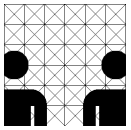
Diploma Thesis
in Informatics

Creating Highly Accurate 3D Representations of Large-Scale Outdoor Areas Using SLAM Algorithms

at the Department of
Technical Aspects of Multimodal Systems,
University of Hamburg

submitted by
Jan Christoph Gries
Jan Girlich
June 3, 2011



supervised by
Prof. Dr. Jianwei Zhang
Dr. Kay Fürstenberg

T|A
M|S

Abstract

Heavy transports are very important undertakings that due to their cost and risk factors require careful planning. The project 3D-Heavy Transport Route Finder (3D-HTRF) was developed by Gustav Seeland GmbH and SICK AG, which resulted in a car equipped with laser range finders and highly accurate, but expensive inertial navigation systems to build maps of possible transportation routes for quicker and cheaper scouting of routes. This work takes a look at the accuracy of the generated maps, as well as suggest and evaluates a method to further improve the maps. FastSLAM is a method from the field of robotics and originally developed for localization and mapping. We adopted the FastSLAM algorithm for 3D application and for using it to improve the match of laser scan data in a post processing step. The evaluation shows that, while the basic concept works, FastSLAM cannot help improving the maps of the 3D-HTRF project using the existing hardware setup. The reasons why are discussed and suggestions on how to overcome the problems faced are made.

Contents

List of Figures

Motivation

1

Heavy transports are vehicles moving objects of unusual large dimensions or weight that cannot be divided, exceeding the lawful limits for public vehicles in traffic. Heavy transports are very important for many areas of modern industry and often cannot be avoided, for example to move windmill propellers, industrial machinery, transformers, turbines, airplane wings and sometimes even small houses.

Those transports can be carried out with ships by water, via air with freight planes or by land on streets. Usually the most complicated transports are the ones on land because of man-made structures like houses, bridges, tunnels and lamp posts. Air and water do not have as many obstacles to circumvent as land transports.

Streets are not built for heavy transports to fit and need to be surveyed to evaluate if a transport will fit around street corners, under bridges and maybe which traffic signs have to be temporarily removed. Often the shortest way is the hardest way to go and detours are easier, but the best option should be found beforehand. For a heavy transport special permits and fees are needed and they are only allowed at certain days and times to minimize the impact on traffic. Main traffic arteries might be partly or fully closed for longer periods of time, since heavy transports usually drive very slowly for safety reasons. Because of the high costs and the critical time conditions special care has to be taken when planning a heavy transport. Mistakes are very expensive and high planning costs to avoid mistakes from happening are justified.

To find the best route and ensure a safe passage of a heavy transport scouts are sent out to examine all critical areas of the intended route and measure heights of bridges, width of roads, check for obstructions on street corners and so forth. Even during non peak hours, this measuring can take up to seven or ten days[1].

This is why the heavy transport company Gustav Seeland GmbH, Technical University Hamburg-Harburg (TUHH) and SICK AG started a joint venture called 3D-HTRF[2] to develop an enhanced way of scouting heavy transport routes on streets.

[1] `http://www.welt.de/print/die_welt/hamburg/article12983598/Gesucht-Die-optimale-Route-fuer-Riesenlaster.html` (April 07, 2011)

[2] `http://www.seeland-messtechnik.com/index.html` (April 07, 2011)

1.1 3D Heavy Transport Route Finder (3D-HTRF)

To ease the scouting process, a car is equipped with Laser Range Finders (LRF) which scans its surroundings while driving along the intended route. To find out where the car is at any given moment an Inertial Navigation System (INS) is installed in the car. An INS is a system with gyroscopes, acceleration sensors and GPS to precisely determine the position, speed and orientation of the vehicle it is installed in (for further information on INS see sect. 4.2). The LRF measurements and the position information from the INS are then combined to create maps which can be evaluated on a computer.

The project is complete and the car is in use by Gustav Seeland GmbH. In its current setup the car drives up to about 80 km/h and takes approximately 40,000 laser range measurements per second. Compared to systems working with SLAM algorithms this is very fast. For example the Pioneer used in [LNHS05] drives less than 20 km/h to get a higher scan point resolution but one requirement of 3D-HTRF was to travel without obstructing the normal traffic. So the typical speed of trucks was chosen.

1.2 Challenges of 3D-HTRF

For planning heavy transport routes the accuracy and the detail of the maps used have to be exceptionally high. Small objects like thin poles, hanging power cables and attachments to tunnel walls might all stop an oversize transport. On the other hand detours are expensive and therefore heavy transport companies work with safety distances as low as 4 cm between the vehicle and any object. This is problematic since the laser range finders already have a statistical error of $\sigma = 2$ cm (see tbl. 2.1). Assuming that the errors causing too long readings are irrelevant and the 4 cm tolerance equals two standard deviations of the laser scanner, $2 \cdot \sigma = 4$ cm, the error margin is met in $\frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\mu+2\cdot\sigma} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) \mathrm{d}x \approx 97.7\%$ of all measurements.

To increase the detail of a map and get more data which can be used to lower the reading error the car could go slower to take more measurements on the same area as with higher speeds. This will not help to increase the point density of a map in all cases, because some detail might be missing because of occlusion. Occlusion can be caused by vehicles, pedestrians or other moving objects. Most of the objects are occlude some certain areas only temporarily and so another possibility could be taken. Driving past area of interest multiple times in different directions and combining the scans increase the level of detail in a map and deal with the occlusions.

In the current system those methods do not increase the accuracy or detail of the maps. In fact combining multiple scans decreases the accuracy of the maps because the alignment has a high error rate. Localizing a vehicle is a difficult task and even high end solutions like the INSs used in the 3D-HTRF project have an error

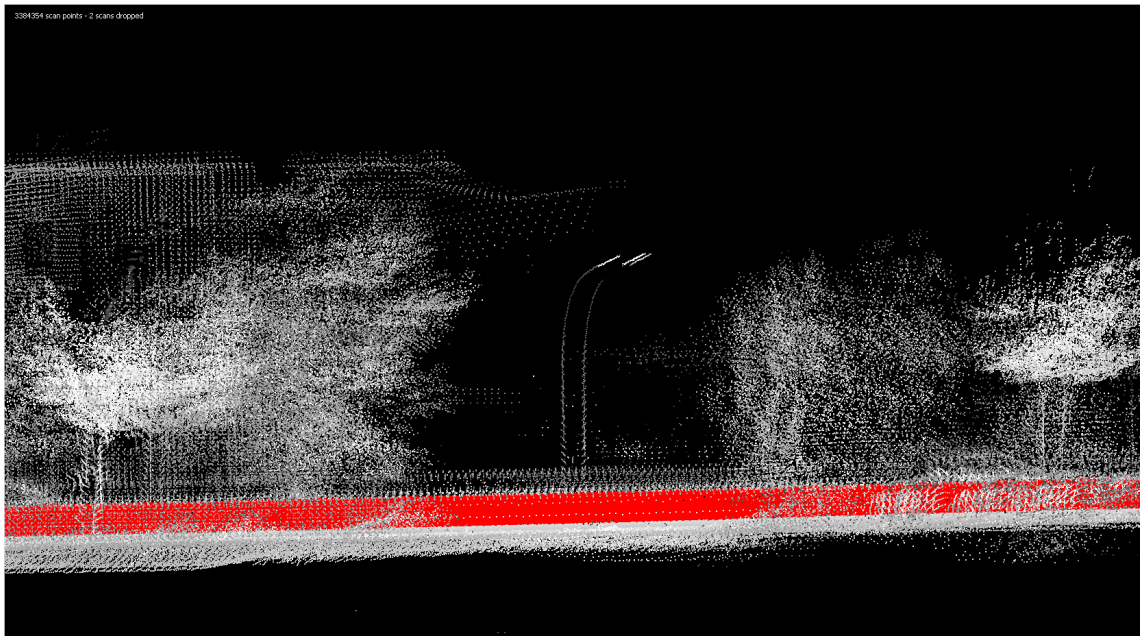


Figure 1.1: A street lamp shows up twice due to the localization error between two composed scan drives. The red line is the trajectory driven by the car

significantly higher than the accuracy needed for matching multiple scans within the desired range of accuracy (compare sect. 1.1 and table 2.3). When combining the maps of several scan drives this error leads to multiple images of objects (see fig. 1.1).

Further problems are tunnels or other constructions which block the GPS signal for a significant timespan. The INSs try to compensate the lost GPS signal by extrapolating the vehicle position by dead-reckoning with the acceleration and gyroscope data. The longer the signal is lost and the more curves are involved the worse these position estimates get. When the INS receives a new GPS position after a while the position gets corrected quickly and a "jump" from the last estimated position to the new GPS position occurs. This problem can be tackled with a technique called relaxation which was implemented and tested as well (see sect. 5.5).

Small changes in the vehicle orientation, e.g. like they occur when driving through a pot hole, might go undetected because of the short time duration within which they happen. But single laser scans taken during this short time frame might be affected. For the scope of this thesis such errors are ignored since they will only affect few scan points and can be regarded as measurement errors.

When a vehicle drives back to a position where it has been before, it created a loop. When working only with dead-reckoning closing such loops is a challenging task, but the 3D-HTRF project handles large loops very well because of the global positioning through the GPS.

For the purposes of the heavy transport company the global position is not of high priority, because the hard problems to solve are in small ranges of less than 50 m only. As long as the heavy transport company has a map which is about 4 cm accurate within any given 50 m radius, they are able to plan their heavy transport routes. Where exactly on the globe the problematic area is does not matter as much.

1.3 Experimental comparison of position data sources

Since localization accuracy is a very important aspect of the 3D-HTRF project, an experiment was set-up and performed to compare the quality of the three different position sources available. This experiment was also intended to get acquainted with the software framework and test vehicle including its sensors used in the following parts of this work.

Since no suitable 3D FastSLAM-package could be found for a comparison the well established SLAM software package SLAM-6D[3] was used. SLAM-6D is based on the Iterative Closest Point approach (see sect. 3.4) and later also used for a comparison to the FastSLAM algorithm implemented (see sect. 6.2.1).

1.3.1 Problem formulation

The experiment was conducted to answer these questions:

- How good are the maps created by the 3D-HTRF system?
- Which of the position sources of the 3D-HTRF test vehicle is the most accurate one?

1.3.2 Experimental setup

Six traffic cones (fig. 1.2(a)) are laid out on each end of an 100 m test track on a straight road in a low-traffic industrial area. The traffic cones serve as a reference to judge the quality of the matching process with the scan data. A straight track was chosen because it is easier for the driver of the car to follow precisely at a steady speed and thus to reproduce multiple times.

To determine the odometry three sources are possible. First the built-in sensors of the car, normally used for ESP (see sect. 2.3.1). The other two possibles are the informations delivered by the INSs Xsens MTi-G (see sect. 2.3.2) and RT3040 (see sect. 2.3.3).

[3] `http://www.openslam.org/slam6d.html` (May 3, 2011)

Figure 1.2: Arrangement of the traffic cones serving as a reference in the scan data.

1.3.3 Conduct of the experiment

The car drove the track up and down and scanned its surroundings a total of ten times. Each run's measurements are combined to one point cloud for each run and are treated as separate scan drives.

The first six cones are used to match the starting position of each run. To do this the first part of the map with the six cones is cut out and the program SLAM6D[4] is used to calculate the transformation matrix matching the ten runs one on another. The algorithm for this matching process is *Iterative Closest Point (ICP)* (see sect. 3.4).

This matrix is then used to transform the point cloud of the whole track. The six cones at the other end of the test track are used to assess and compare the accuracy of the odometry data and the data from the INSs.

1.3.4 Experimental results

The results of the experiment are a little surprise. As can be seen in figure 1.3 the car sensors generate a quite good result. The cones are in an area with a diameter of about 40 cm which is a maximum deviation of 4 ‰.

The data of the RT3040 produces a good result when ignoring the the height information. The diameter is with about 50 cm a little bit larger than the outcome of the car sensors data. When considering the height information there is an error of about ±30 cm as can be seen in figure 1.4.

The results of the Xsens MTi-G are shown in figure 2.3.2. The deviation is in a range of several meters and thus the worst result for all position sources.

[4] `http://www.openslam.org/slam6d.html` (May 3, 2011)

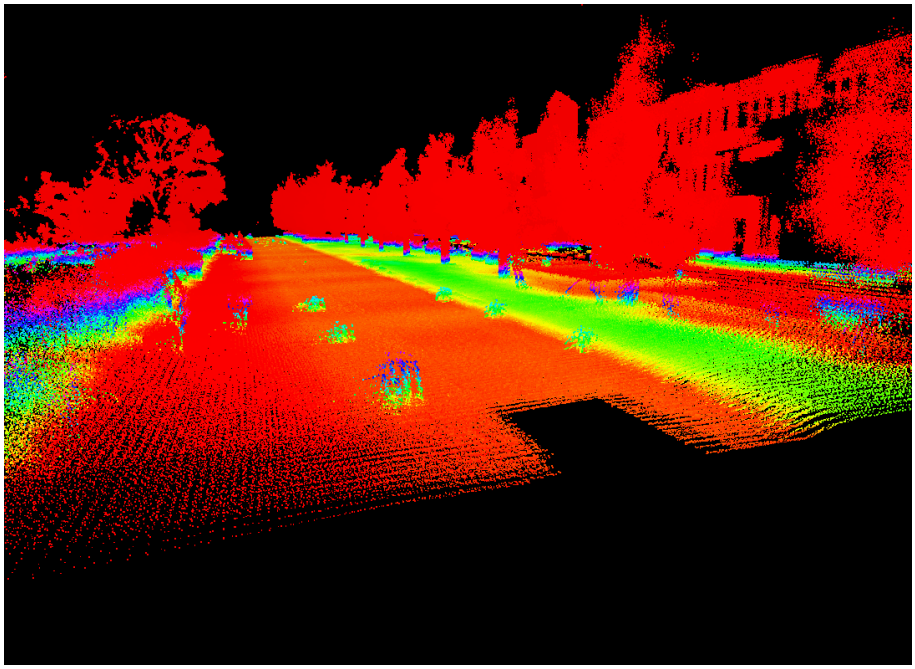


Figure 1.3: The point cloud of the car sensors does not include any height information and the accuracy of the resulting map is very good. The traffic cones of the ten point clouds are placed very close together by the SLAM algorithm forming these quite small clouds of cones.

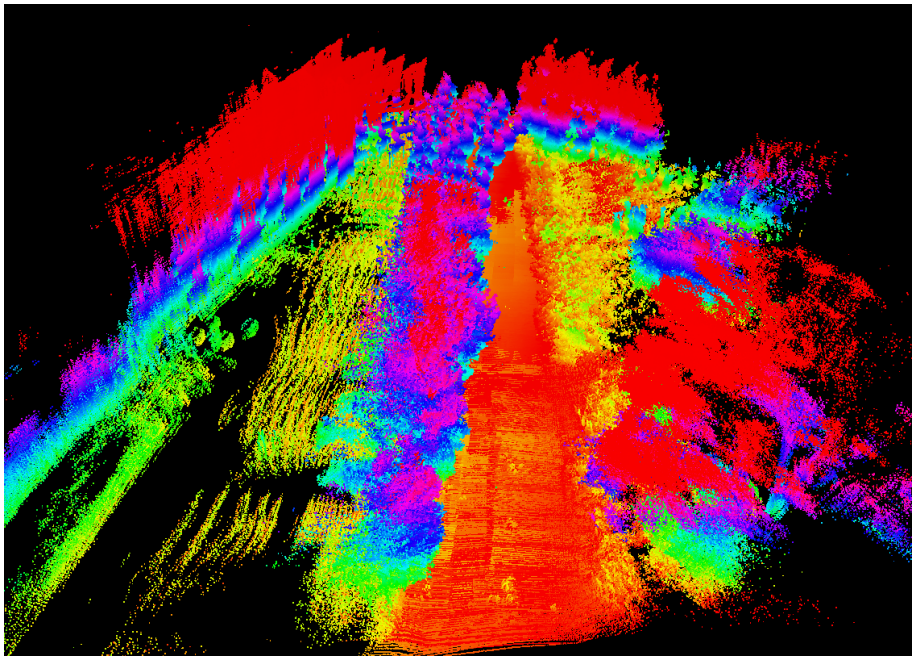


Figure 1.4: The Xsens MTi-G produced the worst results and an aerial view shows that no walls or trees are matched. Everything shows up five or more times in quite a bit of a distance which is enough to conclude that the matching did not work well on the data of this source.

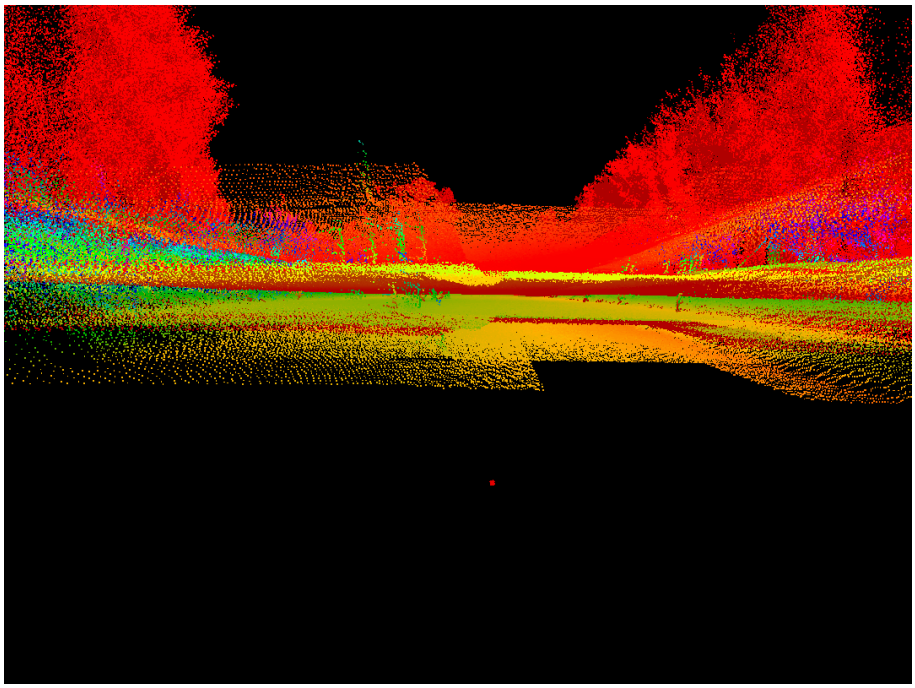


Figure 1.4: Although the Oxford Technical Solutions RT3040 is as expected much better than the results from the Xsens MTi-G, it does not reach the accuracy of the CAN data and lacks consistency in the height information. On this picture one can clearly distinguish the different scans at different heights around ground level.

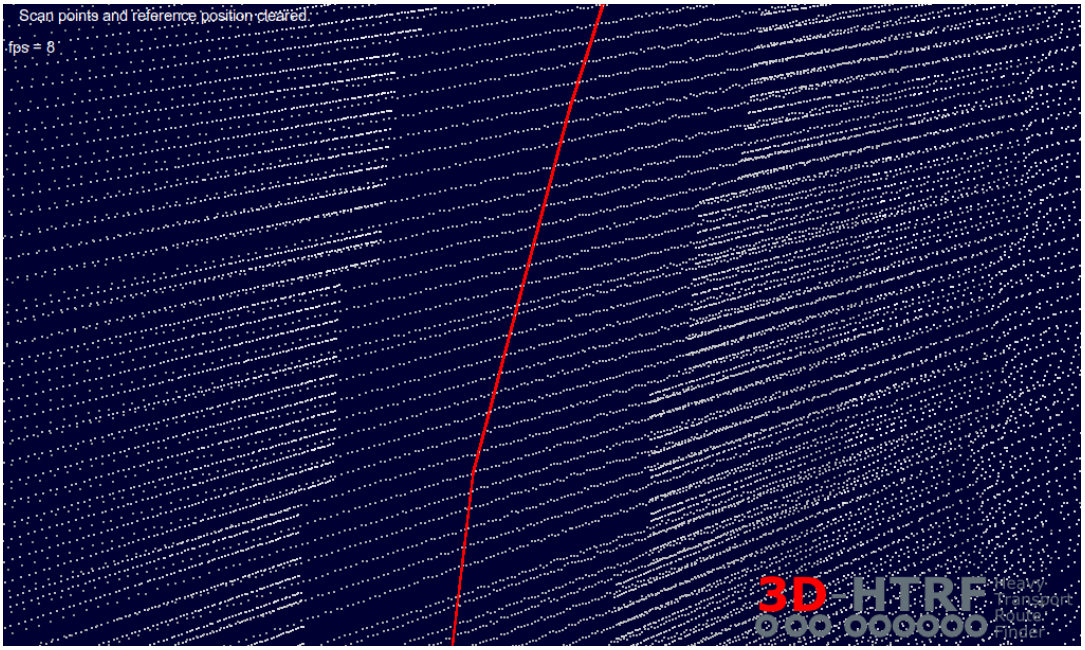


Figure 1.5: One can clearly see about 10 cm long horizontal offsets at the vertical line marking the beginning of the field of vision of the sideways pointing LRFs. Those 'jumps' are probably caused by a correction step by the Xsens MTi-G when a new GPS position has been determined.

1.3.5 Discussion

As shown the best result are the ones using the dead-reckoning data from the car sensors. That the provided height information is matched so well is not surprising since the car has no information about its height. What is surprising is that even without noticing the height the car sensors meet the goal best.

Possible reasons are that the setup the typical error precludes. When the vehicle travels at a constant speed straight ahead the tires never loose contact to the road. Perhaps it would appear different if the test contains curves or acceleration and deceleration would be involved. Anyway the cheapest localization system produces the best results. This possibly can be used to get a cheaper version of the HTRF-system when we deal with the combination of multiple scans with the help of a SLAM algorithm.

The bad results of the Xsens MTi-G can be explained when looking at figure 1.5. The update of the provided pose is very abrupt when a GPS position is received. This makes a precise matching of the scans measured between the start cones very difficult. A mistake made here will be amplified during the course of the test track. To make up for that an algorithm was built in our work called relaxation.

When matching the first scans of the test track ICP is used. The results are good

but this was one of the easiest tasks for a SLAM algorithm. All point clouds have the same borders given by the traffic cones. The measurements are always of the same objects and nearly every point has a corresponding point in the other scan.

1.4 Expanding 3D-HTRF to the field of robotics

As a manufacturer of Laser Range Finders often used in the field of robotics, SICK AG is in a cooperation with the Department of Technical Aspects of Multimodal Systems (TAMS) at the University of Hamburg. Amongst many other areas[5] TAMS is working on acquiring, processing and applying information from multiple sensory sources and developing robot-learning technologies like localization and mapping. The authors of this thesis are students at TAMS and contribute their expertise in the area of localization and mapping to the 3D-HTRF project. In the context of the cooperation this thesis has been created at the premises and with the great support of SICK AG. To further illustrate the specific contributions to the 3D-HTRF project by the field of robotics, the following section will introduce the reader to mapping as a research topic.

1.5 Mapping

Mapping traditionally is an important and well researched topic in the domain of robotics. This thesis will use state of the art approaches of mapping from the field of robotics and try to apply them in a scenario not typical for robotics.

Robots need a representation of their environment to interact with it or navigate within it. Those representations often are maps which include information about the areas the robot can move to and the areas blocked by obstacles.

There are many ways how a robot can retrieve a map, but it boils down to two basic options. Either it was programmed with a ready-made map or it is processing sensor readings and creates a map itself by exploring. On the one hand the latter method allows for dynamic adjustments if the environment changes and the robot can be deployed in unknown areas. On the other hand implementing mapping algorithms is a difficult task and running them costs much more processing power than using static maps.

Mapping algorithms are the base for many advanced robotic tasks with a vast field of applications. Service robots like automated vacuum cleaners build maps to know where they already have been and to avoid falling down stairs [PRSF00]. Architects and building managers can use maps and 3D models of buildings for planning construction works or utility studies and fire fighters can plan and evaluate emergency

[5] `http://tams-www.informatik.uni-hamburg.de/research/index.php` (May, 25th)

operations [HBT03]. A lot of research goes into search and rescue applications, e.g. in disaster recovery, and regularly competitions are held to help speed-up the development of new technologies in this area [JMW+03]. Cities are growing larger and former mining, industrial or other not well charted regions are urbanized. Exploring old mines, sewers and caves is important prior to urbanizing areas but can also help in rescue missions for miners or lost persons [aE01, NSL+04, TTW+04, MFO+06]. Also armed forces are interested in mapping technologies for various tasks, most prominently reconnaissance [TDD+00, Yam04].

These methods can help the 3D-HTRF project since the basic problem of creating maps for robots is very close to the way how maps are created for the 3D-HTRF project. The inertial navigation system used with the car can be regarded as a very accurate odometry source like it is usually assumed to be available for robots and the Laser Range Finders are used like the sensors of a robot. A main difference to typical robot tasks is that the car is controlled by a driver and no vehicle controls or actions have to be determined, but this does not affect the selected mapping algorithms. The main goal of this work is to take the data collected and assembled to maps by the 3D-HTRF car and create maps of better quality.

1.6 Simultaneous Localization and Mapping to improve map accuracy

The term Simultaneous Localization and Mapping (SLAM) is the description of a problem and the name of a class of algorithms to solve this sort of problem. This might be confusing at times since this work is about both: specifying the SLAM problem within 3D-HTRF and solving it using SLAM algorithms. Firstly the problem is described and then three possible algorithms are compared for their potential to solve the problem.

SLAM is about solving the mapping and the localization problem at the same time with the only knowledge being noisy observations from a sensor mounted at the vehicle and an estimate of its movement (for further details on what SLAM is check the theoretical background in chapter 4).

In the 3D-HTRF system Inertial Navigation Systems provide good pose data derived from GPS, movement sensors and dead-reckoning. Sensor observations are available from the three Laser Range Finders on the roof of the car (the hardware is described in chapter 2). Now the 3D-HTRF approach is to do mapping only and align the scan points from the sensors along the trajectory derived from the position data from the Inertial Navigation Systems. Although the systems used are highly accurate top of the line devices, the accuracy wished for by the heavy transport company is not reached yet. Trying to improve the pose information by localizing the vehicle more precisely within the map data does not work either, because the maps do not

exist a priori and suffer from the error one tries to correct. So the only way to solve this, based on the 3D-HTRF hardware and its restrictions, is to try to solve both problems, mapping and localization, at the same time, which is the very definition of the SLAM problem. Interpreting the pose information from the Inertial Navigation Systems as movement estimation and the scan points from the Laser Range Finders as observations, then SLAM can work on the data and create maps from it, which supposedly are more accurate than the input data.

Some adjustments are needed to adapt SLAM algorithms for the 3D-HTRF project. The most prevalent one being that the observations from the Laser Range Finders are taken in an unsuitable fashion. For SLAM to work properly the observations taken at one position need to be found over and over again and associated with other observations taken at other positions. The Laser Range Finders on the 3D-HTRF car are measuring vertical planes only, but the observations of interest for SLAM are to be found in horizontal directions. Thus a method named slicing is used to compose multiple vertical planes to larger point clouds assuming that the local positioning error on short distances is small enough to be ignored (slicing is explained in sect. 5.3.5).

The idea to use SLAM techniques for improving maps of already high quality has not been researched thoroughly yet. In contrast to other works [MTKW03, MR06, KSD+09] which use GPS as ground truth to evaluate the accuracy of their SLAM results, we can not do that because the GPS position is the one we try to improve.

There are many SLAM algorithms which could possibly be suitable for solving the SLAM problem of 3D-HTRF described. In the following sections we take a look at the three most common solutions and reason which one to use for this thesis (more on the different SLAM algorithms is said in chapter 3).

1.6.1 Iterative Closest Point (ICP)

Iterative Closest Point tries to match point clouds by minimizing the quadratic error between the individual points. It is a common and tried method, so we used a ready-made framework called SLAM-6D on the data from the above experiment. Goal of this test was to see if and how well ICP can handle the data from the 3D-HTRF project.

As mentioned before the data from the 3D-HTRF project is not suitable for direct use by SLAM algorithms and thus the data was preprocessed and composed to slices of approximately the length of 1 m prior to processing, but no landmark detection was done since ICP works on the full point clouds (see sect. 5.3 for more on preprocessing).

The result can be seen in figure 1.6 and is disappointing. Even when the position of the car is given as start parameter, ICP can not handle the given data at all. Most

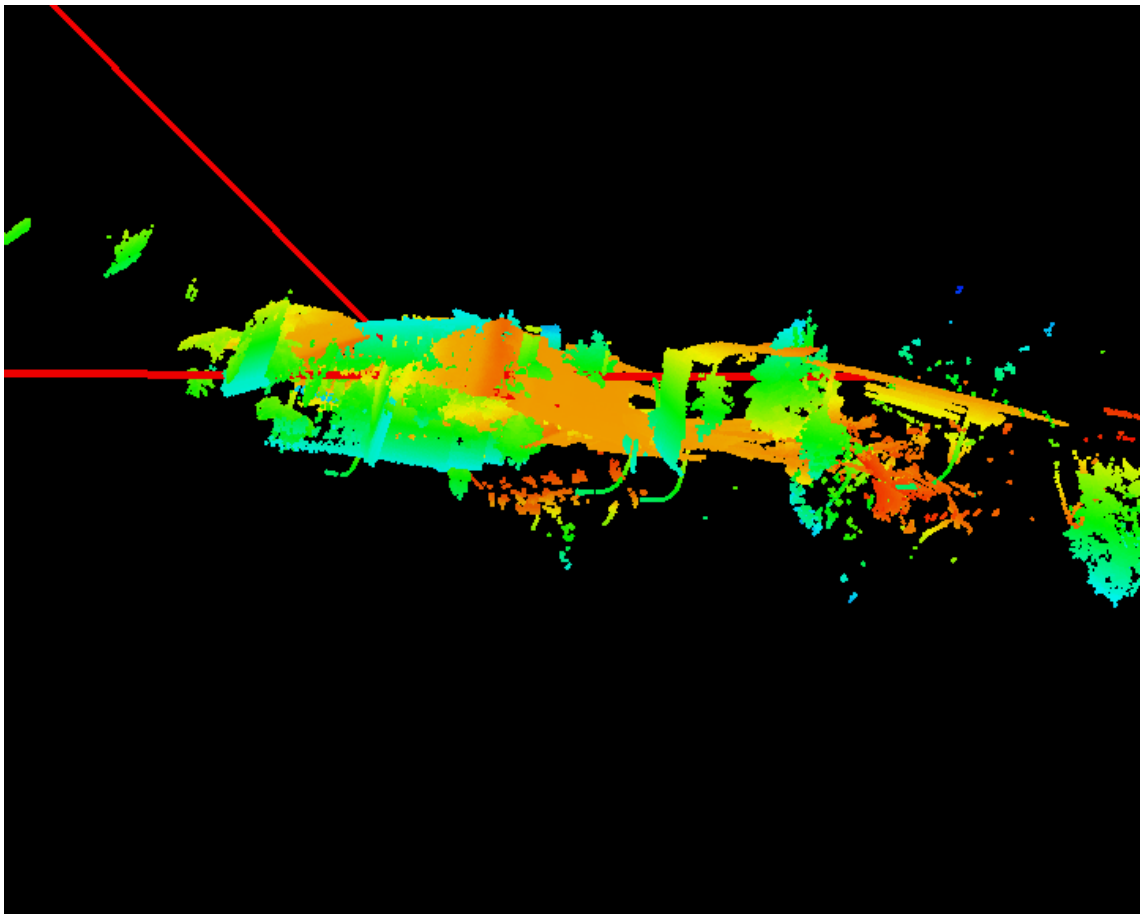


Figure 1.6: The SLAM-6D algorithm based on ICP was applied to scan data preprocessed in the same fashion as the scan data for the FastSLAM algorithm. This figure shows the result. As can be seen the straight street did not get reassembled at all by the ICP SLAM, but each slice is going in different directions and trees are pointing up and down.

likely the reason for this failure is that the point clouds only have little overlap, but ICP heavily relies on the fact, that the point clouds are very similar and overlap for the most part.

Also, matching all points of the point clouds is a calculation intensive approach because the Laser Range Finders take up to 1,000 measurements every 20 ms each. The following two SLAM algorithms reduce the calculations necessary by extracting landmarks (see sect. 4.6 about landmarks) from the point clouds and working on those.

1.6.2 Extended Kalman Filter (EKF)

Extended Kalman filter-SLAM tries to derive the most likely position by calculating a joint probability from an estimated position probability and an assumed position probability from matching observations with known landmarks. A huge disadvantage is that only one most likely position can be determined and in ambiguous situations where the data available allows for multiple interpretations of the most likely position, the EKF-SLAM has to decide on one possibility. Since the algorithm is greedy the decision can not be reevaluated in a later processing step. Thus mistakes accumulate. Since 3D-HTRF is designed for large maps, accumulating errors cannot be tolerated.

1.6.3 FastSLAM

Combining a particle filter and an extended Kalman filter allows to follow multiple hypothesis of the position in parallel and evens out the disadvantages of the EKF. The algorithm doing so is called FastSLAM. FastSLAM is a well researched algorithm which is used in many applications and found to be superior to a pure EKF-SLAM algorithm, e.g. by [SM06] (a complete explanation of FastSLAM is given in sect. 4.7).

With FastSLAM being the SLAM algorithm of our choice this thesis evaluates if it can be used for improving maps of the 3D-HTRF project.

1.7 Contribution of this thesis

This thesis is an evaluation if the hardware of the 3D-HTRF project is a suitable configuration to apply a FastSLAM algorithm on the collected data for improving the quality of the generated maps. In the 3D-HTRF project maps are composed using only the vehicle pose information provided by the INSs. First a way to preprocess and adapt the data collected by the 3D-HTRF car is developed and implemented. This technique is named slicing in this work. Then the FastSLAM algorithm by M. Montemerlo, S. Thrun and B. Siciliano [MTS07] is extended to fully operate in three-dimensional space. After writing an implementation of this extended algorithm it is applied on the 3D-HTRF data. The resulting maps of problematic test cases are then compared to the input data as generated by the INSs and evaluated for improvements in their accuracy.

Two advantages can be gained if the above described postprocessing of the 3D-HTRF data is successful and scan points are aligned better with each others in the resulting maps. This approach can be used for creating maps with much more detail by applying many scans of the same area to one map and the costs of the current 3D-HTRF system can be reduced by replacing expensive INSs with cheap postprocessing of the sensor data.

1.7.1 Reducing financial costs of the current system

The 3D-HTRF project does not require a precise localization, but high accuracy within a comparatively small section. The idea is that all information needed is if a heavy transport will fit through a tunnel, under a bridge or around a corner, but it is not important to know exactly where this particularly hard to maneuver spot is placed on the globe. Since there is no need for a global position, an INS is not necessary as long as the local accuracy conditions are met.

Also the map is not needed during the measuring process. The whole calculation process can be done offline after the data from the measurement vehicle is transfered to a desktop computer. The evaluation if the heavy transport can use the recorded route can be done on the same computer as the post processing step in which the map data is improved. So no new or more powerful hardware is necessary for the FastSLAM approach evaluated in this thesis.

1.7.2 Creating maps of higher accuracy using the 3D-HTRF hardware

Some places need multiple measurement drives because of occlusions created by driving by trucks or stationary structures in the line of sight. Or sometimes just more detail is needed and thus an area is scanned multiple times. FastSLAM can help adding such multiple scans of the same area onto each others in the postprocessing

step using the recorded sensor data. This is a new feature FastSLAM could make possible.

Lastly, a highly accurate, but expensive INS can be replaced by a cheaper, less accurate model while preserving the high accuracy so the system is still able to perform the task it was designed for, but save costs.

1.7.3 Code quality vs. runtime

Since this thesis is designed as a proof-of-concept work, the focus is not on performance or efficiency. In fact less performing but easy to understand code is preferred over fast but complicated code to ensure it works flawless. Time is not a concern for this evaluation as stated in the introduction of this chapter (see sect. 1.1), but the ability to understand and replicate this work is. For this reason many decisions made are in favor of simple structures, but longer runtime.

1.8 Summary & Overview

The 3D-HTRF project introduced in this chapter was built under the assumption that INSs can reach the position accuracy necessary for the heavy transport company. It turns out this accuracy is very hard and expensive to reach and especially hard to repeat when scanning the same place multiple times.

Firstly this thesis assesses how well the charting done in 3D-HTRF works and if the data produced is suitable to use with readily available mapping solutions from the field of robotics.

The experiment conducted in this chapter shows that the more expensive Oxford Technical Solutions RT3040 INS yields better results than the cheaper Xsens MTi-G INS, as expected. It also showed that the dead-reckoning data from the cars CAN-bus is surprisingly accurate, too. Although the overall quality of the maps is quite good, the structure of the scan data is not suitable for directly applying the SLAM-6D algorithm. There is a lot of room for improvement left, though, and an accordingly adopted FastSLAM might yield maps of higher accuracy.

The next chapter (chapter 2) describes the hardware used for the 3D-HTRF project and how it works in detail. This is the same hardware the implementation of this thesis has to work with, so a well-founded knowledge of it is essential for the rest of the work.

In “State of the Art” (chapter 3) an overview of the current scientific state on Simultaneous Localization and Mapping (SLAM) is given and the main approaches are described. This chapter classifies the FastSLAM algorithm in the scientific context of the field of mapping in robotics.

The theoretical background (chapter 4) behind FastSLAM lies in probabilistic methods. This chapter explains the mathematical inner workings of the algorithm.

Then this thesis adopts and implements a FastSLAM algorithm from the field of 2D indoor robotic scenarios to a large scale 3D outdoor environment. Chapter 5 leads through the implementation and explains crucial parts with code examples.

In order to evaluate if the 3D-HTRF project's hardware setup is suitable for FastSLAM and if any of the above specified advantages can be achieved, the resulting maps are examined for improvements. Consequently the results are discussed in chapter 6.

Finally in the last chapter (chapter 7) this work is summed up and an outlook for further research impulses is given.

Hardware

2

This chapter contains some information about the hardware used for the 3D-HTRF project. The hardware is taken unchanged for our work. It consists of a car, a superstructure with three Laser Range Finders and localization devices.

2.1 Car

The vehicle used for testing is a Volkswagen Passat with three SICK Laser Range Finders of the type Laser Measurement System 151[1] (LMS151, tbl. 2.1 & fig. 2.4) as can be seen in fig. 2.2. To each side of the car one LRF is mounted vertically and a third LRF is attached to the back in a way that the scan layer is hitting the ground about 1 m behind the car (see fig. 2.1 & 2.3). The rear laser scanner is skewed sideways by a few degrees to potentially get more reflections from surfaces which do not reflect the laser spots of the sideway scanners in a 90° angle to the car and to get informations from objects like poles that are also 90° to the ground and fit between two scan planes. To some extent this allows to look around the corner of objects and get a better impression of the measured shapes. E.g. detecting a fence can profit from this sensor arrangement because the pickets cannot be distinguished from a flat wall if the sideway scanner only measures the flat picket areas facing the scanner in a 90° angle or measures nothing in the holes between the pickets. But a slightly skewed scanner will more probably scan the inside surfaces between the pickets and thus add some more information about the shape of the pickets as if it was scanning the same surfaces in the same angle.

2.2 Laser Range Finder

The Laser Range Finders are used in the setup of the HTRF-Project are three LMS151 (see fig. 2.4). They belong to the group of outdoor short range finders and has as operating limits 0.5 m to 50 m. The field of view is 270 degree and scanning

[1] `https://mysick.com/saqqara/get.aspx?id=im0031331` (April 19, 2011)

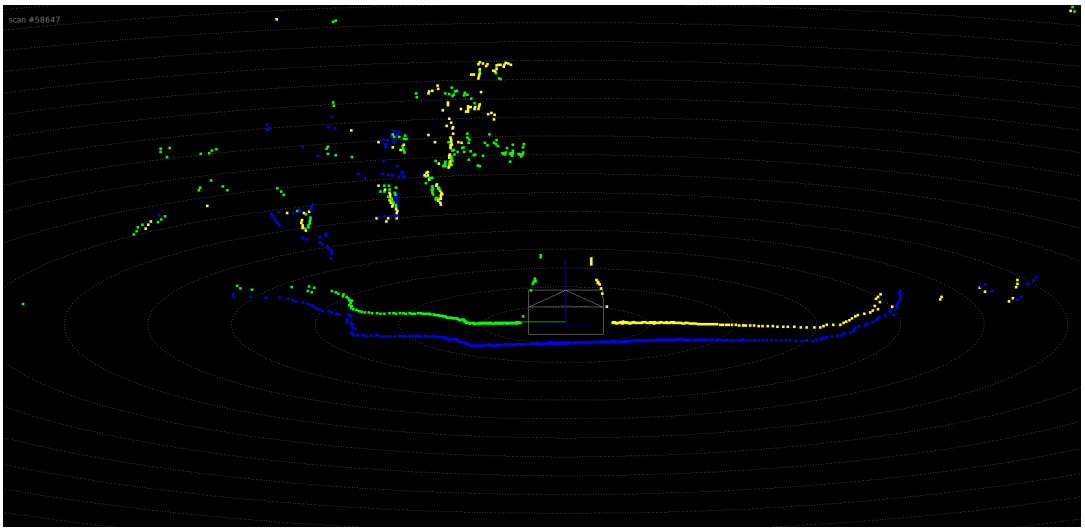


Figure 2.1: Picture from ibeolaserviewer with a tree on the left. The two scanners on the side are green and yellow and the one on the back of the car is blue. In the middle you can see some measured points originated from the vehicle itself.

Figure 2.2: The Volkswagen Passat used by Gustav Seeland GmbH with three SICK LMS151 laser range finders on the roof. One LRF to scan each side of the car and a third on the stern of the car.

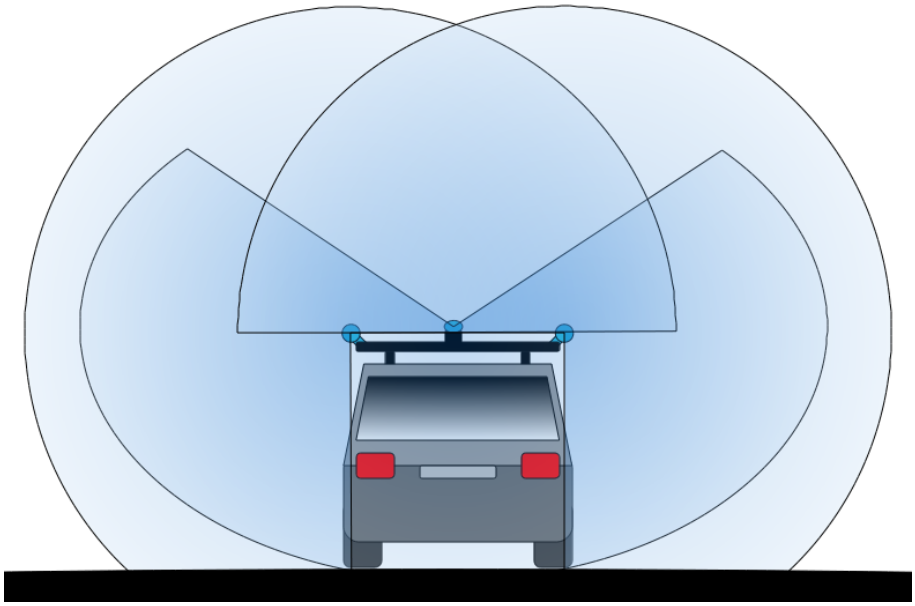


Figure 2.3: The three scanner are attached to roof rack of the car. The two scanner on the side are installed vertical. The one in the middle is tilted and can look behind the car.

Figure 2.4: The Laser Measurement System 151 (LMS151) from SICK.

Price	approx. €3,200
Scan angle	270°
Scanning range	50 m with object remission >75%
Scanning frequency	Either 25 or 50 Hz
Angular resolution	2.5° at 25 Hz and 5.0° at 50 Hz
Opening angle of laser beam	0.86°
Systematic error	Maximal ±40 mm
Statistical error (1σ)	Maximal 20 mm
Temperature drift	0.32 mm/K
Connectors	100 MBit ethernet port, RS232 port, CAN-Bus

Table 2.1: Excerpt from the LMS151 data sheet

frequency 50 Hz. When there is a need for a higher resolution and no need for a high frequency the frequency can be changed to the half. Connectivities are a 100 MBit ethernet port, serial port to standard rs232 and a CAN-Bus connector. CAN-Bus is an bus which mostly used by automotive and aerospace industries. As expected from an outdoor unit the scanner has a rugged housing with enclosure rating of IP 67. Some more technical information can be seen in table 2.1. The actual price is about 4,500 $[2].

2.3 Position data sources

For this work three data sources for vehicle positions are available. The first one is from the built in sensors of the car and the other two sources are ready-made INSs[3] which were bought for the 3D-HTRF project. While cars usually have gyroscopes and accelerometers which are meant for recognition of unstable driving situations and not for global localization of the car, INS additionally have GPS[4] and are able to localization a car precisely anywhere on the surface of th earth. GPS has different service qualities. In this project the free publicly available SPS[5] is used as opposed to the encrypted PPS[6] which is reserved for the US military. GPS position accuracy is mostly measured in meter CEP[7], which is the radius of a circle spanning the area within which the actual position is with 50% probability, or in meter RMS[8] which is basically the same measure as meter CEP but with a 63.2% confidence.

[2] `http://www.robotsinsearch.com` (April 29, 2011)
[3] Inertial Navigation System
[4] Global Positioning System
[5] Standard Positioning Service
[6] Precise Positioning Service
[7] Circular Error Probable
[8] Root Mean Square

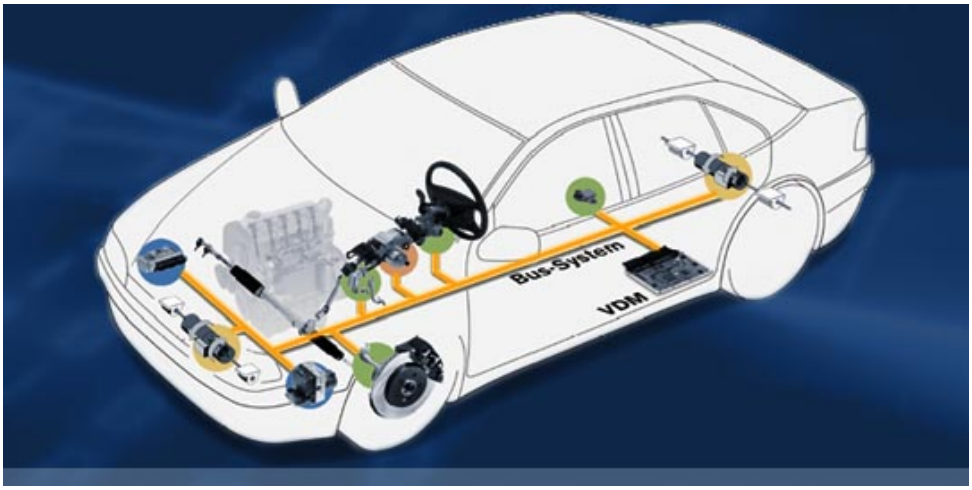


Figure 2.5: The CAN-bus connects all parts of the car and through it one can read out the sensors and control the car functions.

2.3.1 Car sensors

Modern cars are already equipped with accelerometers and gyroscopes. Airbags, ESC[9], ABS[10] and other systems need those sensors in order to work. Usually all sensors of a car are connected to the cars CAN[11] bus (see fig. 2.5) and their measurement data can be read out over an interface. The Volkswagen Passat used in this experiment provides information about

course angle in $[rad]$

yaw rate in $[\frac{rad}{s}]$

steering wheel angle (where the steering wheel is pointing at) in $[rad]$

steer angle (where the front wheel is pointing at) in $[rad]$

cross acceleration in $[\frac{m}{s^2}]$ and

velocity in $[\frac{m}{s}]$.

This information is sufficient to calculate a position but for retrieving a position data dead reckoning has to be used and the position the car was when the system is turned on is the origin of the coordinate system. Using the information about velocity, time and direction the next position is calculated. So there is no global localization.

[9]Electronic stability control
[10]Anti-lock braking system
[11]Controller–area network

(a) Xsens MTi-G (b) Oxford Technical Solutions RT3040

Figure 2.6: These two INSs were used in this thesis. The Xsens MTi-G (a) is a small, cheap and less accurate INS compared to the Oxford Technical Solutions RT3040 (b).

2.3.2 Xsens MTi-G

Xsens is a Dutch company which sells an INS called MTi-G[12] at about €3,500[13] which is a fair price when comparing to RT3040 mentioned in sect. 2.3.3. The picture of it are shown in fig. 2.6(a). The products leaflet[14] of the MTi-G opens with this self-description:

> The MTi-G is a miniature size and low weight 6DOF Attitude and Heading Reference System (AHRS). The MTi-G contains accelerometers, gyroscopes, magnetometers in 3D, an integrated GPS receiver, a static pressure sensor and temperature sensor. Its internal low-power signal processor provides real time and drift-free 3D orientation as well as calibrated 3D acceleration, 3D rate of turn, 3D earth-magnetic field, 3D position and 3D velocity data.

Price	approx. €3,500
Position accuracy SPS	2.5m CEP (50% confidence)
Roll/Pitch accuracy	<0.5°
Heading accuracy	<1°
Connectors	RS232 port, USB port

Table 2.2: Excerpt from the Xsens MTi-G data sheet

[12] `http://www.xsens.com/en/general/mti-g` (April 21, 2011)
[13] `http://damien.douxchamps.net/research/imu/` (April 21, 2011)
[14] `http://www.xsens.com/images/stories/products/PDF_Brochures/mti-g%20leaflet.pdf` (April 21, 2011)

2.3.3 Oxford Technical Solutions RT3040

Oxford Technical Solutions offer a range of high-precision INSs coming with accordingly high pricing ([JD06] mentions some prices). The RT3040[15] shown in fig. 2.6(b) is one of their top models using SPS, and if available also Satellite-Based Augmentation System (SBAS) which is a class of local satellite-based GPS correction signals and operated by local governmental institutions. The concept of SBAS is to have well surveyed stationary receivers which measure satellite signals and other environmental data which might influence the GPS signal and calculate a correction signal which is sent out through satellites and can be received for free by users to correct their GPS signal. In Europe the European Space Agency (ESA) operates the European Geostationary Navigation Overlay Service (EGNOS) which is used in the context of this work since all experiments took place in Germany. Additionally the RT3040 can handle OmniStar HP correction signals, if available. OmniStar is a proprietary, encrypted differential GPS service using its own satellites covering most of the landmass of earth. In order to use it a subscription is needed and for the best accuracy class called HP a yearly fee of $2,500 has to be paid[16].

Price	approx. €41,000 + $2,500/yr
Position accuracy SPS	1.5m CEP (50% confidence)
Position accuracy SBAS	0.6m CEP (50% confidence)
Position accuracy OmniStar HP	0.1m CEP (50% confidence)
Roll/Pitch accuracy	0.03° 1σ-interval (68.27% confidence)
Heading accuracy	0.1° 1σ-interval (68.27% confidence)
Connectors	100 MBit ethernet port, RS232 port

Table 2.3: Excerpt from the Oxford Technical Solutions RT3040 data sheet

2.4 System costs

Summarized the roof rack used for the 3D-HTRF project has a total cost of about €12,000. This contains the rack itself, the lasers, a computer and some cables. When it is decided to use the RT3040, its portion of the costs is more then 75%. Even when the car is included in the total costs of the system, the INS part is over 50%. So when the INS could be replaced by a cheaper INS or completely, the costs of the whole system would be dramatically reduced.

[15] `http://www.oxts.com/default.asp?pageRef=20` (April 21, 2011)
[16] `http://www.omnistar.com/pricing.html` (May 2, 2011)

3· LMS151	€9,600
Xsens MTi-G	€3,500
Oxford Technical Solutions RT3040	€41,000
Computer, cables, rack, etc.	€900
Car	€25,000
Total Costs:	€80,000

Table 2.4: Approximate costs of the complete 3D-HTRF hardware. This table ignores running costs for the satellite corrections signal for the RT3040 which is at $2,500/yr.

2.5 Summary

The 3D-HTRF system is basically a car equipped with three Laser Range Finders, a high precision Inertial Navigation System and a computer to control all the devices and collect their data. The configuration of the Laser Range Finders is catered to a pure mapping task and will cause problems with the usual assumptions of SLAM, as will be discussed in future chapters.

The hardware of the 3D-HTRF project is quite expensive and the high precision INS is more than half of the total costs. Thus aiming for replacing an INS with comparatively cheap processing time will cause tremendous savings on the project.

State of the Art

3

As laid out in the Motivation (chapter 1), SLAM is an approach promising to improve the maps created by the 3D-HTRF project. Thus in the State of the Art an introduction to Simultaneous Localization And Mapping is given and a look on the history will be done, which shows the relevance of SLAM in research. It is presented that SLAM can be used in many scenarios with various sensors. Further the filters which are commonly used for SLAM are named and it is said which ones and why they are used for FastSLAM. The last part describes some typical problems and requirements associated with SLAM.

3.1 The problem of Simultaneous Localization And Mapping

Simultaneous Localization and Mapping is often abbreviated *SLAM* and describes the problem of determining a robot's path and its surroundings at the same time. This is one of the most fundamental problems of robotics and occurs when a robot has neither a knowledge of its surroundings like a map, nor any information about its pose. All the information available to the robot are the readings from a sensor and the controls of its propulsion device. This little information is enough to solve the problem of mapping the environment and localizing within the map. As [MTS07] states each of these problems is well studied and feasible solutions are available, but the combination of those problems are in a chicken-or-egg relation to each others.

When a good map of the environment is available and the robots task is to find its position in the map, its called a *localization* problem. This can be done by observing and detecting landmarks in the sensor readings and matching them with the corresponding landmarks extracted from the map. Localizing the robot then is done by triangulation over these landmarks.

With the knowledge of the exact poses of the robot while taking readings from a sensor, a map can be built by simply combining the pose of the robot, the sensor reading and possibly the sensor's orientation over time. The resulting map will hold all sensor readings in the correct relations to each others. Solving this problem is called *mapping*.

Trying to solve both problems at the same time is harder than solving both problems separately, because they depend heavily on each other.

3.2 History of SLAM

First mentions of the probabilistic SLAM problem go as far back as 1986, when the problem was first formulated on the IEEE Robotics and Automation Conference held in San Francisco [DB06]. In 1995 the acronym 'SLAM' was first coined by [DRN96] on the International Symposium on Robotics Research. Since 2001 research interest in the topic has picked up rapidly. This can be seen in fig. 3.1 where the blue chart shows the new published papers per year occupied with SLAM. The red chart are the ones occupied with 3D SLAM like this thesis.

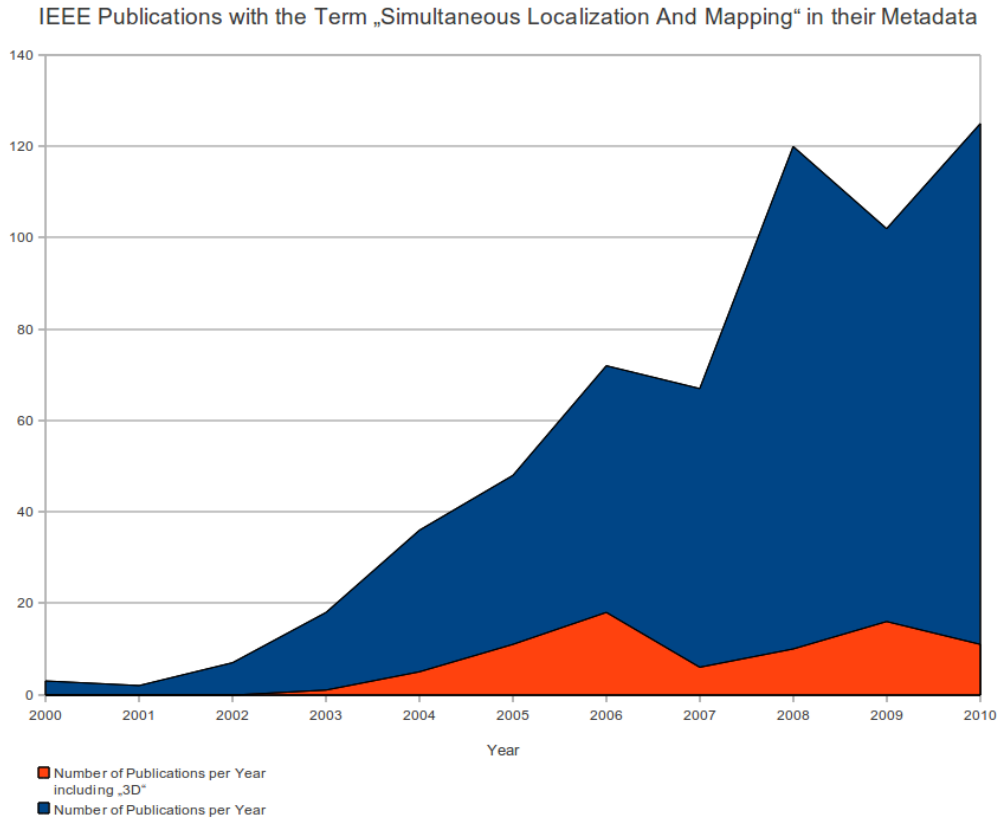


Figure 3.1: Development of scientific interest on *Simultaneous Localization And Mapping* by the number of publications in the IEEE publication database. SLAM is a growing field of research.

Typical applications for SLAM are found in the field of robotics. A robot equipped with one or more sensors is set in an unknown environment and given the task to

navigate and map the area. Managing this task is considered a basic ability for autonomous robots.

SLAM is not restricted to indoor or outdoor land-bound vehicles and for example is actively used in airborne devices [KS07], in underwater scenarios [ESLW06] and to map a mine [NSL+04].

For truly autonomous robots it is critical to process all sensor data and estimate its position in real-time, also called *online*. Without the processing done a robot cannot decide where to go next and thus is forced to wait for the calculations to finish.

In contrast *offline* means that the robot only collects sensory input and is either remote controlled or moving blind, not doing more than collision avoidance. The collected sensor and trajectory data is then processed after the robot finished its tour.

The advantage of offline over online is that in offline processing information from the future can be used since all processing is done after the recording. Usually when using offline SLAM the problem formulation goes a bit further than in online SLAM where the current position is often sufficient while in offline SLAM often the full path is sought. Offline processing then again has no or little time restrictions and the collected data can be transferred and processed on any computer, making large processing power available.

Contemporary 3D scanning technologies cannot take scene impressions instantaneously and need not-negligible time for one scan due to nodding or rotating a 2D LRF, which leads either to an unattractive stop-scan-move paradigm or distortions due to not properly modeled motion having to be accepted (see [NCC+07, HDB+10]).

Cole and Newman also proposed an approach called segmentation in [CN06], which combines several scans to a point cloud large enough for processing. The underlying assumption is that if the vehicle is driving somewhat straight the odometry information is accurate enough to ignore the error for a couple meters.

[DB06] and [BD06] offer a comprehensive overview of the history and the state of SLAM as a scientific problem in 2006.

3.3 Common sensors used with SLAM

The sensor technology used has a high influence on the SLAM algorithm. Depending on the sensor type different information is available in changing frequency, detail and accuracy.

Most systems can be divided in two-dimensional and three-dimensional approaches. With respect to the computing power available in the beginning of the long history of SLAM it is not surprising the first experiments were using two-dimensional approaches only. 2D is usually sufficient for robots which only operate indoors or in similar

environments with a flat plane the robot moves on and a 3D model is not necessary for operating a robot safely. With the ongoing development of robots upcoming new possibilities to move in space create a demand for proper three-dimensional perception and SLAM. Although there are more papers on 2D solutions, interest in researching 3D systems is growing (see fig. 3.1).

3.3.1 LIght Detection And Ranging (LIDAR)

Lidar devices are emitting laser pulses to measure distances to objects. The basic idea is that the light of a laser is only rarely dispersed when traveling through air and reflected back to the emitting source still strong enough to recognize the reflected light using a photo diode. One laser beam only yields one distance measurement. The measurement is quick though, since light travels very fast. To make better use of a Lidar the high measurement speed is utilized by directing the laser beam in different directions and taking measurements in quickly repeating patterns. Because the laser light used usually is close to the range of human-visible light, Lidars produce many scan points representing environments similar to the way the human eye recognizes them. Of course there is no color information or just a rudimentary interpretation of the distortions of the reflected light wave compared to the one sent out. The most common design of Lidars for robotics and similar uses are Laser Range Finders which are explained in detail in section 4.1.

3.3.2 Camera

Cameras capture pictures at a high frequency and can gather a lot of information at once. For example a normal webcam like the Logitech QuickCam 3000 delivers $640 \times 480 \frac{pixel}{f} \cdot 30 \frac{f}{s} \approx 9,000,000 \frac{pixel}{s}$. Compared to a LRF which measures about $1000 \frac{points}{scan} \cdot 50 \frac{scan}{s} = 50,000 \frac{points}{s}$ it is much more but the information of a pixel is not as interesting as the points of the LRF, especially when doing SLAM. The missing depth information using camera pictures makes the building of maps and doing SLAM difficult. Images require a lot of processing to extract landmarks usable for SLAM. Since cameras work in the visible spectrum of light but do not have their own light source they are very dependent on lighting conditions and recognizing the same view again a second time can be difficult, due to different lighting present. Cameras are cheaper than LRFs and do not have any moving parts, which makes them favorable over LRFs in cost-sensitive applications or if shock-resistance is important. Some recent work on SLAM working only with cameras can be found in [Dav03], [SLL05], and [KBO+06].

[ATS02], [NCH06], and [BZB+10] tried combining cameras and LRFs and fusing the data to seize the advantages of both technologies.

3.3.3 Radar

Radar stands for Radio Detection and Ranging and is a technology which uses radio waves to detect metallic objects. Like all electromagnetic waves radio waves travel at the speed of light. Usually an impulse is transmitted and the echo from one direction is observed. Using the time passed and the speed of light, the distance can be calculated. Good sensors even take care of the reflected frequency and compare it with the frequency of the sent impulse and can calculate the relative speed of the target by considering the Doppler effect. R. Rouveure, P. Faure and M.O. Monod use a K2Pi FMCW radar sensor in [RFM10].

3.3.4 Sonar

Sonar stands for Sound Navigation And Ranging and based on the same process as laser scanners and Radar sensors. A short wave is sent out an the echo is looked up. In case of Sonar the wave is a pulse of sound, so in this case the distance has to be calculated with the speed of sound. The consequence of this is that because of the lower speed compared to the speed of light there is no need for a clock that is as exact as the ones used in active Lidar or Radar sensors. On the other hand this has the result that the time needed for one scan increases. The second disadvantage of sound waves is the different speed of sound in different media. The speed in air is different from the speed in water, and even in water, where Sonar sensors are mostly used, the speed depends on the temperature, the depth and the salinity. Besides, sonar sensor arrays usually have a lack of information compared to LRFs, as shown in [TNNL02].

3.4 Iterative Closest Point

Iterative Closest Point or *ICP* is a method researched since the beginning of SLAM research [AHB87]. The idea is to minimize the squared error of the euclidean distance of the closest points between two point clouds in iterative steps. This is done using translation in both directions and one rotation which leads to three degrees of freedom (3DOF) in the two-dimensional case and three directions and three rotations (6DOF) in 3D. Due to sensory noise, different sensor positions and viewangles, and changes in the world between observations, sensors generally do not measure the exact same spot on an object. The points representing the same object in separate point clouds usually differ largely. Thus it is not possible to find a perfect match between two point clouds even if of the same area. Therefore the algorithm is stopped after a minimal error threshold is achieved.

ICP is an old (since 1992: [BM92]), basic and intuitive approach to SLAM which is easy to implement and has widespread use (e.g. in [LM97, NLHS07]). Although

it can be improved (e.g. in [RL01]), it is a computationally very expensive method with major drawbacks.

Especially in non-static environments moving objects which are scanned once with every observation but at different positions will show up multiple times in the final map. These errors cannot be corrected, sum up and increase the complexity of the matching process.

Another problem is that ICP only maintains one hypothesis of the position of the robot. Under some circumstances the position of a point cloud might be ambiguous and due to inaccurate odometry data the seemingly best pose estimate might actually be off by far. In such a case ICP does not have a way to detect or correct this mistake, but has to decide on one solution only.

Many methods are variants of the basic ICP principle or very similar in their approach. In Force Field Simulation [LALM07], for example, every point of a point cloud is interpreted as a mass influencing a gravitational field. Now forces similar to gravitational forces move the point clouds in the gravitational field until a configuration of least energy is reached.

3.5 Analytical approaches

Another approach to the SLAM problem is to model the error of the odometry and sensor data and determine the statistically most likely current position. For this problem Kalman filters are often used and are widespread since long. Rudolf E. Kálmán explained and published the concept of the Kalman filter most prominently in [K+60]. Since then Kalman filters have undergone a lot of research and adaptions and there are many variants. The most important Kalman filter variants used in SLAM are introduced in the following sections.

3.5.1 (Discrete) Kalman Filter

The Kalman filter is named after Rudolf Emil Kálmán, who presented it in *A New Approach to Linear filtering and Prediction Problems* [K+60]. The Kalman filter is an algorithm which calculates a presumed state and a covariance of a measured object. Prerequisites are the availability of several measurements and the knowledge of the error of the used measurement device.

$\mathbf{x}_k$ the state of the system at time k

$\mathbf{P}_k$ covariance of the state of the system

$\mathbf{F}_k$ main transition matrix from one system state to another

$\mathbf{B}_k$ dynamic of control vector

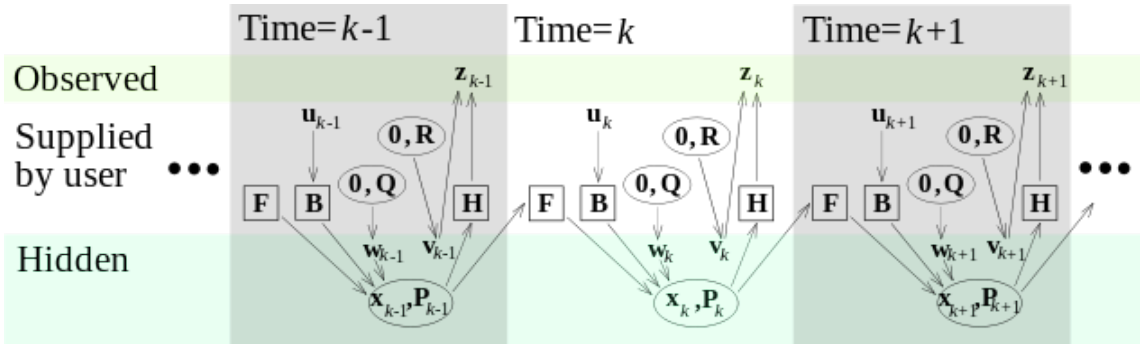


Figure 3.2: This diagram of the Kalman filter updates from time k-1 to k+1 shows how all the different factors influence the new estimate. (Taken from `http://www.marsa4.com/jmla/index.php?option=com_content&view=article&id=52&Itemid=57` on June 1st, 2011)

$\mathbf{u}_k$ control vector

$\mathbf{w}_k$ process noise which is assumed to be drawn from a zero mean multivariate normal distribution N(0,$\mathbf{R}$) for state transition

$\mathbf{z}_k$ observation

$\mathbf{H}_k$ observation conversion matrix

$\mathbf{v}_k$ process noise which is assumed to be drawn from a zero mean multivariate normal distribution N(0,$\mathbf{Q}$) for observations

The first step of a Kalman filter iteration is to estimate a new state $\mathbf{x}_{k|k-1}$. To do so, we need the old state $\mathbf{x}_{k-1}$ and the transition that contains of $\mathbf{F}_k$ and a controlpart $\mathbf{B}_k \cdot u_k$. The noise of the system which generate as a normal distribution is respected when the covariance is calculated.

$$\mathbf{x}_{k|k-1} = \mathbf{F}_k \mathbf{x}_{k-1} + \mathbf{B}_k \mathbf{u}_k \tag{3.1}$$

The second step is to take a look on the measurement. To use the measurement we could have to do a some transformation for example to get the sensor data to the coordinate system we used for $\mathbf{x}_k$ which is mostly a world coordinate system ENU or at least a system where the starting position is the origin. In some scenarios (e.g. in Thruns book FastSLAM [MTS07]) the used system is in a vehicle coordinate system. With the result that each time all landmarks for which calculation the Kalman filter is used have to be updated when the vehicle was moved.

The part $\mathbf{v}_k$ is again the noise represented by a normal distribution which takes part in the covariance calculation.

$$\mathbf{z}_k = \mathbf{H}_k \mathbf{x}_k \tag{3.2}$$

Both $\mathbf{z}_k$ and $\mathbf{x}_k$ consist of a mean value which is a non-probabilistic part and the probabilistic noise represented by a covariance. To get the new mean there are now two possible states and the Kalman filter has to decide which one to believe more and which one less.

$$\mathbf{x}_k = \mathbf{z}_k \cdot \mathbf{K} + \mathbf{x}_{k|k-1} \cdot (1 - \mathbf{K}) \tag{3.3}$$

$\mathbf{K}_k$ is called Kalman gain and can be calculated by taking a look on the covariances. The approach is that the covariance of the new state has to be as small as possible. Considering that even a bad measurement (high covariance) has a self-information and so these states should not be ignored completely the filter takes more respect to the state with a smaller covariance.

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}_k^{\mathrm{T}}\mathbf{S}_k^{-1} \tag{3.4}$$

$\mathbf{P}_{k|k-1}$ is the estimated covariance based on the same parameter as the estimation of the state.

$$\mathbf{P}_{k|k-1} = \mathbf{F}_k\mathbf{P}_{k-1}\mathbf{F}_k^{\mathrm{T}} + \mathbf{Q}_k \tag{3.5}$$

and $\mathbf{S}_k$ is the so called Innovation covariance

$$\mathbf{S}_k = \mathbf{H}_k\mathbf{P}_{k|k-1}\mathbf{H}_k^{\mathrm{T}} + \mathbf{R}_k \tag{3.6}$$

at last the new covariance is

$$\mathbf{P}_k = (I - \mathbf{K}_k\mathbf{H}_k)\mathbf{P}_{k|k-1} \tag{3.7}$$

The Kalman filter can be used in many scenarios like fusion of several measurements of different sensors for the same fact. For example, the localization with the help of GPS and odometry data (see sect. 4.2)

A problem of Kalman filters is that not all systems can be described by the equation (3.1) and (3.2). For example sensor data of laser-scanners mostly given in a range and some angles must be converted in a world coordinate system. This is a nonlinear function and so the result of the conversion of the covariance no longer is a normal distribution or sometimes the measurement itself is a non normal distribution and cannot or should not be estimated as a normal distribution. The first problem the extended Kalman filter is trying to solve by a approximation of the observation conversion matrix with help of a Taylor series (see sect. 3.5.2). For the second problem there is no analytical solution and the only way is to use a numerical approach (see sect. 3.6).

3.5.2 Extended Kalman Filter

Opposed to the Kalman filter the extended Kalman filter assumes that the state transition function and the transformation measurement can be non-linear functions,

too. Therefore the extended version does a linear approximation of the state to calculate the covariance while still using the same main concept of the Kalman filter.

(3.1) with the new state transition function

$$\mathbf{x}_k = f(\mathbf{x}_{k-1}, \mathbf{u}_{k-1}) + \mathbf{w}_{k-1} \tag{3.8}$$

(3.2) with the new measurement transformation

$$\mathbf{z}_k = h(\mathbf{x}_k) + \mathbf{v}_k \tag{3.9}$$

To do so a Taylor series is used. A representation of a function as a Taylor series is a sum of an infinite number of summands. In the normal case the number of summands is because of the performance limited. In this case functions are represented in the surroundings of certain points. The more summands are uses the better is the approximation around the point or the space in which the error is smaller than a given epsilon.

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x-a)^n \tag{3.10}$$

For the extended Kalman filter the first Taylor polynomial is needed.

$$f(a) + \frac{f'(a)}{1!}(x-a) \tag{3.11}$$

When we transfer this function to our case there are two application scenarios. First to calculate the mean x and a are the same, the second summand is zero and only $f(x)$ left. The second scenario is the calculation of the covariance. In this case the only thing needed is the difference between two results. From this and the linearity of the function it follows that no constant value has to be attended and only $f'(a) \cdot x$ left and $f'(a)$ can be insert for $\mathbf{F}_k$ and $\mathbf{H}_k$.

3.5.3 Further Kalman Filter

Over time, there are many more variants of the Kalman filter which are mentioned here but not treated further. The first is the Unscented Kalman filter presented in 1997 [JU97] and the second the Kalman–Bucy filter was the first version that can be used for systems using continuous time [BJ05].

3.6 Numerical Approaches

A numerical analysis based on the idea to calculate continuous mathematical problems by calculating only some values. Based on this results new points are selected and new values are reckoned. Of course this can only create an approximation but for n iterations and $\lim n \to \infty$ it should be the same as the result of the analytical analysis if it exists.

It is generally assumed that the process to be calculated can be represented by a Markov process. Therefore any new state has to be predictable from the previous state without further information from the past. Also it is assumed that the system satisfies the Chapman-Kolmogorov equality.

Chapman-Kolmogorov equation according to [Wen07]:

$$p(x_k|Y_k) = \frac{p(y_k|x_k) \cdot p(x_k|Y_{k-1})}{p(y_k|Y_{k-1})} \tag{3.12}$$

particle filter, also known as Sequential Monte Carlo method (SMC), is a numerical approach to solve the Problem of a state estimation. Like in any other numerical solution there are no functional limitation so that the state transition function and the transformation of the measurement can be out of any function set. Even $\mathbf{w}$ and $\mathbf{v}$ need not to be out of the set of normal distributions.

$$\mathbf{x}_k = f(\mathbf{x}_{k-1}, \mathbf{u}_{k-1}) + \mathbf{w}_{k-1} \tag{3.13}$$

$$\mathbf{z}_k = h(\mathbf{x}_k) + \mathbf{v}_k \tag{3.14}$$

The basic idea is to create some particles which each represents one possible state of the system so that the sum of all particles is an approximation of the distribution of the probability of the system at the time k.

$$\int f(x_{k-1}) p(x_{k-1}|y_0, \ldots, y_{k-1}) dx_{k-1} \approx \frac{1}{P} \sum_{L=1}^{P} f(x_{k-1}^{(L)}) \tag{3.15}$$

Proceeding from the state $\mathbf{x}_{k-1}$ of the particle the new state $\mathbf{x}_k$ of the particle is calculated by 3.13. Based on the measurement each particle gets a rating representing how well the measurement fits with the state. Now the particles with a low rating have to be removed and those which have a good matching should survive or even should be used to get new particles out of them. This procedure is called resampling. Resampling is a kind of importance sampling and there are a lot of algorithms but each one has to interpret the rating as a probability. The probability if it is used one or more times again or not.

3.7 FastSLAM

FastSLAM is an algorithm that was developed by Michael Montemerlo, Sebastian Thrun, Daphne Koller and Ben Wegbreit [MTKW02]. It is based on the idea that particle filters have features, that could be useful to solve the SLAM-problem. This is e.g. the ability to handle multiple possibilities independently and that it can be done over a longer period without neglecting one. Unfortunately this cannot be transferred fully to an implementation because it will not be practical. There are too many unknown variables that have to be presented by a probability distribution. The number to cover this high-dimensional space with particle would be too large. Because of this restriction FastSLAM attempted to use the particle filter only in a part of the problem.

SLAM can easily be divided in two parts. The first is the localization of the vehicle and the second is the update of the map. The localization is done with the help of the odometry and the comparison of the old map of time $k-1$ and the measured data. The new position is a combination of the odometry and the position where the measured data matches best with the data already known from the map. Sometimes there is more than one position that can be evaluated as well. These are mostly not as close together that they are unsuitable for presentation as a normal function. This situation can usually only be solved after a longer time. This makes it difficult for the Kalman filter and suggests to use a particle filter.

The map update step is different, because it is about whether existing points of the map should be postponed again because they have previously been measured incorrectly. The measured values can be approximated as a normal distribution as well as the points of the map that are older measured values. The reunion of both is exactly what a Kalman filter is made for.

So FastSLAM is a combination of a particle filter for localization and a Kalman filter for the map update. An interesting work about FastSLAM made by B. Steux and O. El Hamzaoui [SH09]. Here a minimal minimal version of the algorithm is implementated. Sebastian Thrun has prepared a lot of works. Two of his books are [MTS07] and [TBF05] and with J. Nieto, J. Guivant and E. Nebot he presents a solution for the data association of the map and the measurements [NGNT03]. A mapping in real time algorithm is presented in [Men07].

3.8 Related work

Most works about SLAM do not use GPS for positioning, but rely solely on odometry data for their motion model. If GPS is used it usually is only for comparing the positions derived with a ground truth and to assess the results. The goal basically is to replace GPS by SLAM.

[KS04] and [Car08] go a bit further and use GPS for mapping purposes. SLAM is then used to compensate the loss of a GPS signal, but the results are still only compared to a GPS position. This problem is similar to the problem an INS encounters when driving through a tunnel and loosing the GPS signal for a while.

Some works like [BLO+09] fuse stereo vision and GPS with SLAM to create highly accurate maps of large areas. The difference to our work is that we do use Laser Range Finders instead of stereo vision. The difference in the sensor technology is immense, since cameras and LRFs gather different amount of different data at different speeds. Images can be used for SIFT and similar feature detectors which do not compare to landmark extraction methods on scan point clouds.

Altogether there is only little work done with the aim of using GPS and LRFs to create highly accurate maps using FastSLAM. Usually the accuracy needs are lower than the ones asked for in heavy transport route planning.

3.9 Other SLAM related research topics

A lot of effort is put in improving the quality of the generated maps [WSBC10], speed [LNHS05], complexity [SH09] or other aspects of the above described methods by modifying them slightly. But there are more general approaches on how to solve the SLAM problem. Some approaches are going further than the basic set of methods. Atlas [BNL+03] is a framework which represents the poses of the system as nodes in a tree and the matching of landmarks is done by using a signature. Another complex framework is GridSLAM [HBFT03]. It contains a particle filter and grid maps. Although only designed for 2D it has a huge number of algorithms for example for loop closing (see sec. 3.9.1).

3.9.1 Loop-Closing

Although SLAM is capable of correcting the robot pose and creating a good map of the environment, small errors remain and accumulate. These errors become very obvious when a robot moves in a loop and meets its path at a later point in time again. At such junctions the previously created map must match up with the recent robot pose and sensor data, but often this is not the case because of the accumulated error. The distance between the expected position of the observation and the position of the corresponding landmark is too far. To detect the loop it is necessary to try to match the observations to the whole map. This is very expensive. Therefore this can only be done rarely. When a match is found the trajectory has to be updated because of the correction of the last position. To do so a method called relaxation is used which smooths the trajectory in a recursive manner all around the found loop (see sect. 3.9.2). This whole correction process is named loop-closing and a commonly discussed topic (e.g. in [ED08, SHB04, CN07]).

3.9.2 Relaxation

maps created with a SLAM algorithm often have local distortions. During a typical SLAM process the latest scans point cloud is added to the current model at the most likely position estimated from the last position. This process is repeated iteratively and position errors due to dead reckoning from odometry accumulate. Estimated positions can be represented by dots in a graph in which a line stands for an estimate of one position from another. Correcting each position in relation to its connected neighbors iteratively uses all position estimates in every direction instead of the least one only. For the correction a simple mean position of all estimates about a position is sufficient. Relaxation is shown to yield globally good results [DMS00]. A more complex algorithm is shown in [FLD05]

3.9.3 Kidnapped Robot Problem

In SLAM an estimate of the current robot position is needed. These estimates are often taken from a robots odometry information. If a robot is accidentally or forcefully taken away from its current position without updating the pose estimate, the estimate is wrong and the SLAM algorithm cannot succeed properly. Unfortunately this can happen very easily by unforeseen events. A robot might tumble down some stairs, move in and out an elevator at different levels, drive over a conveyor belt, taken by people and set down at a different place, and come into many more situations where the pose estimation will be rendered useless. This Problem was first mentioned in [EM92] and [Thr02] and [TNNL02] address this issue in detail. If there are possibilities for global localization (see sect. 3.9.4) there is no kidnapped robot problem. This is because in a global localization the system assumes that the starting position is not known and if a kidnapped robot situation is suspected it can be easily solved by restarting the global localization.

3.9.4 Global and local localization

If one needs global localization, one has to assume not to know the starting position. In contrast to a local localization, which creates a map starting from a given or freely chosen position (usually the origin of coordinate system), the global localization is used to create a general map. This general map can subsequently be extended or linked with other maps. It is the task of the system to acquire its own position from the measurement data. This can be done by a given map where the system has the task to find the position at which the measurement data adjusts to the recorded objects in the map. A typical approach to do this is the Monte Carlo Localization which was introduced in [DFBT99]. The algorithm tests some possible positions in the map and around the positions that provide the best results further positions are tested. Sometimes there is the need of more than one scan to solve the localization

problem. Much easier is a global localization when using artificial landmarks such as GPS satellites (see sect. 4.6).

In considering the global localization the biggest problem is the starting position. This can be everywhere on earth, or in other terms, the covariance of the starting position is nearly infinite. To obtain a global position there is the need of a recognition and an identification of a landmark, whose global position is known. This can be a static object, for example a big building, or a dynamic object whose temporary position is known. This is the case when we are looking at GPS satellites, which send signals with their time and an identification, so that the current position can be calculated. At least three landmarks are needed to get a position in three dimensional space.

When we consider local localization we assume that the global starting position is known or a completely new map is built, so that the starting position is unimportant because there are no other entries yet. In this case the origin is typically taken as the first position. Starting from here the trajectory is calculated. To do this odometry data or data of sensors for acceleration and gyroscopes are used.

3.10 Summary

SLAM is a well studied and yet still very active research field. It has many applications and much potential to be improved even further. A multitude of methods and variants thereof exist to choose the best option for any task and sensor type.

The extended Kalman filter and the Particle filter are of special interest to this work because they are the basis of the FastSLAM algorithm whose implementation is a central part of this thesis.

There is only little research in using GPS as a position estimate to create highly accurate maps, but instead most works try to replace GPS by SLAM. This thesis is a novel attempt to improve maps by assuming maps as input for a FastSLAM algorithm.

Lastly some important fields related to SLAM were introduced. The Relaxation technique introduced here will be dealt with later in this work.

Theoretical background

4

After giving an overview of the state of the art in SLAM research and introducing the basic concepts of the main techniques, this chapter gives a detailed introduction to the theoretical background needed to implement a FastSLAM algorithm for the 3D-HTRF project. The chapter starts with a close view on how the hardware used in the 3D-HTRF project works and point out the aspects important to this work. Because of the characteristic errors done by the Xsens MTi-G INS a part of this chapter is dedicated to the technique of relaxation as a measure to correct those errors. Strictly speaking this is not related to FastSLAM, but it can improve the quality of the generated maps and thus helps achieving the goals of this thesis. Then landmarks are introduced and discussed. Landmarks are the basis on which the matching process of the FastSLAM algorithm works. Lastly we go through a complete round of the FastSLAM algorithm, step by step.

4.1 Laser Range Finder (LRF)

A LRF is an instrument for measuring distances and angles with a laser whose beam is reflected by a rotating mirror onto an object and then received by a photo diode after being reflected back by the object hit. There are two possibilities to interpret the difference between the sent signal and the received one. Either the phase shift method or time of flight is used to calculate how far the laser beam traveled. LRFs usually work in a triangular plane and provide accurate distance and angular readings. These capabilities make LRFs an ideal choice for use cases where 2D is sufficient and can be extended with a rotating or a nodding mechanism for the 3D case.

4.1.1 Infrared light

The laser light used for laser range scanners usually is at a wavelength around 900nm (905nm for the LMS151[1]). This is well within the infrared spectrum of light which is invisible to the human eye, but close to the visible range and can be made visible

[1] `https://mysick.com/saqqara/get.aspx?id=im0026550` (May 12, 2011)

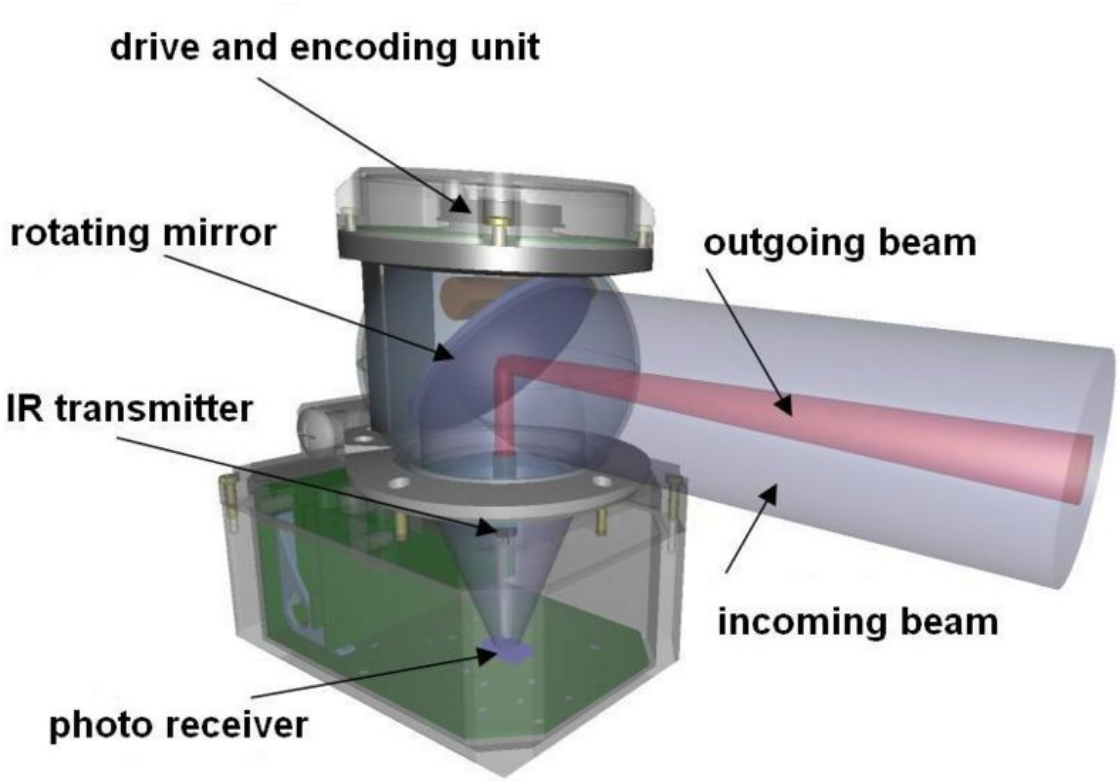


Figure 4.1: This schematic shows an Ibeo Alasca LRF and its inner workings. Ibeo was a subsidiary of SICK AG, the manufacturer of the LRFs used in this thesis. Ibeos Alasca model uses a very typical working principle shared by many LRFs.

with the help of night vision goggles, if necessary. Infrared light is behaving almost identical to visible light in terms of optics as in reflectivity and transparency of certain objects like glass, metal or stone. Due to these similarities to human vision, humans can understand and handle laser range finders intuitively. Generally a LRF can see as well and as much as the human eye, but with less detail due to the laser beams large diameter. Also infrared lasers are well established, cheap and available in high quality.

The scanners mostly belong to the group of class 1 lasers and do not harm or distract any human next to an operating laser range finder.

In the next two sections the two commonly used methods for determining the distance a laser pulse traveled are explained.

4.1.2 Time of flight

The obvious method is to measure the time light needs to travel from the emitting laser to the reflecting object and back to the receiving photo diode. Since the speed of light is constant one can calculate the distance **D** by

$$\mathbf{D} = \frac{\mathbf{c} \cdot \mathbf{t}}{2} \tag{4.1}$$

With **c** being the speed of light and **t** being the time the laser pulse needed to travel from its source to the photo diode.

For a good measurement resolution a good temporal resolution is required. For this method there is a need for frequencies in the GHz range which raises the costs for this technology. For example for resolution of $1cm$ 17GHz is necessary. Then again this technique is very robust and works very well on long distances.

4.1.3 Frequency phase-shift

The sent-out laser pulse's power is altered in a sinusoidally pattern and the simultaneously measured reflection is compared with the signal sent. The shift in the phase of this signal enables one to calculate the distance the signal traveled.

Let

f_0 be the modulation frequency

$\Delta\phi$ the phase-shift between sent and received signal and

D the distance between the LRF.

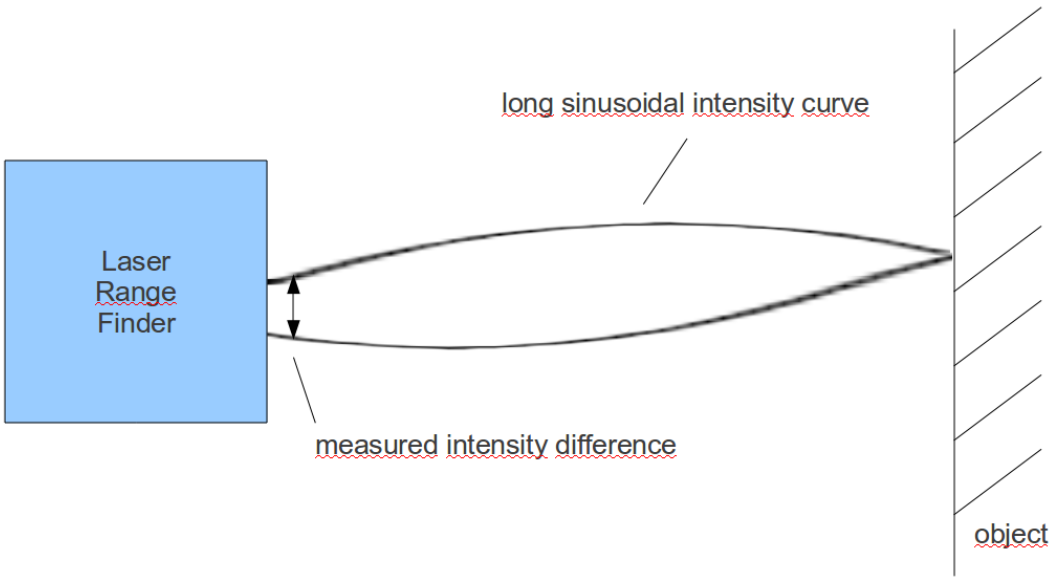


Figure 4.2: The phase-shift technique modifies the laser beams light intensity in a long, sinusoidal curve and measures the incoming laser beams intensity. From the difference the length of the phase-shift can be calculated and the distance to the object derived.

Then the distance to the object is found by this equation [NF06]:

$$\Delta\phi = 4\pi f_0 \frac{\mathbf{D}}{\mathbf{c}} \tag{4.2}$$

$$\mathbf{D} = \frac{\mathbf{c} \cdot \Delta\phi}{4\pi \cdot f_0} \tag{4.3}$$

The phase-shift can be derived from the difference of the outgoing and the received laser intensity as shown in fig. 4.2.

As can be seen in 4.3 the distance is calculated without involving the time. This makes the laser scanner using phase-shift cheaper than the ones using time of flight, because precisely measuring short time spans like the time light needs to travel to a close by object and back is expensive, as explained in the previous section. The first disadvantage is the restriction of range because of the period of the sinus function which only can be increased at the cost of accuracy. Some manufacturers try to circumvent this by using multiple frequencies. The second disadvantage is that sometimes the color and other reflection properties of the measured object have an effect on the reflected laser beams which affect the measurement range negatively.

4.1.4 Echo pulse width

When a laser beam is sent out the scanner records the light that is returning. The intensity of this light can be plotted over time. Each time this function crosses a threshold from low to high it is interpreted as a reflection of an object and a distance is calculated and returned. The echo pulse width of a measurement is the width of the intensity curve at the threshold level.

4.2 Inertial Navigation System (INS)

An INS is a system that estimates the position of an object to which it is attached. They are mainly used in aviation and composed of a GPS part and an inertial part which is called inertial measurement unit (IMU). As GPS part nearly every GPS correction standard (see sect. 4.2.2) is available even sometimes several standards are implemented and are unlocked according to customers request. The information from the IMU is combined with the GPS position for example with the help of a Kalman filter, but is mostly the secret of the INS manufacturing company.

4.2.1 Inertial Measurement Unit (IMU)

An IMU is a combination of three accelerometers and three gyroscopes to determine all 6 degrees of freedom of a 3d position. The accelerometers for the relative position (x, y, z) and the gyroscopes for the orientation (yaw, pitch, roll). A problem of the IMUs is that to get good results a very precise calibration is necessary. A good introduction to INSs is a Technical Report from Oliver J. Woodman [Woo07]. For detailed information on the INSs used see the hardware description in sect. 2.3.

4.2.2 Global Positioning System (GPS)

GPS is a position system based on satellites with atomic clocks, which send out their time and identification to the earth's surface. From the information of three or more satellites the position of the receiver can be calculated (see fig. 4.3. To increase the accuracy there are some services that focus on the problem that GPS has a lack of integrity. These services are processing the GPS data and calculate some correction signals. This signals can be received for example from the Wide Area Augmentation System (WAAS) or the European Geostationary Navigation Overlay Service (EGNOS). Some commercial systems are StarFire and OmniSTAR which provide different correction qualities at different costs. The errors that are corrected are for example some deflections in the atmosphere which depend on the weather and can be calculated because the position of the receiver is already known exactly. A problem that cannot be solved with this method is the so-called Manhattan effect.

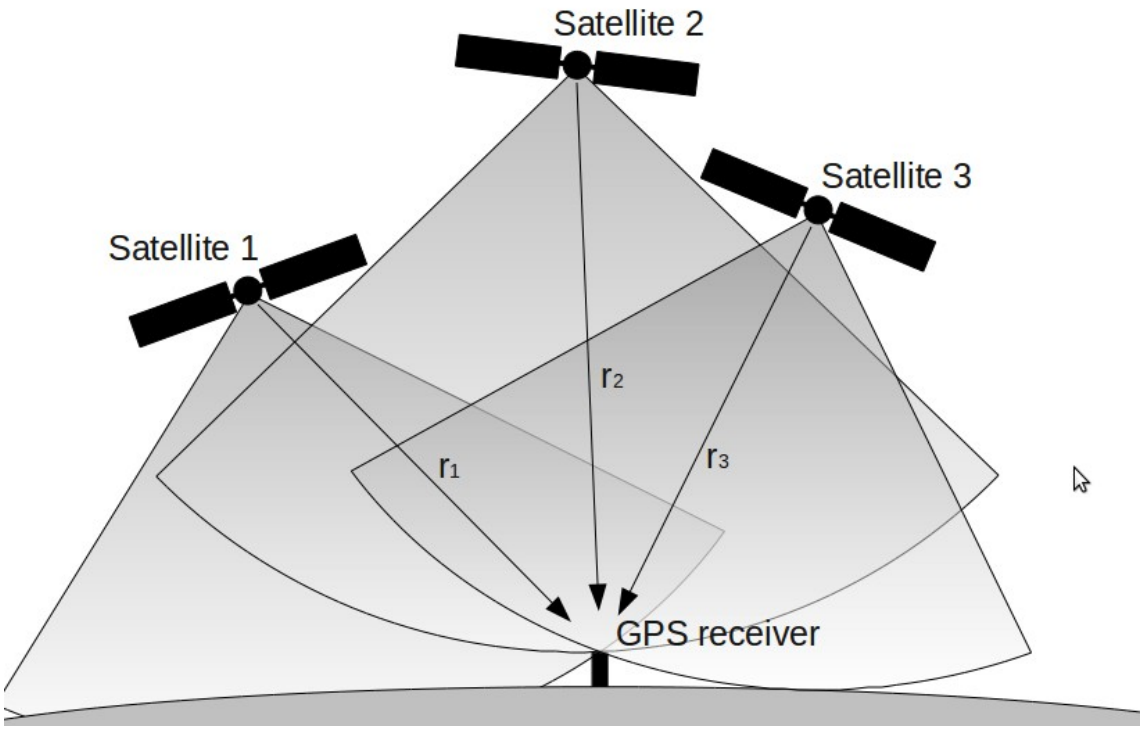


Figure 4.3: A GPS receiver on the earth's surface receives the signal from three GPS satellites and based on the send time the distances r_1 ,r_2 and r_3 can be calculated. In turn this information are sufficient to calculate GPS receivers position.

Streets in big cities naturally are surrounded by large buildings. These houses prevent the direct reception of a signal and instead reflections are picked up. The signal run time of reflections is obviously longer and so the calculated position is wrong. Addressed to this the position that is shown to the user is a combination of more then one measurement. The wrong signals are permanently reflected differently and thus have various running times. If a runtime occurs frequently, it is very likely the non-reflected signal. To support this sometimes several antennas are installed.

4.3 Quality of a map

Maps in this work describe simple point clouds derived from Laser Range Finder (LRF) readings. To better describe the quality of a map in this context the terms *accuracy* and *detail* are used.

accuracy describes how close a point representing a measurement of the LRF is placed at the position in the map where the measured object is in reality. This also includes distances and orientations between multiple points. The closer a point is to the position of the measured object, the higher is the accuracy.

The accuracy highly depends on the error of the sensor and the correction of movements while the sensor was measuring.

detail describes the number of measurements per surface area of objects. The more measurements are made on the same surface area of an object, the more features of this object are discernible and distinguishable. Detail basically refers to the point density of a point cloud.

Another very important aspect of mapping is the computation complexity, because the runtime of the mapping algorithms depends directly on it. Although for the 3D-HTRF project a calculation time of several days would be tolerable since scouting takes up to seven or ten days, saved calculation time can be spent on more precise algorithms.

4.4 Calculation complexity

Mapping methods for robots can be classified by many different criteria. The following four criteria are often used in the field of mapping and have a strong influence on the calculation complexity of mapping algorithms.

indoor vs. outdoor (e.g. [LSNH04]) Indoor as opposed to outdoor environments allow many assumptions which simplify a lot of robotic tasks. In indoor environments like office buildings, homes, factory buildings etc. the floor is generally flat and walls are straight vertically. For most projects so far the above assumptions were sufficient and a two dimensional representation of such areas was enough for most navigational and other robotic tasks. The robot does not need to know of the height of the ceiling or its own pitch angle because those variables never change.

On the contrary, in outdoor environments the floor often is uneven and thus a robot needs to be aware of its full pose and relative to that the positions of its sensors to properly make sense of its sensor readings. Outdoor environments often require three dimensional mapping and navigation. Also a lot of objects are moving, e.g. cars and people, or constantly changing like plant leafs in the wind, which demands algorithms capable of working with changing foreground and probabilistic movement of objects.

Outdoor environments tend to be larger and not to have distinct borders compared to indoor environments which usually are restricted to the inside of one particular building. It has not only to be resistant to dirt and water but because of this larger environment the range of the sensor has to be larger. Mostly in outdoor surroundings the noise is larger and changes because there are more external influences that are not constant. For example working with cameras is more complicated because the quality of the pictures depend on the

weather. When the sun is shining there could be big shadow or it is cloudy and less colors are recognized. Sensors that are not passive like laser scanners have an advantage here.

2D vs. 3D Naturally three dimensional mapping algorithms do require more sensory input than two dimensional ones and thus need more processing power. The mathematical background of most algorithms can be adjusted by extending the corresponding formula by another spatial dimension. Occasionally algorithms are designed specifically for two dimensional problems only and require a major overhaul to adjust them to 3D.

When working in outdoor environments using three dimensions is often required because in most cases the important assumption to have a flat floor for 2D processing is not given anymore.

The dimensionality of a map is not the same as the dimensionality of the mapping or localization problem to solve with the map. This depends on the degrees of freedom (DoF). A 2D problem usually has three DoF: a two dimensional position (x, y) and a rotation θ. For three dimensions this increases to six DoF which are (x, y, z)-coordinate and three angles for the orientation in space (ρ, θ, ϕ) referred to as yaw, pitch and roll. Combinations of 2D pose and 3D sensor input or the other way around are also possible, but this work's subject is the full six DoF problem. With every DoF the calculation complexity grows exponentially [BEL+08].

size of area to map The larger the generated maps get, the more memory and computing time is needed to store them and to add new measurements. Depending on the size of the maps they can contain different levels of detail starting from sparse maps only containing landmarks up to dense surface information. The operations of adding new measurements and retrieving information for e.g. loop closing restrict how large a map can grow if the computational complexity is too large [Kon04].

online vs. offline Online means that the mapping is done incrementally based on the prior sensor readings and in real-time. Such algorithms are needed when the robot has to make decisions based on its current situation and sensor readings, e.g. exploring an unknown territory. Offline methods are those which need all collected sensory data right from the beginning to create a map or are computationally so expensive that it is not possible to finish calculations before the next measurements are taken [GRS+08].

As a basis on which to adopt the FastSLAM algorithm, the 3D-HTRF project's existing hardware, software and its inner workings, capabilities and limits need to be explored and evaluated. For this reason an experiment has been conducted.

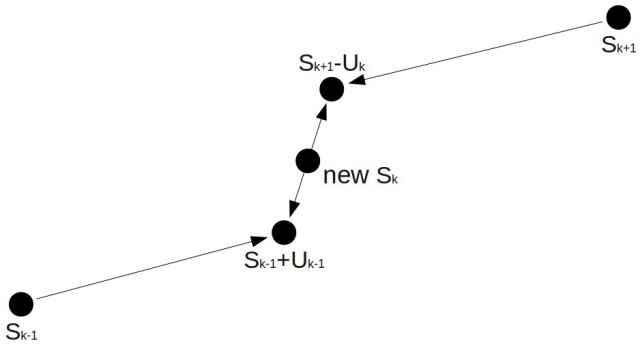


Figure 4.4: Relaxation computed out of the states s_{k+1}, s_{k-1} and the odometry data u_k, u_{k-1} a new position for s_k by building the center between the estimated position

4.5 Relaxation

Sometimes it seems that the trajectory is not smooth enough. It contains jumps that cannot be executed under normal circumstances by the vehicle. These occur in the system, if a measured GPS position is not equivalent to the approximate position that was calculated with help of the odometry data. This happens when the INS only trusts the GPS position and disregards the position calculated with the help of the IMU or when the track without a GPS receive was too long and the errors of odometry have accumulated too much. Then even a slower adaptation as done by a Kalman filter is clearly visible (see Discussion 6.4). Another reason for an uneven trajectory in the context of SLAM could be Loop-Closing (see 3.9.1). When doing Loop-Closing a completely new position is set as current position and there is no causal connection to its previous position anymore. To compensate for this and get a smooth trajectory and therefore a smooth map, there is an algorithm called relaxation.

Relaxation is based on the idea that each point of the trajectory has to be somewhere where it is supposed by its previous (s_{k-1}) and where its successor (s_{k+1}) suspects it came from. So the algorithm calculates with the help of u_k and u_{k-1} this two positions for s_k and computes the euclidean center.

$$s_k = \frac{1}{2} \cdot [(s_{k-1} + u_{k-1}) + (s_{k+1} - u_k)] \tag{4.4}$$

A graphical illustration can be seen in fig. 4.4. When doing relaxation this relaxation step must be run multiple times. How many times it has to be done depends on the quantity of the jump that has to be smoothened. Also the number of points that should be used to get a good result has to be adjusted to the size of the jump.

4.6 Landmarks

When doing localization landmarks are very important. The recognition and if possible identification of such elements are the basic for SLAM algorithms because they are needed as corresponding points to match the point clouds. A landmark in its original meaning is a place or object that can be used for orientation. The requirements for a good landmark are an easy recognition from many direction and a unique identification so that they can be used as corresponding points. Landmarks used in the context of SLAM are either artificial ones which mostly meet the definition or they are automatically generated. Most types of the latter do not fully meet the requirements. Especially a unique identification is usually not given, but with the knowledge about the movement there is a good chance to find the same landmarks again in the new sensor data. The generation of several landmarks and their relationship to each other can be used for matching. Another distinction of landmarks can be made if they are active or passive.

Artificial landmarks

Artificial landmarks are human made objects, which mostly are visible over long distances and can be identified. The most obvious examples are lighthouses. Their light is observable for ships over 20 km and more. Each Lighthouse has its own light characteristic which is listed in every nautical map. In the context of SLAM artificial landmarks are only used in very limited areas like the robocup as shown in fig. 4.5. Typically on streets the landmarks are signs with the number or name of a street and for bigger roads there are street signs that represent milestones, which both together is sufficient for a global localization. Unfortunately the laser scanners of 3D-HTRF look straight left and right. So the laser cannot see the fronts of the street signs. The front of a street sign is mostly made of a material which reflects light very good so that they would have a large echo pulse width. But even if we could recognize the street signs there is no way to read them.

Automatically generated landmarks

Automatically generated landmarks are characteristics of the measurement data. When using a 2D laser scanner there is a quite simple way to extract landmarks of borders of the foreground that masked something in the background (see sect. 6.1.3). This algorithm works quite good for indoor scenarios where the ground is flat and horizontal and every wall is vertical. For outdoor environments the elevation of the ground changes often and so the scanning beams hit the ground, or when hitting a mound a little bit higher or lower, the measured range varies a lot.

To avoid such problems one can work in 3D space. Of course this increases the complexity but to get undistorted 3D maps one needs to take care of all three

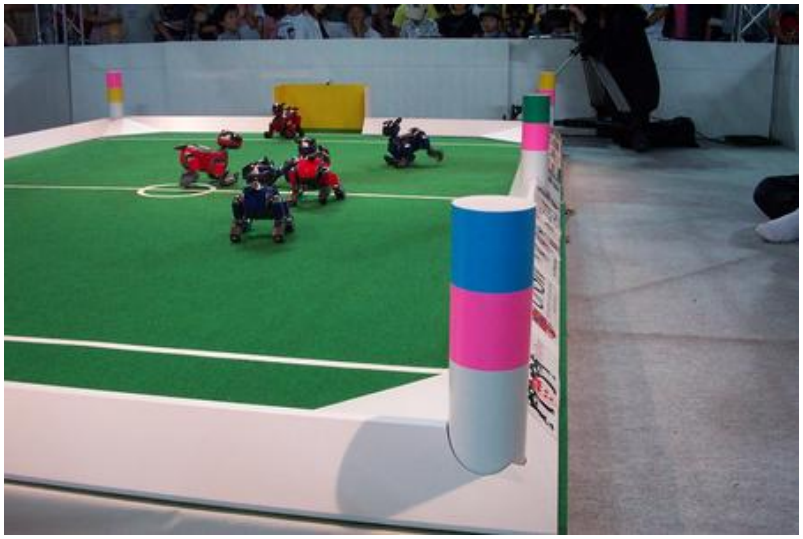


Figure 4.5: A robocup field with some artificial landmarks at the border of the field. Each landmark has its own color code for its identification

dimensions and even of all three directions of the vehicle with the scanners and therefore there is no way to avoid 3D landmarks. Extraction of 3D landmarks is far more difficult than in 2D, because the next points in line are not known in the third dimension. But the basic concept is the same: Find a Point which is not on a plane formed by four of its neighbors.

If there is some extra information it can be used, too. In our case the laser scanners provide a information called Echo pulse width. Extract the planes with the same value and find the corners of these plains by using for example a bounding cube (a bounding box in 3D). The Echo pulse width and the found planes can be combined and for example a landmark can be set on a border line of a 3D plane at the point where the value of the Echo pulse width changes.

Normally when using camera data the only information available is the color. Sometimes you known the size of an object and can guess the range to this object but when generating landmarks with help of the bounding box we only know the direction. Therefore the calculation of the correct landmark position is hard.

Active and passive landmarks

If a landmark is active or passive depends on if the landmark sends out a signal on its own or not. The lighthouse used as an example of as an artificial landmark in section 4.6 is an active landmark because it sends out a light signal all the time. The landmarks of the robocup field are passive. They have to be recognized by the players and the color code have to be extracted. Passive artificial landmarks for laser

Figure 4.6: A SICK laser scanner and a reflector-mark installed on a table leg for an easily recognition

scanners are reflectors which have a significant higher reflection which can be easily found in the measurements (see fig. 4.6).

Non artificial active landmarks are very rare and of course they do not really send out an identification signal. They are very special natural phenomena. This could be in the astronomical context a pulsar which is a star which shows a regular flicker at the night sky. Another example could be the magnetic north and south pole. A dissertation which addresses the classification of landmarks is written by Joachim Jotzo [Jot01].

4.7 FastSLAM

As pointed out in the State of the Art (see sect. 3.7) FastSLAM is a two-step algorithm using a particle filter for the position updates and a Kalman filter for the landmark updates. In this section we explain how exactly an ideal FastSLAM version 1.0 update step works for one observation each time step and how it can be adapted to work in three-dimensional environments. The variable names used (tbl. 4.1) are the same ones Montemerlo and Thrun use in their FastSLAM book [MTS07].

The algorithm starts with the particle set from the previous step in time S_{t-1}, the current observation z_t made by the vehicles sensor and the current vehicle control u_t. The vehicle measurement noise R_t usually does not change. For the first run a particle which holds the position (0,0) and and no landmarks can be added to the set of particles S_{t-1} to initialize the algorithm.

S_t	particle set at time t
z_t	sensor observation at time t
R_t	linearized vehicle measurement noise
u_t	robot control at time t
M	Total number of particles
N	Total number of landmarks
$\mu_{n,t}^{[m]}, \Sigma_{n,t}^{[m]}$	n-th landmark EKF (mean, covariance) in the m-th particle
θ_{n_t}	angle from the vehicle to the n_t-th landmark
$g\left(s_t, \theta_{n_t}\right)$	measurement function
G_θ	Jacobian of measurement model with respect to landmark pose
$\hat{z}_{n_t}$	expected measurement of n_t-th landmark
$z_t - \hat{z}_{n_t}$	measurement innovation
Z_t	innovation covariance matrix
K_t	Kalman gain
$w_t^{[m]}$	importance weight of the m-th particle

Table 4.1: Variable names and their meanings as described by Montemerlo and Thrun. For this work the same conventions are used.

FastSLAM(S_{t-1}, z_t, R_t, u_t)
$S_t = S_{aux} = \emptyset$

Every particle is processed the same way in FastSLAM, so the main part of the algorithm is a for-loop going through all particles.

4.7.1 Particle

The m-th particle is defined as a combination of the complete path s^t up to the current vehicle pose at time t of this particle and a list of all N landmarks associated with it:

$$S_t^{[m]} = \left\langle s^{t,[m]}, \mu_{1,t}^{[m]}, \Sigma_{1,t}^{[m]}, \ldots, \mu_{N,t}^{[m]}, \Sigma_{N,t}^{[m]} \right\rangle \tag{4.5}$$

Processing the m-th particle starts with drawing a sample from it after selecting it.

for $m = 1$ to M
  retrieve m-th particle // loop: all particles
    $\left\langle s_{t-1}^{[m]}, N_{t-1}^{[m]}, \mu_{1,t-1}^{[m]}, \Sigma_{1,t-1}^{[m]}, \ldots, \mu_{N_{t-1}^{[m]},t-1}^{[m]}, \Sigma_{N_{t-1}^{[m]},t-1}^{[m]} \right\rangle$ from S_{t-1}
  draw $s_t^{[m]} \sim p\left(s_t \middle| s_{t-1}^{[m]}, u_t\right)$ // sample new pose

A sample is a probabilistic prediction of the vehicle position of a particle.

4.7.2 Predicted particle pose

A predicted pose is a possible pose of the vehicle in a particle. This pose is estimated from the old pose saved in the particle and a movement vector (control u_t). The movement vector is the difference between the last two odometry poses transformed to polar coordinate system with the old vehicle pose as origin and the alignment as x-axis. The transformation is necessary because each particle has its own alignment and this is the start alignment of the movement vector. After calculating the new pose the new alignment has to be adapted. Each angle is the old one from the particle plus the relative one from the odometry data. The movement vector as well as the relative angles are measurements and interpreted as normal function. The used values are produced from the corresponding function with the help of a random generator (see sect. 5.4.6). The variance of this function depends on the used system, is usually unknown and has to be experimentally determined. This is done in [Eis02]. The variances necessary for moving in three-dimensional space are listed in tbl. 4.2.

α_{yaw1}	variance of the yaw angle of the movement vector
α_{pitch1}	variance of the pitch angle of the movement vector
α_{trans}	variance of the magnitude of the movement vector
α_{yaw2}	variance of the relative yaw angle
α_{pitch2}	variance of the relative pitch angle
α_{roll}	variance of the relative roll angle

Table 4.2: The variances for a 3D motion model.

After the particle has been adjusted for the vehicle movement a for-loop goes over all landmarks saved in the particle.

Montemerlo and Thrun do not delve into how to detect landmarks and assume landmark detection to work. This is justified because the landmark detection is highly depending on the type of sensor data and the FastSLAM description is supposed to be as independent from the type of sensor as possible. So the landmark detection is an individual part of every implementation and will not be handled any further in this chapter, but in the following chapter (see sect. 5.3.2).

4.7.3 Observation

An observation is a landmark found in the sensor data. Those observations get classified either as a new observation or as a match with a known and saved landmark. If an observation is regarded as a previously unseen, new landmark it will be saved for future processing. Otherwise it is considered a repeated detection of a previous landmark it got associated with. In this case the information of the observation is

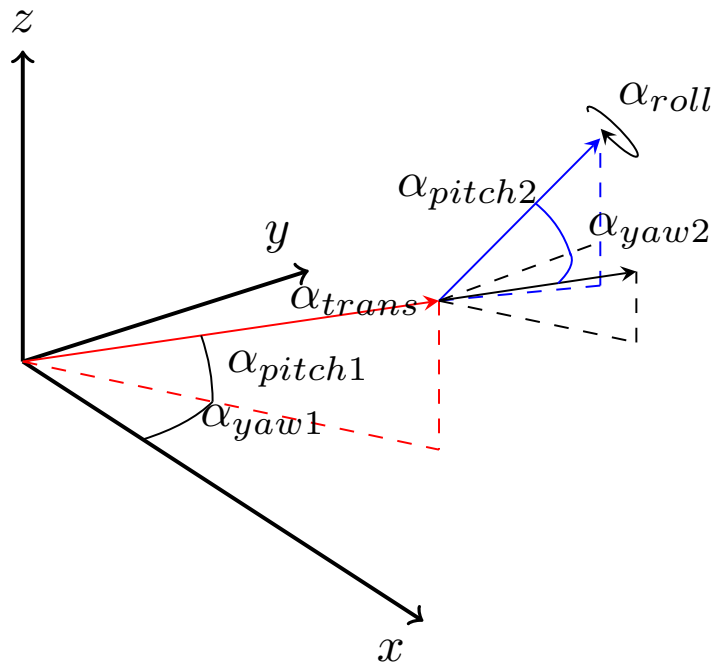


Figure 4.7: An extension of the FastSLAM motion model from 2D to 3D. Additionally to the α_{yaw1}, α_{trans} and α_{yaw2} parameters, α_{pitch1}, α_{pitch2} and α_{roll} are added.

used to either update the robot pose, the position of the landmark it got associated with or both.

4.7.4 Landmark associations

The landmark association is a difficult problem. As long as no other features of the landmark are known the association can only be done by its locality. On how to detect identifiable features, take a look in sect. 7.2.1. The simplest method to find the right association between landmarks and observations is to take the observation and landmark with the smallest distance. Generally the choice of the metric is left to the developer. Of course the most common one is the Euclidean distance, but, e.g. to improve the performance, one can use the Manhattan distance if one is not affected by its disadvantages. In our case we decided to use a proposal from the book “Probabilistic Robotics” [TBF05] and use the maximum likelihood metric for the landmark association, which just associates the observation with the landmark with the highest likelihood value. This is illustrated in fig. 4.8 where the likelihood of Landmark 2 is larger. Another approach is to interpret the likelihood values as a probabilistic distribution so that the landmark to associate an observation with is randomly selected based on this distribution.

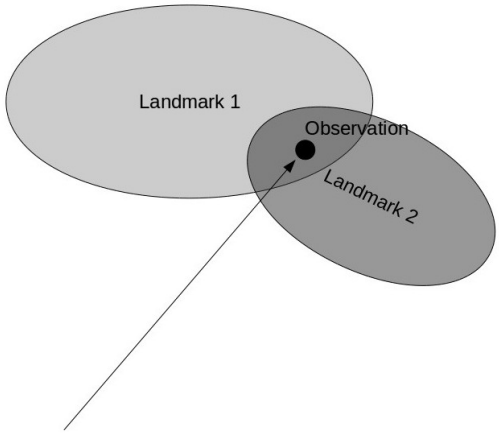


Figure 4.8: An ambiguous data association which could be solved in favor of landmark 2 by the calculation of the likelihood. The landmarks are diagrammed by a ellipsoid represents its probability distribution.

4.7.5 Likelihood

Likelihood is a term used in probability theory. It is used for example in the maximum likelihood estimation which estimates the parameters of a distribution only based on the measurements. The likelihood itself is the probability of the given measurement when the distribution has a certain parameter. The likelihood function has the distributions as domain and the probabilities as image but these are not normalized.

In our case the likelihood calculation is just one value because our landmark model already sets the mean and the covariance but there are more than one possible landmark for a observation. So we have to calculate the likelihood for each landmark with the help of the probability density function which is a multivariate normal distribution and describes the probability that the observation is a random value of the landmark and its noise.

$$p_{n,t}^{[m]} = \frac{1}{(2\pi)^{\frac{p}{2}} |Z_{n,t}|^{\frac{1}{2}}} \exp\left\{-\frac{1}{2}\left(z_t - \hat{z}_{n,t}\right)^T Z_{n,t}^{-1}\left(z_t - \hat{z}_{n,t}\right)\right\} \tag{4.6}$$

p denotes the dimension of the distribution, which is 3 in case of this thesis.

The likelihoods of all possible landmark associations are calculated and then the landmark with the highest likelihood is associated with the current observation z_t. This association is called n_t.

Likelihood values are also useful for deciding if the current observation is a new landmark. If the likelihood values of all landmark associations are below an empirically determined threshold value the observation is considered more likely to be of a new

landmark instead of being associated with a previously observed landmark. In this case a new landmark is created and stored in the current particle like this:

$$\mu_{\hat{n}_t,t}^{[m]} = g^{-1}\left(s_t^{[m]}, \hat{z}_{\hat{n}_t,t}\right)$$
$$\Sigma_{\hat{n}_t,t}^{[m]} = \left(G_{\theta,\hat{n}_t}^T R^{-1} G_{\theta,\hat{n}_t}\right)^{-1}$$

In the other case, when a landmark association exists, the landmark position gets updated. An extended Kalman filter is used to find the most likely position and covariance of the landmark factoring in its previous position and covariance and the associated, new observation including its covariance.

4.7.6 Extended Kalman filter

An observation received as sensor data from a Laser Range Finder is a point in spheric coordinate space, described by a horizontal and a vertical angle and a distance with the sensor being the coordinate system's origin. Conversely the maps built are described in Cartesian coordinates with a global origin.

In the context of this thesis the Jacobian Matrix is used to transform coordinates from the spheric coordinate system of the vehicle's sensor to the cartesian coordinate system of the map and back again.

Jacobian Matrix

The Jacobian Matrix is a matrix of all first-order partial derivatives of a vector function. It describes the orientation of a tangent plane on this function and thus represents a the best linear approximation of the function at a given point. The Jacobian Matrix is the important extension to a Kalman filter which makes it an extended Kalman filter.

The advantage of a Jacobian Matrix over simply transforming all coordinates accordingly is that it can be used in matrix calculations and allows for an convenient, efficient and save implementation.

Since this work is handling FastSLAM in 3D instead of 2D as the original formulation of the algorithm, this section describes an extension of the Jacobian Matrix to three-dimensional space.

To simplify the following formulas these definitions are used for x, y and z:

$$x = \theta_x - s_{t,x} \ ; \ y = \theta_y - s_{t,y} \ ; \ z = \theta_z - s_{t,z}$$

g is the measurement function which describes the coordinate transformation from spheric coordinate space to Cartesian coordinate space.

$$g(s_t, \theta_{n_t}) = \begin{bmatrix} \rho \\ \theta \\ \phi \end{bmatrix} = \begin{bmatrix} \sqrt{x^2+y^2+z^2} \\ \arccos\left(\frac{z}{\sqrt{x^2+y^2+z^2}}\right) \\ \arctan\left(\frac{y}{x}\right) - s_{t,\theta} \end{bmatrix} \tag{4.7}$$

Based on aboves measurement function the partial derivations of its components yield this Jacobian Matrix:

$$J = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2+z^2}} & \frac{y}{\sqrt{x^2+y^2+z^2}} & \frac{z}{\sqrt{x^2+y^2+z^2}} \\ -\frac{y}{x^2+y^2} & \frac{x}{x^2+y^2} & 0 \\ \frac{x \cdot z}{\sqrt{x^2+y^2} \cdot (x^2+y^2+z^2)} & \frac{y \cdot z}{\sqrt{x^2+y^2} \cdot (x^2+y^2+z^2)} & -\frac{x^2+y^2}{x^2+y^2+z^2} \end{bmatrix} \tag{4.8}$$

This Jacobian Matrix is an integral part of the extended Kalman filter and needed for Covariance Matrices, Innovation Matrices and the Kalman Gain for updating landmark positions.

Covariance Matrix

With the help of the Jacobian matrix and the knowledge of the errors of the scanner, that are given in angle errors and distance error, the covariance matrix can be calculated by the extended Kalman filter.

The covariance is a term of probability theory showing how much two or more random variables change together. The definition of the covariance is

$$\mathrm{Cov}(X, Y) := \mathrm{E}\big((X - \mathrm{E}(X))(Y - \mathrm{E}(Y))\big) \tag{4.9}$$

where X and Y are random variables and the $E(X)$ is the expected value of X. Each value thus represents the link between the different variables. This dependence is commutative. So the values over the diagonal correspond to those under it. The interpretation of the diagonal itself is simple. These values represent the variance σ of the probability distribution to its random variable.

When working with FastSLAM every landmark in the map has a covariance. The covariance represents the belief in the accuracy of the calculated position. Each time the landmark is detected the belief increases and the covariances decrease. An observation has always the same covariance given by the measurement inaccuracies of the system. This is given by the measurement noise matrix R_t.

Kalman Update

Using the same measurement function $g(s_t, \theta_{n_t})$ the Jacobian Matrix is based on (sect. 4.7.6) the current observation z_t is transformed to the expected observation's position $\hat{z}_t$

$$\hat{z}_t = g(s_t, \mu_{n_t,t-1}) \tag{4.10}$$

Using the Jacobian Matrix $G_{\theta_{n_t}}$ the Innovation Covariance Matrix $Z_{n,t}$ can be calculated

$$Z_{n,t} = G_{\theta_{n_t}} \Sigma_{n_t,t-1} G_{\theta_{n_t}}^T + R_t \tag{4.11}$$

which in return is used to calculate the Kalman Gain K_t

$$K_t = \Sigma_{n_t,t-1} G_{\theta_{n_t}}^T Z_{n,t}^{-1} \tag{4.12}$$

The difference between the observation and its expected position $z_t - \hat{z}_t$ is called Innovation. The Kalman Gain K_t describes how much of the innovation is most likely gained according to the extended Kalman filter with the given vehicle measurement noise R_t. This leads to the updated landmark position $\mu n_t, t$

$$\mu_{n_t,t} = \mu_{n_t,t-1} + K_t(z_t - \hat{z}_t) \tag{4.13}$$

The update for the landmark's covariance $\Sigma_{n_t,t}$ works similar

$$\Sigma_{n_t,t} = \left(I - K_t G_{\theta_{n_t}}\right) \Sigma_{n_t,t-1} \tag{4.14}$$

All the updated landmarks $(\mu_{n_t,t}, \Sigma_{n_t,t})$ are saved in the particle they belong to for the next round of FastSLAM. This concludes the Kalman Update step of the algorithm and the updated particles are rated.

4.7.7 Particle Filter

The particle filter is the part of the FastSLAM algorithm that enables it to follow multiple hypotheses at the same time and avoid deciding on one hypothesis only. Each particle is processed in the above described manner independently and holds a vehicle pose and a complete map. Then all particles get rated and the good ones are kept and the bad ones are removed.

Importance Weights

This step is a critical correction step in the FastSLAM algorithm. After sampling new poses for particles in the beginning of the algorithm (see sect. 4.7.2), the particles are not distributed according to the desired distribution. This is because the particle poses s^t are sampled based on the latest control information u^t, but are missing the latest observations z^t and their landmark associations n^t at this point. So the posterior sampled from is $p(s^t|z^{t-1}, u^t, n^{t-1})$ instead of the desired $p(s^t|z^t, u^t, n^t)$ including the latest information available.

The particle's proposal distribution after sampling new poses is a Gaussian due to the way the particles poses are predicted from the control information. But the target distribution is determined by the observations and the landmark associations and thus can be arbitrary. The importance weights are higher for particles in regions where the target distribution is larger than the Gaussian proposal distribution and vice versa, so the importance weights of the particles approximate the target distribution.

To calculate an importance weight $w_t^{[m]}$ we start with the proposal and the target distribution:

$$w_t^{[m]} = \frac{targetdistribution}{proposaldistribution} \tag{4.15}$$

$$= \frac{p(s^t|z^t, u^t, n^t)}{p(s^t|z^{t-1}, u^t, n^{t-1})} \tag{4.16}$$

Using Bayes' theorem

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{4.17}$$

but dropping the normalization constant $P(B)$ because the importance weights are normalized in a separate normalization step, above formula can be rearranged to

$$w_t^{[m]} \propto \frac{p(z^t|s^t, z^{t-1}, u^t, n^t) \cdot p(s^{t-1}|z^t, u^t, n^t)}{p(s^t|z^{t-1}, u^t, n^{t-1})} \tag{4.18}$$

Since the particle filter is a Markov process (compare sect. 3.6) and thus the latest association n_t depends on the current observation z_t, which is not part of the right hand term in the numerator, n_t can be dropped from it.

$$w_t^{[m]} = \frac{p(z^t|s^t, z^{t-1}, u^t, n^t) \cdot p(s^{t-1}|z^t, u^t, n^{t-1})}{p(s^t|z^{t-1}, u^t, n^{t-1})} \tag{4.19}$$

$$= p(z^t|s^t, z^{t-1}, u^t, n^t) \tag{4.20}$$

This means the importance weight is equal to the likelihood of the current observation under the current information of control, previous observations, trajectory and landmark associations.

To extend this definition for more than one observation in Thrun's book "Probabilistic Robotics" [TBF05] the likelihoods of all landmark associations are simply multiplied with each others.

Resampling

Based on the importance weights assigned to the particles, in the resampling step the best particles are chosen and duplicated several times. At the same time the worst rated particles are discarded to keep the total number of particles stable.

Montemerlo and Thrun in their book [MTS07] did not specify which method to use for the resampling step. The important aspect to take make sure of when selecting a suitable algorithm is to sample the particles proportional to the importance weights. However the reader is advised to Madow's systematic sampling algorithm which is used in the implementation (see sect. 5.4.5).

This concludes a run of the FastSLAM algorithm. As soon as all necessary data (vehicle control and observation) is available again the next round of the FastSLAM algorithm can be started.

4.8 Summary

Laser Range Finders are Lidar devices which work with laser beams close to the visible spectrum of light. This is advantageous because the measurements are similar to a human understanding. The different techniques of time-of-flight and phase-shift allow for precise measurements of close by as well as far away objects.

With the help of an Inertial Navigation System the global pose and movement of a vehicle can be determined quite precisely due to a combination of sensors measuring the vehicle movement and the Global Positioning System. The different information sources are joined with probabilistic methods to calculate the position with the highest probability.

When an INS has no GPS signal for a longer while like in a tunnel, it has to solely rely on its dead-reckoning capabilities. Since errors add up and cannot be corrected with this technique the pose estimation goes worse over time and when a new GPS pose is retrieved the INS tend to do very abrupt corrections. Relaxation was explained and introduced to smoothen out such errors.

Landmarks are the basis for most matching algorithms which is why a section of this chapter explained what landmarks are and how they can be classified.

With the basic understanding how the FastSLAM algorithm works one can implement it. Although the theoretical description by Thrun leaves some points open and for the reader to decide, the base frame is outlined clearly. In the next chapter a full FastSLAM 1.0 algorithm is implemented and the algorithms used to complement the base frame given by Thrun are explained in detail.

Implementation

5

With the knowledge about the base frame of the FastSLAM algorithm from the previous chapter an implementation was written. Many changes had to be made to adapt the algorithm for the 3D-HTRF project and fill in the open issues not covered by the general description of FastSLAM. Because several components interact to form the whole 3D-HTRF system, this chapter starts with an introduction of the different coordinate systems used by the components.

5.1 Coordinate Systems

All the components of the system, built as part of this thesis, deal with points and angles in three-dimensional coordinate systems, but many different coordinate systems are involved. This section gives an overview which coordinate systems these are and why they are used in which component.

The most important aspect of the different coordinate systems used is the orientation of the axes. Unfortunately all components using Cartesian coordinates use different axes and need a conversion. Starting from the East-North-Up coordinates provided by the INSs, over the AppBase down to the OpenGL engine used for displaying the maps in the end. Additionally the angles for describing a vehicle orientation in space are defined differently as well.

5.1.1 World Geodetic System 1984 (WGS84)

WGS84 is the reference coordinate system used for GPS. It models an ellipsoid representing the earth's surface with its origin in the mass center of the earth. Positions on the surface are given by longitude and latitude in degree and minutes. This coordinate system was developed by the United States Department of Defense beginning in the 1950s and consequently updated since then using data about the shape and density of earth as it became available (see e.g. [Sep74] or [MSWS02]). Since it is the reference coordinate system of the GPS system the raw position data from the INSs is provided in degree and minutes of latitude and longitude and

altitude in meters. For ease of implementation and interpreting data the 3D-HTRF project benefits from using a Cartesian coordinate system based on a flat plane instead of a ellipsoid, though. For this reason the AppBase transparently converts the WGS84 coordinates to East-North-Up coordinates.

East-North-Up (ENU)

East-North-Up is the most intuitive coordinate system. It is formed by placing a tangent plane at earth's surface, fixed to a specific location, being the ENU's origin. This is why ENU sometimes is called "Local Tangent" or "Local Geodetic". The axes then correspond to the AppBase's implementation of the standard map with north pointing up, which is the y-axis and east pointing to the right, which corresponds to the x-axis. The z-axis is pointing up from earth's center, which also is equal to the general use.

For the purposes of the 3D-HTRF project, especially the local accuracy (compare sect. 1.1), ENU is sufficient and tremendously simplifies working with global coordinates. The AppBase sets the coordinate origin at the position of the vehicle when the system is turned on. This way only changes in the position have to be considered and all data can be handled the same way independently on where on earth's surface it was collected.

Although the coordinate systems of the AppBase and the data coming from the INSs are the same, the angles used to describe the vehicle's orientation are different. The data coming from the INSs must be corrected by turning the yaw-angle by 90°, respectively $\frac{\pi}{2}$ radians, because a yaw-angle of 0 is rotated between the AppBase and the INSs.

```
1  MountingPosition SliceWorker::getRelativeMountingPositionWGS(PositionWGS84 currentWGS84,
       PositionWGS84 originWGS84)
2  {
3    Point3D currentRelWGS84;
4    MountingPosition vehicleMountingPosition;
5
6    currentRelWGS84 = currentWGS84.getCartesianRelPos(originWGS84);
7    currentRelWGS84.rotateAroundZ(-originWGS84.getYawAngleInRad()-ibeo::PI_double/2);
8    currentRelWGS84.rotateAroundY(-originWGS84.getPitchAngleInRad());
9    currentRelWGS84.rotateAroundX(-originWGS84.getRollAngleInRad());
10   vehicleMountingPosition = MountingPosition(currentWGS84.getYawAngleInRad()-originWGS84.
         getYawAngleInRad(),currentWGS84.getPitchAngleInRad()-originWGS84.getPitchAngleInRad(),
         currentWGS84.getRollAngleInRad()-originWGS84.getRollAngleInRad(),currentRelWGS84.getX(),
         currentRelWGS84.getY(),currentRelWGS84.getZ());
11
12   return vehicleMountingPosition;
13 }
```

See line 7 for correcting the yaw-angle by $\frac{\pi}{2}$ radians.

OpenGL coordinate system

OpenGL is an API for describing 3D graphics and is used for displaying point clouds. The coordinate system used in OpenGL is again different from the AppBase and INSs. The x-axis is pointing east, the y-axis upwards and the z-axis south, looking at the user in front of a computer. This makes several translations necessary.

```
1    sscanf(line.c_str(), "%f %f %f %f %f %f", &X, &Y, &Z, &R, &G, &B);
2
3    ScanPointCoord[i]=X;
4    ScanPointCoord[i+1]=Z;
5    ScanPointCoord[i+2]=-Y;
6
7    ScanPointColor[i]=R;
8    ScanPointColor[i+1]=G;
9    ScanPointColor[i+2]=B;
```

The z- and y-axis are switched around (lines 4 and 5) and the direction of the z-axis is inversed (line 5).

Since no orientation is modeled within the OpenGL parts of the software, the corrections for angles can be dropped.

5.2 Program layout

The AppBase is a closed source framework by SICK AG and used for processing data from different sources through units called workers. Workers can be connected by paths which transport certain types of data from and to the workers. Data is emitted at a source object and transmitted along the paths until it is sent to the drain object. The path used for this thesis can be seen in figure 5.1. When the AppBase is done sending the data from the source, it starts shutting down all of its components. This very serial processing paradigm leads to a problem when trying to implement FastSLAM within the AppBase. Particle weights in FastSLAM might change later on and later data might suggest a different hypothesis than before, which is exactly what FastSLAM is supposed to do. The best solution would be to add a routine to the destructors of the workers to tell them to write out their data now. Unfortunately this does not work with the AppBase because the order in which the destructors are called is not predictable. If the current state of the FastSLAM processing is forwarded continuously, as a serial design would assume, changing to a different hypothesis is only possible for the following scans, but not the ones already forwarded. The only feasible way to circumvent this is to write out all data up to the current state in regular intervals. In the worst case the data of a complete interval is lost in the end because it did not get written out before the AppBase shut down. Just making the input data a little longer than actually needed makes this data loss irrelevant. Since the data sent to the AppBase drain is written to a binary stream format which would require postprocessing to separate

the different FastSLAM stages again, the data is written directly to the harddrive by the FastSLAM worker, circumventing the AppBase mechanisms.

5.2.1 AppBase configuration

A XML configuration file describes which workers to use and how the data flows between them. The XML configuration file used for this work is schematically represented by the flowchart of fig. 5.1 and can be found in Appendix B.

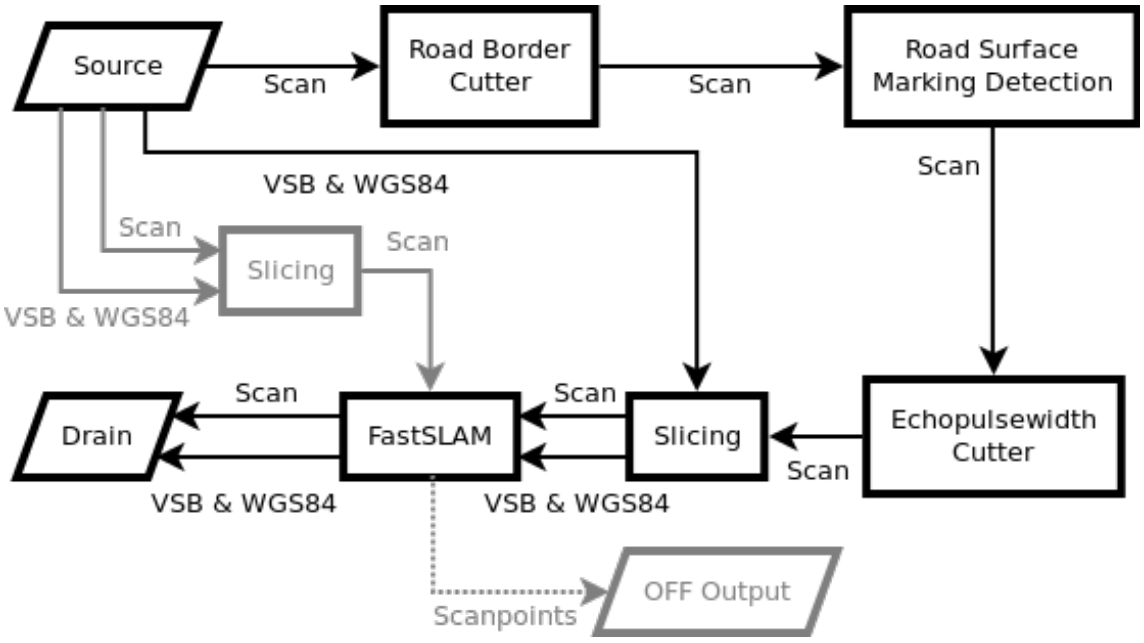


Figure 5.1: This flowchart shows the layout of the implemented AppBase workers and the data paths between them. For output purposes the data is routed along the grey paths, but this does not have any impact on the FastSLAM algorithm. The OFF output is implemented within the FastSLAM worker, but shown here for completeness. VSB stands for VehicleStateBasic which is an object of the AppBase API holding the dead-reckoning data from the car's CAN-bus, while WGS84 objects hold the position data from the INSs in longitude and latitude format.

The names on the arrows are data types used in the AppBase.

5.2.2 AppBase data types

These data types are mostly list objects with some limitations which makes it sometimes hard to work with them.

Scan A Scan object holds all information of a scan sweep of all LRFs. This includes a complete list of all scan points, IDs to tell the LRFs apart and the mounting

position of the LRFs. The maximum number of ScanPoints a Scan can hold is 65,536.

ScanPoint A single measurement from an LRF. Most importantly this object has a position (in AppBase terms a 'point'), an echo pulse width and an ID which tells which LRF it was measured with.

PositionWGS84 These objects are a encapsulation of a 3D pose (in AppBase terms a 'position'), this is a position and an orientation in three-dimensional space.

VehicleStateBasic VSB objects are similar to PositionWGS84 objects as they too contain position information, but the positions in these objects are not retrieved from an INS. The position and orientation data in these objects are calculated through dead-reckoning with the odometry provided by the motion sensors connected to the cars CAN-bus.

These objects are only saved in a serialized binary format by the AppBase. To get the information in those objects part of the SLAMWorker takes care of writing it out in the Object File Format.

5.2.3 Object File Format (OFF)

The Geomview Object File Format[1] is a very simple ASCII file format. All we use from the specifications is the definition of vertices and colors, although the colors are only for visual appearance. An OFF file has a header containing the word 'OFF' in the first line and the number of vertices, faces and edges on the second line, separated by spaces. Each line then describes a vertex with its X, Y and Z coordinates and its color in RGBA floatingpoint values from 0 to 1 (e.g. like in tbl. 5.1).

```
1  OFF
2  2 0 0
3  10.0 20.0 30.0 1.0 0.0 0.0 1.0
4  -1.0 50.0 50.0 0.0 1.0 0.0 0.5
```

Table 5.1: An OFF file describing two points. The first point is at (10, 20, 30) and solid red. The second point is at (-1, 50, 50) and 50% transparent green.

5.2.4 OFF file viewer

In order to watch the OFF files a viewer is needed. No suitable software was found and thus a minimal viewer using OpenGL was implemented.

[1] `http://people.sc.fsu.edu/~jburkardt/data/off/off.html` (May 9, 2011)

The basic idea is to read the OFF file line by line and toss out the two header lines. Then every following line is a vertex which needs only to be split up into its components and assigned to the corresponding variables.

Two arrays represent the points. ScanPointCoord holds the X, Y and Z coordinates consecutively and ScanPointColor holds the RGB portion of the color information consecutively (tbl. 5.2). The transparency information A as specified in the OFF definition (sect. 5.2.3) does not have any correspondence to scan point information and thus has not been implemented.

```
1  std::ifstream in(file);
2  string line;
3
4  getline(in, line);
5  getline(in, line);
6
7  float X, Y, Z, R, G, B;
8  while (! in.eof())
9  {
10   getline(in, line);
11   sscanf(line.c_str(), "%f %f %f %f %f %f", &X, &Y, &Z, &R, &G, &B);
12
13   ScanPointCoord[i]=X;
14   ScanPointCoord[i+1]=Z;  // adjust coordinate system to OpenGL
15   ScanPointCoord[i+2]=-Y; // adjust coordinate system to OpenGL
16
17   ScanPointColor[i]=R;
18   ScanPointColor[i+1]=G;
19   ScanPointColor[i+2]=B;
20   [...]
21 }
```

Table 5.2: Shortened method for reading an OFF file line by line. The Cartesian coordinates and RGB color information is stored in arrays. Transparency information is ignored.

5.3 Preprocessing

FastSLAM is a consuming process whose complexity grows with the number of landmarks. Therefore, the number of landmarks is reduced by some preprocessing steps. Because the echo pulse width is involved, recognizable landmarks should be possible to produced. A further step of preprocessing is slicing. Slicing joins multiple scans to a larger 3D point cloud to make them more useful for our 3D FastSlam algorithm.

5.3.1 Street border cutter

Most streets are used by a heavy transport are bigger ones which nearly always have curbs or a kind of traffic barriers. Therefore the best way to detect the street surface is to detect the different heights of the surface and the border. We first assume that the ground the vehicle stands on is a part of a street. The scan points which are in an area around zero height are cut out to avoid points which definitely not belong to the ground, i.e. trees. Then they are sorted by the distance to the vehicle, separately on each side. Then based on the first points next to the car a tangent is calculated and used to determine the deviation of the next scan point to this tangent and to decide if it belongs to the street or not. If it belongs to the street a new tangent is calculated. This is necessary because most streets are build with a convex surface to give the rain a chance to flow off the street. If the deviation is greater than a given value it must be the border of the street or an object that blocked the view on the roadside. In both cases we are not interested in the points that follow in this direction and they will be ignored.

5.3.2 Street surface marking detector

Street markings are markings on the pavement of a street which signify and separate the lanes to use by the drivers. They are made of a bright material to set them as much apart as possible from the dark pavement and additionally they have some special additives like light reflecting beads. This makes street surface markings quite good objects to create landmarks from. They are found on almost all streets, especially on the bigger ones on which a heavy transport mostly moves. The markings can be easily recognized in the laser scanner data because of their light reflecting materials. Of course this is primarily for better visibility for the human drivers of the cars but this has also advantages for scanning with laser range finders. The reflected laser pulse has a wide echo pulse width and the street surface itself has a low reflectivity in the light spectrum of the Laser Range Finder. Therefore a significant saltus is detectable when the laser beam crosses the border between the pavement and the marking and the other way around. So we take a look on the difference of the echo width of two neighboring points, which is almost the same as the calculation of the first derivative, and cut out the points where the difference to the next point is higher than a given value. This is true for the boundaries of the road markings as can be seen in figure 5.2. The detected echo pulse width do not only depend on the different physical properties of the reflecting object, but also on the angle at which the object was hit and the distance the light traveled. This is why the derivative of the echo pulse width will yield different values for the same objects depending on the distance and orientation to the car's Laser Range Finders. In general the echo pulse width is dropping with the distance and a decreasing angle of incidence. Thus the implementation can also calculate the second derivative which is more stable over

distance and yields similar results for the same objects at different distances and angles. For the most cases the interesting landmarks are found on the street's surface right beneath the car and thus it is configurable if the first or second derivative is used.

```
1  for (std::vector<Scan>::iterator scanIt = allScans.begin(); scanIt != allScans.end(); ++scanIt)
2  {
3    // Vector containing the recent ScanPoints EchoWidths
4    std::vector<float> recentSPs;
5
6    // initialize iterator over current Scan object to fill
7    it = (*scanIt).begin();
8
9    // Add three Scanpoints to the recentSPs-vector to prepare for the
10   // following loop
11   for (int i=0; i<2; ++i)
12   {
13     recentSPs.push_back((*it).getEchoWidth());
14     ++it;
15   }
16
17   while( it != (*scanIt).end() )
18   {
19     // Set the echoWidth to the magnified 2nd derivation
20     (it-2)->setEchoWidth(m_factor * fabs(recentSPs[0]-recentSPs[1]));
21
22     // iterate and move elements around in the recentSPs vector
23     ++it;
24     recentSPs.push_back((*it).getEchoWidth());
25     recentSPs.erase(recentSPs.begin());
26   }
27   // Delete the last two scanpoints in the result
28   (*scanIt).getPointList().erase((*scanIt).end()-2, (*scanIt).end());
29   outputScan.addScan(*scanIt);
30 }
```

The code is simplifying the position of the calculated derivative by putting it on the position of the last scan point which went into the calculation. This is intentional because the this will set a high value to the first scan point on the recognized object, right after the saltus. The full code is in appendix C.2 and also shows the first derivative and handling of the scan points.

Now what is stored in the Scan object are the derivatives of the Echo pulse width, but they need to be interpreted to find the interesting features. This is done by the Echo pulse width cutter.

5.3.3 Echo pulse width cutter

The resulting scans from the street surface marking detector contain the results of the first or second derivative, depending on the configuration settings of the worker, as the new echo pulse width value. The echo pulse width cutter (see appendix C.3) simply removes all scan points below a certain threshold.

```
1  std::remove_copy_if(scan.begin(), scan.end(), std::back_inserter(limitedScans.getPointList()),
       boost::bind(&spBetweenValues, _1, m_scanner1_min, m_scanner1_max, m_scanner2_min,
       m_scanner2_max, m_scanner3_min, m_scanner3_max));
```

All ScanPoint objects with an echo pulse width value not within the limits given by the min and max values, are removed. The function spBetweenValues evaluates if the conditions are met and returns an boolean indicating if the scan point should be removed.

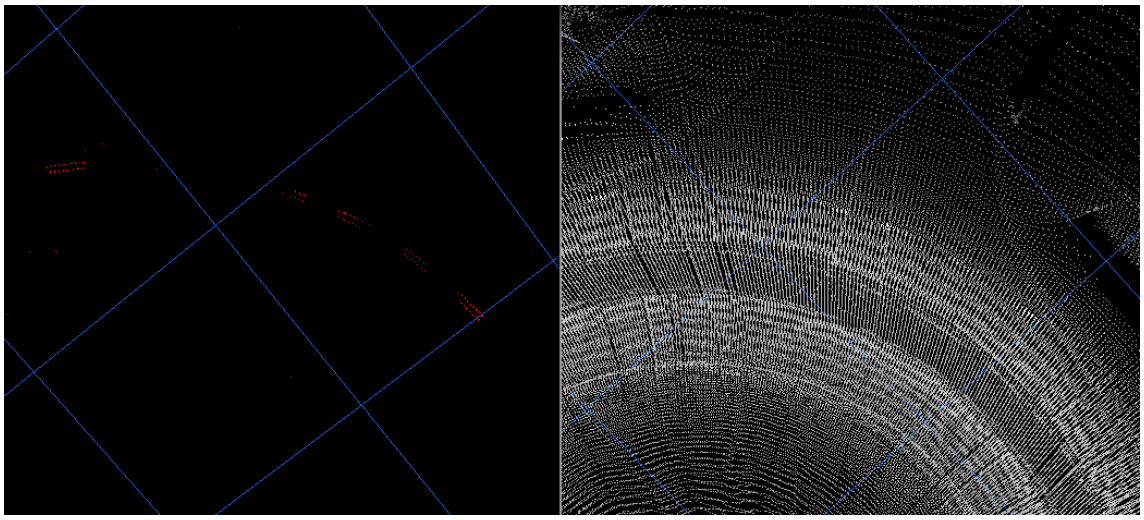


Figure 5.2: On the right hand side the original data with the echo pulse width representation as gray scale can be seen and on the left hand side the result of the street surface marking detector followed by the echo pulse width cutter is shown.

5.3.4 Clustering

Using a clustering algorithm one can associate the feature points extracted by the street surface marking detector. Due to the large sizes of the street surface markings, they are usually cut off halfway in scan slices (see fig. 5.3). It makes more sense and is easier to implement matching on the feature points of the street surface outlines than trying to derive stable information from clusters which represent different parts of the same object. For this reason no further processing of the feature points is done, although a clustering worker was developed and tested, but then discarded.

5.3.5 Slicing

In the existing system one scan consists of one round-turn of the mirror in all three scanners. The resulting scan points are joined together with the knowledge of the mounting positions of the scanners. As a result we get two lines of points. One consists of the data of the scanner on the back and the other of the data of the side scanners. The side scanners are attached mirror-image and at right angles to the car so that they are in the same plane. To fulfill the basic idea of the SLAM algorithm we have to look for some landmarks in the scans and try to find them again in the

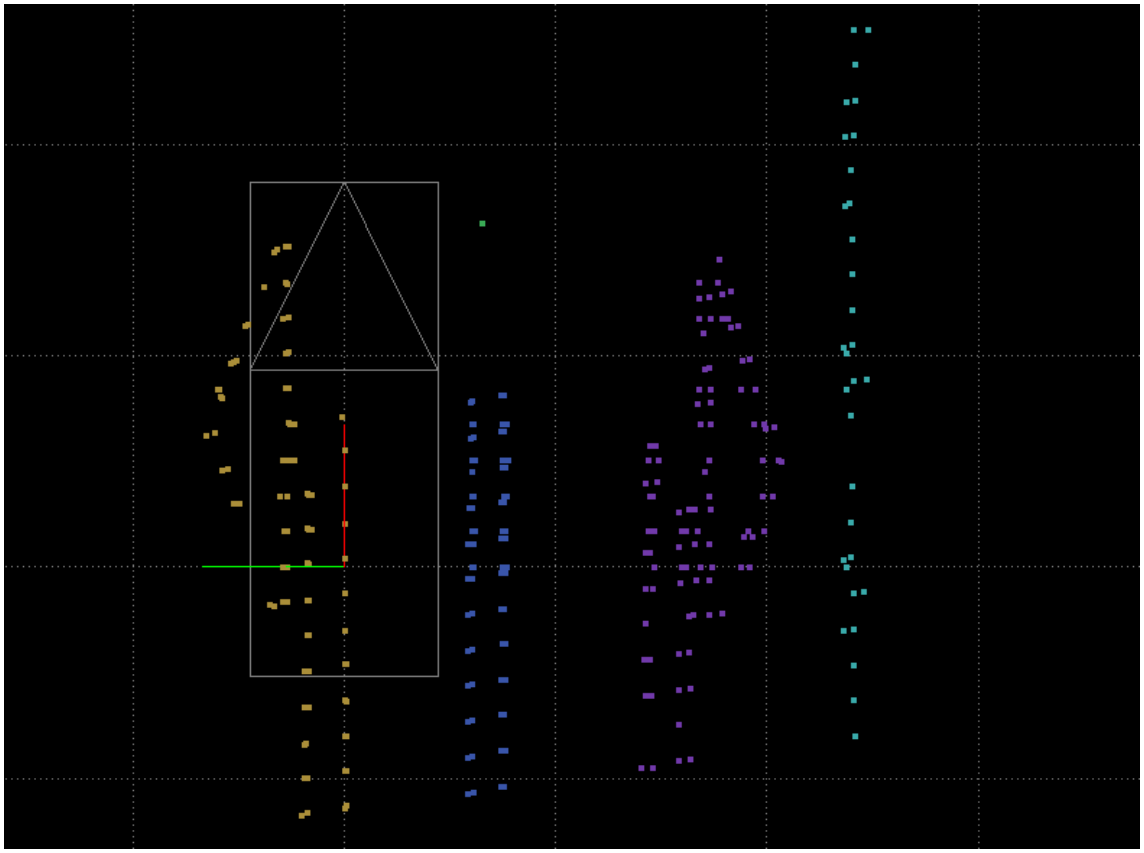


Figure 5.3: The outlines of the street surface markings have been extracted and assigned to clusters. Landmarks belonging to the same cluster have the same color.

new scans. To do so we decided to create some 3D point clouds that have a complete plane of points on street surface. For this purpose we combine several scans and join them together with consideration of the odometry of the vehicle to one new point cloud. The joined scan is called a slice. To motivate the name you have to view it the other way around and take the driving path and cut it into suitable slices. Cole and Newman use a similar process they call 'segmentation' and explain in [CN06].

Because of the limits of the AppBase's Scan object used in the implementation of the slicing algorithm, there cannot be any slices containing more than 65,536 ScanPoints. Since the number of ScanPoints is varying only a safe upper limit can be calculated. If every LRF returns a maximum of 1,000 measurements every scan swipe we can expect up to 60,000 ScanPoints to be combined within 20 scans of 3 LRFs. So, 20 is a safe upper limit for combining scans.

Most scan swipes of the LRFs do return less than 1,000 measurements though and scans of more than 20 scan swipes can be created. But the next limit being hit is the ScannerInfo vector containing the information of all LRFs. A scan can hold no more than 128 ScannerInfo objects and every scan swipe of each LRF adds a new

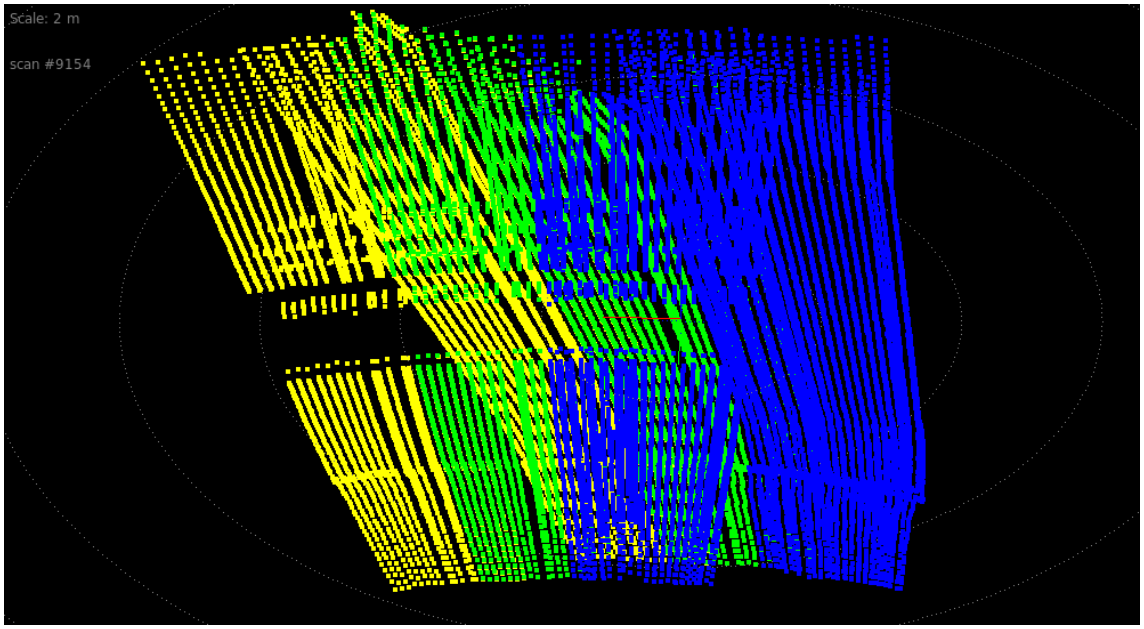


Figure 5.4: This figure shows three differently colored slices. Each slice is a combination of 20 scans of all three scanners and the slices have an overlap of eight scans. The overlap of the yellow and green slices are masked by the next slice but one can see that blue slice consist of 20 and masked slices of $20 - 8 = 12$ scans

ScannerInfo object. This leads to a maximum of 42 combined scan swipes of three LRFs with a total 126 ScannerInfo objects.

As initialization step the first incoming position information is saved as origin position of the slice. When a scan arrives the last position information is saved as current position and the Mounting Position relative to the slice's origin is calculated. It is important that the relative Mounting Position vector has to be in the coordinate system of the origin position. Also the orientation has to be relative to the orientation of the origin position.

```
1  MountingPosition SliceWorker::getRelativeMountingPositionWGS(PositionWGS84 currentWGS84,
       PositionWGS84 originWGS84)
2  {
3    Point3D currentRelWGS84;
4    MountingPosition vehicleMountingPosition;
5
6    currentRelWGS84 = currentWGS84.getCartesianRelPos(originWGS84);
7
8    // calculation of the orientation
9    currentRelWGS84.rotateAroundZ(-originWGS84.getYawAngleInRad()-ibeo::PI_double/2);
10   currentRelWGS84.rotateAroundY(-originWGS84.getPitchAngleInRad());
11   currentRelWGS84.rotateAroundX(-originWGS84.getRollAngleInRad());
12
13   vehicleMountingPosition = MountingPosition(
14         currentWGS84.getYawAngleInRad()-originWGS84.getYawAngleInRad(),
15         currentWGS84.getPitchAngleInRad()-originWGS84.getPitchAngleInRad(),
16         currentWGS84.getRollAngleInRad()-originWGS84.getRollAngleInRad(),
17         currentRelWGS84.getX(),
```

```
18          currentRelWGS84.getY(),
19          currentRelWGS84.getZ());
20
21    return vehicleMountingPosition;
22  }
```

This relative Mounting Position is used to transform the scan data to the origin position coordinate system and to add it to the slice. This is repeated until the slice has the desired size. To start a new slice the next incoming position is saved as origin position and the algorithm starts again.

A feature of the slicing is the parameter overlap. The parameter overlap is the number of scans of the last slice that are used for the next slice, too. So when there is a $overlap > 0$ the starting position and scan are not the first ones to come in, instead they are the ones used as $|overlap|$ last scans in the last slice. This feature allows to create larger slices without increasing the distance between two scans but some points are used twice (see sect. Discussion 6.1.2).

To implement the overlap feature all scans and position data are saved to a *boost::circular_buffer*. These are useful because when the first slice is finished we want to keep the last incoming information for the next slice and delete the old. The *circular_buffer* has the same size as the slices should have, so that every time the buffer is full a slice is finished.

5.4 FastSLAM

FastSLAM is a process which is initiated every time new observations are available (see fig. 5.5). First some Particles are produced. Each particle has a pose which has to be updated by a prediction. Based on the new pose the observations are matched to the landmarks in the map of the particle. This is made with the help of the likelihood table which contains the likelihood values of each observation landmark pair. The best association to an observation is taken and the landmark update is done by using a Kalman filter. At last the importance weight for each particle is calculated and used for resampling which deletes some old particles produces new ones.

5.4.1 Particle

An object of the class Particle represents a possible state of the vehicle at time t. Each one has its own map with the landmarks it has derived from the observations and the predicted position. The predicted position is a sum of the old predicted position and the relative movement vector with some random factors.

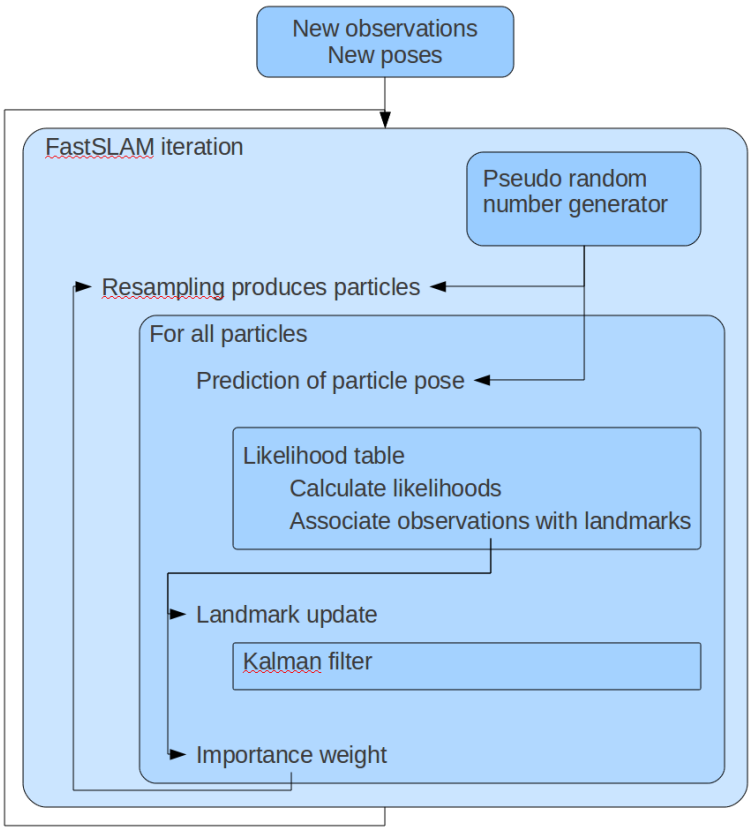


Figure 5.5: Flow chart of FastSLAM

5.4.2 Predicted particle pose

As explained in the previous chapter (see sect. 4.7.2), the motion model in three-dimensional space has six degrees of freedom and thus six different parameters can influence the newly drawn particle poses.

```
ibeo::Position3D Particle::sampleMotionModelOdometry(
  ibeo::Position3D oldPosition,
  ibeo::Position3D newPosition,
  ibeo::Position3D originatingSample,
  float alphaYaw1,
  float alphaPitch1,
  float alphaTrans,
  float alphaYaw2,
  float alphaPitch2,
  float alphaRoll,
```

From the difference of the previous (oldPosition) and the current pose information (newPosition) from the INSs or CAN-bus the vehicle control is derived and added to the previous particle pose (originatingSample). The α-values determine the fraction of the corresponding angles and distances which are used as σ parameter for a Gauss

distributed sample. So if $\alpha_{trans} = 0.1$ and $trans_{u_t} = 2$ the sampled $trans_{m,t}$ will be a value derived from a Gauss distribution with $-\sigma = 2 - 0.1 \cdot 2 = 1.8$ and $\sigma = 2 + 0.1 \cdot 2 = 2.2$.

```
ibeo::Position3D movementVector = newPosition - oldPosition;

// calculating the angles for the first rotation to point in the direction of movementVector
float rot1yaw = atan2 (movementVector.getY(), movementVector.getX()) - oldPosition.getYawAngle();
float rot1pitch = atan2pitch (sqrt(movementVector.getY() * movementVector.getY() + movementVector.
    getX() * movementVector.getX()), movementVector.getZ()) - oldPosition.getPitchAngle();

// calculating the distance to the new position (length of movementVector)
float trans = sqrt(movementVector.getX() * movementVector.getX() + movementVector.getY() *
    movementVector.getY() + movementVector.getZ() * movementVector.getZ());

// calculating the angles to turn to the final orientation at the newPosition
float rot2yaw = newPosition.getYawAngle() - rot1yaw - oldPosition.getYawAngle();
float rot2pitch = newPosition.getPitchAngle() - rot1pitch - oldPosition.getPitchAngle();
float rot2roll = newPosition.getRollAngle() - oldPosition.getRollAngle();

// add noise
float rot1yawSample = rot1yaw - sample( alphaYaw1 * rot1yaw);
float rot1pitchSample = rot1pitch - sample( alphaPitch1 * rot1pitch);

float transSample = trans - sample( alphaTrans * trans );

// rot2 usually depends on rot1, but this dependency is disregarded here
// usually a small rot1 leads to a small rot2
float rot2yawSample = rot2yaw - sample( alphaYaw2 * rot2yaw);
float rot2pitchSample = rot2pitch - sample( alphaPitch2 * rot2pitch);

float rot2rollSample = rot2roll - sample( alphaRoll * rot2roll );
```

The newly sampled parameters are transformed into the global Cartesian coordinate system and set as the new particle's position. For the full particle code see app. D.4.

For visualizing and evaluation purposes test runs showing the sampled particle positions in comparison to the trajectory were plotted. The image shows a typical banana shaped particle cloud like it occurs when the angle error is higher then the translational error, which is the case with the dead-reckoning data from the CAN-bus used to create fig. 5.6.

5.4.3 Likelihood table

The likelihood table holds all likelihood values for all combinations of the current observations and all landmarks in a certain range around the current particle position. This table can grow very large and cost a lot of computation time to calculate. Thus some tricks to reduce computation time were applied. First of all only the landmarks within a reasonable range of the current particles position are fetched and fed into the table. When requesting the landmarks saved inside a particle the particle only returns the ones within the perception range (for more details on particles see sect. 5.4.1). The perception range is a value less than the maximum range the LRFs can scan. For this thesis the perception range is set at 30.0 m although the LRFs are specified

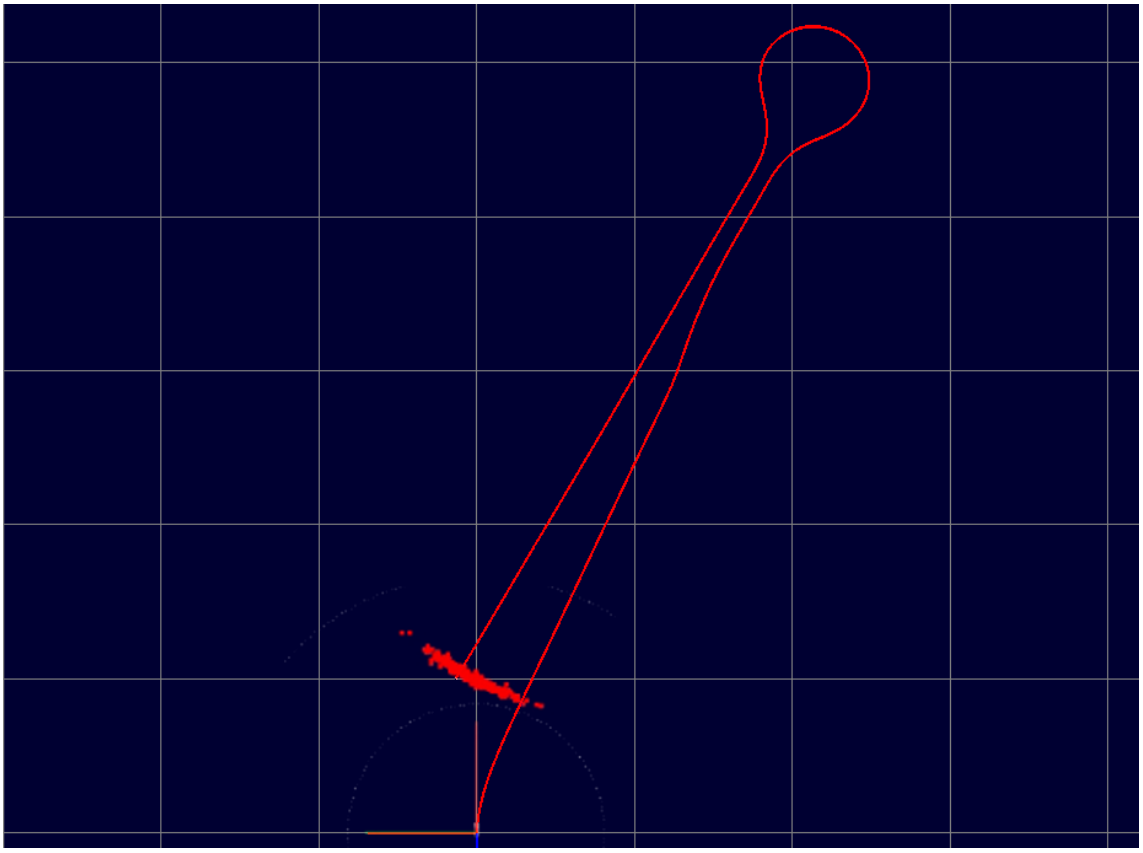


Figure 5.6: The red line shows the vehicle trajectory as plotted by the dead-reckoning data from the car's CAN-bus. The dot cloud at its end is the particle cloud created by repeated pose sampling of 100 particles along the trajectory.

for distances up to 50.0 m, but landmarks in larger distances were found to be hard to recognize reproducibly and scan points further away than the streets curb are cut off anyway (see sect. 5.3.1). Secondly only the likelihood values between observations and landmarks within close vicinity are calculated. As a cut-off distance 3.0 m were chosen because the GPS error (see sect. 2.3) plus the new particles sampled position will always be well within this range.

For the distance comparisons a two-step approach is used. The euclidean distance between the particles position and the landmarks is only calculated after the distances have been checked in a Manhattan like way. Concretely the range is checked per axis. Only when the value for each axis on its own is within the limits the euclidean distance is calculated. Ideally the expensive calculation of three multiplications and one square root operation are saved and replaced by only one comparison.

```
1 bool Likelihoodtable::distLargerLimit(Point3D p1, Point3D p2, float maxDist)
2 {
3   if (fabs(p1.getX() - p2.getX()) > maxDist)
4     return true;
5   else if (fabs(p1.getY() - p2.getY()) > maxDist)
```

```
6       return true;
7     else if (fabs(p1.getZ() - p2.getZ()) > maxDist)
8       return true;
9     else if (p1.dist(p2) > maxDist)
10      return true;
11    else
12      return false;
13  }
```

The proper calculation of the Manhattan distance is simply adding the x, y and z distances of the two points to compare. The name corresponds to the rectangular street grid of Manhattan in New York and is often also called Taxi distance because this is the distance a cab would have to drive on a checker board to get to the given coordinates.

$$\text{Manhattan distance} = x + y + z \tag{5.1}$$

$$\text{Euclidean distance} = \sqrt{x^2 + y^2 + z^2} \tag{5.2}$$

Our FastSLAM algorithm spends a lot of time doing distance comparisons and has to do even more over time. This is caused by the linearly growing numbers of landmarks in the map. In every iteration of FastSLAM each landmark has to compare by the particle to its position. But the goal to get quite stable number of comparisons for the likelihood table was achieved.

5.4.4 Associate Observations with Landmarks

To find the correct association for an Observation to a Landmark one has to look them up in the likelihood-table. The problem is that we want to prevent that two or more Observations are matched to the same Landmark. Therefore it is not sufficient that we seek only the association with the highest likelihood value, but additionally note, whether these Observations can be better allocated to other Landmarks. On the other hand, we do not want to disregard good associations, so we are still looking for other assignments. To find these we have to know all associations with a better likelihood. Therefore the best solution is to start with the best value of the whole likelihood table, remember the corresponding Landmark and Observation and setting all other likelihoods of possible associations for this Observation and Landmark to 0. This is repeated for $\min(|landmark|, |observation|)$ iterations and the likelihood table in the end will hold nothing but the likelihood values of the best associations and 0s.

```
1   void Likelihoodtable::findBestAssociations()
2   {
3     float lastGlobalMax = numeric_limits<float>::max();
4
5     for (size_t i=0; i<min(m_likelihoods.size1(),m_likelihoods.size2()); ++i)
6     {
7       float maxLikelihood = 0;
```

```
    size_t maxRow = std::numeric_limits<std::size_t>::max();
    size_t maxColumn = std::numeric_limits<std::size_t>::max();
    for (size_t row=0; row<m_likelihoods.size1(); ++row)
    {
      for (size_t column=0; column<m_likelihoods.size2(); ++column)
      {
        if ((m_likelihoods(row, column) > maxLikelihood) && (m_likelihoods(row, column) <
            lastGlobalMax))
        {
          maxLikelihood = m_likelihoods(row, column);
          maxRow = row;
          maxColumn = column;
        }
      }
    }

    if ((maxRow < std::numeric_limits<std::size_t>::max()) && (maxColumn < std::numeric_limits<std::
        size_t>::max()))
    {
      lastGlobalMax = maxLikelihood;

      for (size_t row=0; row<m_likelihoods.size1(); ++row)
        if (row != maxRow)
          m_likelihoods(row, maxColumn) = 0;

      for (size_t column=0; column<m_likelihoods.size2(); ++column)
        if (column != maxColumn)
          m_likelihoods(maxRow, column) = 0;
    }
  }
}
```

Whether a Landmark of an association is updated or a new Landmark is created from the Observation depends on a threshold. The likelihoods are compared to this threshold and if the likelihood of the best found association is less than the threshold, the Observation is considered a new Landmark and the associated Landmark stays untouched. The value of this threshold has to be found empirically and depends heavily on the measurement noise model.

5.4.5 Kalman Filter for landmark update

When updating a Landmark position like described in sect. 4.7.6 a new mean and covariance are calculated. This process is nicely encapsulated in our ekf class for the extended Kalman filter. The full code calculating the formula can be found in appendix D.3.

```
  Landmark maxLM = likelihoods.getLandmarkWithMaxLikelihood(*iterZ);

  Matrix associatedJacobian = ekf.landmarkJacobian(iterParticle->getlastOdometryUpdate(), maxLM.mean
      );

  Matrix associatedInnoCov = ekf.innovationCovariance(associatedJacobian, maxLM.covariance,
      measurementNoise);

  // update Kalman Gain (Probabilistic Robotics p. 461, l. 21)
  Matrix associatedGain = ekf.ekfGain(associatedJacobian, maxLM.covariance, associatedInnoCov);

```

```
10    // update Kalman Mean (Probabilistic Robotics p. 461, l. 22)
11    maxLM.mean = ekf.ekfMean(maxLM.mean, associatedGain, *iterZ, maxLM.mean);
12
13    // update Kalman Covariance (Probabilistic Robotics
14    // p. 461, l. 23)
15    maxLM.covariance = ekf.ekfCovariance(associatedJacobian, maxLM.covariance, associatedGain,
          measurementNoise);
16
17    iterParticle->updateLandmark(maxLM);
```

For this step the Jacobian Matrix is needed. The Jacobian Matrix can be problematic and cause numerical instability under certain circumstances which are faced in this implementation. The next section explains these and how the problem is dealt with.

Jacobian Matrix

Using a Jacobian Matrix as described in section 4.7.6 is calculation intensive, because it involves several sinus and cosine calculations for every landmark in every particle every round of the FastSLAM algorithm. Additionally the matrix calculations involved often encounter numerical instability in certain cases, i.e. when the vehicle is only moving very little between scans and the values in the matrices become very small. This makes the Jacobian Matrix and the calculations it involves unreliable and causes hard to correct errors.

```
1  Matrix ExtendedKalmanFilter::landmarkJacobian_original(Position3D estimatedRobotPose, Point3D
       landmarkEKFMean)
2  {
3    const float r = landmarkEKFMean.getX() - estimatedRobotPose.getX();
4    const float t = landmarkEKFMean.getY() - estimatedRobotPose.getY();
5    const float p = landmarkEKFMean.getZ() - estimatedRobotPose.getZ();
6
7    Matrix jacMat (3, 3);
8
9    jacMat(0, 0) = cos(p) * sin(t);
10   jacMat(0, 1) = sin(p) * sin(t);
11   jacMat(0, 2) = cos(t);
12   jacMat(1, 0) = cos(p) * cos(t)/r;
13   jacMat(1, 1) = cos(t) * sin(p)/r;
14   jacMat(1, 2) = -(sin(t)/r);
15   jacMat(2, 0) = -((1/sin(t) * sin(p))/r);
16   jacMat(2, 1) = cos(p) * (1/sin(t))/r;
17   jacMat(2, 2) = 0;
18
19   return jacMat;
20 }
```

A more reliable solution was implemented by replacing the 3D Jacobian Matrix by an identity matrix and transforming all observations into the absolute Cartesian map coordinate system. This way there is no need to convert from spheric to Cartesian coordinate systems or back within the FastSLAM algorithm anymore, but the sensor data has to be transformed to Cartesian map coordinates. Still, this solution comes at virtually no costs since the coordinate system conversions are done within the AppBase already anyway.

```
Matrix ExtendedKalmanFilter::landmarkJacobian(Position3D estimatedRobotPose, Point3D landmarkEKFMean
    )
{
  return IdentityMatrix(3,3);
}
```

All other parts of the FastSLAM algorithm work just as fine as before since the landmarks and their covariances and the vehicle poses are already stored and handled in absolute Cartesian map coordinates. This change introduces minor mistakes into the model, which can be ignored, though. For example the measurement noise and the covariances of the landmarks are modeled according to a vehicle-centric spheric coordinate system. Transforming them to absolute Cartesian coordinates changes their orientation and thus for example the area covered by the covariances changes. But the measurement noise used in this work is modeled as a sphere and thus completely symmetric, so changing the orientation has no effect after all.

Importance weight

As outlined in the previous chapter (see sect. 4.7.7) the importance weights are the basis for the important decision in the following resampling step which particles are a good representation of the mapped environment and to reproduce and which are not and are to be discarded. The original FastSLAM description by Thrun does not include handling of multiple landmark and observation associations and assumes the extension of the algorithm from one to multiple associations is trivial. Only the hint to generate an importance weight for a particle by multiplying the likelihood values for the associations of the particle is given. In practice this turned out to often result in extremely small importance weights or even an importance weight of 0, so it was decided to use the average likelihood value over all associations within the particle.

```
// calculate importance weight average
float importanceWeight = 0.0;
int importanceCounter = 1;

// Loop through all observations
for (ZCollection::iterator iterZ = z_t.begin(); iterZ != z_t.end(); ++iterZ)
{
  float maxLikelihood = likelihoods.getMaxLikelihoodForObservation(*iterZ);

  assert(maxLikelihood == maxLikelihood);

  importanceWeight += maxLikelihood;
  ++importanceCounter;
  [...]
```

Using the average likelihoods the values of the importance weights stay quite stable, are still representing a difference in particles suitability for the resampling step and they are not getting close to the lower floatingpoint value limits.

Resampling

For the resampling step (see sect. 4.7.7) an efficient algorithm by Madow [Mad49] was chosen. It is easy to implement and works in $O(M)$ with M being the number of particles.

```
1  void SLAMWorker::resamplingParticle()
2  {
3    ParticleCollection newParticleCollection;
4    ParticleCollection::iterator iterParticle = m_particles.begin();
5    float sum = iterParticle->getNormRating();
6    float lookingValue = 0;
7
8    // cumulative distribution function
9    for (size_t i=0 ; i<m_maxNumOfParticles ; ++i)
10   {
11     lookingValue = ((uniformRand(0, 1) + i) / m_maxNumOfParticles);
12     assert (lookingValue <= 1);
13     while ( lookingValue > sum )
14     {
15       ++iterParticle;
16       sum += iterParticle->getNormRating();
17     }
18     newParticleCollection.push_back(*iterParticle);
19   }
20
21   m_particles = newParticleCollection;
22
23   assert (sum <= 1.1);
24 }
```

Using INS position information for rating particles

Since the INSs used provide position information and we know from their data sheets how much we can trust their accuracy (see tbl. 2.2 and tbl. 2.3) the position information can be used for rating particle position estimates. Based on the accuracy of 1.5m CEP of the basic SPS GPS service of the RT3040 INS, up to 3m difference of position estimates are considered well within probable limits. Particles with position estimates farther away than 3m from the INS position information are given a penalty. A function calculates a factor in the interval of $]0, 1]$ to apply to the particle rating. If the position difference is higher than the above mentioned limit of 3m an exponentially decreasing function is applied. The result of this function is then multiplied to the particles rating:

5.4.6 Pseudo random number generator

For generating samples for particle positions and for the stochastic select resampling random numbers are needed. On a computer it is really hard to get hold of real random numbers so a pseudo random number generator (PRNG) is used. The boost library has a good implementation which is used for all random numbers needed in

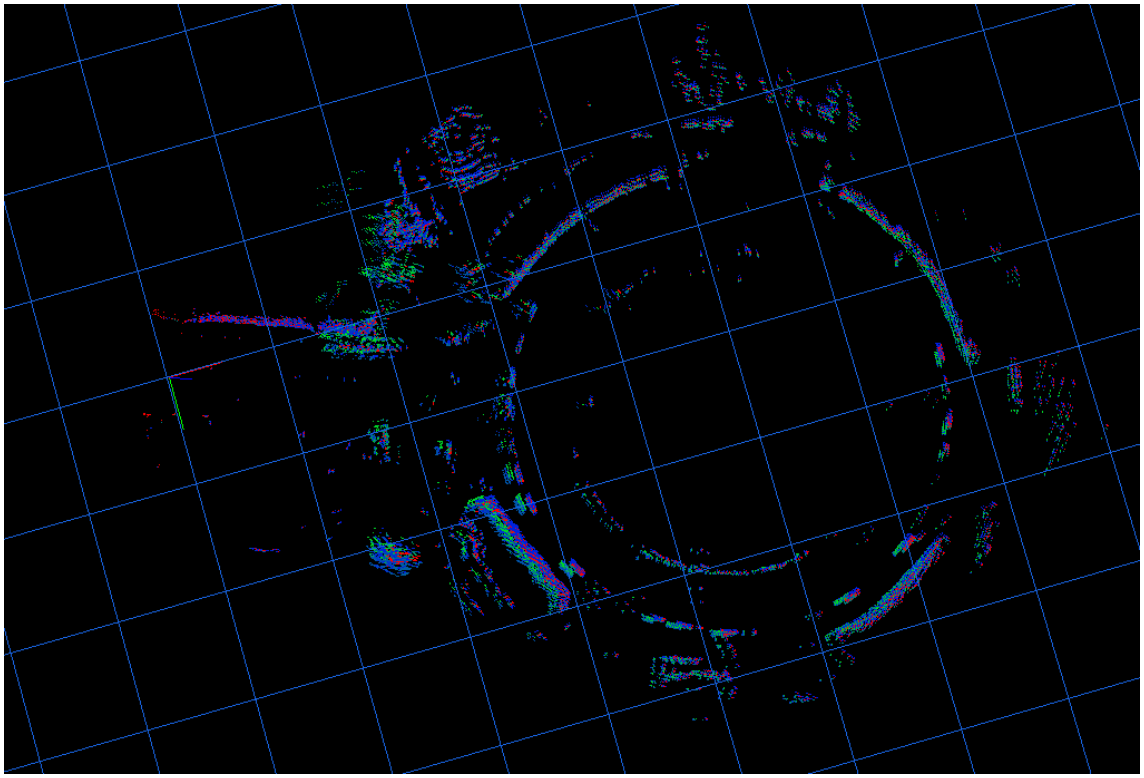


Figure 5.7: This figure shows a view of all Particles saved Landmarks extracted from a Merkurring scan. The best rated Particle's Landmarks are colored red and the other Particle's Landmarks were arbitrarily colored with varying green and blue levels.

this work. The boost PRNG is very configurable and flexible, but also needs some care to set it up properly. First of all a generator is an algorithm that produces numbers which seem random, but are reproducible. That is why a seed is used as a starting point for the algorithm. The current time works well as a seed and ensures that PRNGs started at different times yield different series of numbers. But if more than one generator is started with the current time as a seed in a short timespan, which happens easily on a computer, those generators might get the same seed and thus produce the same series of numbers. This is why a programmer should make sure to only use one generator and access this one from everywhere where random numbers are needed. This means also to take care not to accidentally copy the generator, for example when calling a function with the generator as a parameter. As a generator algorithm the Mersenne-Twister was chosen because it is very fast and proven to generate uniformly distributed series of numbers. It also has a long period of $2^{19937} - 1$ until it starts repeating the same numbers again. When a certain type of distribution is of advantage boost offers mechanisms to do that, too. Different types of distributions can easily be defined and combined with a PRNG to create a variate generator which returns numbers according to the given distribution. In case of the particle samples a normal distribution centered at 0 with $\sigma = 1$ is the best choice.

```
float SLAMWorker::trustInPosRelToGps(float distance)
{
  const float m_lowerGpsBound = 3.0;

  if (distance < m_lowerGpsBound)
    result = 1.0;
  else
    result = exp(m_lowerGpsBound - distance);

  return result;
}
[...]
iterParticle->setRating(iterParticle->getRating() * trustInPosRelToGps(distanceToGpsPos));
```

Table 5.3: Calculate a factor in the interval]0,1] depending on the distance of a particles position estimation to the position information of the INS to apply on the particles rating. For distances less than a bound of 3m no penalty is applied and after this the factor decreases exponentially. The factor then is multiplied with the particles rating.

```
// choosing a generator
static boost::mt19937 generator;
[...]
// choosing a normal distribution centered at 0 with a sigma of 1
boost::normal_distribution<> norm_dist(0, 1);
[...]
// combining generator and distribution to create a
// method which returns a number every time it is called
boost::variate_generator<boost::mt19937&, boost::normal_distribution<> > boost_nrand(generator,
    norm_dist);
```

Now when the method *boost_nrand()* is called it returns the next normal distributed pseudo random number. Madows resampling algorithm (see sect. 5.4.5) needs uniform random numbers in the interval $[0, 1[$. The same generator as defined above is used, but *boost::uniform_real⟨double⟩(min, max)* is used instead of the normal distribution to create another *boost::variate_generator* object.

5.5 Relaxation

For the implementation of the relaxation algorithm shown in sect. 4.5 you need the data from the INS and the data of the vehicle can bus. Because the can bus provides data much more frequently than the INS only that can bus message will be used which follows an INS message. Out of these two message a struct is built and saved to a circular buffer. The size of the circular buffer can be given by a parameter. When the buffer is full iteratively three consecutive entries are taken and a relaxation step is done.

In each step the middle one of the three positions is updated. To do so a relative vector from the previous and one from the successor to the middle one is generated.

This relative vectors are in the vehicle coordinate systems of the the outer points. To transfer this vectors to the global coordinate system the data of the INS of both outer points is used which consist out of a WGS84 message which includes the 3D coordinates and the orientation of the vehicle. Now the arithmetic middle of the vectors can be calculated and is saved as new coordination part of WGS84 information of the middle point.

```
1  inline void RelaxationWorker::relaxationalgo(boost::circular_buffer<State>::iterator iter0 , boost::
       circular_buffer<State>::iterator iter1 , boost::circular_buffer<State>::iterator iter2 )
2  {
3    // calculation of the vector from iter2 to its predecessor based on the angles and vehicleState of
         iter2
4    ibeo::VehicleStateBasic::RelativeVehicle::RelativeVehicle relativeVehicleState((iter1)->
       vehicleState, iter2->vehicleState); //current and previous interchanged because of need of
       negative vector
5
6    ibeo::geom3d::HMatrix  m   =   ibeo::geom3d::rotationRoll  (-(iter2->wgs84.getRollAngleInRad()));
7    m = boost::numeric::ublas::prod (ibeo::geom3d::rotationPitch(-(iter2->wgs84.getPitchAngleInRad()))
         , m);
8    m = boost::numeric::ublas::prod (ibeo::geom3d::rotationYaw(-(iter2->wgs84.getYawAngleInRad()))  ,
       m);
9    ibeo::geom3d::HVector relativeWorldState_21 = boost::numeric::ublas::prod(m ,(ibeo::geom3d::
       makeHVectorRect(relativeVehicleState.getDeltaPos().getX(),relativeVehicleState.getDeltaPos().
       getY(),0)));
10
11   // calculation of the vector from iter0 to its successor based on the angles and vehicleState of
       iter0
12   relativeVehicleState = ibeo::VehicleStateBasic::RelativeVehicle::RelativeVehicle((iter1)->
       vehicleState, iter0->vehicleState);
13
14   m = ibeo::geom3d::rotationRoll  (-(iter0->wgs84.getRollAngleInRad()));
15   m = boost::numeric::ublas::prod (ibeo::geom3d::rotationPitch(-(iter0->wgs84.getPitchAngleInRad()))
         , m);
16   m = boost::numeric::ublas::prod (ibeo::geom3d::rotationYaw(-(iter0->wgs84.getYawAngleInRad()))  ,
       m);
17   ibeo::geom3d::HVector relativeWorldState_01 = boost::numeric::ublas::prod(m ,(ibeo::geom3d::
       makeHVectorRect(relativeVehicleState.getDeltaPos().getX(),relativeVehicleState.getDeltaPos().
       getY(),0)));
18
19   // new Positions for iter1
20   PositionWGS84 wgs84_01 = iter1->wgs84;
21   wgs84_01.transformFromTangentialPlane(relativeWorldState_01(0),relativeWorldState_01(1), iter0->
       wgs84);
22   PositionWGS84 wgs84_21 = iter1->wgs84;
23   wgs84_21.transformFromTangentialPlane(relativeWorldState_21(0),relativeWorldState_21(1), iter2->
       wgs84);
24
25   // arithmetic mean
26   iter1->wgs84.setLatitudeInRad((wgs84_01.getLatitudeInRad() + wgs84_21.getLatitudeInRad())/2);
27   iter1->wgs84.setLongitudeInRad((wgs84_01.getLongitudeInRad() + wgs84_21.getLongitudeInRad())/2);
28   iter1->wgs84.setAltitudeInMeterMSL((iter0->wgs84.getAltitudeInMeterMSL() + iter1->wgs84.
       getAltitudeInMeterMSL() + relativeWorldState_01(2) + relativeWorldState_21(2))/2);
29 }
```

This is done for all trio entries of the circular buffer and if set by a parameter more than one time. Now the oldest entries are removed from the buffer and sent back to the AppBase. How many are removed again can be determined by a parameter. To fill the buffer new INS messages and their related can bus messages are linked and stored. When the buffer is full the relaxation step starts again.

5.6 Summary

The AppBase has some restrictions we had to circumvent, probably because it was designed for a different kind of data processing than necessary for FastSLAM. Nonetheless all problems could be solved, FastSLAM was implemented successfully and scan models can be exported as OFF files for further evaluation without the need to install the proprietary AppBase software framework.

Additionally to the FastSLAM algorithm Relaxation was implemented to iron out the uneven trajectory of the Xsens MTi-G.

Discussion

6

The theoretical description of the FastSLAM algorithm is only a broad outline. For the implementation many decisions had to be made about how to specifically address the open points of this outline. After implementing the FastSLAM algorithm tests were run to evaluate if the implementation is able to improve the maps generated by the 3D-HTRF project. The tests and their results, as well as the above mentioned design decisions are described and discussed in this chapter.

6.1 Design decisions

The FastSLAM algorithm is only specified in a rough outline and leaves many details open. These details can be implemented in many different ways, depending on the specific use-case, the sensors available, the vehicle used and time and resources at hand. Many choices about the algorithms described in the implementation (sect. 5) were made.

When possible an easy to implement option was chosen with only little regard to runtime complexity. The heavy transport company Gustav Seeland GmbH stated that examining the transport route the usual way with scouts being sent out takes seven to ten days (sect. 1.1) and as long as the processing of the maps with FastSLAM does not take longer than this while yielding the same quality or better maps, it is considered an improvement. Consequently fail-safety and code quality was chosen over runtime speed.

6.1.1 FastSLAM 1.0 vs. FastSLAM 2.0

The FastSLAM 2.0 algorithm is an improved version of the FastSLAM 1.0 algorithm implemented in this thesis. The performance of the FastSLAM 1.0 algorithm might actually suffer if the motion information is noisier than the sensors observations, because the proposal distribution of the particle filter will be matched poorly with the posterior probability distribution. Since the LMS151 used in the context of this thesis has errors of up to 2 cm statistically and up to 4 cm systematically and the

sensor is moved around the landmarks, errors can sum up to ±6 cm. Additionally movement of the vehicle during a scan rotation of the LRF is not corrected for which can take up to 44 cm (see sect. 7.1.2). The landmark detection based on the change of the echo pulse width is variable, too, and depends on many factors like distance and angle to the reflecting object, which changes often when moving. Plus during the calculation the position of the landmarks is shifted a bit due to the way the street surface marking detector is implemented (see sect. 5.3.2). In summary it can be said that the sensor error is higher than the position error of the INSs and thus no need for FastSLAM 2.0 is present. FastSLAM 2.0 is mathematically more involved than FastSLAM 1.0 and therefore will need more runtime. It also is more complex to implement (as described by [Hus10]) and since no advantage is to be expected for the purpose of this thesis FastSLAM 1.0 was chosen.

6.1.2 Slicing

FastSLAM was designed for stop-scan-move patterns where large, overlapping point clouds corresponding to each pose are created. This is necessary for FastSLAM to extract landmarks, rediscover them later and use those associations for its extended Kalman filter to refine landmark positions and thus the resulting map. Since it is impractical for a car to repeatedly stop in traffic and because the Laser Range Finder configuration of the 3D-HTRF car is such that it records mostly vertical planes, the scan data of the 3D-HTRF project is not suitable for processing with FastSLAM. Thus a process called 'slicing' similar to the segmentation described in [CN06] was developed. Assuming that within short ranges the odometry information is accurate enough to ignore the error (see the experiment in sect. 1.3), several scan planes are put together and associated with one pose to simulate large point clouds as needed by FastSLAM. The overlap necessary to rediscover landmarks is created by building the slices in a fashion that a certain number of vertical planes is used in both, the current and the previous slice, so they share them.

This is a great way to adapt the FastSLAM algorithm from the stop-scan-move paradigm to a continuously moving vehicle and avoiding problems. For example if the vertical planes were used as they are the FastSLAM algorithm would most likely match the planes all on each others, because of their close proximity and because they are very similar. This still occurs with the vertical planes shared over slices but is desired then, because the similarity is only partly and only the parts belonging together are matched.

6.1.3 Edge and corner detection

In 2D SLAM edge detection is widely used for landmark extraction (six common variants are presented in [NMTS05]). The points transmitted by the scanner are

processed according to the sequence they were determined. This means that they are sorted. If multiple points are on a line the last and first one of this line is searched and marked. The line is the 2D image of a plane and ends of the line are one point of the edge of this line.

For a full six DoF SLAM edges are harder to find because the next neighbors are not known. This problem is caused by the unknown order of the points in 3D. All points have to be tested if the distance is the shortest. A plane has to be extracted to find the edges. Unlike edges in a 2D section of a 3D world, edges in 3D still have one degree of freedom. This can be used for localization (see [SMDW11]), but to get real landmarks the corners have to be calculated. It becomes even more difficult when looking at curved surfaces. An edge extracted out of the scan points of a cylindrical shaped object changes when looking at the same object from another angle.

All in all the creation of landmarks out of corners in 3D point clouds is an expensive and complicated task. Additionally edge and corner detection needs edges and corners to detect. Highly irregular shapes like nature found beneath country roads will challenge the algorithms much more than urban ares with man-made structures. Instead of adapting edge and corner detection for all the different environments expected to encounter it was decided to settle for more efficient alternatives.

6.1.4 Landmark detection

Many different ways to handle the landmarks, i.e. with edge and corner detection like discussed in the previous section, were thought of and tested to be ruled out in the end in favor of the simplest solution of them all. Recognizing and classifying complete street surface markings seems like a good idea at first glance but comes with a lot of computational effort like clustering all the scan points of each scan. This can be done quite well because the shape and sizes of the street surface markings are standardized. These standards can be looked up for every country and used as a basis for a clustering algorithm's parameters (like in sect. 5.3.4). Unfortunately the method used for grouping the scans into slices leads to the problem that almost all slices contain only cut-off parts of street surface markings. This makes it really hard to determine derivative data points for example the center of gravity of a cluster and match street surface markings. Just using the derivative of the echo pulse width without any further processing yields quite good landmarks already. This way every scan point of the border of a street surface marking is a landmark. This might create many erroneous landmarks which cannot be found again, but many more landmarks will match. The number of landmarks is slowing down the runtime of the algorithm, but this simple and safe approach works well for the purposes of this thesis.

6.1.5 Likelihoods

Associating the saved landmarks with the current observations is a crucial part for the FastSLAM algorithm. Some techniques like SIFT and SURF features can be described by a unique vector and thus easily and quickly associated

Associating the saved landmarks with the current observations is a crucial part for the FastSLAM algorithm. A lot of the runtime is spent with calculating the possible associations and their likelihoods to decide which associations are the most likely ones. Instead of comparing each landmark with every observation several approaches exist to tackle this problem.

SIFT or SURF or similar features can be described by a unique vector which is stable between scans and to establish the association nothing more than a comparison of these vectors is needed, but those features are designed for 2D images and need to be adapted to 3D laser range data. Also they need more calculation time to detect. There might still be advantages of using them which is why they are further discussed in the outlook (sect. 7.2.1).

The FastSLAM book ([MTS07]) by Thrun and Montemerlo proposes a method called Monte Carlo Data Association. The idea is to assign associations with a probability according to the likelihood values between landmarks and associations. This approach also addresses the LRFs measurement error by not assuming the most likely associations to be consequently correct and adding a probability. But apart from costing more runtime this approach also needs more particles to represent the different possible associations.

Another idea is to add all landmarks into a data structure which allows for fast nearest neighbor searches, e.g. the data structures described by P. N. Yianilos in [Yia93]. The problem with most data structures for nearest neighbor is that adding landmarks requires to rebuild the complete data structure and new landmarks are added every round of the FastSLAM algorithm to every particle. Special care must be taken in selecting the right data structure to ensure that the gained speed-ups are not being out-balanced by the more costly manipulation operations.

All in all using the plain Maximum Likelihood Data Association proves as simple, yet powerful method for finding landmark and observation associations and the alternatives do not promise a justifiable improvement in accuracy or runtime for the complexity they add to the algorithm or research work needed to adapt them.

6.1.6 Resampling

There are many sampling algorithms available. Madow's systematic sampling algorithm is easy to implement, works accurately and executes in linear time $O(N)$ with the number of landmarks N. This can be improved by not copying all landmarks of

all particles every resampling step, but that would require a more complex particle representation. Since the resampling step is not taking up much runtime and runtime is not a concern of this thesis it was decided to keep the algorithm simple and not add the complexity of an highly efficient data structure.

6.1.7 Modeling of errors and probabilities

Assessing the exact error model for the LRF model or the INSs is out of the scope of this thesis. But this is not necessary because using rough assumptions about the error models taken from the data sheets is sufficient for the algorithm to work. This advantage goes mostly back to the nature of the particle filter which allows for such deliberate errors within some boundaries to be irrelevant. As long as enough particles are sampled from the posterior distribution a sufficient number of particles will be close enough to the real state of the world.

$$R_t = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix} \tag{6.1}$$

The value of 0.1 m for the measurement noise matrix diagonal is a bit above the maximum error a LMS151 can have and proved to work well throughout all conducted tests.

6.1.8 Coordinate Systems

Contrary to Thrun suggestion to handle observations data in polar, or in our case spheric coordinates and transforming back and forth to the Cartesian coordinate system of the map, we eliminated the Jacobian Matrix as a cause of numerical instabilities in the extended Kalman filter calculations. To do so the Jacobian Matrix is replaced by an identity matrix and all scan points are transformed and handled in Cartesian coordinates. This comes at no costs for the implemented FastSLAM algorithm since the design of the AppBase requires to transform the coordinates already. Small differences in the way the covariances are modeled in spheric coordinate space compared to Cartesian coordinate space can be ignored safely. With this decision the implementation gains stability and saves calculation time.

6.1.9 Data structures

For the most part the data structures already implemented by the AppBase are used. This is because the LRF-, position- and map data of the 3D-HTRF project

is all saved in a proprietary binary format and the quickest way to access it is to use the AppBase framework and the data structures it provides. Although the data structures of the AppBase are mainly designed for linear filtering operations on the data and running an algorithm like FastSLAM, which will change data in the past based on current information within the AppBase, is already out of the specifications, but doable. The hardest problem with using the AppBase data structures is the termination problem mentioned in sect. 5.2. Other than that the only advantage from using different data structures would be a better runtime and less memory usage. As mentioned in this chapters introduction this is not a concern, but a well working implementation and easy to understand code are important. In the next chapter (see sect. 7.2.2), more efficient data structures are suggested.

6.1.10 Data output

Since the calling order of the AppBase destructors are unknown the current state of the best map is written out regularly in an OFF file (see sect. 5.2). The last few steps of the FastSLAM matching process might be thus lost, but this problem is easy to overcome by simply feeding the algorithm as much data as could maximally be lost more in the end. The number of scans to add to the end can be calculated by multiplying the size of the interval at which the output is written to disk by the number of non-overlapping scans in a slice. Using OFF as a file output format also has the advantage of avoiding the proprietary binary format used by the AppBase and making the data accessible through freely available software for further inspection and replication.

6.2 Results

The 3D-HTRF project heavily relies on a user to evaluate the generated maps. For automatic analysis the system would have to be very accurate and free of errors in the maps. Many problems in the scanning process are very hard to correct algorithmically, i.e. occlusions and odd reflections. Humans, though, are able to interpret the scan data on a higher level of understanding and can judge from it quite well what the real world must be like. Also, a video is recorded while scanning which can be consulted by the user for a visual confirmation of his or her understanding. Additionally a user with experience in heavy transports can contribute his or her expertise in the interpretation of the maps.

All in all it probably is a tedious and complex work, way out of scope of this thesis, to do automatic evaluation of the maps, which is able to compete with human visual interpretation skills. Thus the evaluation of the maps and their improvements is done by visual inspection and no metric to compare the maps was developed.

Thus comparing the resulting maps from different position data sources before and after processing them with the FastSLAM algorithm implemented in this work is a difficult task. The results are discussed using screenshots of certain problematic areas and comparing them visually with each others. This way the results will be more helpful to the persons working with the 3D-HTRF project than values of an abstract metric.

Figure 6.1: A photo of the Merkurring looking at the central pole from a position a few meters away from the lamp post used for evaluating the map accuracy.

The test track used is a street in Hamburg Rahlstedt called Merkurring. It is located in a business park and does not have much traffic. As shown in figure 6.1[1] it contains a roundabout. Roundabouts are typical obstacles of heavy transports and lead to the situation that the car could drive one time around and come to the same place as it was before. This entails some second scans of the same objects that need to be matched on each others. Now some standing out and easy to recognize object can be chosen and used for assessing the accuracy of the scan matching.

[1] `http://www.meyle.com/_download/press/Meyle_Gebaeude_Merkurring_Hamburg_print_2.5MB.jpg` (May 30th, 2011)

6.2.1 Comparison of ICP to FastSLAM

In the beginning of this work an experiment was conducted (see sect. 1.6.1) to evaluate and compare the different position data sources.

To rate ICP one of this scan drives is sliced and matched with an ICP algorithm (see sect. 3.4). This algorithm is taken from a framework called SLAM-6D. The resulting map (fig. 1.6) clearly shows that this approach does not yield any usable maps. The slices are put together in a seemingly arbitrary fashion and trees are pointing in all directions. The reassembled trajectory in the upper left corner of the picture is going in a big loop which obviously is wrong since the original data is from a straight line only. SLAM-6D failed at processing the test data completely.

ICP is not suitable for the improvement of existing maps. The method to match all points of the point clouds without considering the position of the vehicle is not appropriate. The FastSLAM algorithm has the option to set some parameters which represent the believe in the given pose (see sect. 5.4.2) with the consequence that comparing to ICP FastSLAM produces much better maps.

6.2.2 Replacing an INS

INSs are expensive systems and the hardware used in the 3D-HTRF project costs €3,500 to €41,000 (see chapter 2). Replacing an INS by improving the maps generated from the CAN-data which comes for free with the car would be a great way to reduce costs. Therefore the first test is a comparison of a map generated from CAN-data only.

In the original map (see top picture in figure 6.2(a)) the lamp post is showing up twice, roughly 4 m apart. When looking at the parked car one can see that the offset is mainly sideways which suggests the error being an angular one from going around the roundabout. The translational error is as expected small, like it has been in the experiment in the beginning of this work (see sect. 1.3).

The resulting map (see bottom picture in figure 6.2(b) has the lamp post showing up twice again. This time the distance got even larger, which can be seen easily at the car. So instead of improving the match of the features significant and important to the user, the matching process deteriorated those.

For the CAN data only FastSLAM is not able to match the 3D-HTRF data any better than it lines up already without any matching.

6.2.3 Using the Xsens MTi-G instead of the Oxford Technical Solutions RT3040

Although FastSLAM is often used to entirely replace a GPS receiver, a lot is achieved if the expensive, high precision Oxford Technical Solutions RT3040 can be substituted

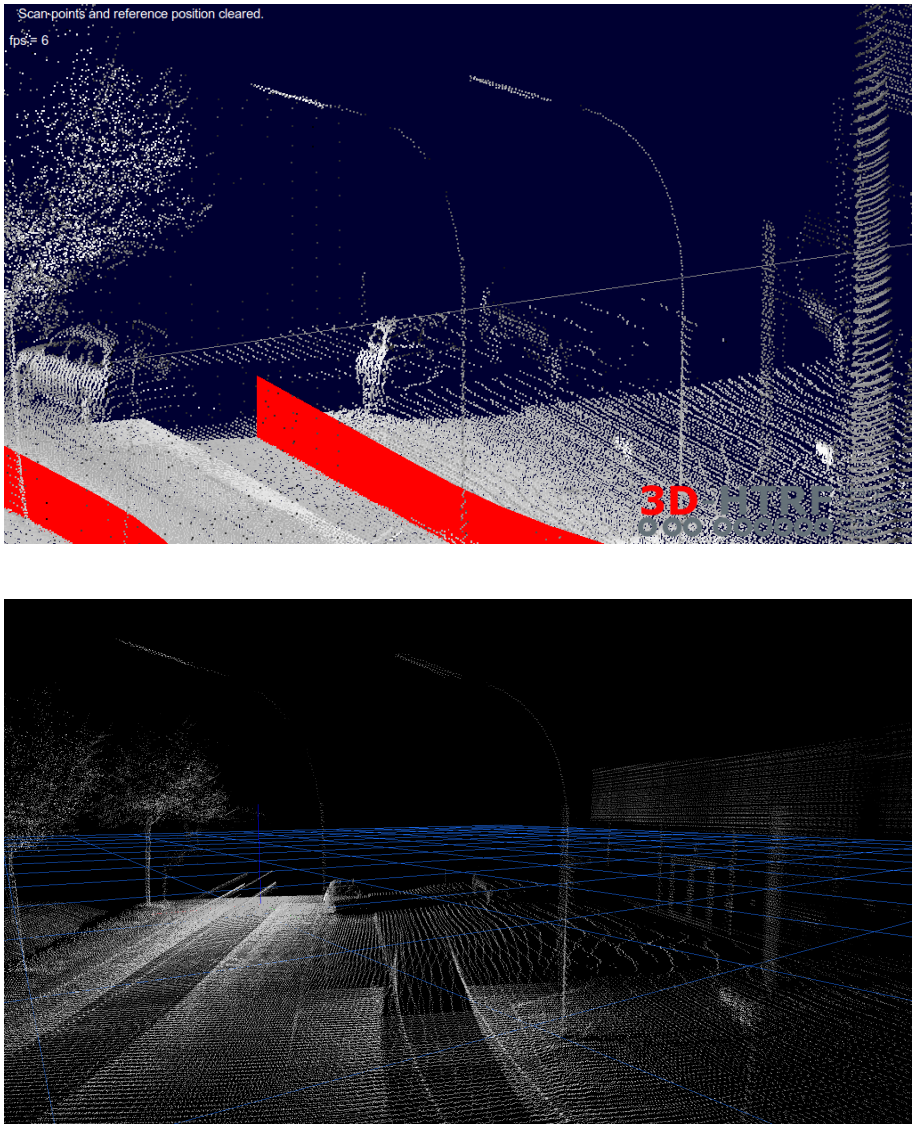


Figure 6.2: On the upper image one can see a map built from CAN data only. It is taken from the 3D-HTRF project without any processing. The scans do not match up and the lamp post is clearly twice in the map. The bottom image shows the same lamp post after applying the implemented FastSLAM algorithm to the map. The lamp post is still visible twice. This shows that FastSLAM is not able to replace an INS in the context of the 3D-HTRF project.

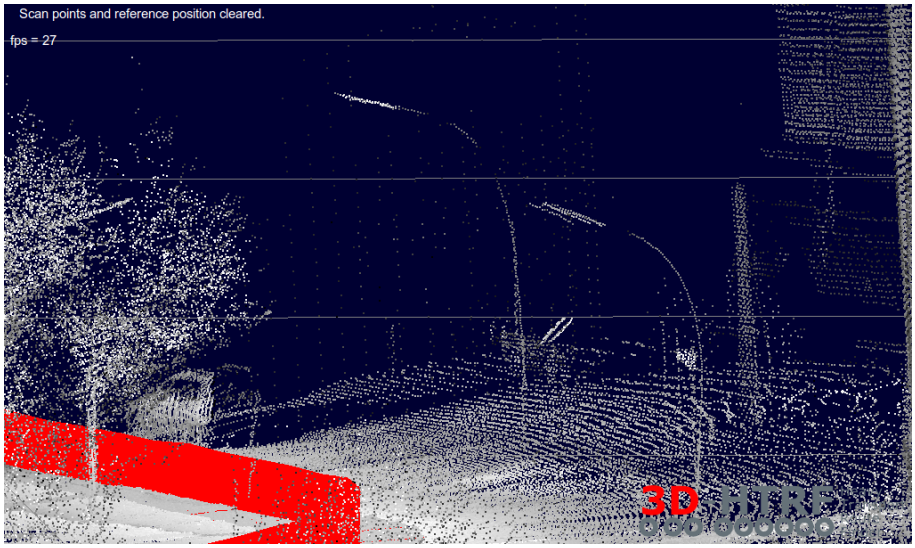
by the Xsens MTi-G combined with FastSLAM postprocessing while preserving map quality. This step would save the difference of the prices of these two INSs, which would be $e\,41,000 - e\,3,500 = e\,37,500$ and thus almost half the system's total hardware costs (compare chapter 2).

To evaluate this question first the Xsens MTi-G is assessed to see if the map was

improved by the FastSLAM postprocessing.

Again the first figure (top picture in figure 6.3(a)) shows the lamp post doubled, but this time recorded by the Xsens MTi-G. The map is not much more accurate than the one built from the CAN data. In this case the translational error is quite high and the scans of the lamp post are mainly separated along the street, in direction of the driving car.

The bottom figure (in figure 6.3(b)) shows the map from above after processing through FastSLAM. The two scans of the lamp post are at about the same positions like before. No improvement can be seen, but the lamp posts are tilted a bit after processing.



The tilted lamp post can be explained by the comparing the next pair of pictures, which was created to take a closer look at the seemingly impairment of the map through FastSLAM processing it.

In the top picture is again the original prior to any processing (figure 6.4(a)). One can clearly see that the trajectory coming from the roundabout is vanishing under the scans from the part of the trajectory going to the roundabout. This is due to the large height error the Xsens MTi-G does make.

The bottom picture (see figure 6.3(b)) shows the same view in the result data and it can be seen that the floor level matches better than in the original data. This is an easy matchable case, that can be matched quite well even with the otherwise impractical LRF mounting design of the 3D-HTRF project. That is because to match the floor level what needed is, is mainly floor data which is available enough of in every slice.

Although FastSLAM was able to improve the map in one aspect this is not the wished for result and not sufficient to substitute the Oxford Technical Solutions

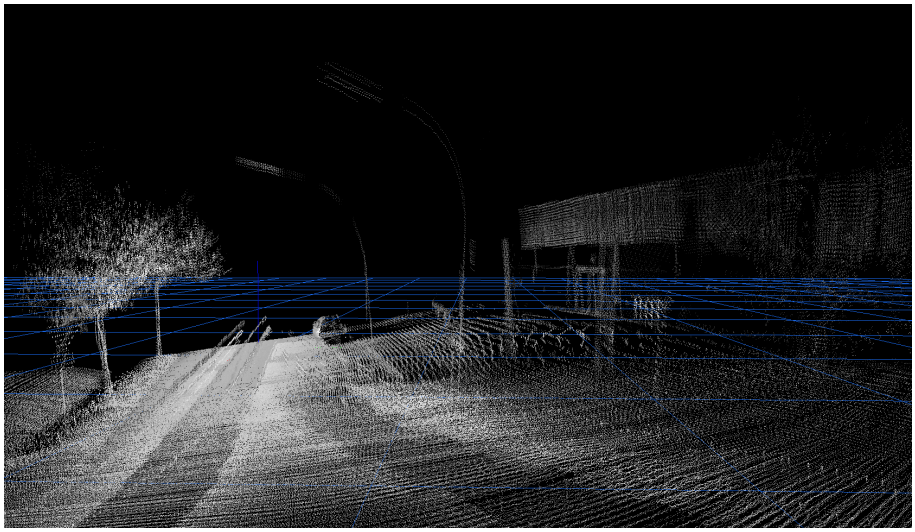
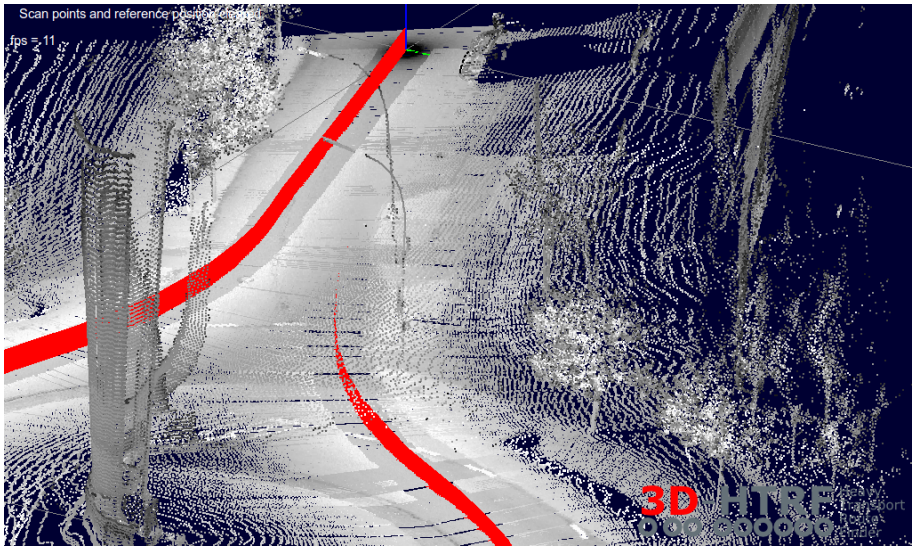


Figure 6.3: Maps built with the Xsens MTi-G are not much better than the maps built from CAN data, shown in the previous section. The lamp post still shows up twice in the unprocessed upper image. In the lower image the two lamp posts are still far away from each other. There is no significant improvement.



RT3040 with the Xsens MTi-G. The partly improvement of the floor level match implies that FastSLAM is working, but the data foundation is not well enough for the matching to work in other orientations or dimensions.

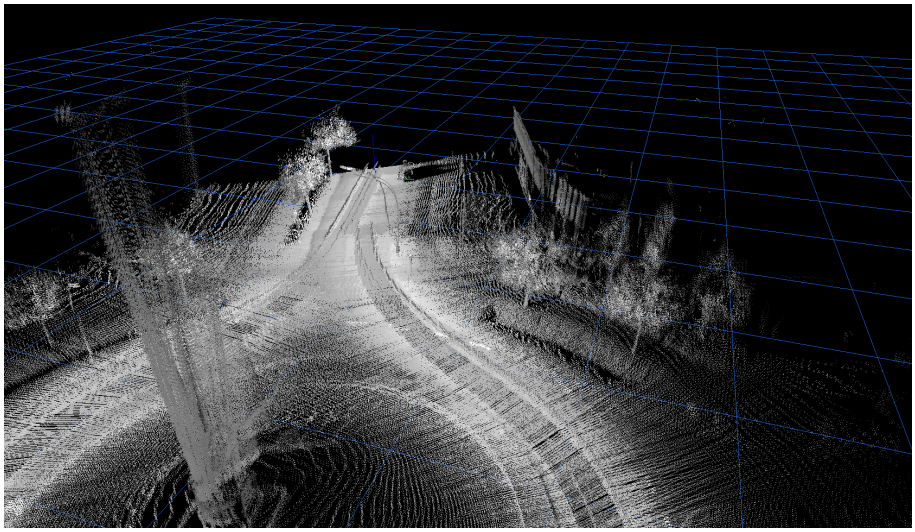


Figure 6.3: Another problem of the Xsens MTi-G is the height information as can be seen on the upper picture where the right hand side trajectory is positioned below the left hand one although the car was on the same street level when recording both trajectories. The lower image shows an improvement in the height of the right hand side trajectory, but the floor plane is still tilted and does not match up perfectly.

6.2.4 Improving maps built from Oxford Technical Solutions RT3040 data

The maps created by the Oxford Technical Solutions RT3040 are the ones with the best quality so far. Improving these would be very helpful towards the accuracy needs of heavy transport companies (compare with sect. 1.2).

Although those maps are the best to come from the 3D-HTRF project, the lamp post still shows up twice, as can be seen in the upper picture of figure 6.4(a). The distance of the two scans is only about 20 cm though.

In the processed version of the map (bottom in figure 6.3(b)), the gap between the two scans of the lamp post grew bigger. All other parts of the map are fine and the floor is matched nicely, but the significant features are not. Matching the floor is easy compared to an object like a lamp post, because the floor is a large plane with lots of scan points that can be matched, even when they are not entirely properly aligned.

Improving the maps generated with the Oxford Technical Solutions RT3040 did not work either.

6.2.5 Conclusion

FastSLAM produces much better results than the ICP algorithm of the SLAM-6D framework using the data from the 3D-HTRF project. It is able to put the maps,

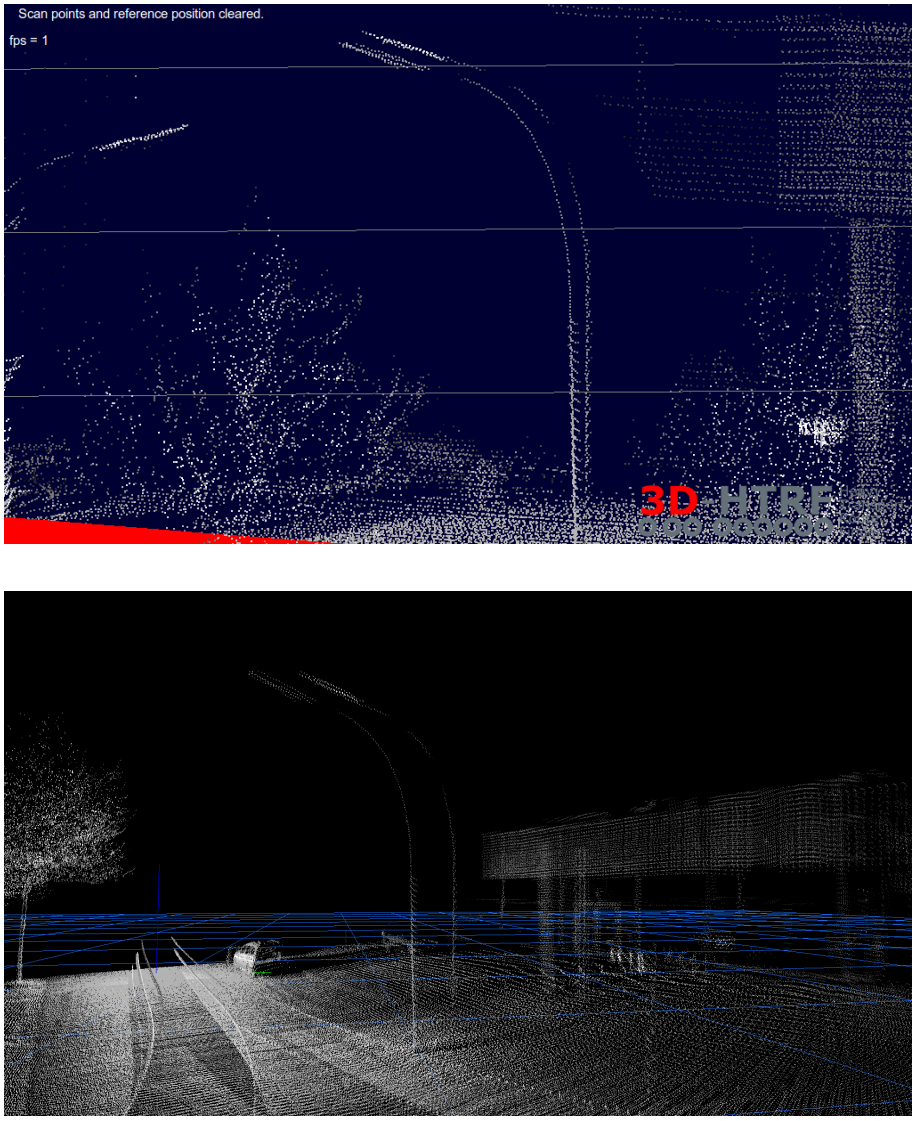


Figure 6.3: Again the upper image shows the map taken from the 3D-HTRF project, this time made using the data from the RT3040 INS. The lamp post is already matched quite well. After applying the FastSLAM algorithm the resulting map is less accurate, as can be seen in the lower image.

which were sliced in preprocessing and reduced to sets of landmarks, back together, which proves that the algorithm is working fine.

The evaluation of the processed maps, though, shows that the hardware of the 3D-HTRF project is not suited for the FastSLAM algorithm to unfold its full potential. Even using slicing as a means to adopt the data for FastSLAM cannot solve this problem by itself.

In order to achieve an improvement in the FastSLAM results, the configuration of

the Laser Range Finders has to be changed. They need to be mounted in a way that allows for creating slices with more coverage of the surrounding environment, so more observations can be extracted per pose. Because then more information for each slice is available and FastSLAM can place them more precisely in relation to the saved landmarks. This will also decrease the undesired influence of the incremental influence of the identical, overlapping scan planes.

Within the preprocessing done in the 3D-HTRF framework the offset of the scan points measured while driving is neglected. Although the Laser Range Finders take about 20 ms to record a 270°swipe, those scan points are treated as if they were taken at the same instant. Especially in curves this offset can be very high due to the angular error.

Another approach would be to additionally use visual data. Visual data usually is available at higher resolutions and gathers more utilizable data like e.g. colors. Scan points from Laser Range Finders are derived from quite large laser beams which can reflect multiple times from multiple objects and then are simplified to one point in space. With SIFT or comparable visual feature extraction techniques very precise matching can be done like [BLO+09] does.

Without changing the hardware setup the maps from the 3D-HTRF project will most likely not be improvable with FastSLAM.

6.3 Statistics

As explained in the results section (see sect. 6.2) there is not much sense in comparing the maps with a metric, so none was developed.

Furthermore measuring improvement of a map is impossible without a reference to compare with. Usually this reference for SLAM algorithms is a GPS generated map, but in this work GPS is already used for creating the maps in the first place and thus cannot be used as ground truth for comparison. Other means to construct a ground truth with the necessary accuracy would be too costly and out of scope in this case.

Although it was not a concern to this work's goals, we took a look at the performance of the FastSLAM algorithm to validate that it behaves as expected.

6.3.1 Runtime

Despite the fact that FastSLAM is only working on landmarks and comparatively efficient, the SLAM problem is calculation intensive. Running a matching process like the roundabout used for evaluation in this chapter, which is about 160 m long, takes several hours up to a day. In this case about 170 matching processes are calculated with 100 particles and a total of several 100,000 landmarks per particle.

Unfortunately longer test runs, or test runs with more particles were not possible since those would take several days each, but the test track is long enough to evaluate if FastSLAM as a method can be used with the 3D-HTRF hardware setup. The runtime depends on the number of particles M used and the number of landmarks N in the way that $O(M \cdot N)$. If the hardware can be changed to work with FastSLAM, then the algorithm could be improved to $O(log(N))$ (as described in [MTS07]) and in the outlook (chapter 7) the steps towards such an algorithm are outlined.

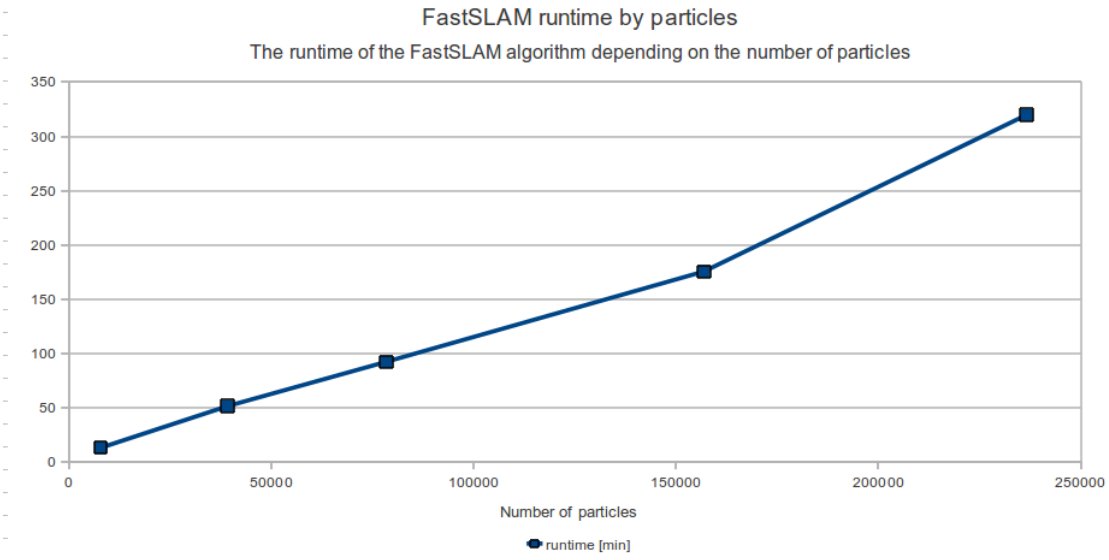


Figure 6.4: The runtime of the basic implementation of the FastSLAM algorithm rises linear with the number of Particles M.

Figures 6.4 and 6.5 show that the FastSLAM implementation's runtime linearly depends on the number of Particles M and also linearly on the number of matched landmarks N. This supports that it has a complexity of $O(M \cdot N)$ like expected.

6.3.2 Memory usage

Every Particle saves its own list of Landmarks and thus many very similar or even identical datasets are stored. With every point cloud processed a number of Landmarks is added to all Particles, thus steadily increasing the memory usage. But even with the 3 GB RAM limit of 32 Bit computer systems this is not a problem for the implemented FastSLAM. As figure 6.6 shows, the memory usage for an extensive test run does not reach the memory limit. The graph also shows that the memory usage increases linearly as predicted.

If memory usage becomes a concern the data structurces mentioned in section 6.1.9 will help reducing it tremendously.

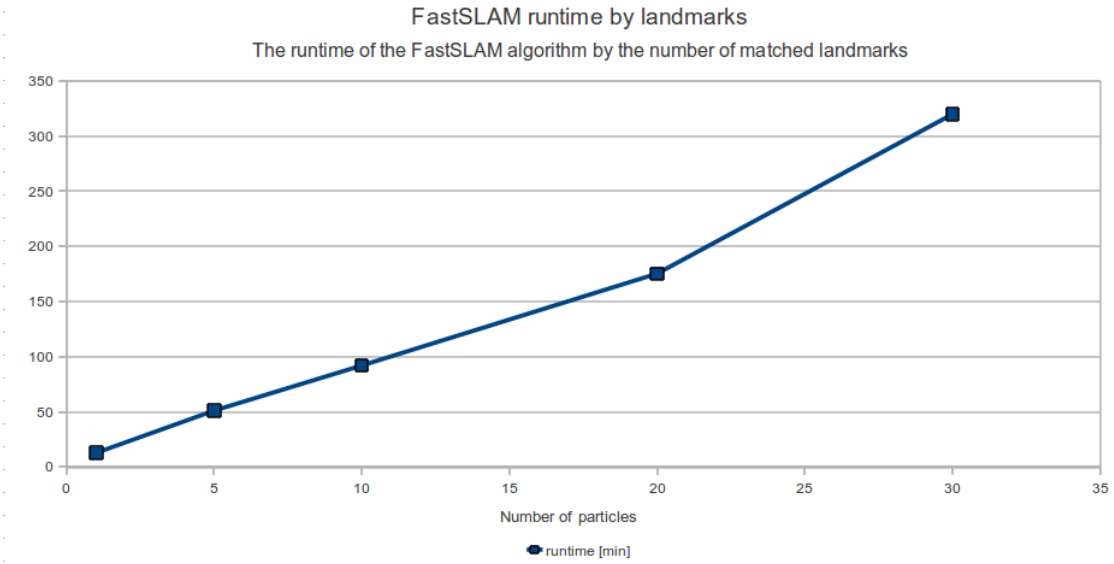


Figure 6.5: The runtime of the basic implementation of the FastSLAM algorithm also rises linear with the number of matched Landmarks N.

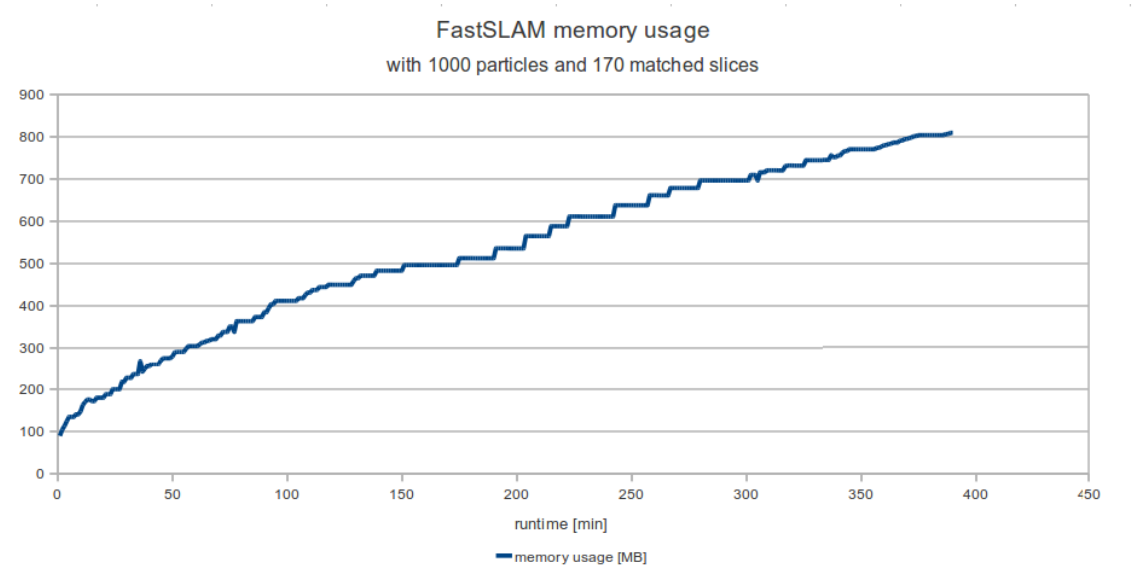


Figure 6.6: This graph shows the memory usage of the implemented FastSLAM over time. The memory usage reflects the number of Landmarks saved because that is the main data structure stored by the Particles, so this graph gives a good impression on how many Landmarks are in use by the FastSLAM algorithm.

6.3.3 Updated Landmarks to Observations ratio

Depending on the likelihood threshold for new landmarks, observations are considered to be a new landmark if the likelihood for any possible association is below this

threshold. So the runtime heavily depends on this threshold, because new Landmarks deeply affect the runtime in the long run.

To reflect this dependency the number of all Landmarks n^t is compared to the number of Observations z^t and a coefficient $\frac{n^t \cdot m}{z^t}$, normalized by the number of Particles M, is calculated.

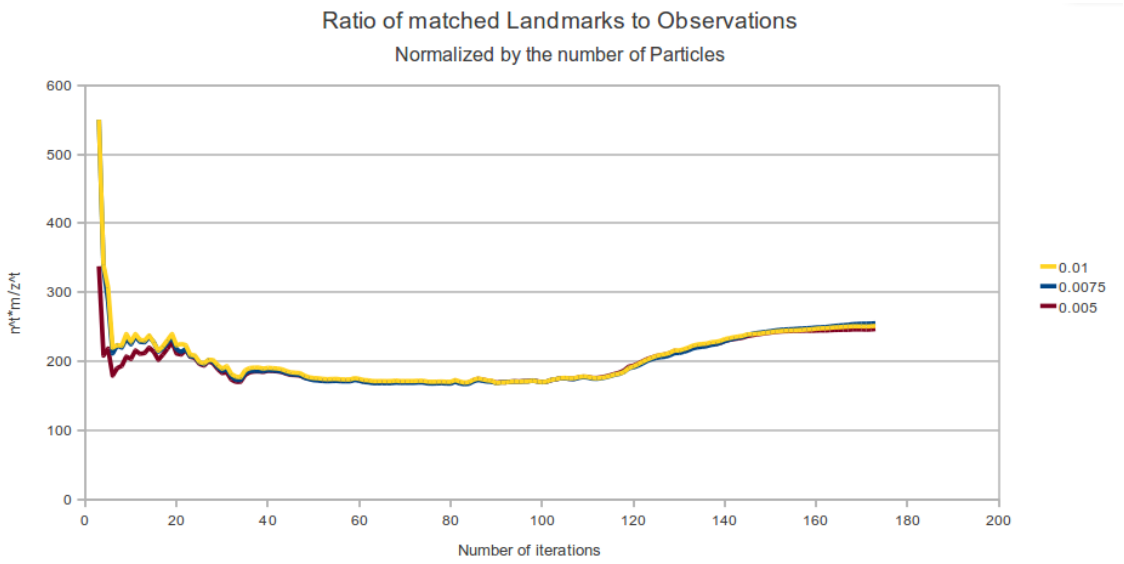


Figure 6.7: The ratio of associated Landmarks n^t and Observations z^t, normalized by the number of Particles M, plotted for multiple likelihood thresholds for new Landmarks. The curves being so close shows that the Landmarks are good associated with the Observations.

That the ratios are so similar for all tested thresholds shows that the Observations are very close to the previous points in space where now is a Landmark saved and the next Landmark the current Observation could be confused with is far away enough to not have many ambiguous associations. This is intended, so the matching is as reproducible as possible. If the threshold is selected much higher than 0.01 or much lower than 0.005 either all Observations are matched or no associations are found.

6.4 Relaxation

Relaxation is not a SLAM algorithm or in any way connected to FastSLAM. It was implemented to smooth the jumps of a trajectory, which FastSLAM is not made for. One of the scenarios it was made for is when the GPS information could not be received for a long period of time and the new GPS position after leaving the tunnel does not match the position estimated by the INS. Figure 6.8(a) shows a map generated at the end of the Elbtunnel in Hamburg. The Elbtunnel is 3325 m long and is therefore a very long track without a chance of retrieving a GPS position of the vehicle. One recognizes the significant jump after about 100 m when exiting the tunnel. Although the INS attempts to incorporate the new GPS position slowly (probably via Kalman filter), the resulting map is without further human intervention useless for the 3D-HTRF project. Any automatic test if a heavy transport would fit would come to the conclusion that the street is too narrow.

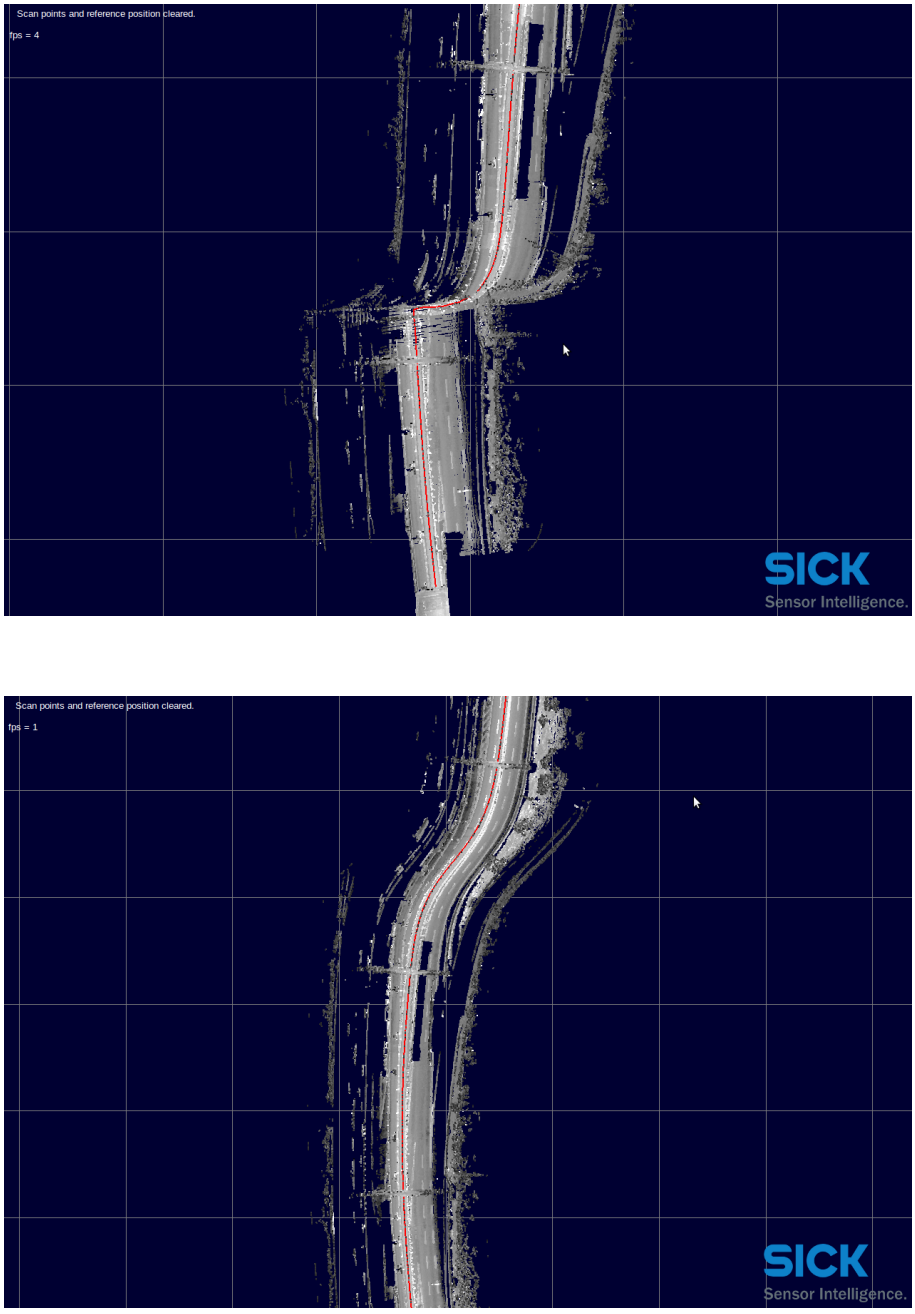


Figure 6.8: Maps when exiting the Elbtunnel before and after relaxation. A quite more smooth course of the road can be seen.

Figure 6.8(b) shows the result of the relaxation algorithm when applied to a track of about 300 m length. Of course the map still does not reflect reality but with regard to the local accuracy it is much better than the original data. To get the best result relaxation has to be done on the whole track in the tunnel (see chapter 7.2.5). For the correction of small jumps as they were presented in the motivation of the Xsens

(see fig. 1.5), relaxation is very useful as shown in figure 6.9. The computational cost is low compared to relaxing large jumps of a few meters like the one of the Elbtunnel.

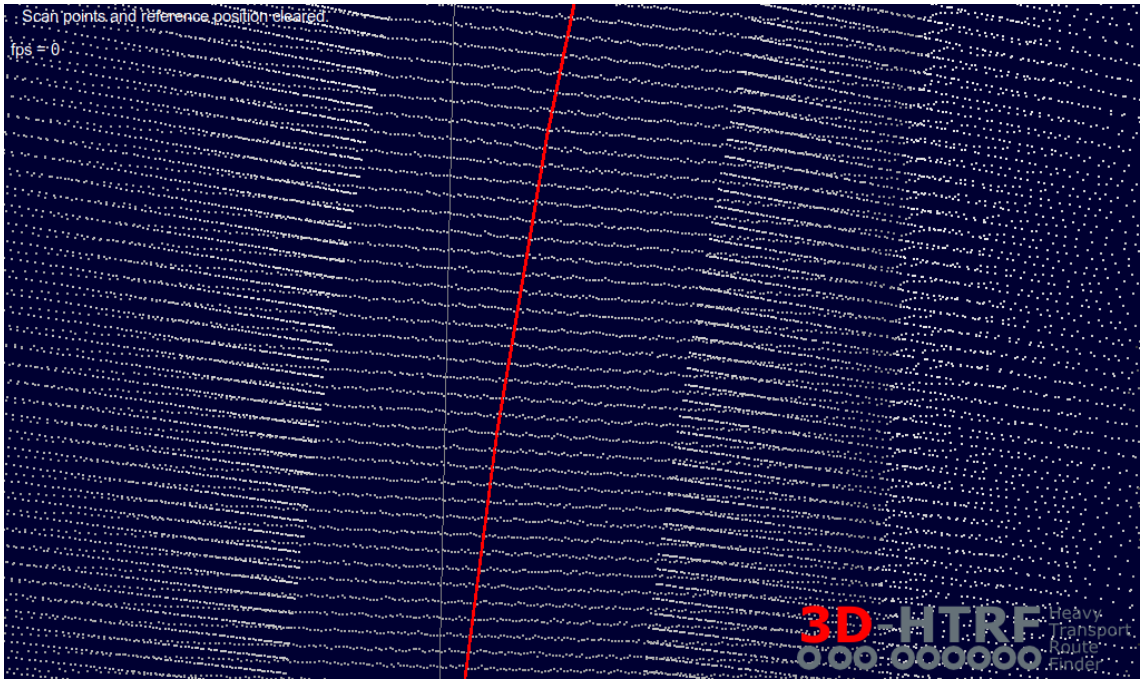


Figure 6.9: Data after using the relaxation algorithm as they are shown in figure 1.5

In FastSLAM the trajectory is determined by how well the current and the previous scan's landmarks match. This may be contrary to the odometry information. When doing relaxation on the resulting trajectory the influence of the odometry information is increased again and a good mixture of both is created.

A problem with the AppBase, that occurs is that when calculating a new position there is a need of information from the future of this position. This means that the system must wait for this next position, and thus delays the calculation. The data from the scanners has to be delayed as long as there are no generally valid time stamps.

6.5 Summary

For the evaluation purposes of this thesis the FastSLAM algorithm in its basic version 1.0 is sufficient. To avoid problems caused by complex data structures and in order to create a straightforward implementation which is easy to understand, most design decisions were made in favor of comprehensible code over high performance.

The evaluation shows that using FastSLAM on the current hardware setup of the 3D-HTRF project works and produces maps, in contrast to the ICP algorithm as

tested with the SLAM-6D implementation. But the evaluation also shows that the current hardware setup is not suitable for improving the maps accuracy further than the level currently achieved already.

According to the statistics our FastSLAM implementation shows the expected runtime behavior of $O(M \cdot N)$. The memory usage is within acceptable limits as well and does not exceed the 3 GB limit of a 32 bit computer system. In other scenarios, where more scan points are processed, a more efficient data structure can be of profit.

For the rough trajectory of the Xsens MTi-G INS, which causes abrupt edges every about 10 cm in the maps, relaxation does help very well. The trajectory can be smoothed and the quality of the maps is improved significantly.

Outlook & Conclusion

7

This chapter deals with the question how further improvements could look like. First a view on the timing and the design of the hardware is taken. Then some implementation changes are proposed, which we expect to solve the problems encountered in this work. The last section is the conclusion of our work.

7.1 Hardware Suggestions

FastSLAM has been shown to basically work on the hardware setup, but the performance is not good enough to improve the map over their current quality. To enhance the results of the FastSLAM algorithm on the hardware of the 3D-HTRF project, some hardware and low level preprocessing changes can be made. When spending more attention to the timing of the system's data and changing the mounting positions of the Laser Range Finders, significant improvements should be achievable.

7.1.1 LRF mounting positions

The initial goal of 3D-HTRF was not preprocessing of the scan data and the mounting positions were chosen with the believe that a very precise position estimate will be provided by an external system. In this context having multiple scans layered on top of each others might make little errors stand out more because clear edges might smudge out. The resulting sensor arrangement (sect. 2.1) only provides little overlap which is being tried to compensate by making the slices overlap (sect. 5.3.5).

For mapping purposes most approaches including FastSLAM use a sensor configuration covering an area as large as possible at once. As a result many landmarks are found per scan and can be utilized for matching. The more landmarks are found and associated between scans, the better the triangulation gets and the higher is the confidence in the matching hypothesis. To profit from a better overlap the sensors mounting positions need to be changed, e.g. as horizontal scanners in the cars bumpers covering the whole region in front of and behind the car. Ideally the LRFs are tilted up and down to cover as much area as possible or, even better, multi-layer

scanners are used and no slicing has to be done anymore. In any case complete 3D point clouds are recorded and matching will work better.

To go a step further splitting up the task of localization and mapping can help. The LRFs in the car's bumpers, as suggested above, are only used for the SLAM and the current LRFs are kept in their configuration, but their data is only saved for later building the maps with the SLAM-improved position data.

7.1.2 Accuracy of timing

Currently the associations between position data and scan data is constructed from the order of appearance in the input channels only. No timestamps are used and no model on delay and its jitter is applied to better correlate scan data and positions. Conducting experiments to determine the delay and jitter of the scanners and INSs and applying a model to correct the timing of the data might improve the accuracy of the resulting maps, especially with regards to the accumulation of errors.

When moving at 80 km/h and the LRFs scanning at 50 Hz, a vehicle travels $80km/h \cdot \frac{1}{3.6}\frac{m/s}{km/h} \cdot 0.02s \approx 44\ cm$ within one scan and thus distorts the line of scan points from the LRF, which takes this long to be measured. Correcting these distortions will improve the position of individual scan points. Additionally the angle speed when turning should be taken into consideration, because for far away scan points on the outside of a curve the effect is amplified.

The problem when trying to improve the timing of the 3D-HTRF system is that there are multiple clocks. One is in the INS, one in each LRF and another one in the computer that collects all the data. The data transmission between this computer and the scanners as well as the connection to the INS is via ethernet and a TCP/IP protocol which has no timing guarantees. On the other hand we can assume that when the transmission is the only one on the bus the time delay is very small. So when minding the postprocessing time, especially of the LRFs, there is a good change to improve the timing and reducing the position and measurement errors.

The second possibility is to use some synchronization channels. The INS retrieves its time via GPS and has a sync output which sends a precise impulse. This can be used by the LRFs to synchronize with. Each LRF itself has a clock which uses ticks, but has no knowledge about the current time. A tick is generated when the rotating mirror is in a typical position which may be given by the user. When the scanner is synchronized the given sync signal is at the same time as the tick. Now a time from the INS can be assigned to the messages from the LRF. When the computer has the same capabilities like for example a real-time operating system, the whole system is timed. If the pose information between the INS messages is interpolated for each scan point where the vehicle pose is known, each scan point can be precisely positioned.

The jitter of the data stream from the LRFs can be reduced by applying a Kalman filter on the data packets to estimate the real time they were sent. This way fluctuations in the TCP/IP data streams can be balanced out.

7.1.3 Accurate determination of model errors

With elaborate measurement methods the errors of all the sensors can be determined accurately and used for the error models of the FastSLAM algorithm. There are two error models that influence FastSLAM. The first is the distance and angular error of the LRFs for the extended Kalman filter's Landmark update and the positioning error of the INSs for the Particle filter's pose estimation step.

Determining these errors is a lot of tedious work since they are comparatively small. For a high precision application this work might be justified and prove helpful, but in the most cases assumptions taken from the data sheets of the sensors are sufficient.

7.2 Improving FastSLAM

There are many ways in which the FastSLAM algorithm can be improved. This chapter introduces some approaches which could further improve the results of FastSLAM on the 3D-HTRF data. These are other data structures for a faster access and other algorithms for the only roughly outlined parts of the FastSLAM algorithm.

7.2.1 Landmark detection

Optimizing the landmark detection is one of the points with the highest potential for improving the FastSLAM results. If the additional information, which is given by the environment, is used one can predict where the next landmark should be observed or landmarks can be classified as certain types like trees, houses, street markings or similar. This classification can be used to associate the observed landmarks with the saved landmarks easier and faster.

Scale-Invariant Feature Transform (SIFT)

In 1999 David Lowe developed an algorithm to describe features in images which are invariant to scaling, translation, and rotation and minimally affected by noise and small distortions [Low99]. Such a feature is very robust and can be used as a landmark in SLAM algorithms. Some of the works mentioned in chapter 3 actually use SIFT features for SLAM, most prominently [SLL05].

SIFTs were developed for images and are based on the difference-of-Gaussian function. For each axis multiple Gaussian kernels with different σ-values are compared and

the differences can be described by a vector which identifies this SIFT. Calcuating thousand of SIFTs takes only very little time. SIFT descriptors can be compared very efficiently and SIFTs describing the same object in different images can be identified an associated. Since calculating the likelihoods of landmark and observation associations is a very computational time consuming task using SIFTs could speed-up FastSLAM substantially.

The above properties make SIFTs a candidate for landmark extraction in FastSLAM. If the echo pulse width of the LRF is interpreted as greyscale intensity information SIFTs could be used on the scanner data. To use them in the context of this thesis' work some more adaptations need to be made. First of all either the three-dimensional point clouds have to be projected on a 2D plane or the SIFT-definition has to be extended to 3D similar to [SAS07]. Additionally the calculation of the importance weights does rely on the existence of likelihood values for the associations between landmarks and observations and a new method for either retrieving likelihood values from SIFTs or another measurement for the importance weights has to be found.

Alternatively Speeded Up Robust Features (SURFs) could be implemented as their inventor claims they are even faster to compute while being as reliable as SIFTs [BTG06]

Street marking detection

Street markings in Germany are normed by the StVO. The width of guideline is set on Autobahnen to 0.15 m and 0.12 m on all other roads. The length is not specified explicitly, instead it depends on the speed limit of the road and is between 3 m and 6 m but the ratio from length of a line to the spacing between lines is on Autobahnen 1:2 and 1:1 on all other streets. This knowledge can help to match the measured data with the map, for example because the end of the center line can be guessed and matched with the ends of street markings found in the measurement data.

A possible approach would be to extract the street surface markings the way it is already done in this work and then further process them. Intelligent clustering with the knowledge of the above mentioned properties of the street markings can isolate them from each other and be a first step to identifying them. When a street marking has been isolated and identified its properties can be used for positioning and orientation. For example the direction they are pointing in can be extracted and the center of gravity of the street marking's scan points can be calculated and used as Landmark position.

For this to work a different sensor configuration is advisable, because right now even the slices of combined scan planes are too small compared to the sizes of street markings and they are cut off almost all the time and some slices do not contain any street markings at all because they are too small.

7.2.2 Reducing calculation complexity and memory usage

The basic variant of the FastSLAM algorithm implemented is calculation intensive and sufficient for an evaluation if the method works with the 3D-HTRF project. When building a system for deployment and use with tracks of hundreds of kilometers of length, the current implementation will reach its limits. Thus the outlines for implementing a better performing variant of the FastSLAM algorithm are given here.

Using efficient data structures

Because the code should be easy to understand and to avoid mistakes caused by a complex construction a simple data structure is used. To get a better performance some changes could be made. Especially the implementation of the likelihood-table is very inefficient. A Matrix is not optimized for accessing its cells and a lot of time is spent searching and accessing elements in a matrix of the size of observations times saved landmarks. A better data structure that allows faster access as e.g. hashtables could improve the runtime of this FastSLAM implementation. Also the number of distance comparisons and likelihood calculations could be decreased significantly or even avoided completely if the data structure used would support a nearest neighborhood search and access.

A further approach for efficient data structures is to save only the new data and avoid redundancy. Every particle has its own landmark map. Because in every resampling step new particles are produced by copying an old one and rather large parts of the maps are saved twice or more often. To prevent this a tree structure could be used. When doing this, first the number of steps to access a landmark is the height of the tree $h = \log N$ with $N =$ number of landmarks and second the pointer to a subtree could be used to share them between more particles. When doing FastSLAM over a longer time period this will save a bulk of the needed memory. A problem of this approach is the garbage collection because old particles that normally would be deleted by the resampling algorithm may contain subtrees that are still needed. So there is a need to remember how many pointers point on a subtree.

In general many calculations are very similar within every particle and repeated very often. It might proof useful to invest some research into caching computation results to minimize the total number of calculations necessary.

Parallelization

As mentioned in the above section many calculations are similar and repeated within every particle, they are also independent. Ideally all particles can be calculated completely in parallel and thus use the full potential of modern CPUs with multiple cores. Since FastSLAM typically uses hundreds of Particles (more on how to determine

a good number of Particles can be found in [BNN06]), it would even be possible to distribute the work over multiple computers. This can be interesting for e.g. semi-autonomous systems where the vehicle sends its sensor data to remote servers which help computing the map.

Advantages of a quick FastSLAM algorithm

If the execution time can be reduced to enable online use of the algorithm, the map will be ready right when the car finished driving along the suggested route. Apart from reducing costs by saving on processing time, a live created map might help detecting problematic areas and the driver can search for an alternative route ad-hoc without waiting for the resulting map after finishing his route. This way redoing whole scouting drives to find alternative routes can be avoided and a lot of time can be saved.

7.2.3 Probabilistic extensions

Thrun's FastSLAM 2.0 is an extension of the FastSLAM 1.0 algorithm using probabilistic methods. The main methods to move towards a FastSLAM 2.0 implementation are outlined in this section.

Landmark association

The likelihood values are crucial to the results of FastSLAM. It is used to associate observations to landmarks by choosing the landmark with the highest likelihood value. A different approach is to choose the landmark by interpreting the likelihood value as the probability to associate the current observation with the related landmark. This suggestion is mentioned in the book *FastSLAM* [MTS07]. This interpretation makes sense only when more possibilities will be investigated. This leads to higher computational costs.

Likelihood value

In our work the likelihood is determined by calculating the value of a normal distribution as it is done in [MTS07]. This forces some problems as noted in sect. 5.4.5. The mathematically correct method would be to combine the normal distribution of the landmark with the normal distribution of the observation. The integral of the resulting distribution would be the correct likelihood value.

Resampling method

In the description of the FastSLAM algorithm the resampling method is discussed briefly only. But the method used can have a significant influence on the ratio between resampled particles with high importance weights and low importance weights. If this ratio is skewed towards the highly rated particles FastSLAM might suffer the same problems like a traditional Kalman filter approach FastSLAM explicitly started out to avoid and result in only a single hypothesis being sustained. On the other hand if the resampling algorithm is selecting too many particles with a bad rating the particle cloud representing all position hypothesis might spread out very thinly and not cover the highly probable positions anymore.

Thus researching and implementing a different resampling method can change the behavior and performance of the FastSLAM algorithm very much. In the book *Integrierte Navigationssysteme* [Wen07] a resampling algorithm is presented that can evaluate whether resampling is necessary at all, which of course can save lots of processing time.

7.2.4 Output

When the AppBase finished sending all data from its source it starts calling the destructor of all worker objects that finished processing all incoming data. Unfortunately writing out the latest and final maps calculated by the FastSLAM algorithm in the destructor fails. As a workaround all maps are written to disk in a specifiable interval. The maps written to disk contain billions of lines of point coordinates and color information and easily reach several hundred megabytes per file which takes a while to write to the harddrive. Reducing the frequency at which to write out the maps will speed-up the execution of the program vastly. In combination with parallelization (sect. 7.2.2) a further improvement in execution speed can be achieved by placing the function for writing the data to disk in its own thread. With appropriate locking the data in the process of being written out can be read for resampling and the newly sampled particles can already be processed.

7.2.5 Relaxation

Relaxation has delivered the results we had hoped for but for large deviations they can be improved. As shown in section 6.4 the smoothing of the exit of the Elbtunnel is probably good enough for the case of HTRF but for a good global map it has to be better. The Problem is that the jump is too large which is because the time during which no GPS signal was received was very long. If we would knew the size of the jump we could react, spend more time and adjust the parameters. So when knowing the state of the GPS receiver the last position before entering the tunnel

can be taken by relaxation as a starting point which can be highly trusted. The point after leaving the tunnel when first getting a GPS update would be the last one used in the algorithm. Based on the number of the points the parameters could be guessed. The more points the more computing time should be used on relaxation.

7.3 Conclusion

In this work we examined the 3D-HTRF project for creating highly accurate maps for scouting heavy transport routes.

The motivation started with an experiment to assess the hardware used and test the suitability of a well known and scientifically tried SLAM algorithm called SLAM-6D of the ICP family. The results showed that the three available position data sources are of varying accuracy. The worst position information are the vehicle data taken from the CAN bus of the car, which is expected as it is derived through dead-reckoning and errors accumulate. Of the two INSs the cheaper Xsens MTi-G is performing better than the dead-reckoning from the car's sensors, but the costly Oxford Technical Solutions RT3040 using the highly accurate satellite based differential GPS naturally delivers the best position information.

No matter which position information was used, the SLAM-6D algorithm was not able to match the simple, straight test track properly (fig. 1.6). This is probably due to the basic principle of ICP which only allows for one hypothesis of the state of the world to be followed. Consequently errors made when little information was available cannot be corrected later when more information becomes available.

FastSLAM is a combination of EKFs and a particle filter which use probabilistic methods to follow multiple hypotheses. Using FastSLAM the problems of an ICP algorithm can be overcome, but FastSLAM had to be adopted and implemented for the three dimensional case as present in the 3D-HTRF project and the hardware used. This was done and described in the main part of this thesis.

The FastSLAM implementation was tested and evaluated to find out

1. if the 3D-HTRF hardware is producing data suitable for the application of a FastSLAM algorithm for improving the generated maps
2. if the costs of an INS can be saved by improving maps created from the CAN data of the car with comparable results
3. if the cheaper Xsens MTi-G in combination with FastSLAM could replace the expensive Oxford Technical Solutions RT3040 INS.

It was shown that the adaptation of the 3D-HTRF data for FastSLAM basically works by combining scan data to create artificial point clouds, named slices. But the results for improving the maps are not satisfactory at all, which is due to the mounting positions of the LRFs. Even with the slicing applied the point clouds are

not sufficient for creating highly accurate maps using FastSLAM. More Landmarks associated with each vehicle pose are needed for better triangulation and matching. Suggestions were made on how to change the 3D-HTRF hardware to achieve the above goals.

The software of the 3D-HTRF project is proprietary and protected by copyrights and the data format used is a proprietary binary format. Thus an exporter had to be written to make the data necessary to examine the results available and a viewer to be able to visualize the results. Additionally, to further improve the data of the Xsens MTi-G, a relaxation algorithm was implemented to straighten out periodic leaps in the position data. These errors most likely result from a too quick correction step of the INS when determining a new GPS position. The tests showed that these leaps can be corrected using our relaxation algorithm.

The FastSLAM algorithm implemented here is a simple and easy variant tailored for evaluation purposes. FastSLAM has a huge potential for runtime and memory usage improvements which will allow the algorithm to work on large maps of hundreds of kilometers.

With these results a way for the 3D-HTRF project to save costs and improve map quality in the future is demonstrated.

Acknowledgement

Writing a diploma thesis is a huge effort representing the final highpoint of a students degree. We are very thankful for all the support we received to make this experience as remarkable as it was. First of all thanks to our supervisors Prof. Dr. Jianwei Zhang[1] Head of the group Technical Aspects of Multimodal Systems (TAMS) at the University of Hamburg and Dr. Kay Fürstenberg, Senior Manager Research Activities at SICK AG[2] for making this thesis possible for us and providing us with everything necessary to conclude this work.

Many thanks go to our thesis advisors Denis Klimentjew and Daniel Westhoff who gave us important input and guidance which helped to shape the outline of our work. They were always there for us when we needed help and advice.

Last but not least we want to thank our friends and family for their moral support and looking over our work to point out all those little things we never would have noticed.

- Hannes Bistry (proofreading)
- David Dittman (proofreading)
- Julian Fietkau (proofreading)
- Monika Nerlich-Girlich (for being a patient and understanding mother)
- Johann Wegener (delicious coffee)

[1] http://tams-www.informatik.uni-hamburg.de/people/zhang/
[2] http://www.sick.com/

Eidesstattliche Erklärung Jan Girlich

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den

Jan Girlich
Matr.- Nr.: 5595529

Eidesstattliche Erklärung Jan Gries

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den

Jan Gries
Matr.- Nr.: 5401333

Aufteilung der Gruppenarbeit

Da diese Arbeit von Jan Girlich und Jan Gries gemeinsam erstellt wurde, werden im Folgenden einzelne Kapitel dem jeweiligen Autor zugeordnet.

Von Jan Girlich, Matr.- Nr.: 5595529, erstellte Kapitel: 1.1, 1.2, 1.4-1.8, 2.3-3.2, 3.7, 3.8, 4.3, 4.4, 4.7-5.2, 5.4, 6.2, 7.2.1, 7.2.2

Von Jan Gries, Matr.- Nr.: 5401333, erstellte Kapitel: 1.3, 2.1, 2.2, 3.3-3.5, 3.9-4.2, 4.5, 4.6, 5.3, 5.5-6.1, 6.3-7.1, 7.2.3-7.2.5

Die hier nicht aufgeführten Textpassagen, genauso wie der gesamte Quellcode, wurde in Zusammenarbeit erstellt beziehungsweise programmiert.

Bibliography

[aE01] B.A. am Ende. 3D mapping of underwater caves. *Computer Graphics and Applications, IEEE*, 21(2):14–20, 2001.

[AHB87] K. S. Arun, T. S. Huang, and S. D. Blostein. Least-Squares fitting of two 3-D point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(5):698–700, 1987.

[ATS02] K. O Arras, N. Tomatis, and R. Siegwart. Multisensor on-the-fly localization using laser and vision. volume 1, page 462–467, 2002.

[BD06] T. Bailey and H. Durrant-Whyte. Simultaneous localization and mapping (SLAM): part II. *Robotics & Automation Magazine, IEEE*, 13(3):108–117, 2006.

[BEL+08] D. Borrmann, J. Elseberg, K. Lingemann, A. N\üchter, and J. Hertzberg. Globally consistent 3D mapping with scan matching. *Robotics and Autonomous Systems*, 56(2):130–142, 2008.

[BJ05] R. S Bucy and P. D Joseph. *Filtering for stochastic processes with applications to guidance*. Chelsea Pub Co, 2005.

[BLO+09] L. M. Bergasa, M. E. Lopez, M. Ocana, R. Barea, and D. Schleicher. Real-Time hierarchical outdoor SLAM based on stereovision and GPS fusion. *IEEE Transactions on Intelligent Transportation Systems*, 10(3):440–452, 2009.

[BM92] Paul J. Besl and Neil D. McKay. A method for registration of 3-D shapes. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(2):239–256, 1992.

[BNL+03] M. Bosse, P. Newman, J. Leonard, M. Soika, W. Feiten, and S. Teller. An atlas framework for scalable mapping. pages 1899–1906, Taipei, Taiwan, 2003.

[BNN06] T. Bailey, J. Nieto, and E. Nebot. Consistency of the FastSLAM algorithm. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, page 424–429, 2006.

[BTG06] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *Computer Vision–ECCV 2006*, page 404–417, 2006.

[BZB+10] Paulo Borges, Robert Zlot, Michael Bosse, Stephen Nuske, and Ashley Tews. Vision-based localization using an edge map extracted from 3D laser range data. pages 4902–4909, Anchorage, AK, USA, 2010.

[Car08] J. Carlson. *Mapping Large Urban Environments with GPS-Aided SLAM.* PhD thesis, Citeseer, 2008.

[CN06] D. M Cole and P. M Newman. Using laser range data for 3D SLAM in outdoor environments. page 1556–1563, 2006.

[CN07] M. Cummins and P. Newman. Probabilistic appearance based navigation and loop closing. In *Robotics and Automation, 2007 IEEE International Conference on*, page 2042–2048, 2007.

[Dav03] Andrew J Davison. Real-time simultaneous localisation and mapping with a single camera. *null*, pages 1403—1410, 2003.

[DB06] H. Durrant-Whyte and T. Bailey. Simultaneous localisation and mapping (SLAM): part i the essential algorithms. *Robotics and Automation Magazine*, 13(2):99–110, 2006.

[DFBT99] F. Dellaert, D. Fox, W. Burgard, and S. Thrun. Monte carlo localization for mobile robots. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 2, page 1322–1328, 1999.

[DMS00] T. Duckett, S. Marsland, and J. Shapiro. Learning globally consistent maps by relaxation. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 4, page 3841–3846, 2000.

[DRN96] H. Durrant-Whyte, D. Rye, and E. Nebot. Localisation of automatic guided vehicles. *Robotics Research: The 7th International Symposium (ISRR 1995)*, 1996.

[ED08] E. Eade and T. Drummond. Unified loop closing and recovery for real time monocular slam. In *British Machine Vision Conference*, 2008.

[Eis02] Stephan Eisenlauer. Bestimmung der fahrzeugeigenbewegung auf basis von 2D entfernungsprofilen eines laserscanners, 2002.

[EM92] S. P Engelson and D. V McDermott. Error correction in mobile robot map learning. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, page 2555–2560, 1992.

[ESLW06] R. M Eustice, H. Singh, J. J Leonard, and M. R Walter. Visually mapping the RMS titanic: Conservative covariance estimates for SLAM information filters. *The International Journal of Robotics Research*, 25(12):1223, 2006.

[FLD05] U. Frese, P. Larsson, and T. Duckett. A multilevel relaxation algorithm for simultaneous localization and mapping. *Robotics, IEEE Transactions on*, 21(2):196–207, 2005.

[GRS+08] G. Grisetti, L. Rizzini, C. Stachniss, E. Olson, and W. Burgard. Online constraint network optimization for efficient maximum likelihood

map learning. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, page 1880–1885, 2008.

[HBFT03] D. Hahnel, W. Burgard, D. Fox, and S. Thrun. An efficient fastslam algorithm for generating maps of large-scale cyclic environments from raw laser range measurements. pages 206–211, Las Vegas, Nevada, USA, 2003.

[HBT03] D. Hähnel, W. Burgard, and S. Thrun. Learning compact 3D models of indoor and outdoor environments with a mobile robot. *Robotics and Autonomous Systems*, 44(1):15–27, 2003.

[HDB+10] D. Holz, D. Droeschel, S. Behnke, S. May, and H. Surmann. Fast 3D perception for collision avoidance and SLAM in domestic environments. 2010.

[Hus10] S. F Husain. Evaluation of methods for 3D environment reconstruction with respect to navigation and manipulation tasks for mobile robots. 2010.

[JD06] P. Johnson and M. Danis. Unmanned aerial vehicle as the platform for lightweight laser sensing to produce sub-meter accuracy terrain maps for less than $5/km2. *Mech. Eng. Dept., Columbia Univ., New York*, 2006.

[JMW+03] A. Jacoff, E. Messina, B. A Weiss, S. Tadokoro, and Y. Nakagawa. Test arenas and performance metrics for urban search and rescue robots. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 4, page 3396–3403, 2003.

[Jot01] D. I.J Jotzo. Aktive landmarken zur positionsbestimmung von autonomen fahrzeugen. *TU Chemnitz*, 2001.

[JU97] Simon J Julier and Jeffrey K Uhlmann. A new extension of the kalman filter to nonlinear systems. 3068:182—193, 1997.

[K+60] R. E Kalman et al. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

[KBO+06] N. Karlsson, E. Di Bernardo, J. Ostrowski, L. Goncalves, P. Pirjanian, and M. E Munich. The vSLAM algorithm for robust localization and mapping. page 24–29, 2006.

[Kon04] K. Konolige. Large-scale map-making. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, page 457–463, 2004.

[KS04] J. Kim and S. Sukkarieh. SLAM aided GPS/INS navigation in GPS denied and unknown environments. In *The 2004 International Symposium on GNSS/GPS, Sydney*, page 6–8, 2004.

[KS07] J. Kim and S. Sukkarieh. Real-time implementation of airborne inertial-SLAM. *Robotics and Autonomous Systems*, 55(1):62–71, 2007.

[KSD+09] R. K\ümmerle, B. Steder, C. Dornhege, M. Ruhnke, G. Grisetti, C. Stachniss, and A. Kleiner. On measuring the accuracy of SLAM algorithms. *Autonomous Robots*, 27(4):387–407, 2009.

[LALM07] R. Lakaemper, N. Adluru, L. J Latecki, and R. Madhavan. Multi robot mapping using force field simulation. *Journal of Field Robotics*, 24(8-9):747–762, 2007.

[LM97] F. Lu and E. Milios. Robot pose estimation in unknown environments by matching 2d range scans. *Journal of Intelligent and Robotic Systems*, 18(3):249–275, 1997.

[LNHS05] K. Lingemann, A. N\üchter, J. Hertzberg, and H. Surmann. High-speed laser localization for mobile robots. *Robotics and Autonomous Systems*, 51(4):275–296, 2005.

[Low99] D. G Lowe. Object recognition from local scale-invariant features. In *iccv*, page 1150, 1999.

[LSNH04] Kai Lingemann, H. Surmann, A. Nuchter, and J. Hertzberg. Indoor and outdoor localization for fast mobile robots. pages 2185–2190, Sendai, Japan, 2004.

[Mad49] W. G Madow. On the theory of systematic sampling, II. *The Annals of Mathematical Statistics*, 20(3):333–354, 1949.

[Men07] Marco Mengelkoch. *Implementieren des FastSLAM Algorithmus zur Kartenerstellung in Echtzeit*. PhD thesis, Universität Koblenz, Landau, 2007.

[MFO+06] Aaron Morris, Dave Ferguson, Zachary Omohundro, David Bradley, David Silver, Chris Baker, Scott Thayer, Chuck Whittaker, and William Whittaker. Recent developments in subterranean robotics. *Journal of Field Robotics*, 23(1):35–57, 2006.

[MR06] A. I Mourikis and S. I Roumeliotis. Analytical characterization of the accuracy of slam without absolute orientation measurements. In *Proc. Robotics: Science and Systems Conf*, August 2006.

[MSWS02] M. J Merrigan, E. R Swift, R. F Wong, and J. T Saffel. A refinement to the world geodetic system 1984 reference frame. In *Proceedings of the ION-GPS-2002*, 2002.

[MTKW02] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM: a factored solution to the simultaneous localization and mapping problem. page 593–598, 2002.

[MTKW03] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *International Joint Conference on Artificial Intelligence*, volume 18, page 1151–1156, 2003.

[MTS07] M. Montemerlo, S. Thrun, and B. Siciliano. *FastSLAM: A scalable method for the simultaneous localization and mapping problem in robotics*. Springer Verlag, 2007.

[NCC+07] P. Newman, M. Chandran-Ramesh, D. Cole, M. Cummins, A. Harrison, I. Posner, and D. Schroeter. Describing, navigating and recognising urban Spaces-Building an End-to-End SLAM system. 2007.

[NCH06] P. Newman, D. Cole, and K. Ho. Outdoor SLAM using visual appearance and laser ranging. page 1180–1187, 2006.

[NF06] S. M Nejad and K. Fasihi. A new design of laser phase-shift range finder independent of environ-mental conditions and thermal drift. In *Proceedings of the 9th Joint Conference on Information Sciences, JCIS*, volume 2006, 2006.

[NGNT03] J. Nieto, J. Guivant, E. Nebot, and S. Thrun. Real time data association for FastSLAM. volume 1, page 412–418, 2003.

[NLHS07] Andreas Nüchter, Kai Lingemann, Joachim Hertzberg, and Hartmut Surmann. 6D SLAM—3D mapping outdoor environments. *Journal of Field Robotics*, 24(8-9):699–722, 2007.

[NMTS05] V. Nguyen, A. Martinelli, N. Tomatis, and R. Siegwart. A comparison of line extraction algorithms using 2D laser rangefinder for indoor mobile robotics. In *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*, page 1929–1934, 2005.

[NSL+04] A. Nüchter, H. Surmann, K. Lingemann, J. Hertzberg, and S. Thrun. 6D SLAM with an application in autonomous mine mapping. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 2, page 1998–2003, 2004.

[PRSF00] E. Prassler, A. Ritter, C. Schaeffer, and P. Fiorini. A short history of cleaning robots. *Autonomous Robots*, 9(3):211–226, 2000.

[RFM10] R. Rouveure, P. Faure, and M. O. Monod. Radar-based SLAM without odometric sensor. 2010.

[RL01] S. Rusinkiewicz and M. Levoy. Efficient variants of the ICP algorithm. In *3dim*, page 145, 2001.

[SAS07] P. Scovanner, S. Ali, and M. Shah. A 3-dimensional sift descriptor and its application to action recognition. In *Proceedings of the 15th international conference on Multimedia*, page 357–360, 2007.

[Sep74] T. O Seppelin. The department of defense world geodetic system 1972. Technical report, WORLD GEODETIC SYSTEM COMMITTEE WASHINGTON DC, 1974.

[SH09] B. Steux and O. El Hamzaoui. CoreSLAM: a SLAM algorithm in less than 200 lines of c code. *Mines ParisTech-Center of Robotics, Paris*, 2009.

[SHB04] C. Stachniss, D. Hahnel, and W. Burgard. Exploration with active loop-closing for FastSLAM. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 2, page 1505–1510, 2004.

[SLL05] S. Se, D. Lowe, and J. Little. Vision-based mobile robot localization and mapping using scale-invariant features. volume 2, page 2051–2058, 2005.

[SM06] J. Z. Sasiadek and A. Monjazeb. A comparison between EKF-SLAM and Fast-SLAM. 2006.

[SMDW11] Matthias R. Schmid, Mirko Mählisch, Jürgen Dickmann, and Hans-Joachim Wünsche. Straight feature based Self-Localization for urban scenarios. 2011.

[TBF05] S. Thrun, W. Burgard, and D. Fox. *Probabilistic robotics*. MIT Press, Camebridge, MA, 2005.

[TDD+00] S. Thayer, B. Digney, M. Diaz, A. Stentz, B. Nabbe, and M. Hebert. Distributed robotic mapping of extreme environments. In *Proceedings of SPIE*, volume 4195, 2000.

[Thr02] S. Thrun. Particle filters in robotics. In *Proceedings of the 17th Annual Conference on Uncertainty in AI (UAI)*, volume 1, 2002.

[TNNL02] J. D Tardós, J. Neira, P. M Newman, and J. J Leonard. Robust mapping and localization in indoor environments using sonar data. *The International Journal of Robotics Research*, 21(4):311, 2002.

[TTW+04] S. Thrun, S. Thayer, W. Whittaker, C. Baker, W. Burgard, D. Ferguson, D. Hahnel, D. Montemerlo, A. Morris, Z. Omohundro, et al. Autonomous exploration and mapping of abandoned mines. *Robotics & Automation Magazine, IEEE*, 11(4):79–91, 2004.

[Wen07] J. Wendel. *Integrierte Navigationssysteme: Sensordatenfusion, GPS und Inertiale Navigation*. Oldenbourg Wissenschaftsverlag, 2007.

[Woo07] O. J Woodman. An introduction to inertial navigation. *University of Cambridge, Computer Laboratory, Tech. Rep. UCAMCL-TR-696*, 2007.

[WSBC10] Jochen Welle, Dirk Schulz, Thomas Bachran, and Armin B. Cremers. Optimization techniques for laser-based 3D particle filter SLAM. pages 3525–3530, Anchorage, AK, USA, 2010.

[Yam04] B. Yamauchi. PackBot: a versatile platform for military robotics. In *Proceedings of SPIE*, volume 5422, page 228–237, 2004.

[Yia93] P. N Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, page 311–321, 1993.

FastSLAM 1.0 pseudo code

A

This is the pseudo code by Montemerlo and Thrun [MTS07] explaining how FastSLAM works. It is simplified in the regard that just one observation is taken at every time step and leaves the landmark detection for the reader to fill in.

FastSLAM(S_{t-1}, z_t, R_t, u_t)
$S_t = S_{aux} = \emptyset$
for $m = 1$ to M
  retrieve m-th particle //loop: all particles
    $\left\langle s_{t-1}^{[m]}, N_{t-1}^{[m]}, \mu_{1,t-1}^{[m]}, \Sigma_{1,t-1}^{[m]}, \ldots, \mu_{N_{t-1}^{[m]},t-1}^{[m]}, \Sigma_{N_{t-1}^{[m]},t-1}^{[m]} \right\rangle$ from S_{t-1}
  draw $s_t^{[m]} \sim p\left(s_t | s_{t-1}^{[m]}, u_t\right)$ //sample new pose
  for $n = 1$ to $N_{t-1}^{[m]}$ //loop over potential data associations
    $G_{\theta,n} = \nabla_{\theta_n} g\left(\theta_n, s_t\right)\big|_{\theta_n = \mu_{n,t-1}^{[i]}; s_t = s_t^{[i]}}$
    $\hat{z}_{n,t} = g\left(s_t^{[m]}, \mu_{n,t-1}^{[m]}\right)$
    $Z_{n,t} = G_{\theta,n} \Sigma_{n,t-1}^{[m]} G_{\theta,n}^T + R_t$
    $p_{n,t}^{[m]} = \left|2\pi Z_{n,t}\right|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}\left(z_t - \hat{z}_{n,t}\right)^T Z_{n,t}^{-1}\left(z_t - \hat{z}_{n,t}\right)\right\}$
  end for
  $p_{N_{t-1}^{[m]}+1,t}^{[m]} = p_0$
  $\hat{n}_t = \text{argmax}_n\ p_{n,t}^{[m]}$ or draw $\hat{n}_t$ with prob. $\propto p_{n,t}^{[m]}$ //pick data assoc.
  if $\hat{n}_t = N_{t-1}^{[m]} + 1$ //is new feature?
    $N_t^{[m]} = N_{t-1}^{[m]} + 1$
    $\mu_{\hat{n}_t,t}^{[m]} = g^{-1}\left(s_t^{[m]}, \hat{z}_{\hat{n}_t,t}\right)$
    $\Sigma_{\hat{n}_t,t}^{[m]} = \left(G_{\theta,\hat{n}_t}^T R^{-1} G_{\theta,\hat{n}_t}\right)^{-1}$
  else //or is a known feature?
    $N_t^{[m]} = N_{t-1}^{[m]}$
    $K_{\hat{n}_t,t} = \Sigma_{\hat{n}_t,t-1} G_{\theta,\hat{n}_t}^T Z_{\hat{n}_t,t}^{-1}$
    $\mu_{\hat{n}_t,t}^{[m]} = \mu_{\hat{n}_t,t-1}^{[m]} + K_{\hat{n}_t,t}\left(z_t - \hat{z}_{\hat{n}_t,t}\right)$
    $\Sigma_{\hat{n}_t,t}^{[m]} = \left(I - K_{\hat{n}_t,t} G_{\theta,\hat{n}_t}\right) \Sigma_{\hat{n}_t,t-1}^{[m]}$

end if
for $n = 1$ to $N_t^{[m]}$ do //unobserved features
if $n \neq \hat{n}_t$
$\mu_{\theta_n,t}^{[m]} = \mu_{\theta_n,t-1}^{[m]}$
$\Sigma_{\theta_n,t}^{[m]} = \Sigma_{\theta_n,t-1}^{[m]}$
end if
end for
$w_t^{[m]} = p_{\hat{n}_t,t}^{[m]}$ //save weighted particle
add $\left\langle s_t^{[m]}, N_t^{[m]}, \mu_{1,t}^{[m]}, \Sigma_{1,t}^{[m]}, \ldots, \mu_{N_t^{[m]},t}^{[m]}, \Sigma_{N_t^{[m]},t}^{[m]}, w_t^{[m]} \right\rangle$ to S_{aux}
end for
for $m = 1$ to M //resample
draw random particle from S_{aux} with probability $\propto w_t^{[m]}$
add new particle to S_t
end for
return S_t

AppBase configuration

B

The AppBase is configured using XML files describing the workers used and the dataflow. This is the file used for the FastSLAM implementation.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE appbaseconf SYSTEM "appbaseconf.dtd">
<appbaseconf>
  <system/>
  <processinggraph>
  <configlist>
    <config type="SOURCE" name="SOURCE">
    <param name="AbortOnOverflow">false</param>
    </config>
    <config type="DRAIN" name="DRAIN">
    <param name="EnableBuffering">false</param>
    </config>
    <config type="SliceWorker" name="Slicing">
    <param name="numberOfScansInSlice">20</param>
    <param name="numberOfScansOverlap">15</param>
    <param name="DeviceID">42</param>
    <param name="GroundLabeled">true</param>
    </config>
    <config type="SliceWorker" name="fullSlicing">
    <param name="numberOfScansInSlice">20</param>
    <param name="numberOfScansOverlap">5</param>
    <param name="DeviceID">42</param>
    <param name="GroundLabeled">false</param>
    </config>
    <config type="RoadSurfaceRecognitionWorker" name="RoadBorderRec">
    <param name="maxDifStreet">0.22 m</param>
    </config>
    <config type="EPWAnalyzeWorker" name="EPWRec">
    </config>
   <config type="ScanEPWCutWorker" name="ScanEPWCutWorker">
      <param name="Scanner1_min">0.16 m</param>
      <param name="Scanner2_min">0.10 m</param>
      <param name="Scanner3_min">0.16 m</param>
    </config>
    <config type="SLAMWorker" name="Fastslam">
    <param name="DeviceID">42</param>
    <param name="maxNumOfParticles">10</param>
    <param name="AlphaYaw1">0.001</param>
    <param name="AlphaPitch1">0.001</param>
    <param name="AlphaYaw2">0.001</param>
    <param name="AlphaPitch2">0.001</param>
    <param name="AlphaRoll">0.001</param>
    <param name="AlphaTrans">0.001</param>
    <param name="FYaw1">0</param>
    <param name="FYaw2">0</param>
    <param name="Threshold">0.001</param>
```

```
47
48      </config>
49    </configlist>
50    <connectionlist>
51      <connection from="SOURCE" type="Scan" to="fullSlicing"/>
52      <connection from="fullSlicing" type="Scan" to="Fastslam"/>
53      <connection from="SOURCE" type="Scan" to="RoadBorderRec"/>
54      <connection from="RoadBorderRec" type="Scan" to="EPWRec"/>
55      <connection from="EPWRec" type="Scan" to="ScanEPWCutWorker"/>
56      <connection from="ScanEPWCutWorker" type="Scan" to="Slicing"/>
57      <connection from="Slicing" type="Scan" to="Fastslam"/>
58      <connection from="Slicing" type="Scan" to="DRAIN"/>
59      <connection from="SOURCE" type="CANMessage" to="DRAIN"/>
60      <connection from="SOURCE" type="VehicleState" to="Slicing"/>
61      <connection from="Slicing" type="VehicleState" to="DRAIN"/>
62      <connection from="SOURCE" type="Image" to="DRAIN"/>
63      <connection from="SOURCE" type="PositionWGS84" to="Slicing"/>
64      <connection from="SOURCE" type="PositionWGS84" to="fullSlicing"/>
65      <connection from="Slicing" type="PositionWGS84" to="Fastslam"/>
66      <connection from="Slicing" type="PositionWGS84" to="DRAIN"/>
67    </connectionlist>
68    </processinggraph>
69  </appbaseconf>
```

Preprocessing workers

C

This appendix lists the sourcecode of all the workers used for preprocessing the scan and position data before feeding it to the FastSLAM worker.

C.1 Street border cutter

Almost all streets have a curb which can be detected easily and used as a line for cutting off the scanpoints beyond the streets.

```
1  // RoadSurfaceRecognitionWorker.hpp
2  // created: 2011/01/26
3  // author: Jan Gries
4
5  #ifndef RoadSurfaceRecognitionWorker_HPP
6  #define RoadSurfaceRecognitionWorker_HPP
7
8  // includes for class generation
9  #include "ibeobasic/ibeobasicdecl.hpp"
10 #include <IbeoAPI/Configurable.hpp>
11 #include "ibeograph/Preferences.hpp"
12
13 // includes for drains and sources
14 #include "ibeograph/drain/ScanDrain.hpp"
15 #include "ibeograph/source/ScanSource.hpp"
16
17 // includes for data types.
18 #include <IbeoAPI/Scan.hpp>
19 #include <vector>
20
21 class Scan;
22
23 #define RoadSurfaceRecognitionWorker_VERSION "1.0"
24
25 namespace ibeo {
26 namespace appbase {
27 namespace worker {
28
29 /**
30  * This worker detects the curb of a street and cuts off all scanpoints
31  * beyond the detected curbline.
32  *
33  * Output Data: Filtered Scan object
34  *
35  * Author(s): Jan Girlich
36  */
37 class IBEOBASICDECL RoadSurfaceRecognitionWorker : public Configurable
```

```
38                  , public ibeo::appbase::drain::ScanDrain
39                  , public ibeo::appbase::source::ScanSource
40 {
41 public:
42   /**
43    * Constructor
44    */
45   RoadSurfaceRecognitionWorker(const ibeo::Preferences& preferences, const std::string&
       configBlockName);
46
47   /**
48    * destructor
49    */
50   virtual ~RoadSurfaceRecognitionWorker();
51
52   static const std::string& getDefaultTypeName() { return CONFIG_TYPE; }
53   virtual void setScan(Scan& scan);
54
55 // member variables
56 private:
57
58
59   void medianFilter3();
60   /** Name of the type of the configuration block for this file. */
61   static const std::string CONFIG_TYPE;
62   float m_minZ;
63   float m_maxZ;
64   float m_maxDifStreet;
65   std::vector<size_t> m_deviceIDs;
66   std::vector<Scan::iterator> m_minScans;
67   std::vector<Scan::iterator> m_lastScans;
68   Scan m_currentScan;
69   Scan m_letzterScan;
70 };
71
72 } // worker
73 } // appbase
74 } // ibeo
75
76 #endif
```

```
1  // RoadSurfaceRecognitionWorker.cpp
2  // created: 2011/01/26
3  // author: Jan Gries
4
5  #include "RoadSurfaceRecognitionWorker.hpp"
6  #include <cmath>
7  #include <boost/circular_buffer.hpp>
8  #include "Point3D.hpp"
9
10 namespace ibeo{
11 namespace appbase{
12 namespace worker{
13
14 const std::string RoadSurfaceRecognitionWorker::CONFIG_TYPE = "RoadSurfaceRecognitionWorker";
15
16 RoadSurfaceRecognitionWorker::RoadSurfaceRecognitionWorker(const ibeo::Preferences& preferences,
       const std::string& objectname)
17 : Configurable(getDefaultTypeName(), objectname)
18 {
19   define (new ParamLength ("minZ", m_minZ, "Minimum to keep on Z axis", "-0.3 m"));
20   define (new ParamLength ("maxZ", m_maxZ, "Maximum to keep on Z axis", "0.4 m"));
21   define (new ParamLength ("maxDifStreet", m_maxDifStreet, "Maximum slope of the street", "0.27 m"))
         ;
```

```
22
23    // Load values from the config file, fill them into our
24    // parameters, and throw an exception if something was invalid.
25    fillValuesValidateOrExcept(preferences.findByTypeAndName(getDefaultTypeName(), objectname));
26  }
27
28  RoadSurfaceRecognitionWorker::~RoadSurfaceRecognitionWorker()
29  {
30  }
31
32  bool isTooHighOrLow(const ScanPoint& sp, float minZ, float maxZ)
33  {
34    return !((sp.getZ() > minZ) && (sp.getZ() < maxZ));
35  }
36
37  static bool hasSmallerY (const ScanPoint& P1, const ScanPoint& P2)
38  {
39    return (P1.getY() > P2.getY());
40  }
41
42  static bool hasSmallerHA (const ScanPoint& P1, const ScanPoint& P2)
43  {
44    return (P1.getHAngle() > P2.getHAngle());
45  }
46
47  static bool hasSmallerVA (const ScanPoint& P1, const ScanPoint& P2)
48  {
49    return (P1.getVAngle() > P2.getVAngle());
50  }
51
52  void RoadSurfaceRecognitionWorker::setScan(Scan& scan)
53  {
54    m_deviceIDs.clear();
55    m_minScans.clear();
56
57    if (!scan.isVehicleCoordinates()){
58      bool success;
59      success = scan.transformToScannerCoordinates();
60      if (!success){
61        traceError("") << "transformToVehicleCoordinates() failed";
62        return;
63      }
64    }
65
66    // create output scan without scanning points
67    Scan outputScan(scan);
68    outputScan.resize(0);
69
70    // create current scan with points near the ground
71    m_currentScan = Scan(scan);
72    m_currentScan.getPointList().clear();
73    std::remove_copy_if(scan.begin(), scan.end(), std::back_inserter(m_currentScan.getPointList()),
          boost::bind(&isTooHighOrLow, _1, m_minZ, m_maxZ));
74
75    hasSmallerHA(*scan.getPointList().begin(),*scan.getPointList().begin());
76    hasSmallerY(*scan.getPointList().begin(),*scan.getPointList().begin());
77    hasSmallerVA(*scan.getPointList().begin(),*scan.getPointList().begin());
78    std::sort(m_currentScan.getPointListBegin(), m_currentScan.getPointListEnd(), hasSmallerY);
79
80    std::vector<ibeo::ScanPoint>::iterator iter;
81    //looking for device used
82    for (std::vector<ibeo::ScannerInfo>::iterator iterInfo = m_currentScan.getScannerInfos().begin();
          iterInfo != m_currentScan.getScannerInfos().end(); ++iterInfo){
83      m_deviceIDs.push_back((size_t)iterInfo->getDeviceID());
84
```

```
85        bool foundID = false;
86        for ( iter = m_currentScan.getPointList().begin(); iter != m_currentScan.getPointList().end()
              ; iter++){
87          if (iter->getEchoNum() == 0){
88            if ((size_t)(iter->getDeviceID()) == (m_deviceIDs.back())){
89              foundID =true;
90              break;
91            }
92          }
93        }
94        if (!foundID) traceError("") << "no ScanPoint of Scanner in Infovector! \n";
95
96        m_minScans.push_back(iter);
97    }
98
99    //looking for starting point in the middle of the scan, because we assume that the car is on the
          road
100   for ( iter = m_currentScan.getPointList().begin(); iter!=m_currentScan.getPointList().end(); ++
          iter){
101     if (iter->getEchoNum() == 0){
102       size_t indexOfID;
103
104       bool foundID = false;
105       for ( indexOfID=0; indexOfID != m_deviceIDs.size(); ++indexOfID){
106         if (m_deviceIDs[indexOfID] == (size_t)(iter->getDeviceID())){
107           foundID =true;
108           break;
109         }
110       }
111       if (!foundID) traceError("") << "ScanPoint of Scanner not in Infovector! \n";
112
113       if (( m_minScans[indexOfID]->getZ()-0.05) * (m_minScans[indexOfID]->getZ()-0.05)*8 + fabs(
              m_minScans[indexOfID]->getY()) > (iter->getZ()-0.05) * (iter->getZ()-0.05)*8 + fabs(iter
              ->getY()) ){
114         //-0.05 is typical for Seeland-Passat
115         m_minScans[indexOfID] = iter;
116       }
117     }
118   }
119
120   //looking for the boarder of the street starting at m_minScan Pointer
121   size_t sizeOfCb = 16;
122   for( size_t indexOfID=0 ; indexOfID < m_minScans.size() ; ++indexOfID ){
123     outputScan.getPointList().push_back(*m_minScans[indexOfID]);
124     size_t ready = 0;
125     boost::circular_buffer<std::vector<ibeo::ScanPoint>::iterator> cb(sizeOfCb);
126     for (size_t i=0; i< cb.capacity(); i++){
127       cb.push_back(m_minScans[indexOfID]);
128     }
129     int possibleCut = 0;
130     for ( iter=m_minScans[indexOfID]-- ; iter>=(m_currentScan.getPointList().begin()) ;--iter){
131       if (iter->getEchoNum() == 0){
132         if ( m_deviceIDs[indexOfID] == (size_t)(iter->getDeviceID())){
133           if (ready < cb.size()){
134             cb.push_back(iter);
135             bool isOutOfRange = false;
136             for (size_t i=1; i<cb.size(); i++){
137               isOutOfRange = isOutOfRange || fabs(cb[i-1]->getZ()-cb[i]->getZ()) > (m_maxDifStreet *
                      2);
138             }
139             if (isOutOfRange){
140               break;
141             }
142             ready++;
143           }
```

```
144           else{
145             if (ready == cb.size()){
146               for (size_t i=0; i<cb.size(); i++){
147                  outputScan.getPointList().push_back(*cb[i]);
148               }
149               ready++;
150             }
151
152             // Calculation of tangent out of cbuffer
153             Point3D averagePointFront(0.0,0.0,0.0);
154             Point3D averagePointRear(0.0,0.0,0.0);
155             size_t divFront = 0;
156             size_t divRear = 0;
157
158             for (size_t i=0; i<cb.size(); i++){
159               if (i< (cb.size()/2)) {
160                 averagePointFront += cb[i]->toPoint3D();
161                 divFront++;
162               }
163               else{
164                 averagePointRear += cb[i]->toPoint3D();
165                 divRear++;
166               }
167             }
168             averagePointFront /= divFront;
169             averagePointRear /= divRear;
170             double averageTangent = (averagePointRear.getZ() - averagePointFront.getZ()) / (
                    averagePointRear.getY() - averagePointFront.getY());
171             double newDivTangentPoint = iter->getZ() - (((iter->getY() - averagePointRear.getY()) *
                    averageTangent) + averagePointRear.getZ());
172
173             // loop termination condition calculation (find last scanpoint on road)
174             if (std::fabs( newDivTangentPoint) > m_maxDifStreet){
175               if (newDivTangentPoint > 0){
176                 if (possibleCut == 1) break;
177                 possibleCut++;
178               }
179               else{
180                 if (possibleCut == -1) break;
181                 possibleCut--;
182               }
183             }
184             else{
185               cb.push_back(iter);
186               possibleCut = 0;
187               outputScan.getPointList().push_back(*iter);
188             }
189           }
190         }
191       }
192     }
193   }
194
195   for( size_t indexOfID=0 ; indexOfID < m_minScans.size() ; ++indexOfID ){
196     outputScan.getPointList().push_back(*m_minScans[indexOfID]);
197     size_t ready = 0;
198     boost::circular_buffer<std::vector<ibeo::ScanPoint>::iterator> cb(sizeOfCb);
199     for (size_t i=0; i< cb.capacity(); i++){
200       cb.push_back(m_minScans[indexOfID]);
201     }
202     int possibleCut = 0;
203     std::vector<std::vector<ibeo::ScanPoint>::iterator> cache;
204     for ( iter=m_minScans[indexOfID]++ ; iter<(m_currentScan.getPointList().end()) ;++iter){
205       if (iter->getEchoNum() == 0){
206         if ( m_deviceIDs[indexOfID] == (size_t)(iter->getDeviceID())){
```

```
207           if (ready < cb.size()){
208             cb.push_back(iter);
209             bool isOutOfRange = false;
210             for (size_t i=1; i<cb.size(); i++){
211               isOutOfRange = isOutOfRange || fabs(cb[i-1]->getZ()-cb[i]->getZ()) > (m_maxDifStreet *
                      2);
212             }
213             if (isOutOfRange){
214               break;
215             }
216             ready++;
217           }
218           else{
219             if (ready == cb.size()){
220               for (size_t i=0; i<cb.size(); i++){
221                  outputScan.getPointList().push_back(*cb[i]);
222               }
223               ready++;
224             }
225
226             // Calculation of tangent out of cbuffer
227             Point3D averagePointFront(0.0,0.0,0.0);
228             Point3D averagePointRear(0.0,0.0,0.0);
229             size_t divFront = 0;
230             size_t divRear = 0;
231
232             for (size_t i=0; i<cb.size(); i++){
233               if (i< (cb.size()/2)) {
234                 averagePointFront += cb[i]->toPoint3D();
235                 divFront++;
236               }
237               else{
238                 averagePointRear += cb[i]->toPoint3D();
239                 divRear++;
240               }
241             }
242             averagePointFront /= divFront;
243             averagePointRear /= divRear;
244             double averageTangent = (averagePointRear.getZ() - averagePointFront.getZ()) / (
                    averagePointRear.getY() - averagePointFront.getY());
245             double newDivTangentPoint = iter->getZ() - (((iter->getY() - averagePointRear.getY()) *
                    averageTangent) + averagePointRear.getZ());
246
247             // loop termination condition calculation (find last scanpoint on road)
248             if (std::fabs( newDivTangentPoint) > m_maxDifStreet){
249               if (newDivTangentPoint > 0){
250                 if (possibleCut == 2) break;
251                 possibleCut++;
252                 cache.push_back(iter);
253               }
254               else{
255                 if (possibleCut == -2) break;
256                 possibleCut--;
257                 cache.push_back(iter);
258               }
259             }
260             else{
261               cb.push_back(iter);
262               possibleCut = 0;
263               while (!cache.empty()){
264                 outputScan.getPointList().push_back(*(cache.back()));
265                 cache.pop_back();
266               }
267               outputScan.getPointList().push_back(*iter);
268             }
```

```
269           }
270         }
271       }
272     }
273   }
274
275   m_signalScan(outputScan);
276 }
277
278 // this function implements a low pass filter, but is not used anymore
279 void RoadSurfaceRecognitionWorker::medianFilter3()
280 {
281   if (m_currentScan.getPointList().size() < 3){
282     traceError("") << "not enough points in scan \n";
283     return;
284   }
285
286   for( size_t indexOfID=0 ; indexOfID < m_minScans.size() ; ++indexOfID ){
287     // circular_buffer with 2 times the first point of the scan to avoid null-pointer access
288     boost::circular_buffer<std::vector<ibeo::ScanPoint>::iterator> cb(3);
289     for (std::vector<ibeo::ScanPoint>::iterator iter = m_currentScan.getPointList().begin() ; iter
          != m_currentScan.getPointList().end(); ++iter){
290       if ( ((size_t)iter->getDeviceID()) == ((size_t)m_deviceIDs[indexOfID]) ){
291         cb.push_back(iter);
292         cb.push_back(iter);
293       }
294     }
295     boost::circular_buffer<double> tempZ(2);
296     tempZ.push_back(cb[0]->getZ());
297     tempZ.push_back(cb[0]->getZ());
298
299     for (std::vector<ibeo::ScanPoint>::iterator iter = (m_currentScan.getPointList().begin())+1 ;
          iter != (m_currentScan.getPointList().end()) ; ++iter){
300       if ( ((size_t)iter->getDeviceID()) == ((size_t)m_deviceIDs[indexOfID] )){
301
302         cb.push_back(iter);
303         bool is0bigger1 = cb[0]->getZ() > cb[1]->getZ();
304         bool is1bigger2 = cb[1]->getZ() > cb[2]->getZ();
305         bool is0bigger2 = cb[0]->getZ() > cb[2]->getZ();
306
307         if ((!is0bigger1 && is0bigger2) || (is0bigger1 && !is0bigger2)){
308           tempZ.push_back(cb[0]->getZ());
309         }
310         else if ((!is1bigger2 && !is0bigger1) || (is1bigger2 && is0bigger1)){
311           tempZ.push_back(cb[1]->getZ());
312         }
313         else {
314           tempZ.push_back(cb[2]->getZ());
315         }
316         cb[0]->setCartesian(cb[0]->getX(),cb[0]->getY(), tempZ[0]);
317       }
318     }
319   }
320 }
321
322 }
323 }
324 }
```

C.2 Street surface marking detection

The street surface markings found on streets do reflect the light from the LRFs very differently than the surrounding pavement and thus make easily identifiable landmarks.

```
// EPWAnalyzeWorker.hpp
// created: 2011-01-26
// author: Jan Girlich

#ifndef EPWAnalyzeWorker_HPP
#define EPWAnalyzeWorker_HPP

// includes for class generation
#include "ibeobasic/ibeobasicdecl.hpp"
#include <IbeoAPI/Configurable.hpp>
#include "ibeograph/Preferences.hpp"

// includes for drains and sources
#include "ibeograph/drain/ScanDrain.hpp"
#include "ibeograph/source/ScanSource.hpp"

// includes for data types.
#include <IbeoAPI/Scan.hpp>
#include <IbeoAPI/Point3D.hpp>

// standard C++ includes
#include <vector>
#include <set>

class Scan;

#define EPWAnalyzeWorker_VERSION "1.0"

namespace ibeo {
namespace appbase {
namespace worker {

/**
 * This worker calculates the first or second derivative between
 * scanpoints and writes the result in the echo pulse width of the
 * scanpoints.
 *
 * Output Data: Filtered Scan object
 *
 * Author(s): Jan Girlich
 */
class IBEOBASICDECL EPWAnalyzeWorker : public Configurable
                  , public ibeo::appbase::drain::ScanDrain
                  , public ibeo::appbase::source::ScanSource
{
public:
  /**
   * Constructor
   */
  EPWAnalyzeWorker(const ibeo::Preferences& preferences, const std::string& configBlockName);

  /**
   * destructor
   */
  virtual ~EPWAnalyzeWorker();

  static const std::string& getDefaultTypeName() { return CONFIG_TYPE; }
```

```
58
59    virtual void setScan(Scan& scan);
60
61  // member variables
62  private:
63    /** Name of the type of the configuration block for this file. */
64    static const std::string CONFIG_TYPE;
65
66    Scan lowPassFilter(Scan scan);
67    float m_factor;
68          UINT8 m_differential;
69  };
70
71  } // worker
72  } // appbase
73  } // ibeo
74
75  #endif
```

```
1   // EPWAnalyzeWorker.cpp
2   // created: 2011-01-26
3   // author: Jan Girlich
4
5   #include "EPWAnalyzeWorker.hpp"
6   #include <IbeoAPI/Scan.hpp>
7   #include <IbeoAPI/Segment.hpp>
8
9   namespace ibeo{
10  namespace appbase{
11  namespace worker{
12
13  const std::string EPWAnalyzeWorker::CONFIG_TYPE = "EPWAnalyzeWorker";
14
15  EPWAnalyzeWorker::EPWAnalyzeWorker(const ibeo::Preferences& preferences, const std::string&
        objectname)
16  : Configurable(getDefaultTypeName(), objectname)
17  {
18    define (new ParamFloat  ("factor", m_factor, "Factor by which the EPW is multiplied", "10"));
19    define (new ParamUINT8  ("differential", m_differential, "differential ", "2"));
20    // Load values from the config file, fill them into our
21    // parameters, and throw an exception if something was invalid.
22    fillValuesValidateOrExcept(preferences.findByTypeAndName(getDefaultTypeName(), objectname));
23  }
24
25  EPWAnalyzeWorker::~EPWAnalyzeWorker()
26  {
27  }
28
29  bool spReflection (const ScanPoint& sp)
30  {
31    return (sp.getSubchannel() != 0);
32  }
33
34  bool notMatchDeviceID (const ScanPoint& sp, const int& spDevID)
35  {
36    return (sp.getDeviceID() != spDevID);
37  }
38
39  // Sort criterion. This method is passed to std::sort() to sort scan
40  // points by descending cartesian Y-coordinate.
41  static bool isYCoordinate (const ScanPoint& P1, const ScanPoint& P2)
42  {
43    return (P1.getY() > P2.getY());
44  }
```

```
45
46  void EPWAnalyzeWorker::setScan(Scan& scan)
47  {
48    // Create a Scan object with all meta data, but no ScanPoints
49    Scan limitedScans(scan);
50    limitedScans.resize(0);
51
52    // Remove all Scanpoints which are not the first reflection
53    std::remove_copy_if(scan.begin(), scan.end(), std::back_inserter(limitedScans.getPointList()),
          boost::bind(&spReflection, _1));
54
55    // collecting all DeviceIDs
56    std::set<int> deviceIDs;
57    for (std::vector<ibeo::ScannerInfo>::iterator it = limitedScans.getScannerInfos().begin(); it !=
          limitedScans.getScannerInfos().end(); ++it)
58      deviceIDs.insert(it->getDeviceID());
59
60    // Split up Scans by DeviceIDs
61    std::vector<Scan> allScans;
62    for (std::set<int>::iterator devIDit = deviceIDs.begin(); devIDit != deviceIDs.end(); ++devIDit)
63    {
64      Scan newDevice(scan);
65      newDevice.resize(0);
66      std::remove_copy_if(limitedScans.begin(), limitedScans.end(), std::back_inserter(newDevice.
            getPointList()), boost::bind(&notMatchDeviceID, _1, *devIDit));
67
68      if (newDevice.getNumPoints() < 3)
69      {
70        traceWarning("") << "Scan has less than three ScanPoints per Scanner and is dropped." << std::
              endl;
71      }
72      else
73      {
74        // Sort new Scan object by cartesian Y-coordinate
75        std::sort(newDevice.getPointListBegin(), newDevice.getPointListEnd(), isYCoordinate);
76
77        allScans.push_back(newDevice);
78      }
79    }
80
81    // Get an iterator over all Scans
82    Scan::iterator it;
83
84    // For every scanners Scan object go through all ScanPoints and
85    // detect road surface markings
86    Scan outputScan(scan);
87    outputScan.resize(0);
88
89    if (m_differential == (UINT8)2)
90    {
91      for (std::vector<Scan>::iterator scanIt = allScans.begin(); scanIt != allScans.end(); ++scanIt)
92      {
93        // Vector containing the recent ScanPoints EchoWidths
94        std::vector<float> recentSPs;
95
96        // initialize iterator over current Scan object to fill
97        it = (*scanIt).begin();
98
99        // Add three Scanpoints to the recentSPs-vector to prepare for the
100       // following loop
101       for (int i=0; i<3; ++i)
102       {
103         recentSPs.push_back((*it).getEchoWidth());
104         ++it;
105       }
```

```
106
107       while( it != (*scanIt).end() ) {
108
109         // Set the echoWidth to the magnified 2nd derivation
110         (it-3)->setEchoWidth(m_factor * fabs( fabs(recentSPs[0]-recentSPs[1]) - fabs(recentSPs[1] -
                recentSPs[2]) ));
111
112         // iterate and move elements around in the recentSPs vector
113         ++it;
114         recentSPs.push_back((*it).getEchoWidth());
115         recentSPs.erase(recentSPs.begin());
116       }
117       // Delete the last three scanpoints in the result
118       (*scanIt).getPointList().erase((*scanIt).end()-3, (*scanIt).end());
119       outputScan.addScan(*scanIt);
120     }
121   }
122
123   if (m_differential == (UINT8)1)
124   {
125     for (std::vector<Scan>::iterator scanIt = allScans.begin(); scanIt != allScans.end(); ++scanIt)
126     {
127       // Vector containing the recent ScanPoints EchoWidths
128       std::vector<float> recentSPs;
129
130       // initialize iterator over current Scan object to fill
131       it = (*scanIt).begin();
132
133       // Add three Scanpoints to the recentSPs–vector to prepare for the
134       // following loop
135       for (int i=0; i<2; ++i)
136       {
137         recentSPs.push_back((*it).getEchoWidth());
138         ++it;
139       }
140
141       while( it != (*scanIt).end() )
142       {
143         // Set the echoWidth to the magnified 2nd derivation
144         (it-2)->setEchoWidth(m_factor * fabs(recentSPs[0]-recentSPs[1]));
145
146         // iterate and move elements around in the recentSPs vector
147         ++it;
148         recentSPs.push_back((*it).getEchoWidth());
149         recentSPs.erase(recentSPs.begin());
150       }
151       // Delete the last two scanpoints in the result
152       (*scanIt).getPointList().erase((*scanIt).end()-2, (*scanIt).end());
153       outputScan.addScan(*scanIt);
154     }
155   }
156   m_signalScan(outputScan);
157 }
158 }
159 }
160 }
```

C.3 Echo pulse width cutter

After calculating the echo pulse widths in the street surface marking detector worker this worker removes the scanpoints below a configurable threshold.

```
// ScanEPWCutWorker.hpp
// created: 2011-01-26
// author: Jan Girlich

#ifndef ScanEPWCutWorker_HPP
#define ScanEPWCutWorker_HPP

// includes for class generation
#include "ibeobasic/ibeobasicdecl.hpp"
#include <IbeoAPI/Configurable.hpp>
#include "ibeograph/Preferences.hpp"

// includes for drains and sources
#include "ibeograph/drain/ScanDrain.hpp"
#include "ibeograph/source/ScanSource.hpp"

// includes for data types.
#include <IbeoAPI/Scan.hpp>

class Scan;

#define ScanEPWCutWorker_VERSION "1.0"

namespace ibeo {
namespace appbase {
namespace worker {

/**
 * This worker removes any point from scans whose echo pulse width is
 * not within a given span.
 *
 * Output Data: Filtered Scan object
 *
 * Author(s): Jan Girlich
 */
class IBEOBASICDECL ScanEPWCutWorker : public Configurable
                  , public ibeo::appbase::drain::ScanDrain
                  , public ibeo::appbase::source::ScanSource
{
public:
  /**
   * Constructor
   */
  ScanEPWCutWorker(const ibeo::Preferences& preferences, const std::string& configBlockName);

  /**
   * destructor
   */
  virtual ~ScanEPWCutWorker();

  static const std::string& getDefaultTypeName() { return CONFIG_TYPE; }

  virtual void setScan(Scan& scan);

// member variables
private:
  /** Name of the type of the configuration block for this file. */
  static const std::string CONFIG_TYPE;
```

```
59
60    float m_scanner1_min;
61    float m_scanner2_min;
62    float m_scanner3_min;
63    float m_scanner1_max;
64    float m_scanner2_max;
65    float m_scanner3_max;
66  };
67
68  } // worker
69  } // appbase
70  } // ibeo
71
72  #endif
```

```
1   // ScanEPWCutWorker.cpp
2   // created: 2011-01-26
3   // author: Jan Girlich
4
5   #include "ScanEPWCutWorker.hpp"
6
7   #include <IbeoAPI/Scan.hpp>
8
9   namespace ibeo{
10  namespace appbase{
11  namespace worker{
12
13  const std::string ScanEPWCutWorker::CONFIG_TYPE = "ScanEPWCutWorker";
14
15  ScanEPWCutWorker::ScanEPWCutWorker(const ibeo::Preferences& preferences, const std::string&
        objectname)
16  : Configurable(getDefaultTypeName(), objectname)
17  {
18    define (new ParamLength ("Scanner1_min", m_scanner1_min, "Minimum EPW to keep", "0.0 m"));
19    define (new ParamLength ("Scanner2_min", m_scanner2_min, "Minimum EPW to keep", "0.0 m"));
20    define (new ParamLength ("Scanner3_min", m_scanner3_min, "Minimum EPW to keep", "0.0 m"));
21    define (new ParamLength ("Scanner1_max", m_scanner1_max, "Maximum EPW to keep", "1000000000.0 m"))
          ;
22    define (new ParamLength ("Scanner2_max", m_scanner2_max, "Maximum EPW to keep", "1000000000.0 m"))
          ;
23    define (new ParamLength ("Scanner3_max", m_scanner3_max, "Maximum EPW to keep", "1000000000.0 m"))
          ;
24
25    // Load values from the config file, fill them into our
26    // parameters, and throw an exception if something was invalid.
27    fillValuesValidateOrExcept(preferences.findByTypeAndName(getDefaultTypeName(), objectname));
28  }
29
30  ScanEPWCutWorker::~ScanEPWCutWorker()
31  {
32  }
33
34  bool spBetweenValues(const ScanPoint& sp, float s1min, float s1max, float s2min, float s2max, float
        s3min, float s3max)
35  {
36    bool bad = true;
37
38    if (((int)sp.getDeviceID() == 1) && (sp.getEchoWidth() > s1min) && (sp.getEchoWidth() < s1max))
39      bad = false;
40    if (((int)sp.getDeviceID() == 2) && (sp.getEchoWidth() > s2min) && (sp.getEchoWidth() < s2max))
41      bad = false;
42    if (((int)sp.getDeviceID() == 3) && (sp.getEchoWidth() > s3min) && (sp.getEchoWidth() < s3max))
43      bad = false;
44
```

```
45    return bad;
46  }
47
48  void ScanEPWCutWorker::setScan(Scan& scan)
49  {
50    Scan limitedScans(scan);
51
52    limitedScans.resize(0);
53
54    // add the points with EPW levels in the expected range
55    std::remove_copy_if(scan.begin(), scan.end(), std::back_inserter(limitedScans.getPointList()),
          boost::bind(&spBetweenValues, _1, m_scanner1_min, m_scanner1_max, m_scanner2_min,
          m_scanner2_max, m_scanner3_min, m_scanner3_max));
56
57    m_signalScan(limitedScans);
58  }
59
60  }
61  }
62  }
```

C.4 Coordinate conversion

Converting spheric to cartesian coordinates and vice versa.

```
1   // coordconversion.hpp
2   // Copyright (c) Ibeo Automobile Sensor GmbH, 2011
3   // created: 2011-02-24
4   // author: Jan Girlich
5
6   #ifndef IBEO_FASTSLAM_COORDCONVERSION_HPP
7   #define IBEO_FASTSLAM_COORDCONVERSION_HPP
8
9   #include <iostream>
10  #include <math.h>
11  #include <assert.h>
12
13  using namespace std;
14  namespace ibeo {
15  namespace appbase {
16  namespace worker {
17
18  struct sphere
19  {
20    float rho;
21    float theta;
22    float phi;
23  } ;
24
25  struct cart
26  {
27    float x;
28    float y;
29    float z;
30  } ;
31
32
33  /// This class converts 3D cartesian coordinats to spherical coordinates
34  /// and vice versa. For details see:
35  /// http://www.math.montana.edu/frankw/ccp/multiworld/multipleIVP/spherical/learn.htm
```

```
36 class coordconversion
37 {
38 public:
39   coordconversion() {}
40
41   sphere cart2sphere(cart c)
42   {
43     sphere s;
44     if ((c.x == 0) && (c.y == 0) && (c.z == 0))
45     {
46       s.rho = 0;
47       s.theta = 0;
48     }
49     else
50     {
51       s.rho = sqrt( c.x*c.x + c.y*c.y + c.z*c.z );
52       s.theta = acos( c.z / s.rho );
53     }
54
55     if ((c.x == 0) && (c.y == 0))
56     {
57       if (c.z >= 0)
58         s.phi = M_PI * 0.5;
59       else
60         s.phi = -M_PI * 0.5;
61     }
62     else
63     {
64       float S = sqrt( c.x*c.x + c.y*c.y );
65       if (c.x >= 0)
66         s.phi = asin ( c.y / S );
67       else
68         s.phi = M_PI - asin ( c.y / S );
69     }
70     return s;
71   }
72
73   cart sphere2cart(sphere s)
74   {
75     cart c;
76     c.x = s.rho * sin( s.theta ) * cos( s.phi ) ;
77     c.y = s.rho * sin( s.theta ) * sin( s.phi );
78     c.z = s.rho * cos( s.theta );
79     return c;
80   }
81
82 };
83
84
85 } // worker
86 } // appbase
87 } // ibeo
88
89 #endif // IBEO_FASTSLAM_COORDCONVERSION_HPP
```

C.5 Slicing

The slice worker joins multiple scans into overlapping slices of configurable length.

```
1 // SliceWorker.hpp
```

```
2  // Copyright (c) Ibeo Automobile Sensor GmbH, 2008−2009
3  // created: 2009/02/26
4
5  #ifndef SliceWorker_HPP
6  #define SliceWorker_HPP
7
8  #include <IbeoAPI/Scan.hpp>
9  #include <IbeoAPI/MountingPosition.hpp>
10 #include <IbeoAPI/PositionWGS84.hpp>
11 #include <IbeoAPI/VehicleStateBasic.hpp>
12 #include <IbeoAPI/Point3D.hpp>
13
14 #include <IbeoAPI/Configurable.hpp>
15
16 #include "ibeobasic/ibeobasicdecl.hpp"
17
18 #include "ibeograph/drain/PositionWGS84Drain.hpp"
19 #include "ibeograph/drain/VehicleStateDrain.hpp"
20 #include "ibeograph/drain/ScanDrain.hpp"
21 #include "ibeograph/source/PositionWGS84Source.hpp"
22 #include "ibeograph/source/VehicleStateSource.hpp"
23 #include "ibeograph/source/ScanSource.hpp"
24
25 #include "ibeograph/Preferences.hpp"
26
27 #include <iostream>
28 #include <fstream>
29 #include <string>
30 #include <boost/circular_buffer.hpp>
31
32
33 using namespace std;
34 namespace ibeo {
35 namespace appbase {
36 namespace worker {
37
38 /**
39  * \brief This worker combines Scans
40  */
41 class IBEOBASICDECL SliceWorker : public Configurable,
42           public ibeo::appbase::drain::PositionWGS84Drain,
43           public ibeo::appbase::drain::VehicleStateDrain,
44           public ibeo::appbase::drain::ScanDrain,
45           public ibeo::appbase::source::PositionWGS84Source,
46           public ibeo::appbase::source::VehicleStateSource,
47           public ibeo::appbase::source::ScanSource
48 {
49 public:
50   /// constructor (loads the mounting position out of the configuration file)
51   SliceWorker(const ibeo::Preferences& preferences, const std::string& objectname = "");
52
53   ~SliceWorker(){};
54
55   static const char* getDefaultTypeName() { return "SliceWorker"; }
56
57   virtual void setScan(Scan &scan);
58
59     /*
60      * set the first position of file if necessary
61      * set the relative position to the first position of the current scan
62      */
63   virtual void setPositionWGS84 (const PositionWGS84 &posWGS84);
64
65     /*
66      * set the first position of file if necessary
```

```
67      * set the relative position to the first position of the current scan
68      */
69    virtual void setVehicleState (const VehicleStateBasic &vehicleStateBasic);
70
71  private:
72    void transformToMountingPosition(Scan &scan, MountingPosition vehicleMountingPosition);
73    MountingPosition getRelativeMountingPositionVS(VehicleStateBasic currentVehicleState,
          VehicleStateBasic originVehicleState);
74    MountingPosition getRelativeMountingPositionWGS(PositionWGS84 currentWGS84, PositionWGS84
          originWGS84);
75    /// Device ID of scanner whose mounting position should be changed
76    INT16 m_deviceID;
77
78    enum m_processingState {noPosition, newSlice, fillingSlice, finishedSlice} m_processingState;
79
80    Scan m_scan;
81    Scan m_scanSlice;
82    PositionWGS84 m_currentWGS84;
83    PositionWGS84 m_originWGS84;
84
85    VehicleStateBasic m_currentVehicleState;
86    VehicleStateBasic m_originVehicleState;
87
88    INT8 m_numOfScansInSlice;
89    INT8 m_numOfScansInScan;
90    INT8 m_numOfScansOverlap;
91    bool m_groundLabeled;
92    boost::circular_buffer<Scan> m_overlapScan;
93    boost::circular_buffer<PositionWGS84> m_overlapWGS;
94    boost::circular_buffer<VehicleStateBasic> m_overlapVS;
95  };
96
97  } // worker
98  } // appbase
99  } // ibeo
100
101 #endif
```

```
1   #include "SliceWorker.hpp"
2
3   #include <IbeoAPI/ParamTime.hpp>
4   #include <IbeoAPI/ScannerInfo.hpp>
5   #include <vector>
6   #include <IbeoAPI/StringToolbox.hpp>
7   #include "fastslam/coordconversion.hpp"
8
9   using namespace std;
10
11  namespace ibeo{
12  namespace appbase{
13  namespace worker{
14
15  SliceWorker::SliceWorker(const ibeo::Preferences &preferences, const std::string& objectname)
16    : Configurable(getDefaultTypeName(), objectname)
17    , m_deviceID(-1)
18  {
19    define (new ParamINT8 ("numberOfScansInSlice", m_numOfScansInSlice, "Number of Scans >0 merged to
          one new Scan object. Do not put more than (ScanPointLimit/Scan) ScanPoints into one Scan
          object.","20"));
20    define (new ParamINT8 ("numberOfScansOverlap", m_numOfScansOverlap, "Number of Scans for each
          slice to overlap with the previous one.","5"));
21    define (new ParamINT16 ("DeviceID", m_deviceID, "DeviceID to be used for localisation. Default:
          127 (VehicleState)","127"));
```

```
22    define (new ParamBool ("GroundLabeled", m_groundLabeled, "Set to true if the scans are
          preprocessed for ground detection","false"));
23
24    fillValuesOrException (preferences.getConfigValues (getDefaultTypeName(), objectname));
25
26    m_processingState = noPosition;
27    m_overlapScan = boost::circular_buffer<Scan>(m_numOfScansOverlap);
28    m_overlapWGS = boost::circular_buffer<PositionWGS84>(m_numOfScansOverlap);
29    m_overlapVS = boost::circular_buffer<VehicleStateBasic>(m_numOfScansOverlap);
30    // Load values from the config file, fill them into our
31    // parameters, and throw an exception if something was invalid.
32    fillValuesValidateOrExcept(preferences.findByTypeAndName(getDefaultTypeName(), objectname));
33  }
34
35  void SliceWorker::setScan(Scan &scan)
36  {
37    traceDebug("") << "Processing state: " << m_processingState << endl;
38
39    if (m_processingState == noPosition)
40    {
41      traceWarning("") << "No VehicleState or PositionWGS84 Information. Nothing to be done." << endl;
42    } else {
43      m_scan = scan;
44      m_overlapScan.push_back(scan);
45      if (m_deviceID == 127)
46      {
47        m_overlapVS.push_back(m_currentVehicleState);
48      }
49      if(m_deviceID == 30 || m_deviceID == 31 || m_deviceID == 0 || m_deviceID == 41 || m_deviceID ==
            42 || m_deviceID == 43 || m_deviceID == 44 || m_deviceID == 45)
50      {
51        m_overlapWGS.push_back(m_currentWGS84);
52      }
53
54      // new mountingposition set to vehicleposition relative to the startposition of each file
55      MountingPosition vehicleMountingPosition;
56
57      if(m_deviceID == 127)
58      {
59        vehicleMountingPosition = getRelativeMountingPositionVS(m_currentVehicleState,
              m_originVehicleState);
60      }
61      else
62      {
63        if(m_deviceID == 30 || m_deviceID == 31 || m_deviceID == 0 || m_deviceID == 41 || m_deviceID
              == 42 || m_deviceID == 43 || m_deviceID == 44 || m_deviceID == 45)
64        {
65          vehicleMountingPosition = getRelativeMountingPositionWGS(m_currentWGS84, m_originWGS84);
66        }
67        else
68        {
69          traceError("") << "DeviceID " << m_deviceID << " not specified" << endl;
70        }
71      }
72
73      transformToMountingPosition(m_scan, vehicleMountingPosition);
74
75      // put m_numOfScansInSlice scans in one file
76      if (m_processingState == newSlice)
77      {
78        m_processingState = fillingSlice;
79        m_numOfScansInScan = m_overlapScan.size()+1;
80        if (m_overlapScan.size() > 0)
81        {
82          ScanPoint pt = ScanPoint();
```

```
83          m_scanSlice = m_scan;
84          m_scanSlice.resize(0, pt);
85
86          for (size_t i=0 ; i<m_overlapScan.size(); ++i)
87          {
88            assert(m_overlapScan.size() > i);
89            Scan scan = m_overlapScan[i];
90            if(m_deviceID == 127)
91            {
92              if (i == 0) m_originVehicleState = m_overlapVS[0];
93              assert(m_overlapVS.size() > i);
94              vehicleMountingPosition = getRelativeMountingPositionVS(m_overlapVS[i],
                    m_originVehicleState);
95            }
96            else
97            {
98              if(m_deviceID == 30 || m_deviceID == 31 || m_deviceID == 0 || m_deviceID == 41 ||
                    m_deviceID == 42 || m_deviceID == 43 || m_deviceID == 44 || m_deviceID == 45)
99              {
100               assert(m_overlapWGS.size() > i);
101               vehicleMountingPosition = getRelativeMountingPositionWGS(m_overlapWGS[i],
                      m_originWGS84);
102             }
103             else
104             {
105               traceError("") << "DeviceID " << m_deviceID << " not specified" << endl;
106             }
107           }
108
109           transformToMountingPosition(scan, vehicleMountingPosition);
110           scan.setVehicleCoordinates(true);
111           m_scanSlice.setVehicleCoordinates(true);
112           m_scanSlice.addScan(scan);
113         }
114       }
115       else
116       {
117         m_scanSlice = m_scan;
118       }
119     }
120     else
121     {
122       if (m_numOfScansInScan < m_numOfScansInSlice)
123       {
124         m_scan.setVehicleCoordinates(true);
125         m_scanSlice.setVehicleCoordinates(true);
126         m_scanSlice.addScan(m_scan);
127         m_numOfScansInScan++;
128       }
129       else
130       {
131         m_scan.setVehicleCoordinates(true);
132         m_scanSlice.setVehicleCoordinates(true);
133         m_scanSlice.addScan(m_scan);
134         m_processingState = finishedSlice;
135         traceDebug("") << "Finished slice" << endl;
136       }
137     }
138   }
139 }
140
141 MountingPosition SliceWorker::getRelativeMountingPositionVS(VehicleStateBasic currentVehicleState,
      VehicleStateBasic originVehicleState)
142 {
143   coordconversion converter;
```

```
144
145   cart c;
146   c.x = currentVehicleState.getX()-originVehicleState.getX();
147   c.y = currentVehicleState.getY()-originVehicleState.getY();
148   c.z = 0;
149
150   sphere s = converter.cart2sphere(c);
151
152   s.phi = s.phi - originVehicleState.getCourseAngle();
153
154   c = converter.sphere2cart(s);
155
156   MountingPosition vehicleMountingPosition;
157   vehicleMountingPosition = MountingPosition(
158     currentVehicleState.getCourseAngle()-originVehicleState.getCourseAngle(),
159     0.0f,
160     0.0f,
161     c.x,
162     c.y,
163     0.0f);
164   return vehicleMountingPosition;
165 }
166
167 MountingPosition SliceWorker::getRelativeMountingPositionWGS(PositionWGS84 currentWGS84,
      PositionWGS84 originWGS84)
168 {
169   Point3D currentRelWGS84;
170   MountingPosition vehicleMountingPosition;
171
172   currentRelWGS84 = currentWGS84.getCartesianRelPos(originWGS84);
173   currentRelWGS84.rotateAroundZ(-originWGS84.getYawAngleInRad()-ibeo::PI_double/2);
174   currentRelWGS84.rotateAroundY(-originWGS84.getPitchAngleInRad());
175   currentRelWGS84.rotateAroundX(-originWGS84.getRollAngleInRad());
176   vehicleMountingPosition = MountingPosition(currentWGS84.getYawAngleInRad()-originWGS84.
        getYawAngleInRad(),currentWGS84.getPitchAngleInRad()-originWGS84.getPitchAngleInRad(),
        currentWGS84.getRollAngleInRad()-originWGS84.getRollAngleInRad(),currentRelWGS84.getX(),
        currentRelWGS84.getY(),currentRelWGS84.getZ());
177
178   return vehicleMountingPosition;
179 }
180
181 void SliceWorker::transformToMountingPosition(Scan &scan, MountingPosition vehicleMountingPosition)
182 {
183   traceDebug("") << "transforming mountingPosition!" << endl;
184   scan.setVehicleCoordinates(false);
185   for ( vector<ScannerInfo>::iterator iter = scan.getScannerInfos().begin() ; iter != scan.
        getScannerInfos().end() ; ++iter)
186   {
187     iter->setMountingPosition(vehicleMountingPosition);
188   }
189   scan.transformToVehicleCoordinatesUnsorted();
190 }
191
192 /*
193  * set the first position of file if necessary
194  * set the relativ position to the first position of the current scan
195  * only used if deviceID == 127 (VehicleState)
196  */
197 void SliceWorker::setVehicleState (const VehicleStateBasic &vehicleStateBasic)
198 {
199   if (m_deviceID == 127)
200   {
201     if (m_processingState == noPosition)
202     {
203       m_processingState = newSlice;
```

```
204       m_originVehicleState = vehicleStateBasic;
205     }
206
207     if (m_processingState == finishedSlice)
208     {
209       // signal position before scan so you know where to place the scan
210       m_signalVehicleState(m_originVehicleState);
211
212       if (m_groundLabeled)
213         m_scanSlice.setGroundLabeled(true);
214       else
215         m_scanSlice.setGroundLabeled(false);
216
217       m_scanSlice.setVehicleCoordinates(true);
218       m_signalScan(m_scanSlice);
219       m_originVehicleState = vehicleStateBasic;
220       m_processingState = newSlice;
221     }
222     m_currentVehicleState = vehicleStateBasic;
223   }
224 }
225
226 void SliceWorker::setPositionWGS84 (const PositionWGS84 &WGS84Basic)
227 {
228   if ((int)WGS84Basic.getDeviceID() == m_deviceID)
229   {
230     m_currentWGS84 = WGS84Basic;
231     if (m_processingState == noPosition)
232     {
233       m_processingState = newSlice;
234       m_originWGS84 = WGS84Basic;
235     }
236     if (m_processingState == finishedSlice)
237     {
238       // signal position before scan so you know where to place the scan
239       m_signalPositionWGS84(m_originWGS84);
240
241       if (m_groundLabeled)
242         m_scanSlice.setGroundLabeled(true);
243       else
244         m_scanSlice.setGroundLabeled(false);
245
246       m_scanSlice.setVehicleCoordinates(true);
247       m_signalScan(m_scanSlice);
248       m_originWGS84 = WGS84Basic;
249       m_processingState = newSlice;
250     }
251   }
252   else
253   {
254     traceDebug("") << "DeviceID: " << (int)WGS84Basic.getDeviceID() << endl;
255   }
256 }
257
258
259 } // namespace ibeo
260 } // namespace appbase
261 } // namespace worker
```

FastSLAM

D

The FastSLAM worker is the most complex and largest one. Some of its functionality is outsourced to keep a better overview and make the code better maintainable.

D.1 SLAMWorker

Within this file the main processing for FastSLAM is done.

```
1  // SLAMWorker.hpp
2  // created: 2011-02-11
3  // author: Jan Gries
4
5  #ifndef SLAMWorker_HPP
6  #define SLAMWorker_HPP
7
8  #include <IbeoAPI/Scan.hpp>
9  #include <IbeoAPI/MountingPosition.hpp>
10 #include <IbeoAPI/PositionWGS84.hpp>
11 #include <IbeoAPI/VehicleStateBasic.hpp>
12 #include <IbeoAPI/Point3D.hpp>
13 #include <IbeoAPI/Position3D.hpp>
14 #include <list>
15
16 #include <IbeoAPI/Configurable.hpp>
17
18 #include "ibeobasic/ibeobasicdecl.hpp"
19
20 #include "ibeograph/drain/PositionWGS84Drain.hpp"
21 #include "ibeograph/drain/ScanDrain.hpp"
22 #include "ibeograph/drain/VehicleStateDrain.hpp"
23 #include "ibeograph/source/PositionWGS84Source.hpp"
24 #include "ibeograph/source/ScanSource.hpp"
25 #include "ibeograph/source/VehicleStateSource.hpp"
26
27 #include <boost/numeric/ublas/matrix.hpp>
28
29 #include "ibeobasic/worker/fastslam/Particle.hpp"
30 #include "ibeobasic/worker/fastslam/Landmark.hpp"
31 #include "ibeobasic/worker/fastslam/Likelihoodtable.hpp"
32
33 #include "ibeograph/Preferences.hpp"
34
35 #include <vector>
36 #include <cmath>
37 #include <iostream>
38 #include <fstream>
39 #include <string>
```

```
40
41  using namespace std;
42  namespace ibeo {
43  namespace appbase {
44  namespace worker {
45
46  /**
47   * \brief This worker tries to improve the trajectory using the Scandata. No online Algorithm!
48   *
49   * Potential Limitations: One Scan must have an overlap with its successor
50   *
51   * Input Data Assumptions: Scan must be in vehicle coordinate system.
52   *
53   * Output Data: new VehicleState messages and the Scan belonging to
54   *
55   * Author(s): Jan Gries
56   */
57
58  typedef boost::numeric::ublas::matrix<float> Matrix;
59  typedef vector<Particle> ParticleCollection;
60
61  class IBEOBASICDECL SLAMWorker : public Configurable,
62            public ibeo::appbase::drain::PositionWGS84Drain,
63            public ibeo::appbase::drain::VehicleStateDrain,
64            public ibeo::appbase::drain::ScanDrain,
65            public ibeo::appbase::source::PositionWGS84Source,
66            public ibeo::appbase::source::VehicleStateSource,
67            public ibeo::appbase::source::ScanSource
68  {
69  public:
70    /**
71     * constructor
72     */
73    SLAMWorker(const ibeo::Preferences& preferences, const std::string& objectname = "");
74
75    /**
76     * destructor
77     */
78    ~SLAMWorker(){};
79
80    static const char* getDefaultTypeName() { return "SLAMWorker"; }
81
82    virtual void setScan(Scan &scan);
83    virtual void setPositionWGS84 (const PositionWGS84 &posWGS84);
84    virtual void setVehicleState (const VehicleStateBasic &vehicleState);
85
86  private:
87
88    void setScanPointsToMap ();
89    void startSlam (Scan scan);
90    void eliminateBadParticle ();
91    float findLimitForElimination ();
92    void eliminateParticlesWithEvaluationUnder(float limit);
93    bool isRatingLessThanLimit(const Particle &particle, float limit);
94    void updateParticleLandmarks();
95    void resamplingParticle();
96    void normRatings();
97    float trustInPosRelToGps(float distance);
98
99    double uniformRand(double min, double max);
100   Scan scanToMountingPosition(Scan scan, Position3D position);
101   void writeMapToFile (string string, Particle particle, int outputnumber);
102   void writeBestMapToFile (string string, Particle particle, int outputnumber);
103   void writeLandmarksToFile (string string, Particle particle);
104   void writeInputScansToFile (string string, Particle particle);
```

```
105
106   UINT8 m_deviceID;
107   UINT16 m_maxNumOfParticles;
108   size_t m_FileOutputNum;
109   size_t m_numHits;
110
111   PositionWGS84 m_originWGS84;
112   VehicleStateBasic m_originVehicleState;
113
114   Position3D m_oldPosition;
115   Position3DCollection m_odometryTrajectory;
116
117   float m_threshold;
118
119   float m_alphaYaw1;
120   float m_alphaPitch1;
121   float m_alphaTrans;
122   float m_alphaYaw2;
123   float m_alphaPitch2;
124   float m_alphaRoll;
125   float m_fYaw1;
126   float m_fYaw2;
127
128   UINT8 m_outputFreq;
129
130   list<Scan> m_scanList;
131   list<Scan> m_fullScansList;
132   ParticleCollection m_particles;
133   ParticleCollection m_particlesInWork;
134
135   PositionWGS84 m_startingPointWGS84;
136   VehicleStateBasic m_startingPointVehicleState;
137
138   enum ProcessingStateOdometry {noOdometry, waitForNewOdometry, gotOdometry};
139   enum ProcessingStateScan     {noScan, waitForNewScan, gotScan};
140   ProcessingStateOdometry m_processingStateOdometry;
141   ProcessingStateScan m_processingStateScan;
142 };
143
144 } // worker
145 } // appbase
146 } // ibeo
147
148 #endif
```

```
1  // SLAMWorker.cpp
2  // created: 2011-02-11
3  // author: Jan Gries
4
5  #include "SLAMWorker.hpp"
6
7  #include <IbeoAPI/ParamTime.hpp>
8  #include <IbeoAPI/ScannerInfo.hpp>
9  #include <vector>
10 #include <IbeoAPI/StringToolbox.hpp>
11
12 #include <fstream>
13 #include <sstream>
14 #include <ostream>
15
16 using namespace std;
17
18 namespace ibeo{
19 namespace appbase{
```

```
20 namespace worker{
21
22 SLAMWorker::SLAMWorker(const ibeo::Preferences &preferences, const std::string& objectname)
23   : Configurable(getDefaultTypeName(), objectname)
24 {
25   define (new ParamUINT16 ("maxNumOfParticles", m_maxNumOfParticles, "maximum number of particles
       that are calculated per position", "100"));
26   define (new ParamUINT8 ("DeviceID", m_deviceID, "DeviceID of INS to be used for localisation", "30
       "));
27   define (new ParamFloat ("Threshold", m_threshold, "Likelihood value under which an observation is
       considered a new landmark", "0.1"));
28   define (new ParamUINT8 ("OutputFrequency", m_outputFreq, "Every X scans output files are generated
       , X is the frequency", "20"));
29
30   // Parameters to calculate the particle cloud
31
32   define (new ParamFloat  ("AlphaYaw1", m_alphaYaw1, "alpha for yaw angle distribution of the
       transformation vector", "0.01"));
33   define (new ParamFloat  ("AlphaPitch1", m_alphaPitch1, "alpha for pitch angle distribution of the
       transformation vector", "0.01"));
34   define (new ParamFloat  ("AlphaTrans", m_alphaTrans, "alpha for distribution of the transformation
        vector length ", "0.01"));
35   define (new ParamFloat  ("AlphaYaw2", m_alphaYaw2, "alpha for yaw distribution after
       transformation", "0.01"));
36   define (new ParamFloat  ("AlphaPitch2", m_alphaPitch2, "alpha for pitch distribution after
       transformation", "0.01"));
37   define (new ParamFloat  ("AlphaRoll", m_alphaRoll, "alpha for roll distribution after
       transformation", "0.01"));
38   define (new ParamFloat  ("FYaw1", m_fYaw1, "static yaw error rate", "0.01"));
39   define (new ParamFloat  ("FYaw2", m_fYaw2, "static yaw error rate", "0.01"));
40
41
42   fillValuesOrException (preferences.getConfigValues (getDefaultTypeName(), objectname));
43
44   // Load values from the config file, fill them into our
45   // parameters, and throw an exception if something was invalid.
46   fillValuesValidateOrExcept(preferences.findByTypeAndName(getDefaultTypeName(), objectname));
47   m_processingStateOdometry = noOdometry;
48
49   m_FileOutputNum = 0;
50   m_numHits = 0;
51 }
52
53 bool compareParticleRating(const Particle& i, const Particle& j)
54 {
55   // '>' is descending order
56   return i.getRating() > j.getRating();
57 }
58
59 void SLAMWorker::setScan(Scan &scan)
60 {
61   traceDebug("") << "Scan received" << endl;
62
63   // put the unprocessed scans in m_fullScansList. Only needed for .off
64   // export later on.
65   if (scan.isGroundLabeled())
66     m_scanList.push_back(scan);
67   else
68     m_fullScansList.push_back(scan);
69
70   m_processingStateScan = gotScan;
71   if (m_processingStateOdometry == gotOdometry)
72   {
73     startSlam(scan);
74   }
```

```
75  }
76
77  void SLAMWorker::setPositionWGS84 (const PositionWGS84 &WGS84Basic)
78  {
79    traceDebug("") << "WGS84-ID: " << (int)WGS84Basic.getDeviceID() << " received" << endl;
80    if ((int)WGS84Basic.getDeviceID() == m_deviceID)
81    {
82      // but for initalization there is no update
83      if(m_processingStateOdometry == noOdometry)
84      {
85        // save position as coordinate origin
86        m_originWGS84 = WGS84Basic;
87
88        Position3D newPosition = Position3D(
89          WGS84Basic.getYawAngleInRad(),
90          WGS84Basic.getPitchAngleInRad(),
91          WGS84Basic.getRollAngleInRad(),
92          WGS84Basic.getCartesianRelPos(m_originWGS84));
93
94        m_odometryTrajectory.push_back(newPosition);
95        m_processingStateOdometry = gotOdometry;
96        Particle tempParticle = Particle(newPosition);
97        tempParticle.setRating(1.0);
98        tempParticle.setNormRating(1.0);
99        for (size_t i=0; i<m_maxNumOfParticles; ++i)
100       {
101         m_particles.push_back(tempParticle);
102       }
103       return;
104     }
105     m_processingStateOdometry = gotOdometry;
106
107     // calculate the current position relative from the first
108     // WGS84Basic
109     traceDebug("") << "performing relPos transform on (" << WGS84Basic.getYawAngleInRad() << "," <<
          WGS84Basic.getPitchAngleInRad() << "," << m_originWGS84.toString() << "," << WGS84Basic.
          getCartesianRelPos(m_originWGS84) << endl;
110     Position3D newPosition = Position3D(
111       WGS84Basic.getYawAngleInRad(),
112       WGS84Basic.getPitchAngleInRad(),
113       WGS84Basic.getRollAngleInRad(),
114       WGS84Basic.getCartesianRelPos(m_originWGS84));
115     // save the current position
116     m_odometryTrajectory.push_back(newPosition);
117   }
118 }
119
120 void SLAMWorker::setVehicleState (const VehicleStateBasic &vehicleState)
121 {
122   traceDebug("") << "VehicleState received" << endl;
123     // but for initalization there is no update
124     if(m_processingStateOdometry == noOdometry)
125     {
126       // save position as coordinate origin
127       m_originVehicleState = vehicleState;
128
129       Position3D newPosition = Position3D(
130         vehicleState.getCourseAngle()+ibeo::PI/2,
131         0,
132         0,
133         m_originVehicleState.getX()-vehicleState.getX(),
134         m_originVehicleState.getY()-vehicleState.getY(),
135         0);
136
137       m_odometryTrajectory.push_back(newPosition);
```

```
138       m_processingStateOdometry = gotOdometry;
139       Particle tempParticle = Particle(newPosition);
140       tempParticle.setRating(1.0);
141       tempParticle.setNormRating(1.0);
142       for (size_t i=0; i<m_maxNumOfParticles; ++i)
143       {
144         m_particles.push_back(tempParticle);
145       }
146       return;
147     }
148     m_processingStateOdometry = gotOdometry;
149
150     // calculate the current position relative to the first
151     // vehicleState
152     traceDebug("") << "performing relPos transform on (" << vehicleState.getCourseAngle() << ","  <<
            m_originVehicleState.toString() << "," << vehicleState.getX()-m_originVehicleState.getX()
          << vehicleState.getY()-m_originVehicleState.getY() << endl;
153     Position3D newPosition = Position3D(
154       vehicleState.getCourseAngle()+ibeo::PI/2,
155       0,
156       0,
157       m_originVehicleState.getX()-vehicleState.getX(),
158       m_originVehicleState.getY()-vehicleState.getY(),
159       0);
160
161     // save the current position
162     m_odometryTrajectory.push_back(newPosition);
163 }
164
165 void SLAMWorker::startSlam (Scan scan)
166 {
167   // measurement noise for the LRF: 2 cm max statistical error (1 sigma)
168   // +/- 4 cm max systematic error. Since we have multiple LRF
169   // probably scanning the same landmarks the systematic error might add up.
170   Matrix measurementNoise = IdentityMatrix(3,3) * 0.1;
171
172   // calculate likelihoods for all landmark-observation associations
173   for ( ParticleCollection::iterator iterParticle = m_particles.begin(); iterParticle != m_particles
          .end(); ++iterParticle)
174   {
175     ExtendedKalmanFilter ekf = ExtendedKalmanFilter();
176
177     Matrix jacobian = Matrix(3,3);
178     Matrix innovationCovariance = Matrix(3,3);
179     ZCollection z_t;
180
181     Scan scanRelativeToParticle = scanToMountingPosition(scan, iterParticle->getlastOdometryUpdate()
            );
182     for (Scan::iterator iterScan = scanRelativeToParticle.begin(); iterScan !=
            scanRelativeToParticle.end(); ++iterScan)
183     {
184       z_t.push_back(iterScan->getPoint3D());
185     }
186
187     Position3DCollection::reverse_iterator rit = m_odometryTrajectory.rbegin();
188     assert (m_odometryTrajectory.size() > 0);
189     Position3D one = *rit;
190     Position3D two;
191     if (m_odometryTrajectory.size() < 2)
192     {
193       two = *rit;
194     }
195     else
196     {
197       ++rit;
```

```
198       two = *rit;
199     }
200     iterParticle->calculateNewPose(
201       two,
202       one,
203       m_alphaYaw1,
204       m_alphaPitch1,
205       m_alphaTrans,
206       m_alphaYaw2,
207       m_alphaPitch2,
208       m_alphaRoll,
209       m_fYaw1,
210       m_fYaw2);
211
212     // Create Likelihood table
213     Likelihoodtable likelihoods(iterParticle->getlastOdometryUpdate(), iterParticle->
          getSecondlastOdometryUpdate(), IdentityMatrix(3,3)/10);
214
215     vector<Landmark> landmarks = iterParticle->getMap(iterParticle->getlastOdometryUpdate());
216
217     likelihoods.initLandmarks(landmarks);
218     likelihoods.initObservations(z_t);
219
220     traceDebug("") << "After adding Observations: " << z_t.size() << endl;
221
222     likelihoods.calculateLikelihoods();
223     likelihoods.findBestGlobalAssociations();
224     likelihoods.findBestAssociations();
225
226     // calculate importance weight average
227     float importanceWeight = 0.0;
228     int importanceCounter = 1;
229
230     // Loop through all observations
231     for (ZCollection::iterator iterZ = z_t.begin(); iterZ != z_t.end(); ++iterZ)
232     {
233       float maxLikelihood = likelihoods.getMaxLikelihoodForObservation(*iterZ);
234
235       assert(maxLikelihood == maxLikelihood);
236
237       importanceWeight += maxLikelihood;
238       ++importanceCounter;
239
240       // if the features best likelihood is less than a fixed
241       // threshold, the feature is considered a new landmark
242       if (maxLikelihood < m_threshold)
243       {
244         // observation iterZ is a new landmark
245         Landmark newLandmark;
246
247         // initialize mean (Probabilistic Robotics, 2nd Edition,
248         // p. 461, l. 17)
249         newLandmark.mean = *iterZ;
250
251         // calculate new jacobian (Probabilistic Robotics, 2nd
252         // Edition, p. 461, l. 18)
253         Matrix invJacobian = IdentityMatrix(3,3);
254         if (! InvertMatrix(ekf.landmarkJacobian(iterParticle->getlastOdometryUpdate(), newLandmark.
              mean), invJacobian))
255         {
256           traceError("") << "Inversion of Matrix failed. Identity Matrix returned.\n";
257         }
258
259         // initialize covariance (Probabilistic Robotics, 2nd
260         // Edition, p. 461, l. 18)
```

```
261         Matrix invJaMeasure = prod(trans(invJacobian), measurementNoise);
262         newLandmark.covariance = prod(invJaMeasure, invJacobian);
263         for (size_t i=0; i<newLandmark.covariance.size1(); ++i)
264           for (size_t j=0; j<newLandmark.covariance.size2(); ++j)
265             if (! (newLandmark.covariance(i,j) == newLandmark.covariance(i,j)))
266             {
267               traceError("") << "Covariance matrix has NaN values\n";
268               cout << "===new covariance: " << newLandmark.covariance << endl;
269               cout << "invJacobian: " << invJacobian << endl;
270               cout << "invJaMeasure: " << invJaMeasure << endl;
271               assert(false);
272             }
273
274         iterParticle->addLandmarkToMap(newLandmark);
275       }
276       // associate the current observation with the landmark with
277       // the highest likelihood
278       else
279       {
280         m_numHits++;
281         Landmark maxLM = likelihoods.getLandmarkWithMaxLikelihood(*iterZ);
282
283         Matrix associatedJacobian = ekf.landmarkJacobian(iterParticle->getlastOdometryUpdate(),
                maxLM.mean);
284
285         Matrix associatedInnoCov = ekf.innovationCovariance(associatedJacobian, maxLM.covariance,
                measurementNoise);
286
287         // update Kalman Gain (Probabilistic Robotics p. 461, l. 21)
288         Matrix associatedGain = ekf.ekfGain(associatedJacobian, maxLM.covariance, associatedInnoCov)
                ;
289
290         // update Kalman Mean (Probabilistic Robotics p. 461, l. 22)
291         maxLM.mean = ekf.ekfMean(maxLM.mean, associatedGain, *iterZ, maxLM.mean);
292
293         // update Kalman Covariance (Probabilistic Robotics
294         // p. 461, l. 23)
295         maxLM.covariance = ekf.ekfCovariance(associatedJacobian, maxLM.covariance, associatedGain,
                measurementNoise);
296
297         iterParticle->updateLandmark(maxLM);
298       }
299     }
300
301     iterParticle->setRating(importanceWeight/importanceCounter);
302
303     // unobserved landmarks remain untouched
304   }
305
306   normRatings();
307
308   assert(m_particles.size() > 0);
309
310   for ( ParticleCollection::iterator iterParticle = m_particles.begin(); iterParticle != m_particles
        .end(); ++iterParticle)
311   {
312     // apply trust in position relative to GPS position
313     float distanceToGpsPos = iterParticle->getlastOdometryUpdate().getPoint().dist(
          m_odometryTrajectory.back().getPoint());
314     iterParticle->setRating(iterParticle->getRating() * trustInPosRelToGps(distanceToGpsPos));
315   }
316
317   sort(m_particles.begin(), m_particles.end(), bind(&compareParticleRating, _1, _2));
318
319   Particle bestParticle = m_particles.front();
```

```
320   ParticleCollection::iterator bestParticleIt = m_particles.begin();
321
322   cout << "Best particles Importance Weight: " << bestParticle.getRating() << endl;
323
324   if ((m_FileOutputNum % m_outputFreq) == 0)
325   {
326     writeMapToFile("output", bestParticle, m_FileOutputNum);
327     writeBestMapToFile("bestmap", bestParticle, m_FileOutputNum);
328   }
329   ++m_FileOutputNum;
330
331   resamplingParticle();
332
333   m_processingStateOdometry = waitForNewOdometry;
334   m_processingStateScan = waitForNewScan;
335
336   cout << "Number of adjusted Landmarks: " << m_numHits << endl;
337 }
338
339 void SLAMWorker::writeMapToFile (string string, Particle particle, int outputnum)
340 {
341   stringstream ss;
342   ss << string << setfill('0');
343   ss << setw(3) << outputnum << ".off";
344
345   ofstream out(ss.str().c_str());
346   unsigned int MAX_POINTS = 0;
347
348   for (list<Scan>::iterator iterScanList = m_scanList.begin(); iterScanList != m_scanList.end(); ++
          iterScanList)
349   {
350     MAX_POINTS += iterScanList->getNumPoints();
351   }
352
353   MAX_POINTS *= m_maxNumOfParticles;
354
355   //create header
356   out << "OFF\n" << MAX_POINTS << " 0 0\n";
357
358   list<Scan>::iterator iterScanList = m_scanList.begin();
359
360   Position3DCollection odometryList = particle.getOdometryList();
361   Position3DCollection::iterator iterOdometry = odometryList.begin();
362
363   assert (m_scanList.size() == odometryList.size());
364   while (!(iterScanList == m_scanList.end() || iterOdometry == odometryList.end()))
365   {
366     Scan output = scanToMountingPosition(*iterScanList, *iterOdometry);
367     ++iterScanList;
368     ++iterOdometry;
369
370     for (Scan::iterator iterOutput = output.begin(); iterOutput != output.end(); ++iterOutput)
371     {
372       out << iterOutput->getX() << " " << iterOutput->getY() << " " << iterOutput->getZ() << " 1.0
              0.0 0.0 1.0" << "\n";
373     }
374   }
375
376   int colorCounter = 0;
377   for ( ParticleCollection::iterator iterParticle = m_particles.begin(); iterParticle != m_particles
          .end(); ++iterParticle)
378   {
379     ++colorCounter;
380     list<Scan>::iterator iterScanList = m_scanList.begin();
381
```

```
382     Position3DCollection odometryList = iterParticle->getOdometryList();
383     Position3DCollection::iterator iterOdometry = odometryList.begin();
384
385     assert (m_scanList.size() == odometryList.size());
386     while (!(iterScanList == m_scanList.end() || iterOdometry == odometryList.end()))
387     {
388       Scan output = scanToMountingPosition(*iterScanList, *iterOdometry);
389       ++iterScanList;
390       ++iterOdometry;
391
392       for (Scan::iterator iterOutput = output.begin(); iterOutput != output.end(); ++iterOutput)
393       {
394         if (iterParticle->getRating() != particle.getRating())
395         {
396           out << iterOutput->getX() << " " << iterOutput->getY() << " " << iterOutput->getZ() << "
                  0.0 " <<  (float)colorCounter/m_maxNumOfParticles << " " << 1 - (float)colorCounter/
                  m_maxNumOfParticles << " " << "1.0" << "\n";
397         }
398       }
399     }
400   }
401
402   out.close();
403 }
404
405 void SLAMWorker::writeBestMapToFile (string string, Particle particle, int outputnum)
406 {
407   stringstream ss;
408   ss << string << setfill('0');
409   ss << setw(3) << outputnum << ".off";
410
411   ofstream out(ss.str().c_str());
412   unsigned int MAX_POINTS = 0;
413
414   for (list<Scan>::iterator iterScanList = m_fullScansList.begin(); iterScanList != m_fullScansList.
          end(); ++iterScanList)
415   {
416     MAX_POINTS += iterScanList->getNumPoints();
417   }
418
419   //create header
420   out << "OFF\n" << MAX_POINTS << " 0 0\n";
421
422   list<Scan>::iterator iterScanList = m_fullScansList.begin();
423
424   Position3DCollection odometryList = particle.getOdometryList();
425   Position3DCollection::iterator iterOdometry = odometryList.begin();
426
427   assert (m_scanList.size() == odometryList.size());
428   while (!(iterScanList == m_fullScansList.end() || iterOdometry == odometryList.end()))
429   {
430     Scan output = scanToMountingPosition(*iterScanList, *iterOdometry);
431     ++iterScanList;
432     ++iterOdometry;
433
434     for (Scan::iterator iterOutput = output.begin(); iterOutput != output.end(); ++iterOutput)
435     {
436       float echoWidth = iterOutput->getEchoWidth();
437       echoWidth *= 7;
438       if (echoWidth > 1) echoWidth = 1;
439       out << iterOutput->getX() << " " << iterOutput->getY() << " " << iterOutput->getZ() << " " <<
              echoWidth << " " << echoWidth << " " << echoWidth << " 1.0" << "\n";
440     }
441   }
442
```

```
443   out.close();
444 }
445
446 void SLAMWorker::resamplingParticle()
447 {
448   ParticleCollection newParticleCollection;
449   ParticleCollection::iterator iterParticle = m_particles.begin();
450   float sum = iterParticle->getNormRating();
451   float lookingValue = 0;
452
453   // cumulative distribution function
454   for (size_t i=0 ; i<m_maxNumOfParticles ; ++i)
455   {
456     lookingValue = ((uniformRand(0, 1) + i) / m_maxNumOfParticles);
457     assert (lookingValue <= 1);
458     while ( lookingValue > sum )
459     {
460       ++iterParticle;
461       sum += iterParticle->getNormRating();
462     }
463     newParticleCollection.push_back(*iterParticle);
464   }
465
466   m_particles = newParticleCollection;
467
468   assert (sum <= 1.1);
469 }
470
471 Scan SLAMWorker::scanToMountingPosition(Scan scan, Position3D position)
472   {
473   assert (scan.isVehicleCoordinates());
474
475   scan.setVehicleCoordinates(false);
476
477   // adjust INS yaw angle to AppBase yaw angle
478   position.setYawAngle(position.getYawAngle()+ibeo::PI_double/2);
479
480   MountingPosition mountingPos(position);
481
482   for ( vector<ScannerInfo>::iterator iter = scan.getScannerInfos().begin() ; iter != scan.
        getScannerInfos().end() ; ++iter)
483   {
484     iter->setMountingPosition( mountingPos );
485   }
486
487   bool worked = scan.transformToVehicleCoordinatesUnsorted();
488   if (!worked) traceWarning("") << "Scan::transformToVehicleCoordinatesUnsorted() returned false" <<
        std::endl;
489   return scan;
490 }
491
492 void SLAMWorker::normRatings()
493 {
494   double sum = 0.0;
495   for (ParticleCollection::iterator iter = m_particles.begin(); iter != m_particles.end(); ++iter)
496   {
497     sum += iter->getRating();
498   }
499   if (sum > 0)
500   {
501     for (ParticleCollection::iterator iter = m_particles.begin(); iter != m_particles.end(); ++iter)
502     {
503       iter->setNormRating(iter->getRating()/sum);
504     }
505   }
```

```
506   else
507   {
508     for (ParticleCollection::iterator iter = m_particles.begin(); iter != m_particles.end(); ++iter)
509     {
510       iter->setNormRating(1.0/m_particles.size());
511     }
512   }
513 }
514
515 double SLAMWorker::uniformRand(double min, double max)
516 {
517   boost::mt19937 generator;
518   boost::uniform_real<double> uni_dist(min, max);
519   boost::variate_generator<boost::mt19937&, boost::uniform_real<double> > getSample(generator,
        uni_dist);
520   return getSample();
521 }
522
523 float SLAMWorker::trustInPosRelToGps(float distance)
524 {
525   const float m_lowerGpsBound = 3.0;
526   float result = 0.0;
527
528   // Apply exp(-(x+m_lowerGpsBound)) to avoid dropping down to 0
529   if (distance < m_lowerGpsBound)
530     result = 1.0;
531   else
532     result = exp(m_lowerGpsBound - fabs(distance));
533
534   return result;
535 }
536
537 } // namespace ibeo
538 } // namespace appbase
539 } // namespace worker
```

D.2 Landmark

Definition of a landmark struct.

```
1  // Landmark.hpp
2  // created: 2011-03-16
3  // author: Jan Girlich
4
5  #ifndef LANDMARK_HPP
6  #define LANDMARK_HPP
7
8  #include <IbeoAPI/Point3D.hpp>
9  #include <boost/numeric/ublas/matrix.hpp>
10
11 using namespace std;
12 using namespace ibeo;
13
14 struct Landmark
15 {
16   Point3D mean;
17   boost::numeric::ublas::matrix<float> covariance;
18   size_t index;
19 } ;
20
21 #endif
```

D.3 Extended Kalman Filter

This is an implementation of an extended Kalman filter used in the FastSLAM worker.

```
#include <IbeoAPI/Point3D.hpp>
#include <IbeoAPI/Position3D.hpp>

#include <boost/numeric/ublas/matrix.hpp>
#include "ibeoextended/recttracking/InvertMatrix.hpp"
#include "./coordconversion.hpp"
#include <ibeobasic/worker/fastslam/Landmark.hpp>

typedef boost::numeric::ublas::matrix<float> Matrix;
typedef boost::numeric::ublas::identity_matrix<float> IdentityMatrix;
typedef boost::numeric::ublas::zero_matrix<float> ZeroMatrix;

using namespace std;
using namespace ibeo;

class ExtendedKalmanFilter
{
public:
  /**
   * Constructor, nothing done here
   **/
  ExtendedKalmanFilter();

  /**
   * Destructor, nothing done here
   **/
  ~ExtendedKalmanFilter();

  /**
   * Assume the first given Point3D to be the origin of a local
   * coordinate system and return the spherical coordinates of the
   * second point within this coordinate system.
   *
   * @param   Point3D origin of the local coordinate system
   * @param   Point3D to return in local, spherical coordinates
   * @return  Point3D the second param in local, spherical coordinates
   *
   * See formula (3.29) on page 36 in FastSLAM by Sebastian Thrun
   **/
  Point3D getRelativeSphericCoords_deprecated(Point3D origin, Point3D point);

  /**
   * This function basically is an inverse function of
   * getRelativeSphericCoords.
   *
   * @param   Point3D origin of the local coordinate system
   * @param   Point3D to return in global, cartesian coordinates
   * @return  Point3D the second param in global, cartesian coordinates
   *
   * See formula (3.29) on page 36 in FastSLAM by Sebastian Thrun
   **/
  Point3D getAbsoluteCartesianCoords_deprecated(Point3D origin, Point3D point);

  /**
   * Calculate Jacobian matrix with respect to a landmark from the landmarks position and the
   * updated robot pose of the particle.
   *
   * @param   Current, estimated robot pose of the particle
```

```
59      * @param   Mean position of the landmark before updating
60      * @return  Jacobian matrix
61      *
62      * See formula (3.36) on page 37 in FastSLAM by Sebastian Thrun
63      **/
64     Matrix landmarkJacobian(Position3D estimatedRobotPose, Point3D landmarkEKFMean);
65     Matrix landmarkJacobian_original(Position3D estimatedRobotPose, Point3D landmarkEKFMean);
66
67     /**
68      * Calculate Innovation Covariance matrix using a Jacobian matrix, the landmarks EKF covariance
69      * and the linearized measurement noise
70      *
71      * @param   Jacobian matrix with respect to the landmark
72      * @param   Covariance matrix of the landmarks EKF
73      * @param   Linearized vehicle measurement noise
74      * @return  Innovation Covariance matrix
75      *
76      * See formula (3.31) on page 36 in FastSLAM by Sebastian Thrun
77      **/
78     Matrix innovationCovariance(Matrix landmarkJacobian, Matrix landmarkEKFCovariance, Matrix
            measurementNoise);
79
80     /**
81      * Calculate EKF Gain from earlier EKF Covariance matrix, Jacobian and Innovation Covariance
82      *
83      * @param   Jacobian matrix with respect to the landmark
84      * @param   Covariance matrix of the landmarks EKF
85      * @param   Innovation Covariance matrix
86      * @return  EKF gain matrix
87      *
88      * See formula (3.32) on page 36 in FastSLAM by Sebastian Thrun
89      **/
90     Matrix ekfGain(Matrix landmarkJacobian, Matrix landmarkEKFCovariance, Matrix innovationCovariance)
            ;
91
92     /**
93      * Calculate EKF Mean from earlier EKF Mean position, EKF Gain and the positions of sensor
            observation
94      * and position of expected measurement of the landmark
95      *
96      * @param   Landmarks previous EKF Mean
97      * @param   EKF Gain
98      * @param   Position of the landmarks observation
99      * @param   Position of the landmarks expected measurement
100     * @return  EKF Mean landmark position
101     *
102     * See formula (3.33) on page 36 in FastSLAM by Sebastian Thrun
103     **/
104    Point3D ekfMean(Point3D landmarkEKFMean, Matrix ekfGain, Point3D observedLandmarkPosition, Point3D
            expectedLandmarkPosition);
105
106    /**
107     * Calculate EKF Covariance from EKF Gain matrix, Jacobian matrix and previous EKF Covariance
108     *
109     * @param   Jacobian matrix with respect to the landmark
110     * @param   Covariance matrix of the landmarks EKF
111     * @param   EKF Gain
112     * @return  EKF Covariance matrix
113     *
114     * See formula (3.34) on page 36 in FastSLAM by Sebastian Thrun
115     **/
116    Matrix ekfCovariance(Matrix landmarkJacobian, Matrix landmarkEKFCovariance, Matrix ekfGain, Matrix
            measurementNoise);
117
118    /**
```

```
119     * Convert a Point3D from cartesian coordinates to spherical
120     * coordinates
121     *
122     * @param Point3D containing x, y and z coordinate
123     * @return Point3D containing rho, phi and theta coordinate
124     **/
125    Point3D cart2sphere_deprecated(Point3D input);
126
127    /**
128     * Convert a Point3D from spherical coordinates to cartesian
129     * coordinates
130     *
131     * @param  Point3D containing rho, phi and theta coordinate
132     * @return Point3D containing x, y and z coordinate
133     **/
134    Point3D sphere2cart_deprecated(Point3D input);
135
136  private:
137    /**
138     * Create a 3,1-matrix representing a standing vector of a point
139     * Including a cartesian to spherical coordinates transformation
140     *
141     * @param  Point3D to create a matrix of
142     * @return A 3,1-matrix containing x, y and z of the input point
143     **/
144    Matrix createMatrixFromPoint3D(Point3D pt);
145
146    /**
147     * Create a Point3D from a 3,1-matrix holding the x, y and z
148     * coordinate
149     *
150     * @param A 3,1-matrix containing x, y and z coordinate
151     * @return A Point3D object
152     **/
153    Point3D createPoint3DFromMatrix(Matrix mat);
154  };
```

```
1   #include <IbeoAPI/Math.hpp>
2   #include "./ekf.hpp"
3   #include <iostream>
4   #include <boost/numeric/ublas/io.hpp>
5
6   using namespace ibeo;
7
8   ExtendedKalmanFilter::ExtendedKalmanFilter()
9   {
10  }
11
12  ExtendedKalmanFilter::~ExtendedKalmanFilter()
13  {
14  }
15
16  Matrix ExtendedKalmanFilter::landmarkJacobian(Position3D estimatedRobotPose, Point3D landmarkEKFMean
        )
17  {
18    return IdentityMatrix(3,3);
19  }
20
21  Matrix ExtendedKalmanFilter::landmarkJacobian_original(Position3D estimatedRobotPose, Point3D
        landmarkEKFMean)
22  {
23    const float r = landmarkEKFMean.getX() - estimatedRobotPose.getX();
24    const float t = landmarkEKFMean.getY() - estimatedRobotPose.getY();
25    const float p = landmarkEKFMean.getZ() - estimatedRobotPose.getZ();
```

```
26
27    Matrix jacMat (3, 3);
28
29    jacMat(0, 0) = cos(p) * sin(t);
30    jacMat(0, 1) = sin(p) * sin(t);
31    jacMat(0, 2) = cos(t);
32    jacMat(1, 0) = cos(p) * cos(t)/r;
33    jacMat(1, 1) = cos(t) * sin(p)/r;
34    jacMat(1, 2) = -(sin(t)/r);
35    jacMat(2, 0) = -((1/sin(t) * sin(p))/r);
36    jacMat(2, 1) = cos(p) * (1/sin(t))/r;
37    jacMat(2, 2) = 0;
38
39    return jacMat;
40  }
41
42  Matrix ExtendedKalmanFilter::innovationCovariance(Matrix landmarkJacobian, Matrix
        landmarkEKFCovariance, Matrix measurementNoise)
43  {
44    Matrix innoCovMat (3, 3);
45
46    innoCovMat = prod(landmarkJacobian, landmarkEKFCovariance);
47    innoCovMat = prod(innoCovMat, trans(landmarkJacobian));
48    innoCovMat += measurementNoise;
49
50    return innoCovMat;
51  }
52
53  Matrix ExtendedKalmanFilter::ekfGain(Matrix landmarkJacobian, Matrix landmarkEKFCovariance, Matrix
        innovationCovariance)
54  {
55    Matrix innoCovMatInv (ZeroMatrix(3, 3));
56
57    if (InvertMatrix(innovationCovariance, innoCovMatInv))
58    {
59      Matrix gain (3, 3);
60      gain = prod(landmarkEKFCovariance, trans(landmarkJacobian));
61      gain = prod(gain, innoCovMatInv);
62      return gain;
63    }
64    else
65    {
66      return IdentityMatrix();
67    }
68  }
69
70  Point3D ExtendedKalmanFilter::ekfMean(Point3D landmarkEKFMean, Matrix ekfGain, Point3D
        observedLandmarkPosition, Point3D expectedLandmarkPosition)
71  {
72    Matrix measurementInnovation (3, 1);
73
74    measurementInnovation = createMatrixFromPoint3D(observedLandmarkPosition -
          expectedLandmarkPosition);
75
76    Matrix oldMean = createMatrixFromPoint3D(landmarkEKFMean);
77
78    Matrix newMean = ZeroMatrix(3, 1);
79
80    newMean = oldMean + prod(ekfGain, measurementInnovation);
81
82    Point3D newMeanPt(createPoint3DFromMatrix(newMean));
83
84    return newMeanPt;
85  }
86
```

```
87  Matrix ExtendedKalmanFilter::ekfCovariance(Matrix landmarkJacobian, Matrix landmarkEKFCovariance,
        Matrix ekfGain, Matrix measurementNoise)
88  {
89    // I = IdentityMatrix
90    // K = Gain
91    // H = Jacobian
92    // P = Covariance
93    // Q = Measurment Noise
94
95    // IKH   = IdentityMatrix(sizeOfStateVector) - prod(K, H);
96    // PIKHt = prod(P, trans(IKH));
97    // QKt   = prod(Q, trans(K));
98    // P     = prod(IKH, PIKHt) + prod(K, QKt);
99
100   Matrix idGainJa = IdentityMatrix(3,3) - prod(ekfGain, landmarkJacobian);
101   Matrix gainIdJa = prod(landmarkEKFCovariance, trans(idGainJa));
102   Matrix measureGain = prod(measurementNoise, trans(ekfGain));
103   Matrix newCov = prod(idGainJa, gainIdJa) + prod(ekfGain, measureGain);
104
105   return newCov;
106 }
107
108 Matrix ExtendedKalmanFilter::createMatrixFromPoint3D(Point3D pt)
109 {
110   // standing vectors
111   Matrix result (3, 1);
112   result(0,0) = pt.getX();
113   result(1,0) = pt.getY();
114   result(2,0) = pt.getZ();
115   return result;
116 }
117
118 Point3D ExtendedKalmanFilter::createPoint3DFromMatrix(Matrix mat)
119 {
120   // standing vectors
121   Point3D result;
122   result.setX(mat(0,0));
123   result.setY(mat(1,0));
124   result.setZ(mat(2,0));
125   return result;
126 }
```

D.4 Particle

Implementation of a particle for the particle filter.

```
1  // Particle.hpp
2  // created: 2011-03-16
3  // author: Jan Gries, Jan Girlich
4
5  #ifndef PARTICLE_HPP
6  #define PARTICLE_HPP
7
8  #include <IbeoAPI/Point2D.hpp>
9  #include <IbeoAPI/Point3D.hpp>
10 #include <IbeoAPI/Position3D.hpp>
11 #include <ibeobasic/worker/fastslam/Landmark.hpp>
12 #include <IbeoAPI/Configurable.hpp>
13 #include <vector>
14 #include <deque>
```

```
15 #include <list>
16 #include <string>
17
18 // Boost includes
19 #include <boost/random/linear_congruential.hpp>
20 #include <boost/random/variate_generator.hpp>
21 #include <boost/random/normal_distribution.hpp>
22 #include <boost/random/uniform_real.hpp>
23 #include <boost/random/mersenne_twister.hpp>
24 #include <boost/date_time/posix_time/posix_time.hpp>
25
26 using namespace std;
27 using namespace ibeo;
28
29 typedef vector<Landmark> Sector;
30 typedef deque<deque<Sector> > SectorMap;
31 typedef list<Position3D> Position3DCollection;
32 typedef vector<Point3D> ZCollection;
33
34 class Particle
35 {
36
37 private:
38   Position3DCollection m_odometryUpdateList;
39   Sector m_map;
40   float m_rating;
41   float m_normRating;
42   Point2D m_zeroOfMap;
43   Position3D m_startPosition;
44   static boost::mt19937 generator;
45   vector<Landmark> simpleLmList;
46   size_t m_landmarkindex;
47   float m_perceptionrange;
48   string m_id;
49
50   ibeo::Position3D sampleMotionModelOdometry(
51     ibeo::Position3D oldPosition,
52     ibeo::Position3D newPosition,
53     ibeo::Position3D originatingSample,
54     float alphaYaw1,
55     float alphaPitch1,
56     float alphaTrans,
57     float alphaYaw2,
58     float alphaPitch2,
59     float alphaRoll,
60     float fYaw1,
61     float fYaw2);
62   Sector::iterator findLandmarkiteratorToIndexOfLandmark(size_t index);
63   double sample(double deviation);
64   double atan2pitch(double y, double x);
65   void extendSectorMap (const Landmark &landmark);
66   void addLandmark (Landmark landmark);
67   bool distLargerLimit(Point3D p1, Point3D p2, float maxDist);
68
69 public:
70   /**
71    * constuctor
72    * creates a particle with rating of 1.0
73    *
74    * @param Position3D startingposition
75    */
76   Particle(Position3D);
77
78   /**
79    * destuctor
```

```
80      */
81     ~Particle(){};
82
83     /**
84      * add a landmark to the simple map of the particle
85      */
86     void simpleAddLandmark (Landmark landmark);
87
88     /**
89      * returns a list of landmarks that stored in the map
90      */
91     vector<Landmark> simpleGetLmList();
92
93     /**
94      * add a landmark to the map of the particle
95      */
96     void addLandmarkToMap (Landmark landmark);
97
98     /**
99      * overwrites the landmark in the map
100     */
101    void updateLandmark (Landmark landmark);
102
103    /**
104     * add odometrydata to the trajectory of the particle
105     */
106    void addOdometryUpdate (const Position3D &odometryUpdate);
107
108    /**
109     * returns the last position of the particle
110     */
111    Position3D getlastOdometryUpdate();
112
113    /**
114     * returns the second last position of the particle
115     */
116    Position3D getSecondlastOdometryUpdate();
117
118    /**
119     * returns the rating of the particle, that represents whether the new observations are consistent
            to the landmarks stored in the map
120     */
121    float getRating() const;
122
123    /**
124     * sets the rating of the particle, that represents whether the new observations are consistent to
            the landmarks stored in the map
125     */
126    void setRating(float rating);
127
128    /**
129     * returns the normed rating of the particle, set by setNormedRating()
130     *
131     * @return float Normed rating
132     */
133    float getNormRating ();
134
135    /**
136     * sets the normed rating of the particle
137     *
138     * @param float Normed rating
139     */
140    void setNormRating(float rating);
141
142    /**
```

```
143     * returns a vector with the stored landmarks arount the position
144     */
145   Sector getMap (Position3D);
146
147   /**
148     * calculates a new position using the trajectory of the particle, the odomitridaten of the
          vehicle and some random parameter
149     */
150   void calculateNewPose(
151     Position3D &lastPosition,
152     Position3D &newPosition,
153     float alphaYaw1,
154     float alphaPitch1,
155     float alphaTrans,
156     float alphaYaw2,
157     float alphaPitch2,
158     float alphaRoll,
159     float fYaw1,
160     float fYaw2);
161
162   /**
163     * return the hole trajectory of the particle
164     */
165   Position3DCollection getOdometryList();
166
167   /**
168     * Gives all Landmarks stored in this Particle
169     *
170     * @return Sector of all stored Landmark objects
171     **/
172   Sector getAllLandmarks();
173 };
174
175 #endif
```

```
1  // Particle.cpp
2  // created: 2011-03-16
3  // author: Jan Gries, Jan Girlich
4
5  #include <ibeobasic/worker/fastslam/Particle.hpp>
6  #include <sstream>
7
8  typedef boost::numeric::ublas::identity_matrix<float> IdentityMatrix;
9
10 ibeo::Position3D Particle::sampleMotionModelOdometry(
11   ibeo::Position3D oldPosition,
12   ibeo::Position3D newPosition,
13   ibeo::Position3D originatingSample,
14   float alphaYaw1,
15   float alphaPitch1,
16   float alphaTrans,
17   float alphaYaw2,
18   float alphaPitch2,
19   float alphaRoll,
20   float fYaw1,
21   float fYaw2)
22 {
23   // the angles in movementVector are not used
24   ibeo::Position3D movementVector = newPosition - oldPosition;
25   traceWarning("") << "movementVector: " << movementVector.getX() << " " << movementVector.getY() <<
         " " << movementVector.getZ() << std::endl ;
26   if (movementVector.getX() == 0 && movementVector.getY() == 0 && movementVector.getZ() == 0)
27     traceWarning("") << "sampleMotionModelOdometry = (0,0,0)" << std::endl;
28
```

```
29    // calculating the angles for the first rotation to point in the direction of movementVector
30    float rot1yaw = atan2 (movementVector.getY(), movementVector.getX()) - oldPosition.getYawAngle();
31    float rot1pitch = atan2pitch (sqrt(movementVector.getY() * movementVector.getY() + movementVector.
          getX() * movementVector.getX()), movementVector.getZ()) - oldPosition.getPitchAngle();
32
33    // calculating the distance to the new position (length of movementVector)
34    float trans = sqrt(movementVector.getX() * movementVector.getX() + movementVector.getY() *
          movementVector.getY() + movementVector.getZ() * movementVector.getZ());
35
36    // calculating the angles to turn to the final orientation at the newPosition
37    float rot2yaw = newPosition.getYawAngle() - rot1yaw - oldPosition.getYawAngle();
38    float rot2pitch = newPosition.getPitchAngle() - rot1pitch - oldPosition.getPitchAngle();
39    float rot2roll = newPosition.getRollAngle() - oldPosition.getRollAngle();
40
41    // add noise, could be improved by considering the curves actually travelled and its dependencies
42    float rot1yawSample = rot1yaw + (fYaw1 * rot1yaw) - sample( alphaYaw1 * rot1yaw);
43    float rot1pitchSample = rot1pitch - sample( alphaPitch1 * rot1pitch);
44
45    float transSample = trans - sample( alphaTrans * trans );
46
47    // rot2 usually depends on rot1, but this dependency is disregarded here
48    // usually a small rot1 leads to a small rot2
49    float rot2yawSample = rot2yaw + (fYaw2 * rot2yaw) - sample( alphaYaw2 * rot2yaw);
50    float rot2pitchSample = rot2pitch - sample( alphaPitch2 * rot2pitch);
51
52    float rot2rollSample = rot2roll - sample( alphaRoll * rot2roll );
53
54    // minimize the influence of small movements like they occur from
55    // noisy GPS data when the vehicle is not moving, to prevent
56    // sudden unwanted extreme orientation changes
57    float factor = 1;
58    if (movementVector.getPoint().distFromOrigin() < 0.1)
59    {
60      factor = movementVector.getPoint().distFromOrigin();
61    }
62
63    // now convert the resulting samples into cartesian coordinates
64    // first convert movementVectorSample:
65    ibeo::Position3D relativePositionSample;
66    relativePositionSample.setX( transSample * cos(rot1yawSample + originatingSample.getYawAngle()) *
          cos(rot1pitchSample + originatingSample.getPitchAngle()) );
67    relativePositionSample.setY( transSample * sin(rot1yawSample + originatingSample.getYawAngle()) *
          cos(rot1pitchSample + originatingSample.getPitchAngle()) );
68    relativePositionSample.setZ( -transSample * sin(rot1pitchSample + originatingSample.getPitchAngle
          ()) );
69    relativePositionSample.setYawAngle( originatingSample.getYawAngle() + (rot1yawSample +
          rot2yawSample)*factor );
70    relativePositionSample.setPitchAngle( originatingSample.getPitchAngle() + (rot1pitchSample +
          rot2pitchSample)*factor );
71    relativePositionSample.setRollAngle( originatingSample.getRollAngle() + (rot2rollSample)*factor );
72
73    // add the estimated motion to the originating sample position in an
74    // absolute coordinate system
75    ibeo::Position3D absolutePositionSample;
76    absolutePositionSample.setX( relativePositionSample.getX() + originatingSample.getX() );
77    absolutePositionSample.setY( relativePositionSample.getY() + originatingSample.getY() );
78    absolutePositionSample.setZ( relativePositionSample.getZ() + originatingSample.getZ() );
79    absolutePositionSample.setYawAngle( relativePositionSample.getYawAngle() );
80    absolutePositionSample.setPitchAngle( relativePositionSample.getPitchAngle() );
81    absolutePositionSample.setRollAngle( relativePositionSample.getRollAngle() );
82
83    return absolutePositionSample;
84  }
85
86  Position3DCollection Particle::getOdometryList()
```

```
87  {
88    return m_odometryUpdateList;
89  }
90
91  void Particle::simpleAddLandmark(Landmark landmark)
92  {
93    simpleLmList.push_back(landmark);
94  }
95
96  vector<Landmark> Particle::simpleGetLmList()
97  {
98    return simpleLmList;
99  }
100
101 double Particle::atan2pitch(double y, double x)
102 {
103   double result = 0;
104
105   if (x != 0) // if x == 0 then do nothing and return 0
106   {
107     result = - atan(y / x) + M_PI / 2;
108
109     if (result > M_PI / 2)
110     {
111       result -= M_PI;
112     }
113   }
114
115   return result;
116 }
117
118 double Particle::sample(double deviation)
119 {
120   boost::normal_distribution<> norm_dist(0, 1);
121   boost::variate_generator<boost::mt19937&, boost::normal_distribution<> > boost_nrand(generator,
          norm_dist);
122
123   return boost_nrand() * fabs(deviation);
124 }
125
126 boost::mt19937 Particle::generator(static_cast<unsigned int>(std::time(NULL)));
127
128 Particle::Particle(Position3D startPosition)
129 {
130   m_id = "0";
131   m_rating = 0;
132   m_landmarkindex = 0;
133   m_perceptionrange = 30.0;
134   m_startPosition = startPosition;
135 }
136
137 void Particle::addLandmarkToMap (Landmark landmark)
138 {
139   addLandmark(landmark);
140 }
141
142 void Particle::addLandmark(Landmark landmark)
143 {
144   landmark.index = m_landmarkindex;
145   ++m_landmarkindex;
146   m_map.push_back(landmark);
147 }
148
149 void Particle::calculateNewPose(Position3D &lastPosition, Position3D &newPosition, float alphaYaw1,
      float alphaPitch1, float alphaTrans, float alphaYaw2, float alphaPitch2, float alphaRoll, float
```

```
      fYaw1, float fYaw2)
{
  Position3D temp = sampleMotionModelOdometry(lastPosition, newPosition, getlastOdometryUpdate(),
       alphaYaw1, alphaPitch1, alphaTrans, alphaYaw2, alphaPitch2, alphaRoll, fYaw1, fYaw2);
  traceWarning("") << "Particle::sampleMotionModelOdometry() returns " << temp.getX() << " , " <<
       temp.getY() << " , " << temp.getZ() << " Yaw:" << temp.getYawAngle() << "  Pitch:" << temp.
       getPitchAngle() << "  Roll: " << temp.getRollAngle() << std::endl;
  addOdometryUpdate(temp);
}


void Particle::addOdometryUpdate (const Position3D &odometryUpdate)
{
  m_odometryUpdateList.push_back(odometryUpdate);
}


Position3D Particle::getlastOdometryUpdate()
{
  if (m_odometryUpdateList.size() < 1)
    return m_startPosition;
  else
    return m_odometryUpdateList.back();
}

Position3D Particle::getSecondlastOdometryUpdate()
{

  if (m_odometryUpdateList.size() < 1)
    return m_startPosition;
  else if (m_odometryUpdateList.size() == 1)
    return m_startPosition;
  else
  {
    Position3DCollection::reverse_iterator rit = m_odometryUpdateList.rbegin();
    return *(rit);
  }
}


float Particle::getRating() const
{
  return m_rating;
}

void Particle::setRating (float rating)
{
  m_rating = rating;
  assert(getRating() == rating);
}

float Particle::getNormRating()
{
  return m_normRating;
}

void Particle::setNormRating (float rating)
{
  m_normRating = rating;
  assert(getNormRating() == rating);
}

void Particle::updateLandmark(Landmark landmark)
{
  Sector::iterator iterSector = findLandmarkiteratorToIndexOfLandmark(landmark.index);
```

```
211   assert(iterSector->index == landmark.index);
212   *iterSector = landmark;
213 }
214
215 Sector::iterator Particle::findLandmarkiteratorToIndexOfLandmark(size_t index)
216 {
217   Sector::iterator iterSector;
218   for (iterSector = m_map.begin(); iterSector != m_map.end(); ++iterSector)
219   {
220     if (iterSector->index == index)
221     {
222       return iterSector;
223     }
224   }
225   traceError("") << "Index: " << index << endl;
226   assert(false); //you should not be here!
227   return iterSector;
228 }
229
230 Sector Particle::getMap (Position3D position)
231 {
232   Sector returnSector;
233   for (Sector::iterator iterSector = m_map.begin(); iterSector != m_map.end(); ++iterSector)
234   {
235     if(! distLargerLimit(position.getPoint(), iterSector->mean, m_perceptionrange))
236     {
237       returnSector.push_back(*iterSector);
238     }
239   }
240   return returnSector;
241 }
242
243 bool Particle::distLargerLimit(Point3D p1, Point3D p2, float maxDist)
244 {
245   if (fabs(p1.getX() - p2.getX()) > maxDist)
246     return true;
247   else if (fabs(p1.getY() - p2.getY()) > maxDist)
248     return true;
249   else if (fabs(p1.getZ() - p2.getZ()) > maxDist)
250     return true;
251   else if (p1.dist(p2) > maxDist)
252     return true;
253   else
254     return false;
255 }
256
257 Sector Particle::getAllLandmarks()
258 {
259   return m_map;
260 }
```

D.5 Likelihood table

The likelihood table holds all likelihoods between the landmarks within a particle and the observations made.

```
1
2 #ifndef likelihoodtable_HPP
3 #define likelihoodtable_HPP
4
```

```
5  #include <math.h>
6
7  #include <boost/numeric/ublas/matrix.hpp>
8  #include <boost/numeric/ublas/lu.hpp>
9  #include "ibeoextended/recttracking/InvertMatrix.hpp"
10 typedef boost::numeric::ublas::matrix<float> Matrix;
11 typedef boost::numeric::ublas::permutation_matrix<size_t> PermMatrix;
12 typedef boost::numeric::ublas::identity_matrix<float> IdentityMatrix;
13
14 #include <IbeoAPI/ScanPoint.hpp>
15 #include <IbeoAPI/Point3D.hpp>
16 #include <IbeoAPI/Position3D.hpp>
17 #include "ibeograph/Preferences.hpp"
18 #include "ibeobasic/worker/fastslam/ekf.hpp"
19
20 using namespace std;
21 namespace ibeo {
22 namespace appbase {
23 namespace worker {
24
25 class Likelihoodtable
26 {
27 public:
28   /**
29    * Constructor
30    *
31    * @param Position3D The particles current robot pose
32    * @param Position3D The particles previous robot pose
33    * @param Matrix(3,3) measurement noise of the sensor
34    **/
35     Likelihoodtable(Position3D lastRobotPose, Position3D secondLastRobotPose, Matrix matrix);
36
37   /**
38    * Destructor
39    **/
40     ~Likelihoodtable();
41
42   /**
43    * Adds a new expected Landmark position to the table and calculates
44    * the likelihoods.
45    *
46    * @param Landmark the Landmark
47    **/
48     void addLandmark(Landmark landmark);
49
50   /**
51    * Adds a list of new expected Landmark positions to the table and
52    * calculates the likelihoods. Replaces all prior added Landmarks.
53    *
54    * @param vector<Landmark> Landmarks
55    **/
56   void initLandmarks(vector<Landmark> landmarks);
57
58   /**
59    * Adds a new observation to the table and calculates the
60    * likelihoods.
61    *
62    * @param Point3D Observation
63    **/
64     void addObservation(Point3D point3D);
65
66   /**
67    * Adds all observations at once to the table and calculates the
68    * likelihoods. Replaces all prior added Observations.
69    *
```

```
70      * @param vector<Point3D> Observations
71      **/
72     void initObservations(vector<Point3D> observations);
73
74     /**
75      * Call this method after adding Landmarks and Observations to
76      * calculate the Likelihoods and fill the table.
77      **/
78     void calculateLikelihoods();
79
80     /**
81      * Get the value of the highest Likelihood for a given Observation.
82      *
83      * @param Point3D Observation
84      * @return float maximum Likelihood
85      **/
86       float getMaxLikelihoodForObservation(Point3D point3D);
87
88     /**
89      * Get the expected Landmark position of the Landmark with the
90      * highest Likelihood for a given Observation.
91      *
92      * @param Point3D Observation
93      * @return Point3D expected Position of the Landmark with the maximum Likelihood
94      **/
95       Landmark getLandmarkWithMaxLikelihood(Point3D point3D);
96
97     /**
98      * Finds the globally best Likelihood and removes all other
99      * Likelihoods for this Landmark and Observation. Does this for all
100     * Likelihoods and thus adding Landmarks or Observations afterwards
101     * won't work.
102     **/
103    void findBestAssociations();
104
105    /**
106     * Finds the globally best Likelihood and removes all other
107     * Likelihoods for all Landmarks and Observations. Might leave
108     * Landmarks without any association in some cases where an
109     * observation cannot be associated unambigously.
110     **/
111    void findBestGlobalAssociations();
112
113    /**
114     * Returns the internal table storing all Likelihoods. For debugging
115     * purposes only.
116     **/
117    Matrix getTable();
118
119  private:
120    Matrix m_likelihoods;
121    Position3D m_lastRobotPose;
122    Position3D m_secondLastRobotPose;
123    Matrix m_measurementNoise;
124
125    // colum indicators
126    vector<Point3D> m_observations;
127
128    // row indicators
129    vector<Landmark> m_landmarks;
130    vector<Point3D> m_expLmPos;
131
132    /**
133     * Returns the Observation with the highest likelihood for the
134     * given expected Landmark position. Needed for checking if the
```

```
135     * expected Landmark Position does not have a better possible
136     * association.
137     *
138     * @param Point3D expected Landmark position
139     * @return Point3D Observation
140     **/
141      Point3D getObsWithMaxLikelihood(Landmark landmark);
142
143      /**
144       * Calculate the Likelihood for the cell in 'row' and 'column' and
145       * insert it.
146       *
147       * @param size_t Row
148       * @param size_t Column
149       **/
150      void calculateLikelihood(size_t row, size_t column);
151
152    /**
153     * Fetches the index of the Observation list indicating the
154     * Observation with the highest Likelihood value. This function is a
155     * helper for fetching the Likelihood value or the associated
156     * landmark.
157     *
158     * @param Point3D Observation
159     * @return size_t index of the Observation list
160     **/
161    size_t getIndexOfObservation(Point3D observation);
162
163    /**
164     * Fetches the index of the Landmark list indicating the Landmark
165     * with the highest Likelihood value. This function is a helper for
166     * fetching the Likelihood value or the associated landmark.
167     *
168     * @param size_t index of the Observation list
169     * @return size_t index of the Landmark list
170     **/
171    size_t getIndexOfLandmarkWithMaxLikelihood(size_t observation);
172
173    /**
174     * Returns the sign of the determinant of the given matrix. This is
175     * a helper function to calculate the determinant.
176     *
177     * @param Matrix matrix for which to calculate the determinants sign
178     * @return int sign of determinant of given matrix
179     **/
180    int determinant_sign(const PermMatrix& permMatrix);
181
182    /**
183     * Returns the the determinant of the given matrix.
184     *
185     * @param Matrix matrix for which to calculate the determinant
186     * @return float determinant of given matrix
187     **/
188    float determinant(Matrix& matrix);
189
190    /**
191     * Efficiently tests if the distance between two given points is
192     * larger than a maximum.
193     *
194     * @param Point3D first point
195     * @param Point3D second point
196     * @param float maximum distance
197     * @return bool true if distance between point 1 and point 2 is
198     * larger than maximum distance, else false.
199     **/
```

```
200   bool distLargerLimit(Point3D p1, Point3D p2, float maxDist);
201 };
202
203 }
204 }
205 }
206
207 #endif
```

```
1  // Likelihoodtable.cpp
2  // Likelihood table implementation using a Matrix container
3  // created: 2011-03-11
4  // author: Jan Girlich
5
6  #include <ibeobasic/worker/fastslam/Likelihoodtable.hpp>
7  #include <limits>
8  #include <algorithm>
9  #include <boost/bind.hpp>
10
11 using namespace std;
12 namespace ibeo {
13 namespace appbase {
14 namespace worker {
15
16 struct rowMaximum
17 {
18   size_t rowIndex;
19   size_t columnIndex;
20   float likelihood;
21 } ;
22
23 float m_maxDistance = 3; // in meters
24
25 Likelihoodtable::Likelihoodtable(Position3D lastRobotPose, Position3D secondLastRobotPose, Matrix
       measurementNoise)
26 {
27   m_lastRobotPose = lastRobotPose;
28   m_secondLastRobotPose = secondLastRobotPose;
29   m_measurementNoise = measurementNoise;
30 }
31
32 Likelihoodtable::~Likelihoodtable()
33 {
34 }
35
36 void Likelihoodtable::addLandmark(Landmark landmark)
37 {
38   m_landmarks.push_back(landmark);
39   m_likelihoods.resize(m_likelihoods.size1()+1, m_likelihoods.size2());
40
41   for (size_t i=0; i<m_likelihoods.size2(); ++i)
42   {
43     calculateLikelihood(m_likelihoods.size1()-1, i);
44   }
45 }
46
47 void Likelihoodtable::initLandmarks(vector<Landmark> landmarks)
48 {
49   m_landmarks = landmarks;
50 }
51
52 void Likelihoodtable::addObservation(Point3D observation)
53 {
54   m_observations.push_back(observation);
```

```
55    m_likelihoods.resize(m_likelihoods.size1(), m_likelihoods.size2()+1);
56
57    for (size_t i=0; i<m_likelihoods.size1(); ++i)
58    {
59      calculateLikelihood(i, m_likelihoods.size2()-1);
60    }
61  }
62
63  void Likelihoodtable::calculateLikelihoods()
64  {
65    m_likelihoods.resize(m_landmarks.size(), m_observations.size());
66    traceNote("") << "Likelihood Table rows: " << m_likelihoods.size1() << " columns: " <<
          m_likelihoods.size2() << endl;
67    for (size_t i=0; i<m_likelihoods.size1(); ++i)
68    {
69      for (size_t j=0; j<m_likelihoods.size2(); ++j)
70        calculateLikelihood(i, j);
71    }
72  }
73
74  void Likelihoodtable::initObservations(vector<Point3D> observations)
75  {
76    m_observations = observations;
77  }
78
79
80  float Likelihoodtable::getMaxLikelihoodForObservation(Point3D observation)
81  {
82    size_t observationIndex = getIndexOfObservation(observation);
83    size_t lmIndex = getIndexOfLandmarkWithMaxLikelihood(observationIndex);
84
85    if ((m_likelihoods.size1() < 1) || (m_likelihoods.size2() < 1))
86      return -1;
87    else
88      return m_likelihoods(lmIndex, observationIndex);
89  }
90
91  Landmark Likelihoodtable::getLandmarkWithMaxLikelihood(Point3D observation)
92  {
93    size_t observationIndex = getIndexOfObservation(observation);
94    size_t lmIndex = getIndexOfLandmarkWithMaxLikelihood(observationIndex);
95
96    if (m_landmarks.size() < 1)
97    {
98      traceWarning("") << "Landmark requested from Likelihood Table when none was added before" <<
            endl;
99      Landmark lm;
100     lm.mean = Point3D(0,0,0);
101     lm.covariance = IdentityMatrix(3,3);
102     return lm;
103   }
104   else
105     return m_landmarks[lmIndex];
106 }
107
108 // Used for checking for better associations
109 Point3D Likelihoodtable::getObsWithMaxLikelihood(Landmark landmark)
110 {
111   // find index for observation
112   size_t landmarkIndex = 0;
113   for (size_t i=0; i<m_landmarks.size(); ++i)
114     if (m_landmarks[i].mean == landmark.mean)
115       landmarkIndex = i;
116
117   float maxLikelihood = 0.0;
```

```
118   size_t index = 0;
119   for (size_t i=0; i<m_likelihoods.size2(); ++i)
120     if (m_likelihoods(index, landmarkIndex) > maxLikelihood)
121     {
122       maxLikelihood = m_likelihoods(landmarkIndex, i);
123       index = i;
124     }
125
126   return m_observations[index];
127 }
128
129 void Likelihoodtable::findBestAssociations()
130 {
131   float lastGlobalMax = numeric_limits<float>::max();
132
133   for (size_t i=0; i<min(m_likelihoods.size1(),m_likelihoods.size2()); ++i)
134   {
135     float maxLikelihood = 0;
136     size_t maxRow = std::numeric_limits<std::size_t>::max();
137     size_t maxColumn = std::numeric_limits<std::size_t>::max();
138     for (size_t row=0; row<m_likelihoods.size1(); ++row)
139     {
140       for (size_t column=0; column<m_likelihoods.size2(); ++column)
141       {
142         if ((m_likelihoods(row, column) > maxLikelihood) && (m_likelihoods(row, column) <
                lastGlobalMax))
143         {
144           maxLikelihood = m_likelihoods(row, column);
145           maxRow = row;
146           maxColumn = column;
147         }
148       }
149     }
150
151     if ((maxRow < std::numeric_limits<std::size_t>::max()) && (maxColumn < std::numeric_limits<std::
           size_t>::max()))
152     {
153       lastGlobalMax = maxLikelihood;
154
155       for (size_t row=0; row<m_likelihoods.size1(); ++row)
156         if (row != maxRow)
157           m_likelihoods(row, maxColumn) = 0;
158
159       for (size_t column=0; column<m_likelihoods.size2(); ++column)
160         if (column != maxColumn)
161           m_likelihoods(maxRow, column) = 0;
162     }
163   }
164 }
165
166 bool compareRowMaxima(const rowMaximum& i, const rowMaximum& j)
167 {
168   // '>' is descending order
169   return i.likelihood > j.likelihood;
170 }
171
172 void Likelihoodtable::findBestGlobalAssociations()
173 {
174   vector<rowMaximum> globalRowMaxima;
175
176   // get maximum likelihood for each row
177   for (size_t row=0; row<m_likelihoods.size1(); ++row)
178   {
179     rowMaximum maxRowLikelihood;
180     maxRowLikelihood.likelihood = 0.0;
```

```
181
182     for (size_t column=0; column<m_likelihoods.size2(); ++column)
183     {
184       const float testedLikelihood = m_likelihoods(row, column);
185       if (testedLikelihood > maxRowLikelihood.likelihood)
186       {
187         maxRowLikelihood.likelihood = testedLikelihood;
188         maxRowLikelihood.rowIndex = row;
189         maxRowLikelihood.columnIndex = column;
190       }
191     }
192     if (maxRowLikelihood.likelihood > 0.0)
193       globalRowMaxima.push_back(maxRowLikelihood);
194   }
195
196   sort(globalRowMaxima.begin(), globalRowMaxima.end(), bind(&compareRowMaxima, _1, _2));
197
198   for (vector<rowMaximum>::iterator maxIt = globalRowMaxima.begin(); maxIt != globalRowMaxima.end();
          ++maxIt)
199   {
200     for (size_t row=0; row<m_likelihoods.size1(); ++row)
201       if (row != maxIt->rowIndex)
202         m_likelihoods(row, maxIt->columnIndex) = 0;
203
204     for (size_t column=0; column<m_likelihoods.size2(); ++column)
205       if (column != maxIt->columnIndex)
206         m_likelihoods(maxIt->rowIndex, column) = 0;
207   }
208 }
209
210 Matrix Likelihoodtable::getTable()
211 {
212   return m_likelihoods;
213 }
214
215
216 size_t Likelihoodtable::getIndexOfObservation(Point3D observation)
217 {
218   // find index for observation
219   size_t observationIndex = 0;
220   for (size_t i=0; i<m_observations.size(); ++i)
221     if (m_observations[i] == observation)
222       observationIndex = i;
223
224   return observationIndex;
225 }
226
227 size_t Likelihoodtable::getIndexOfLandmarkWithMaxLikelihood(size_t observationIndex)
228 {
229   float maxLikelihood = 0.0;
230   size_t index = 0;
231   for (size_t i=0; i<m_likelihoods.size1(); ++i)
232     if (m_likelihoods(i, observationIndex) > maxLikelihood)
233     {
234       maxLikelihood = m_likelihoods(i, observationIndex);
235       index = i;
236     }
237
238   return index;
239 }
240
241 // for the formulas see Thrun, Montemerlo: FastSLAM, p. 45
242 void Likelihoodtable::calculateLikelihood(size_t row, size_t column)
243 {
244   ExtendedKalmanFilter ekf = ExtendedKalmanFilter();
```

```
245   float likelihood = 0.0;
246
247   if (! distLargerLimit(m_landmarks[row].mean, m_observations[column], m_maxDistance))
248   {
249     Point3D pt = m_observations[column] - m_landmarks[row].mean;
250     Matrix innovation (3, 1);
251     innovation(0,0) = pt.getX();
252     innovation(1,0) = pt.getY();
253     innovation(2,0) = pt.getZ();
254
255     Matrix jacobian = ekf.landmarkJacobian(m_lastRobotPose, m_landmarks[row].mean);
256
257     Matrix innovationCovariance = ekf.innovationCovariance(jacobian, m_landmarks[row].covariance,
          m_measurementNoise);
258
259     // check for calculation instabilities. This if clause will
260     // come true if the matrix holds a NaN
261     if (! (innovationCovariance(0,0) == innovationCovariance(0,0)))
262     {
263       cout << "=== InnoCov: " << innovationCovariance << endl;
264       cout << "jacobian: " << jacobian << endl;
265       cout << "covariance: " << m_landmarks[row].covariance << endl;
266       cout << "measurementNoise: " << m_measurementNoise << endl;
267       assert(false);
268     }
269
270     Matrix invInnoCov = IdentityMatrix(3,3);
271     if (! InvertMatrix(innovationCovariance, invInnoCov))
272       traceError("") << "Inversion of Matrix failed. Identity Matrix used instead.\n";
273
274     Matrix likelihoodMatrix = prod(trans(innovation), invInnoCov);
275     likelihoodMatrix = prod(likelihoodMatrix, innovation);
276
277     assert(innovationCovariance(0,0) == innovationCovariance(0,0));
278
279     double denominator = pow(2*M_PI,1.5)*sqrt(determinant(innovationCovariance));
280     double numerator = exp( -0.5 * likelihoodMatrix(0,0) );
281     likelihood = numerator/denominator;
282   }
283
284   m_likelihoods(row, column) = likelihood;
285 }
286
287 // Function taken from http://www.anderswallin.net/2010/05/matrix-determinant-with-boostublas/
288 int Likelihoodtable::determinant_sign(const PermMatrix& pm)
289 {
290   int pm_sign=1;
291   size_t size = pm.size();
292   for (size_t i = 0; i < size; ++i)
293     if (i != pm(i))
294       pm_sign *= -1.0; // swap_rows would swap a pair of rows here, so we change sign
295   return pm_sign;
296 }
297
298 float Likelihoodtable::determinant( Matrix& m )
299 {
300   PermMatrix pm(m.size1());
301   float det = 1.0;
302   if( boost::numeric::ublas::lu_factorize(m,pm) ) {
303     det = 0.0;
304   } else {
305     for(int i = 0; i < (int)m.size1(); i++)
306       det *= m(i,i); // multiply by elements on diagonal
307     det = det * determinant_sign( pm );
308   }
```

```
309   return det;
310 }
311
312 bool Likelihoodtable::distLargerLimit(Point3D p1, Point3D p2, float maxDist)
313 {
314   if (fabs(p1.getX() - p2.getX()) > maxDist)
315     return true;
316   else if (fabs(p1.getY() - p2.getY()) > maxDist)
317     return true;
318   else if (fabs(p1.getZ() - p2.getZ()) > maxDist)
319     return true;
320   else if (p1.dist(p2) > maxDist)
321     return true;
322   else
323     return false;
324 }
325
326 }
327 }
328 }
```

Relaxation

E

The relaxation worker was implemented for smoothing the jumpy trajectory of the Xsens MTi-G data and to smoothen out large correction steps after longer periods of dead reckoning like in tunnels.

```
/*
 * File:    relaxationWorker.hpp
 * Author: Jan Gries
 * RelaxationWorker.hpp
 * Copyright (c) Ibeo Automobile Sensor GmbH, 2008-2009
 * Created on November 24, 2010, 10:18 AM
 */

#ifndef _RELAXATIONWORKER_HPP
#define _RELAXATIONWORKER_HPP

#include <IbeoAPI/PositionWGS84.hpp>
#include <IbeoAPI/VehicleStateBasic.hpp>

#include <IbeoAPI/Configurable.hpp>

#include "ibeobasic/ibeobasicdecl.hpp"

#include "ibeograph/drain/PositionWGS84Drain.hpp"
#include "ibeograph/drain/VehicleStateDrain.hpp"
#include "ibeograph/source/PositionWGS84Source.hpp"

#include "ibeograph/Preferences.hpp"

#include <boost/circular_buffer.hpp>

using namespace std;
namespace ibeo {
namespace appbase {
namespace worker {

/**
 * \brief This worker modifies the PositionWGS84-data according to the
 * vehicle state, to get a smoothed trajectory.
 *
 * This Worker delayes the PositionWGS Messages!
 *
 * Output Data: new WGS84-dates
 */

class IBEOBASICDECL RelaxationWorker :  public ibeo::appbase::drain::PositionWGS84Drain,
                    public ibeo::appbase::drain::VehicleStateDrain,
                    public ibeo::appbase::source::PositionWGS84Source,
                    public Configurable
{
```

```
46  public:
47    /// constructor (loads the mounting position out of the configuration file)
48    RelaxationWorker(const ibeo::Preferences& preferences, const std::string& objectname = "");
49
50    ~RelaxationWorker(){};
51
52    static const std::string& getDefaultTypeName() { return CONFIG_TYPE; }
53
54  private:
55    /** Name of the type of the configuration block for this file. */
56    static const std::string CONFIG_TYPE;
57
58    struct State {
59      VehicleStateBasic vehicleState;
60      PositionWGS84 wgs84;
61    };
62
63    INT16 m_deviceID;
64    void setPositionWGS84 (const PositionWGS84 &posWGS84);
65    void setVehicleState (const VehicleStateBasic &vehicleStateBasic);
66    void relaxation();
67    void relaxationalgo(boost::circular_buffer<State>::iterator , boost::circular_buffer<State>::
          iterator , boost::circular_buffer<State>::iterator );
68
69    INT16 m_numOfWGS84ToSmooth;
70    INT8 m_numOfIterations;
71    INT8 m_usedDeviceID;
72    INT16 m_numWaitingSteps;
73    VehicleStateBasic m_vehicleState;
74    bool m_getVehicleState;
75    PositionWGS84 m_WGS84;
76    boost::circular_buffer<State> m_statebuffer;
77    int m_newStatesLeft;
78  };
79
80  }
81  }
82  }
83
84  #endif  /* _RELAXATIONWORKER_HPP */
```

```
1   #include "RelaxationWorker.hpp"
2
3   #include "Geom3D.hpp"
4   #include <boost/numeric/ublas/matrix.hpp>
5   #include <boost/numeric/ublas/vector.hpp>
6   #include <boost/numeric/ublas/io.hpp>
7   #include <boost/numeric/ublas/operation.hpp>
8
9   using namespace std;
10
11  namespace ibeo{
12  namespace appbase{
13  namespace worker{
14
15  const std::string RelaxationWorker::CONFIG_TYPE = "RelaxationWorker";
16
17  RelaxationWorker::RelaxationWorker(const ibeo::Preferences &preferences, const std::string&
        objectname)
18    : Configurable(getDefaultTypeName(), objectname)
19    , m_deviceID(-1)
20  {
21    define (new ParamINT16 ("numOfWGS84ToSmooth", m_numOfWGS84ToSmooth, "Number of WGS84 >0 included
          to the relaxationalgorithm.","40"));
```

```
22    define (new ParamINT8 ("numOfIterations", m_numOfIterations, "Times the algorithm repeats to
          improve the result before geting new PositionWGS84","2"));
23    define (new ParamINT8 ("usedDeviceID", m_usedDeviceID, "DeviceID of the Sensor(IMU) should been
          used ","31"));
24    define (new ParamINT16 ("numWaitingSteps", m_numWaitingSteps, "Number waiting Steps (in WGS84
          message) before restarting relaxationalgorithm.","10"));
25
26    fillValuesOrException (preferences.getConfigValues (getDefaultTypeName(), objectname));
27
28    // Load values from the config file, fill them into our
29    // parameters, and throw an exception if something was invalid.
30    fillValuesValidateOrExcept(preferences.findByTypeAndName(getDefaultTypeName(), objectname));
31
32    m_statebuffer = boost::circular_buffer<State>(m_numOfWGS84ToSmooth);
33  }
34
35  /*
36   * saves the VehicleState and starts the algorithm
37   */
38  void RelaxationWorker::setVehicleState (const VehicleStateBasic &vehicleStateBasic)
39  {
40    if (m_getVehicleState)
41    {
42      m_vehicleState = vehicleStateBasic;
43      State state;
44      state.vehicleState = vehicleStateBasic;
45      state.wgs84 = m_WGS84;
46      m_statebuffer.push_back(state);
47
48      if (m_statebuffer.full())
49      {
50        for (int i=0; i<m_numOfIterations; ++i)
51        {
52          relaxation();
53        }
54        for (int i=0; i<m_numWaitingSteps; ++i)
55        {
56          m_signalPositionWGS84(m_statebuffer.front().wgs84);
57          m_statebuffer.pop_front();
58        }
59      }
60    }
61  }
62
63  /*
64   * saves the position
65   */
66  void RelaxationWorker::setPositionWGS84 (const PositionWGS84 &WGS84Basic)
67  {
68    if(WGS84Basic.getDeviceID() == m_usedDeviceID)
69    {
70      m_WGS84 = WGS84Basic;
71      m_getVehicleState = true;
72    }
73  }
74
75  /*
76   * smooth the States saved in m_statebuffer
77   */
78  void RelaxationWorker::relaxation()
79  {
80    traceDebug("") << " start of relaxation" << "\n";
81    for (boost::circular_buffer<State>::iterator iter2 = (m_statebuffer.begin()) + 2; iter2 !=
          m_statebuffer.end() ; iter2 = iter2 +1 )
82    {
```

```
83      boost::circular_buffer<State>::iterator iter1 = iter2 -1;
84      boost::circular_buffer<State>::iterator iter0 = iter1 -1;
85      relaxationalgo(iter0 , iter1 , iter2);
86
87    }
88  }
89
90  inline void RelaxationWorker::relaxationalgo(boost::circular_buffer<State>::iterator iter0 , boost::
        circular_buffer<State>::iterator iter1 , boost::circular_buffer<State>::iterator iter2 )
91  {
92    // calculation of the vector from iter2 to its predecessor based on the angles and vehicleState of
          iter2
93    ibeo::VehicleStateBasic::RelativeVehicle::RelativeVehicle relativeVehicleState((iter1)->
          vehicleState, iter2->vehicleState); //current and previous interchanged because of need of
          negative vector
94
95    ibeo::geom3d::HMatrix  m   =   ibeo::geom3d::rotationRoll  (-(iter2->wgs84.getRollAngleInRad()));
96    m = boost::numeric::ublas::prod (ibeo::geom3d::rotationPitch(-(iter2->wgs84.getPitchAngleInRad()))
            , m);
97    m = boost::numeric::ublas::prod (ibeo::geom3d::rotationYaw(-(iter2->wgs84.getYawAngleInRad()))  ,
          m);
98    ibeo::geom3d::HVector relativeWorldState_21 = boost::numeric::ublas::prod(m ,(ibeo::geom3d::
          makeHVectorRect(relativeVehicleState.getDeltaPos().getX(),relativeVehicleState.getDeltaPos().
          getY(),0)));
99
100   // calculation of the vector from iter0 to its successor based on the angles and vehicleState of
          iter0
101   relativeVehicleState = ibeo::VehicleStateBasic::RelativeVehicle::RelativeVehicle((iter1)->
          vehicleState, iter0->vehicleState);
102
103   m = ibeo::geom3d::rotationRoll  (-(iter0->wgs84.getRollAngleInRad()));
104   m = boost::numeric::ublas::prod (ibeo::geom3d::rotationPitch(-(iter0->wgs84.getPitchAngleInRad()))
            , m);
105   m = boost::numeric::ublas::prod (ibeo::geom3d::rotationYaw(-(iter0->wgs84.getYawAngleInRad()))  ,
          m);
106   ibeo::geom3d::HVector relativeWorldState_01 = boost::numeric::ublas::prod(m ,(ibeo::geom3d::
          makeHVectorRect(relativeVehicleState.getDeltaPos().getX(),relativeVehicleState.getDeltaPos().
          getY(),0)));
107
108   // new Positions for iter1
109   PositionWGS84 wgs84_01 = iter1->wgs84;
110   wgs84_01.transformFromTangentialPlane(relativeWorldState_01(0),relativeWorldState_01(1), iter0->
          wgs84);
111   PositionWGS84 wgs84_21 = iter1->wgs84;
112   wgs84_21.transformFromTangentialPlane(relativeWorldState_21(0),relativeWorldState_21(1), iter2->
          wgs84);
113
114   // arithmetic mean
115   iter1->wgs84.setLatitudeInRad((wgs84_01.getLatitudeInRad() + wgs84_21.getLatitudeInRad())/2);
116   iter1->wgs84.setLongitudeInRad((wgs84_01.getLongitudeInRad() + wgs84_21.getLongitudeInRad())/2);
117   iter1->wgs84.setAltitudeInMeterMSL((iter0->wgs84.getAltitudeInMeterMSL() + iter1->wgs84.
          getAltitudeInMeterMSL() + relativeWorldState_01(2) + relativeWorldState_21(2))/2);
118 }
119
120 }
121 }
122 }
```

OFF viewer

F

The OFF viewer is implemented in OpenGL and split up in an interface and a backend.

F.1 ibeo3DVisioFileReader.cpp

This minimal interface defines the mouse control, takes care of displaying the vertices and painting a grid and axis for orientation.

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>

#include <math.h>
#include <iostream>
#include <list>
#include <ScanPointArray.cpp>

const char* VERSION="0.1";

using std::list;

// object holding all the scanpoints
ScanPointArray scanpoints = ScanPointArray();

// camera settings for moving
float zoom = 15.0f;
float rotx = 0;
float roty = 0.001f;
float tx = 0;
float ty = 0;
int lastx=0;
int lasty=0;
unsigned char Buttons[3] = {0};

/**
 * creates points from the read OFF file
 **/
void pointdraw()
{

  // activate and specify pointer to vertex & color array
  glEnableClientState(GL_VERTEX_ARRAY);
  glEnableClientState(GL_COLOR_ARRAY);

  GLfloat* vertices = scanpoints.GetCoords();
```

```
38    GLfloat* colors = scanpoints.GetColors();
39
40    glVertexPointer(3, GL_FLOAT, 0, &vertices[0]);
41    glColorPointer(3, GL_FLOAT, 0, &colors[0]);
42
43    // draw our points
44    glDrawArrays(GL_POINTS, 0, scanpoints.getNumPoints());
45
46    // deactivate vertex arrays after drawing
47    glDisableClientState(GL_VERTEX_ARRAY);
48    glDisableClientState(GL_COLOR_ARRAY);
49
50    // Redraw everything to show newly added points
51    glutPostRedisplay();
52
53  }
54
55  /**
56   * actual drawing function
57   **/
58  void display()
59  {
60    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
61
62    glLoadIdentity();
63
64    glTranslatef(0,0,-zoom);
65    glTranslatef(tx,ty,0);
66    glRotatef(rotx,1,0,0);
67    glRotatef(roty,0,1,0);
68
69    // draw grid
70    glColor3f(0.1,0.4,0.9);
71    glBegin(GL_LINES);
72    int i;
73    for(i=-10;i<=10;++i) {
74      glVertex3f(i*10,-0.5,-10*10);
75      glVertex3f(i*10,-0.5,10*10);
76
77      glVertex3f(10*10,-0.5,i*10);
78      glVertex3f(-10*10,-0.5,i*10);
79    }
80
81    // draw three axes for orientation
82    glColor3f(1,0,0);
83    glVertex3f(0,0,0);
84    glVertex3f(5,0,0);
85
86    glColor3f(0,1,0);
87    glVertex3f(0,0,0);
88    glVertex3f(0,0,-5);
89
90    glColor3f(0,0,1);
91    glVertex3f(0,0,0);
92    glVertex3f(0,5,0);
93
94    glEnd();
95    pointdraw();
96    glutSwapBuffers();
97  }
98
99  /**
100  * adjusts viewport to window if window is changed
101  **/
102 void reshape(int w, int h)
```

```
103 {
104   // prevent divide by 0 error when minimized
105   if(w==0)
106     h = 1;
107 
108   glViewport(0,0,w,h);
109   glMatrixMode(GL_PROJECTION);
110   glLoadIdentity();
111   gluPerspective(45,(float)w/h,0.1,500);
112   glMatrixMode(GL_MODELVIEW);
113   glLoadIdentity();
114 }
115 
116 
117 /**
118  * calculate rotation and zoom from mouse movement
119  *
120  * @param int x Mouse movement on x-axis
121  * @param int y Mouse movement on y-axis
122  **/
123 void Motion(int x,int y)
124 {
125   int diffx=x-lastx;
126   int diffy=y-lasty;
127   lastx=x;
128   lasty=y;
129 
130   if( Buttons[0] && Buttons[2] )
131   {
132     zoom += (float) 0.1f * diffy;
133   }
134   else if( Buttons[2] )
135   {
136     rotx += (float) 0.5f * diffy;
137     roty += (float) 0.5f * diffx;
138   }
139   else if( Buttons[0] )
140   {
141     tx += (float) 0.05f * diffx;
142     ty -= (float) 0.05f * diffy;
143   }
144 }
145 
146 /**
147  * mouse event handler
148  *
149  * @param int b Button identifier
150  * @param int s Button status
151  * @param int x Mouse movement on x-axis
152  * @param int y Mouse movement on y-axis
153  **/
154 void mouse(int b,int s,int x,int y)
155 {
156   lastx=x;
157   lasty=y;
158   switch(b)
159   {
160   case GLUT_LEFT_BUTTON:
161     Buttons[0] = ((GLUT_DOWN==s)?1:0);
162     break;
163   case GLUT_MIDDLE_BUTTON:
164     Buttons[1] = ((GLUT_DOWN==s)?1:0);
165     break;
166   case GLUT_RIGHT_BUTTON:
167     Buttons[2] = ((GLUT_DOWN==s)?1:0);
```

```
168     break;
169   default:
170     break;
171   }
172 }
173 
174 /**
175  * main method -
176  * displays program information, handles commandline parameters and
177  * calls the glut functions for the 3D interface
178  *
179  * @param int argc Number of parameters (including the binaries name)
180  * @param char** argv Array holding all space separated cmd-parameters
181  **/
182 int main (int argc, char** argv)
183 {
184   std::cout << "Simple OFF-Viewer version " << VERSION << ", Copyright (c) 2010-2011 Jan Gries and
        Jan Girlich" << endl;
185 
186   if (argc != 2)
187   {
188     std::cout << "usage: " << argv[0] << " off-file" << endl << endl;
189   }
190   else
191   {
192     std::cout << "opening file " << argv[1] << endl;
193 
194     scanpoints.readModel(argv[1]);
195 
196     glutInit(&argc,argv);
197     glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA|GLUT_DEPTH);
198     glutInitWindowSize(640,480);
199     glutInitWindowPosition(100,100);
200     glutCreateWindow("Ibeo-Scan Visualisierung");
201 
202 
203     glutDisplayFunc(display);
204     glutReshapeFunc(reshape);
205     glutMouseFunc(mouse);
206     glutMotionFunc(Motion);
207     glutIdleFunc(NULL);
208 
209     glEnable(GL_DEPTH_TEST);
210 
211     glutMainLoop();
212   }
213 
214   return 0;
215 }
```

F.2 ScanPointArray.cpp

This file serves as backend for several programs written to handle vertices from 3D LRFs. For the above interface only the constructor and the readmodel() method are used.

```
1 #include <GL/gl.h>
2 #include <GL/glu.h>
3 #include <GL/glut.h>
```

```
4  #include <iostream>
5  #include <fstream>
6  #include <string>
7
8  using namespace std;
9
10 class ScanPointArray {
11
12 // global data structure for vertices and colors
13 const static size_t MAX_POINTS = 10000000;
14 size_t scanPointIterator;
15
16 GLfloat ScanPointCoord[MAX_POINTS*3];
17 GLfloat ScanPointColor[MAX_POINTS*3];
18
19 public:
20 ScanPointArray()
21 {
22   scanPointIterator = 0;
23
24   // initialize with negative values, because negative values are not painted
25   for (size_t i=0; i < MAX_POINTS*3; ++i)
26   {
27     ScanPointCoord[i] = 0;
28     ScanPointColor[i] = 0;
29   }
30 }
31
32 void ScanPointAddPoint(GLfloat Xcoord, GLfloat Ycoord, GLfloat Zcoord, GLfloat Greyvalue)
33 {
34   // at MAX_POINTS elements start overwriting from the beginning
35   if (scanPointIterator >= MAX_POINTS*3)
36   {
37     scanPointIterator = 0;
38   }
39
40   // add coordinates of a vertex
41   ScanPointCoord[scanPointIterator]=Xcoord;
42   ScanPointCoord[scanPointIterator+1]=Ycoord;
43   ScanPointCoord[scanPointIterator+2]=Zcoord;
44
45   // set color
46   ScanPointColor[scanPointIterator]=Greyvalue;
47   ScanPointColor[scanPointIterator+1]=Greyvalue;
48   ScanPointColor[scanPointIterator+2]=Greyvalue;
49
50   scanPointIterator += 3;
51 }
52
53 /**
54  * reduce brightness of all vertices by delta
55  *
56  * @param GLfloat delta
57  **/
58 void ReduceAllBrightnessBy(GLfloat delta)
59 {
60
61   for (size_t i=0; i < MAX_POINTS; ++i)
62   {
63     // avoid colors to turn negative
64     if (ScanPointColor[i] >= delta)
65     {
66       ScanPointColor[i] -= delta;
67     }
68   }
```

```
69  }
70
71  GLfloat *GetCoords()
72  {
73    return ScanPointCoord;
74  }
75
76  GLfloat *GetColors()
77  {
78    return ScanPointColor;
79  }
80
81  size_t getNumPoints()
82  {
83    return MAX_POINTS;
84  }
85
86  /**
87   * write out OFF file to a file called 'model.off'
88   **/
89  void saveModel()
90  {
91    std::ofstream out("model.off");
92
93    // create OFF header
94    out << "OFF\n" << MAX_POINTS << " 0 0\n";
95
96    for (unsigned int i=0;i<MAX_POINTS*3;)
97    {
98      out << ScanPointCoord[i] << " " << ScanPointCoord[i+1] << " " << ScanPointCoord[i+2] << " " << "
          \n";
99      i += 3;
100   }
101
102   out.close();
103 }
104
105 /**
106  * read OFF file
107  *
108  * @param const char* the name of the OFF file
109  **/
110 void readModel(const char* file)
111 {
112   std::ifstream in(file);
113   string line;
114
115   // remove OFF header
116   getline(in, line);
117   getline(in, line);
118
119   float X, Y, Z, R, G, B;
120   unsigned int i = 0;
121   while (! in.eof())
122   {
123     getline(in, line);
124     sscanf(line.c_str(), "%f %f %f %f %f %f", &X, &Y, &Z, &R, &G, &B);
125
126     ScanPointCoord[i]=X;
127     ScanPointCoord[i+1]=Z;
128     ScanPointCoord[i+2]=-Y;
129
130     ScanPointColor[i]=R;
131     ScanPointColor[i+1]=G;
132     ScanPointColor[i+2]=B;
```

```
133
134     i += 3;
135     if (i >= MAX_POINTS*3)
136       std::cout << "Too many points to load! Maximum is " << MAX_POINTS << endl;
137   }
138 }
139 };
```