which does not permit their clear-cut separation. The car and the driver form, in a sense, an individuum.

This becomes especially true when modelling the interactions on a detailed level, like the neuromuscular systems steering the hands or feet. Adding these systems will bring inner feedback loops into the driver–vehicle system depicted in figure 7.1. Examples for this, though more related to lateral steering, are given in [Don78].

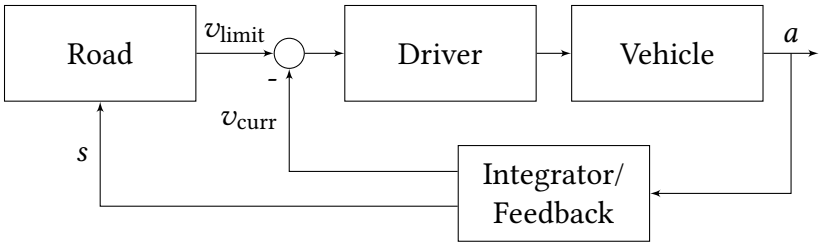


Figure 7.1: Overview of the simulation models

As one intent of this thesis was to get a possibility for driver parametrization, a new integrated driver–vehicle model should still try to isolate certain driver properties that can be modified. Examples for these properties could be the aggressiveness (eagerness to accelerate) and the obedience to speed limits. This would allow modelling different usage patterns of a vehicle, which might result in greatly varying driving cycles.

7.2 Desirable features for RoadHopper

The current simulator, though fully functional, still lacks a few features that would make experiments easier and faster to conduct. Also the simulation models could be extended in some aspects to deliver more accurate results.

Some of those features which are easy to reach in a short term are described here. A broader vision of what RoadHopper could become in the future is detailed in section 7.3.

7.2.1 Simulator engine

To conduct tests more reliably and reproducible, the internal state of the simulator at the start of the simulation should be stored. Such a state vector could then be passed in again to restore the exact same state. This would make A/B analyses of model variations a lot easier.

In addition to that, the models in general should support more parameters that are not hardcoded in their implementation, but passed in from the outside. This way, simulations could be run with different model parameters without the need to recompile and restart RoadHopper.

7.2.2 Signal processing

The registration of signals should be simplified—currently it is rather complicated: A component (or the code instantiating it) must manually register all sensitivities (signals the component listens to). The major drawback of this is that it scatters signal registration code all over the

codebase, making the signal system hard to understand. To improve the simulation startup, components should be able to declare their sensitivities, which would then be automatically registered on instantiation.

The same goes for defining new signals: A process should be able to tell that it will write to some signal, and then the signal name should be automatically registered. Another major improvement would be strongly typed signals—if the type of a signal value is defined together with the signal value, the Scala compiler could detect if a write is valid or not. The same goes for read accesses, which would automatically return the correct type.

Closely linked to the signal write updates is another concept—guards. They should limit the allowed values for signals, e.g. in the pedals or for the engine torque. If a value is exceeded, either a notice should be logged or the simulation should be cancelled with an error, depending on the gravity of the error. This would help detecting mistakes in simulator components that would otherwise go unnoticed.

A problem that sometimes occurred during development are loops in the signal path. Problematic about them are endless loops that will occur if signals update each other or are updated in a daisy chain pattern (A→B→C→A→…). Such errors can be avoided by introducing a deadtime block that delays a signal, thereby breaking the chain of updates. In the example, if C was a deadtime block, the value would only be written back to A after a delay, thus not blocking the current time step.

To help detecting such problems, the signal paths should be checked. The following approaches are possible to tackle this:

1. Check the paths at startup time, when the signals are registered. A problem here is that the block types are not known to the relevant code parts, therefore deadtime blocks could not be reliably detected. Additionally, the “writes-to” relations of blocks to signals are not explicitly defined, while “reads-from” is defined through the sensitivities.

2. A more feasible, but slower, approach is to use the signal bus: It can keep all names of signals that were updated in the current time step and can then throw an error if signals are updated more than once[25].

3. An approach that would address the problem even earlier is static analysis. With static analysis, the source code of all components would be checked for read and write calls to signals, making it possible to get an exact graph of signal–block relations. However, this

[25] In some situations, multiple updates to a signal might be sensible. In this case, the limit should of course be raised.

would incur a greater amount of work, as no work into that direction has been done for this project. Also, for certain general-purpose blocks like PT_x blocks, the signal names are passed as parameters, hence also the process instantiations would need to be checked, in addition to the processes source code.

In the long run, implementing the first concept seems like the best solution; other parts could also profit from a more explicit definition of the relations between blocks and signals. The third approach would allow for a similar benefit, but is more cumbersome to implement.

7.2.3 Vehicle model

The top goal for developing the vehicle model part is creating a more realistic representation of the vehicle, to improve accuracy of the simulated driving cycles.

To get a more realistic simulation, traffic should be added besides improving the vehicle itself. This could be realised in two ways:

1. Define an abstract "traffic load" that influences the driver's decisions, e.g. by accelerating and decelerating more often. This is a refined version of the general load factor of 0.6 already applied in the experiments described in section 6.1.2.

2. Simulate real vehicles that drive on the streets with independent drivers. The single vehicle's positions must be known to all drivers, to allow proper steering and avoid collisions. Such a model would likely require lateral steering and a more fine-grained road implementation.

Implementing more than one vehicle would require namespacing the signals, or using different signal busses for the vehicle. The latter might be preferable for performance reasons, though the effect of a larger number of writes to a single bus has not been examined. If the vehicle busses are split, there must be some way to communicate information about surrounding cars to a vehicle. One possibility would be a common global bus which hold at least the position and heading, possibly also the velocity of each vehicle. Every process would then be able to access this bus and get the required information from there.

7.2.4 Road model

The current road model also contains a bit of information that is only relevant for the driver, e.g. the speed reduction before a turn. This information should be moved to the driver, freeing

the road from unrelated data. The driver should instead create a kind of driving schedule at the start, which could be used to already plan such low-speed areas and areas of higher attention, e.g. near traffic lights, once they are fully implemented.

Although traffic lights are already part of the road model, they are currently in an always-green mode, i.e. they have no state and behaviour attached and the driver also does not recognize them. As for the behaviour part, several models can be imagined:

1. The most simple one would be a fixed schedule of e.g. 45 s for every phase, alternating with a three or five seconds yellow period in between. The driver would then need to watch the traffic light while approaching it and react if its state changes.

2. A more sophisticated traffic light scheduling would take the road category into account and prolong the green phase on major roads, while shortening it for minor roads. This would require embedding deeper knowledge about the type of intersection into GraphHopper's routing graph, as currently the category of the other crossing roads is unknown. It is also possible that for some traffic lights the schedule is already present in the OSM data set.

The initial state of a traffic light might be randomized as the most simple approach. At the start of the simulation, for every traffic light the current state (red/green) and the time into this state would be randomly selected. This way, the start conditions would change with every simulation run, leading to more diverse results.

Alternatively, simulations should be initializable with a fixed state vector—as described above already—to get reproducible results. The vector for the traffic light state should be stored together with the other simulation parameters, so that experiments could be reliably repeated.

Another part that could improve the simulation in various ways are road categories. In OpenStreetMap, a number of road categories are distinguished[26], ranging from ”motorway“ down to “service” for on-site roads within industrial areas etc. Knowledge of the road category could be used by the driver e.g. to better estimate the possible velocity for a turn (for residential roads, the achievable velocity will likely be a lot smaller than for main roads in cities).

Additionally, the road categories would be necessary for the sophisticated traffic light model described above.

Road categories are currently not part of GraphHopper's data model, but could be added as edge properties like described in section 4.1.2 (there, adding node properties is described, but the process is the same for edges).

[26] see https://wiki.openstreetmap.org/wiki/Key:highway; visited 2015/10/01

For a more sophisticated model of the road segments themselves, the number of lanes should also be evaluated. The usefulness of this information largely depends on the degree to which this is part of the OSM data already. Further research into this area would be necessary before investing more time into creating a detailed multilane model with all the implications (implement overtaking in the driver model etc.)

As for nodes, "give way" signs should be added, as they incur a necessary decrease in velocity. This decrease could be done in two ways: either by implementing a slowdown segment right before the sign, like it is done for turns, or by letting the driver react to the sign. The former would be easier to implement, while the latter is a cleaner approach, as the road model should not have to hold all the information that governs the driver behaviour; instead, the driver should apply own intelligence to gather the target speed from all data present in the road model.

When implementing give way signs, a general model for right of way handling should be introduced. There are different situations like traffic lights, stop signs, or no signs at all, that require different, but similar handling. For turns, no delays are currently applied, as the road is always assumed to be empty. Until a full traffic model is in place, a random delay could be inserted at left turns to accommodate for oncoming traffic.

Such a right of way model could also take street names into account—they are already encoded in roads, but currently not evaluated. The street names could e.g. be used to detect a change from one street to the other, to tell turns and road bends apart.

7.2.5 User interface

For the user interface, a new implementation approach has already been started. It uses AngularJS as the underlying framework, allowing a better separation of the tasks and responsibilites than the old plain JavaScript solution.

The new user interface should allow parameterizing the simulation. This includes both a coarse-grained selection like the vehicle model or a specific driver implementation, but also finer-grained control of the simulator and model parameters.

Such finer grained control could include setting the initial state of various components like traffic lights, as it is already detailed above, and modifying parameters like delays or gains of some processes.

During a simulation, the user interface currently displays the time that has passed. This could be extended to show further information on the simulation progress. Desirable information would be the current position and travelled as well as remaining distance, to estimate the necessary time until the simulation run is completed. Additionally, internal telemetric data of

the vehicle, like engine torque, and driver parameters (current target speed and pedal positions) could be displayed.

To get a quick glance at a simulation run's results, currently a t–v diagram is displayed. Especially for longer running simulations, the result is hard to judge just based on this diagram, as it does not allow zooming and is relatively small. This diagram should be improved and a few further diagram types added. Further literature research concerning statistical evaluation of driving cycles should be performed first.

Using the tool (Art.Kinema) and parameters mentioned in [Bar+09], a basis for comparing different simulated cycles could also be established.

7.2.6 Tests

Currently, only some parts of RoadHopper are properly covered with unit and functional tests. Integration tests that validate the whole system are not implemented, due to a lack of fitting test model roads on which certain properties of a simulation run (length, achieved maximum speed, …) could be validated afterwards.

To get a chance to detect errors in newly introduced or changed components of RoadHopper, such system-level tests should be integrated. They could e.g. run a full simulation on a predefined small road network, allowing validation of the general simulation run and single simulation properties. The simulation properties that should be validated must be carefully selected, as otherwise the tests might be prone to breakage, e.g. when asserting a too narrow band for the allowed maximum velocity during the simulation or for the total time taken for the simulation.

The road data for these test scenarios should either be fetched once from a map and then stored statically in the test bed, or be artificially generated as a whole. The required infrastructure for the latter is already present in the form of the `RoadBuilder` class, which is used in various places. For the former, a custom serialization format for the road should be defined, e.g. based on JSON.

Complementary to these system level tests, the unit tests that verify the functionality of single classes should also be extended. Where applicable, the processes should also have unit tests or at least functional tests with a very limited scope, so that each component is tested in an as small-sized isolation as possible.

7.3 Vision for RoadHopper

This section should give a broader outlook on the direction RoadHopper could be developed into.

Simulating driving cycles is desirable for a lot of applications, from assessing emissions or energy usage to laying up components. Depending on the exact scope of the experiment, models with different characteristics are required—for designing a part, the surroundings of this particular element need to be modelled as closely as possible. Other parts might be modelled with only rough approximations if their influence is negligible.

As it was designed with flexibility in mind, RoadHopper could become a standard solution for all these kinds of different use cases. Standardized models could be delivered as plug-and-play components that can be loaded from data files, directly usable for simulations. Parts of these models could then be exchanged with custom implementations to perform different kinds of tests.

For defining such generic, interchangeable models, currently Scala code has to be written, which can be cumbersome. This task would become a lot easier if instead of writing plain Scala, a custom domain-specific language (DSL) could be used to define models. Such a DSL must include defining components and their behaviour (possibly by extending standard components like PT_1 controllers) and wiring them together. This language could e.g. be inspired by VHDL, which already was the inspiration for the signal model in RoadHopper.

Models defined that way should also be graphically displayed, using existing approaches for drawing graphs from data structures (or letting users manually order components). In such a graphical display, errors like unconnected components or signal loops would be easier to spot than in scattered source code.

In addition to custom models written in Scala or a custom Scala-based DSL, interfaces to external modelling software would be very useful. This way, existing models, e.g. in Matlab, or external simulation software e.g. for traffic could be connected to a simulation in RoadHopper.

Any kind of future model definition, whether it is graphical or via a DSL, should include model checking to detect hidden errors before even starting the simulation. This becomes especially necessary when more components from different sources are plugged together, as the potential for errors rises with different modelling styles and nomenclatures e.g. for signal names.

As for the driving visualization, a more appealing live/replay view could be realised by using existing 3D technologies, e.g. like in the OpenDS driving simulator[27].

The data model of RoadHopper could be extended with data from other data sources than the OpenStreetMap database. An example would be aggregated traffic loads to have a data basis for

[27] http://opends.de/

emulating traffic. Such a traffic model could also incorporate live data, e.g. fetched from the Traffic Message Channel (TMC) system used by radio stations worldwide.

Appendices

A Measurement processing

This appendix explains the algorithm that was used for postprocessing the measurement data from [Rap13]. The data was measured during several test drives performed with an Opel Ampera from ETI-HEV.

The data consists of multiple measurement files, each in the range of a few dozen minutes to multiple hours.

A.1 The algorithm

In general, due to their length the measurements cannot directly be compared to a simulation, at least not at the current state: There are too many interruptions where the vehicle stood still, which greatly distorts data like the average velocity or the duration of the trip. Therefore, comparison should instead be done on small driving pulses as defined in [Lia06].

The algorithm loops over all lines, ignoring those with a velocity of 0.0 (lines 30, 32). For every measurement point, the position, vehicle orientation and and velocity are extracted (line 42). These points are put into one object at the end of the driving pulse (line 20).

To get the driving pulses, the data is chopped into small pieces, based on traffic flow interruptions. For such an interruption, 10 s is assumed as the minimum standing time (line 36).

A.2 Source Code

```
1  val lines: Iterator[String]
2
3  var roadBuilder: Option[RoadBuilder] = None
4
5  // skip the first line, as it contains only header data
6  lines.next()
7
8  lazy val measurements: List[Measurement] = {
9    val items = new ListBuffer[Measurement]()
```

```
10
11    val buffer = new ListBuffer[DataPoint]()
12    // the counter for the seconds since stopping
13    var timeSinceStopping = 0
14
15    var group = 0
16    def endMeasurementGroup(): Unit = {
17      val newSet = buffer
18      if (newSet.nonEmpty) {
19        if (roadBuilder.isDefined && roadBuilder.get.segments.nonEmpty) {
20          items.append(new Measurement(name + "_" + group, buffer.toList,
21            roadBuilder.map(_.build).get))
22        }
23        roadBuilder = None
24        group += 1
25      }
26      buffer.clear()
27      timeSinceStopping = 0
28    }
29
30    for (line <- lines) {
31      // using ";0.000;" as an indicator that the speed is 0
32      if (line.indexOf(";0.000;") > -1) {
33        timeSinceStopping += 1
34      } else {
35        // vehicle did not move for more than ten seconds => start new measurement
36        if (timeSinceStopping > 10) {
37          log.debug(s"Starting new measurement group after line $c")
38          endMeasurementGroup()
39        }
40      }
41
42      val Array(time, _latitude, _longitude, _velocity, _heading) = line.split(";").map(_.trim)
43      // NOTE only some of our files had a velocity in knots; therefore, we assume km/h for now.
44      val velocityKmh = _velocity.replace(",", ".").toDouble
45      val latitude = _latitude.replace(",", ".").toDouble
46      val longitude = _longitude.replace(",", ".").toDouble
47      val orientation = _heading.replace(",", ".").toDouble.toRadians
48
49      // only include one measurement per second
50      if (time.indexOf(",") == -1  time.split(",").apply(1).equals("00")) {
51        // ignore slow movements for creating the road
52        if (velocityKmh > 1.0) {
53          handlePointForRoad(latitude, longitude, velocityKmh)
54        }
55
56        try {
57          val date = (time.substring(0, 2).toLong * 3600 + time.substring(2, 4).toLong * 60
58            + time.substring(4, 6).toLong) * 1000 + time.substring(7, 9).toLong * 10
59
60          buffer += DataPoint(date, Point(latitude, longitude, 0.0), velocityKmh / 3.6, orientation)
```

```
61        } catch {
62          case ex: NumberFormatException =>
63            log.error(s"Could not parse time '$time': ${ex.getMessage}")
64        }
65      }
66    }
67    endMeasurementGroup()
68
69    items.toList
70  }
71
72  def handlePointForRoad(latitude: Double, longitude: Double, velocity: Double) = {
73    val point = Point(latitude, longitude)
74
75    roadBuilder match {
76      case None => roadBuilder = Some(new RoadBuilder(point))
77
78      case _ => roadBuilder map {
79        _.addSegment(point)
80      }
81    }
82  }
```

B Velocity control state machine

This approach at controlling the velocity was developed as part of the first simple driver model and is currently not used in RoadHopper. It is described here because it might prove to be useful again in the future when a more sophisticated driver model is implemented.

Internally, the velocity control uses a state machine to tell different operation modes apart. It is still described here because its general model might be useful as a blueprint for future driver implementations that uses operation modes like those in the state machine.

The state machine is responsible for reacting to the road conditions and vehicle state which are passed to it from the outside. It controls the vehicle velocity by directly setting the desired acceleration/deceleration, an approach which was used in the first very simple proof-of-concept vehicle model, but is unfeasible for real models.

The three implemented states are

1. `initial`,
2. `free` and
3. `stop at position`.

State `initial` was only used for initializing the driver, after which it switched to the appropriate of the two other states.

To get the desired acceleration, the current speed and the driver's state were considered. The usual driving state is `free`, which denotes a mode where no obstacle to driving at the full allowed speed is currently known (e.g. a stop sign or [not implemented] other vehicles in front going slower). In state `free`, the driver watches the target speed (which is fed from the outside) and adjusts the acceleration as necessary.

When a stop sign is encountered along the way, the driver switches to state `stop at position` with a stopping position p. It then watches the current speed and remaining distance to the obstacle $\Delta s = p - s$ and starts braking (acceleration < 0) as soon as the required deceleration $a = {}^{v^2}/_{2\Delta s}$ hits a given threshold (i.e. s drops below the so-called *comfortable braking distance*).

Both velocity controlling states `free` and `stop at position` have no sophisticated mathematical model backing them. Instead, simple thresholds were used to reduce and increase acceleration depending on the remaining velocity difference Δv. The model also does not include a lookahead for determining the allowed speed, but only uses the current value. Therefore, speed limits were always missed when the vehicle approached them from a road segment with higher speed limit.

List of Figures

List of Tables

List of Listings

Glossary

Akka A toolkit for implementing the actor model. See section 2.6.2. 19, 28, 43

Delta cycle One execution step within a time step. During a delta cycle, only processes run; signal updates are delayed until the end of the cycle. 16, 78

Driving pulse A short period of movement in a larger velocity measurement. At the beginning and end of the pulse, the vehicle is standing still. 68, 89

Jetty A Java-based Servlet engine and HTTP server. Used by GraphHopper and RoadHopper. See `https://eclipse.org/jetty/`. 27

Meridian A line going along the earth's surface from one pole to the other. 9

Process An independent, sequentially executed set of instructions, called every time one of the signals it listens to (sensitivities) changes. 15, 78

Scala The programming language used for implementing the main parts of RoadHopper. See section 2.7. 20, 28, 80, 85

Sensitivity list The list of signals a process listens to. The process is executed each time one of these signals changes its value. This concept exists in VHDL, but is currently not implemented in RoadHopper. 15, 78

Thread A thread is a unit of computation within a process managed by a computer's operating system. A process can have multiple threads running in parallel which share data. 18

Bibliography

[Agh85] Gul Abdulnabi Agha. "Actors: A Model of Concurrent Computation in Distributed Systems". PhD thesis. Cambridge, MA: Massachusetts Institute of Technology, 1985. URL: https://dspace.mit.edu/handle/1721.1/6952 (visited on 2015-09-15).

[Akka] *Akka Scala Documentation for Release 2.3.13*. 2015-09. URL: http://doc.akka.io/docs/akka/2.3.13/AkkaScala.pdf.

[Bar+09] T. J. Barlow et al. *A reference book of driving cycles for use in the measurement of road vehicle emissions*. Published Project Report PPR354. TRL Limited, 2009-06, p. 276. URL: https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/4247/ppr-354.pdf (visited on 2015-09-27).

[Bat+09] Gernot Veit Batz et al. "Time-Dependent Contraction Hierarchies." In: *ALENEX*. Vol. 9. SIAM. 2009.

[Bat+14] Sarah E. Battersby et al. "Implications of Web Mercator and Its Use in Online Mapping". In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 49.2 (2014-06), pp. 85–101. ISSN: 0317-7173, 1911-9925. DOI: 10.3138/carto.49.2.2313. URL: http://utpjournals.press/doi/10.3138/carto.49.2.2313 (visited on 2015-10-07).

[Bra11] Oliver Braun. *Scala: objektfunktionale Programmierung*. München: Hanser, 2011. ISBN: 978-3-446-42399-2.

[Bra13] Hans-Hermann Braess. *Vieweg Handbuch Kraftfahrzeugtechnik*. Ed. by Ulrich Seiffert. 7., aktual. Aufl. 2013. Wiesbaden: Springer Vieweg, 2013. ISBN: 978-365-80169-1-3. DOI: 10.1007/978-3-658-01691-3.

[Bus90] David W. Bustard. *Concepts of Concurrent Programming*. Carnegie Mellon University, Software Engineering Institute, 1990.

[Cli81] William D. Clinger. "Foundations of Actor Semantics". PhD thesis. Cambridge, MA: Massachusetts Institute of Technology, 1981. URL: https://dspace.mit.edu/handle/1721.1/6935 (visited on 2015-09-15).

[DNE08] Zhen Dai, Deb Niemeier, and Douglas Eisinger. "Driving cycles: a new cycle-building method that better represents real-world emissions". In: *Department of Civil and Environmental Engineering, University of California, Davis* (2008). URL: http://www.dot.ca.gov/hq/env/air/research/ucd_aqp/Documents/2008-Dai-arterial-cycles-final.pdf.

[Don78] Edmund Donges. "A two-level model of driver steering behavior". In: *Human Factors: The Journal of the Human Factors and Ergonomics Society* 20.6 (1978). Don78, pp. 691–707. URL: http://hfs.sagepub.com/content/20/6/691.short (visited on 2015-08-17).

[DSW14] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. "Customizable Contraction Hierarchies". English. In: *Experimental Algorithms*. Ed. by Joachim Gudmundsson and Jyrki Katajainen. Vol. 8504. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 271–282. ISBN: 978-3-319-07958-5. DOI: 10.1007/978-3-319-07959-2_23. URL: http://dx.doi.org/10.1007/978-3-319-07959-2_23.

[EPSGReg] *EPSG Geodetic Parameter Registry*. URL: https://epsg-registry.org/ (visited on 2015-10-01).

[Est+01] A. Esteves-Booth et al. "The measurement of vehicular driving cycle within the city of Edinburgh". In: *Transportation Research Part D: Transport and Environment* 6.3 (2001), pp. 209–220. ISSN: 1361-9209. DOI: http://dx.doi.org/10.1016/S1361-9209(00)00024-9. URL: http://www.sciencedirect.com/science/article/pii/S1361920900000249.

[Far+07] T. G. Farr et al. "The Shuttle Radar Topography Mission". In: *Rev. Geophys.* 45 (2007). DOI: 10.1029/2005RG000183.

[FN98] Emmanuel Felipe and Francis Navin. "Automobiles on Horizontal Curves: Experiments and Observations". In: *Transportation Research Record: Journal of the Transportation Research Board* 1628 (1998-01), pp. 50–56. DOI: 10.3141/1628-07. URL: http://dx.doi.org/10.3141/1628-07.

[Gei08] Robert Geisberger. "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks". Diploma Thesis. Universität Karlsruhe (TH), Institut für Theoretische Informatik, 2008.

[GME07] David Wenzhong Gao, Chris Mi, and Ali Emadi. "Modeling and Simulation of Electric and Hybrid Vehicles". In: *Proceedings of the IEEE* 95.4 (2007-04), pp. 729–745. ISSN: 0018-9219. DOI: 10.1109/JPROC.2006.890127. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4168023 (visited on 2015-07-31).

[Gos+15] James Gosling et al. *The Java Language Specification. Java SE 8 Edition*. 2015-03. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[HM90] R. A. Hess and A. Modjtahedzadeh. "A control theoretic model of driver steering behavior". In: *Control Systems Magazine, IEEE* 10.5 (1990), pp. 3–8. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=60415 (visited on 2015-08-17).

[Hun+07] W.T. Hung et al. "Development of a practical driving cycle construction methodology: A case study in Hong Kong". In: *Transportation Research Part D: Transport and Environment* 12.2 (2007), pp. 115–128. DOI: 10.1016/j.trd.2007.01.002.

[Ire+09] C. Ireland et al. "A Classification of Object-Relational Impedance Mismatch". In: *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA '09. First International Conference on*. 2009-03, pp. 36–43. DOI: 10.1109/DBKDA.2009.11.

[KAR78] JH Kent, GH Allen, and G Rule. "A driving cycle for Sydney". In: *Transportation Research* 12.3 (1978), pp. 147–152. DOI: 10.1016/S1352-2310(99)00074-6.

[Lee06] Edward A. Lee. *The Problem with Threads*. UCB/EECS-2006-1. Berkeley: University of California, Electrical Engineering and Computer Sciences, 2006. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf (visited on 2015-09-15).

[Lia06] Bor Yann Liaw. "Fuzzy logic based driving pattern recognition for driving cycle analysis". In: *Journal of Asian Electric Vehicles* (Vol 2 2004-06). URL: https://www.jstage.jst.go.jp/article/jaev/2/1/2_1_551/_pdf (visited on 2015-09-15).

[Lin+15] Tim Lindholm et al. *The Java Virtual Machine Specification. Java SE 8 Edition.* 2015-02-13. URL: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf.

[LM09] Jan Charles Lenk and Claus Möbus. *Modelling Lateral and Longitudinal Control of Human Drivers with Multiple Linear Regression Models.* 2009. URL: http://oops.uni-oldenburg.de/1762/1/Lenk_Jan_156.pdf.

[LN02] Jie Lin and Debbie A. Niemeier. "An exploratory analysis comparing a stochastic driving cycle to California's regulatory cycle". In: *Atmospheric Environment* 36.38 (2002), pp. 5759–5770. DOI: 10.1016/S1352-2310(02)00695-7.

[Lu14] Zhiping Lu. *Geodesy: Introduction to Geodetic Datum and Geodetic Systems.* Ed. by Yunying Qu and Shubo Qiao. SpringerLink: Bücher. Berlin, Heidelberg: Springer, 2014. ISBN: 978-364-24124-5-5. DOI: 10.1007/978-3-642-41245-5.

[LWS94] Gunther Lehmann, Bernhard Wunder, and Manfred Selz. *Schaltungsdesign mit VHDL.* Franzis, 1994. ISBN: 978-3-7723-6163-0.

[Mac03] Charles C. Macadam. "Understanding and Modeling the Human Driver". In: *Vehicle System Dynamics* 40.1 (2003). Mac03, pp. 101–134. ISSN: 0042-3114. DOI: 10.1076/vesd.40.1.101.15875. URL: http://www.tandfonline.com/doi/abs/10.1076/vesd.40.1.101.15875 (visited on 2015-08-17).

[Mar12] Robert C. Martin. *Agile software development: principles, patterns, and practices.* Pearson Prentice Hall, 2012. ISBN: 978-0-13-276058-4.

[Mey10] Thomas H. Meyer. *Introduction to geometrical and physical geodesy : foundations of geomatics.* Redlands, Calif.: ESRI Press, 2010. ISBN: 978-1-58948-215-9; 1-58948-215-8.

[MH93] A. Modjtahedzadeh and R. A. Hess. "A model of driver steering control behavior for use in assessing vehicle handling qualities". In: *Journal of Dynamic Systems, Measurement, and Control* 115.3 (1993). MH93, pp. 456–464. URL: http://dynamicsystems.asmedigitalcollection.asme.org/article.aspx?articleid=1406356 (visited on 2015-08-17).

[Mit14] Manfred Mitschke. *Dynamik der Kraftfahrzeuge.* Ed. by Henning Wallentowitz. 5., überarb. u. erg. Aufl. 2014. SpringerLink : Bücher. Wiesbaden: Springer Vieweg, 2014. ISBN: 978-365-80506-8-9. DOI: 10.1007/978-3-658-05068-9.

[MS07] Mark Moir and Nir Shavit. "Handbook of Data Structures and Applications". In: Chapman & Hall, 2007. Chap. Concurrent data structures. ISBN: 978-1584884354. URL: https://www.cs.tau.ac.il/~shanir/concurrent-data-structures.pdf (visited on 2015-09-15).

[NGIA14] *Implementation Practice Web Mercator Map Projection.* National Geospatial Intelligence Agency, 2014-02. URL: http://earth-info.nga.mil/GandG/wgs84/web_mercator/(U)%20NGA_SIG_0011_1.0.0_WEBMERC.pdf (visited on 2015-10-07).

[NHG11] Phi Hung Nguyen, E. Hoang, and M. Gabsi. "Performance Synthesis of Permanent-Magnet Synchronous Machines During the Driving Cycle of a Hybrid Electric

Vehicle". In: *Vehicular Technology, IEEE Transactions on* 60.5 (2011-06), pp. 1991–1998. ISSN: 0018-9545. DOI: 10.1109/TVT.2011.2118776.

[NIMA00] *World Geodetic System 84*. Technical Report 8350.2 Third Edition. National Imagery and Mapping Agency, 2000. URL: http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf (visited on 2015-09-19).

[ODbL] *The Open Database License*. URL: http://opendatacommons.org/licenses/odbl/summary/.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. First Edition. Artima, 2008. URL: http://www.artima.com/pins1ed/.

[RAL] *Richtlinien für die Anlage von Landstraßen*. Forschungsgesellschaft für Straßen- und Verkehrswesen FGSV, Arbeitsgruppe Straßenentwurf, 2012.

[Rap13] Martin Rapierski. "Generierung fahrer- und fahrsituationsabhängiger Fahrzyklen". Bachelor's thesis. KIT, 2013.

[RT09] Frederik Ramm and Jochen Topf. *OpenStreetMap : die freie Weltkarte nutzen und mitgestalten*. 2., überarb. und erw. Aufl. Berlin: Lehmanns Media, 2009. ISBN: 978-3-86541-320-8.

[SHB10] Dieter Schramm, Manfred Hiller, and Roberto Bardini. *Modellbildung und Simulation der Dynamik von Kraftfahrzeugen*. Berlin, Heidelberg: Springer, 2010. ISBN: 978-3-540-89315-8. DOI: 10.1007/978-3-540-89315-8.

[Ski08] Steven S. Skiena. *The Algorithm Design Manual*. London, 2008. DOI: 10.1007/978-1-84800-070-4.

[Sto86] Michael Stonebraker. "The case for shared nothing". In: *IEEE Database Eng. Bull.* 9.1 (1986), pp. 4–9.

[StVO] *Straßenverkehrsordnung*. 2015-09-26. URL: https://dejure.org/gesetze/StVO/.

[Sue12] Joshua D. Suereth. *Scala in depth*. Shelter Island: Manning, 2012. ISBN: 978-1-935182-70-2.

[TM12] Wolfgang Torge and Jürgen Müller. *Geodesy*. 4th ed. De Gruyter textbook. Berlin [u.a.]: De Gruyter, 2012. ISBN: 978-3-11-020718-7.

[VHDL] IEEE. *VHDL Language Reference Manual*. 2000.

[Vin75] Thaddeus Vincenty. "Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations". In: *Survey review* 23.176 (1975), pp. 88–93.

[Wen+05] T. A. Wenzel et al. "Closed-loop driver/vehicle model for automotive control". In: *Systems Engineering, 2005. ICSEng 2005. 18th International Conference on*. IEEE, 2005, pp. 46–51. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1562827 (visited on 2015-08-17).

[WFB] Central Intelligence Agency. *The CIA World Factbook*. 2015. URL: https://www.cia.gov/library/publications/the-world-factbook/index.html (visited on 2015-09-24).

[ZC09] W. Zeng and R. L. Church. "Finding shortest paths on real road networks: the case for A*". In: *International Journal of Geographical Information Science* 23.4 (2009), pp. 531–543. DOI: 10.1080/13658810801949850.

[Zie10] Benedikt Zierke. "Sichere Gestaltung von Landstraßen durch definierte Straßentypen". PhD thesis. 2010.