FREIE UNIVERSITÄT BERLIN

Lane Localization for Autonomous Model Cars

Lukas Maischak

B-14-03
July 2014

FACHBEREICH MATHEMATIK UND INFORMATIK
SERIE B • INFORMATIK

This document was produced in the context of the "Berlin United" Carolo-Cup project of *Freie Universität Berlin*. During my time there, the following members were also part of the team:

Jan Boldt designed and assembled most of the hardware, and was the project leader until he left in early 2014.

Severin Junker then took over as project leader and worked primarily on hardware and the microcontroller.

Jannis Ihrig focused on vision, especially obstacle detection.

Ercan Kücükkaraca also worked on vision. He provided the camera calibration and the image in figure 4.3 used for mapping.

Daniel Krakowczyk focused on motion control, which also included working on the microcontroller.

Shayan Nagananda mainly implemented an alternative driving behavior based on lane detection.

I wish to express my special thanks to these people in particular, and also to the members of the *FUManoids* team, for their support. I would also like to thank Prof. Rojas for his guidance during this project.

Contents

1 Introduction **5**

2 The Platform **7**
2.1 Hardware . . . 7
2.2 Software . . . 8
2.2.1 Framework . . . 10
2.2.2 Architecture . . . 10
2.3 Weaknesses . . . 11

3 Problem Description **12**
3.1 Localization . . . 13
3.1.1 Taxonomy . . . 13
3.1.2 Probabilistic View . . . 15
3.1.3 Mapping . . . 17
3.1.4 SLAM . . . 18
3.2 Application to the Model Environment . . . 18
3.2.1 Environment . . . 19
3.2.2 Localization Requirements . . . 20
3.3 The Laboratory Track . . . 21

4 Approach: Tracking with Force Field **23**
4.1 Idea . . . 23
4.2 Implementation . . . 26
4.2.1 Motion Model . . . 26
4.2.2 Sensor Model . . . 27
4.2.3 Map . . . 30
4.2.4 Localization . . . 33
4.3 Navigation . . . 35
4.4 Evaluation . . . 35

5 Approach: Tracking with Particles **39**
5.1 Idea . . . 39
5.2 Implementation . . . 40
5.2.1 Motion Model . . . 40
5.2.2 Map . . . 41
5.2.3 Localization . . . 42
5.3 Evaluation . . . 45

6 Conclusion **48**

List of Algorithms **50**

Bibliography **51**

1 Introduction

Mobile robotics is a rapidly growing field and has countless applications including exploration, logistics, rescue operations, as well as domestic and military use. One particularly interesting example of its use is the construction of autonomous, "self-driving" cars. Imagine that car accidents caused by human error are a thing of the past, or that your car can find its own parking spot after you have left the vehicle.

In many cases, mobile robots need to plan and make decisions autonomously while interacting with their environment. A necessary prerequisite for them to execute most non-trivial tasks is to have a concept of their environment and their location in it. Determining this location is a fundamental problem in mobile robotics known as localization. Autonomous cars need to know where they are on the road, both on a small scale to stay in lane and on a large scale to navigate.

This project explores mobile robotics problems on a small scale. The objective is to construct a small model car that is able to drive autonomously in an indoor model environment like those displayed in figures 1.1 and 1.2.

This document focuses on a mechanism for localizing autonomous model cars within such an environment. The next chapter gives a brief overview of the target platform – the model car in use. Chapter 3 defines the problem this document is trying to solve in more detail. Chapters 4 and 5 describe and evaluate two alternative – though related – solutions to that problem.

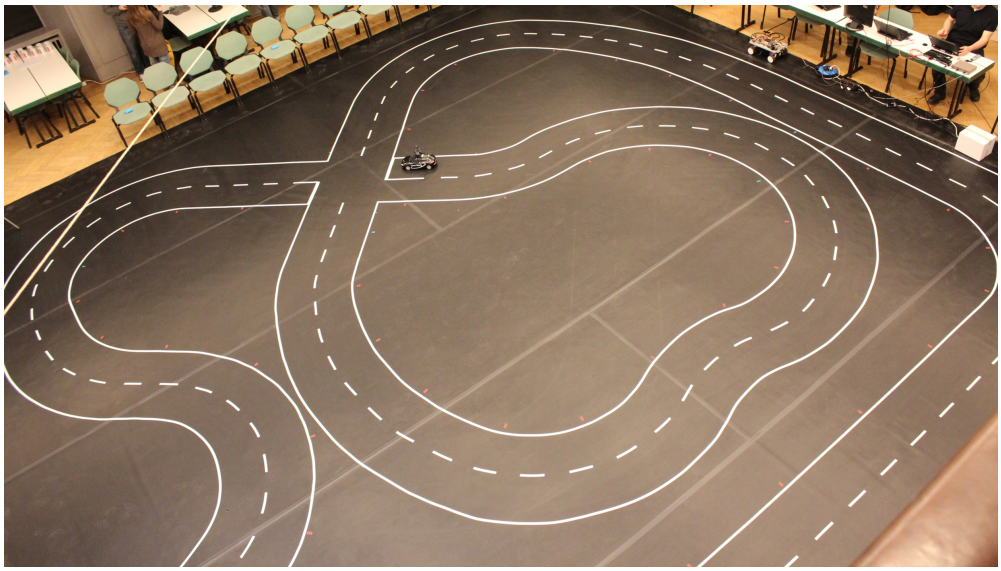


Figure 1.1: The Carolo-Cup 2014 training track.

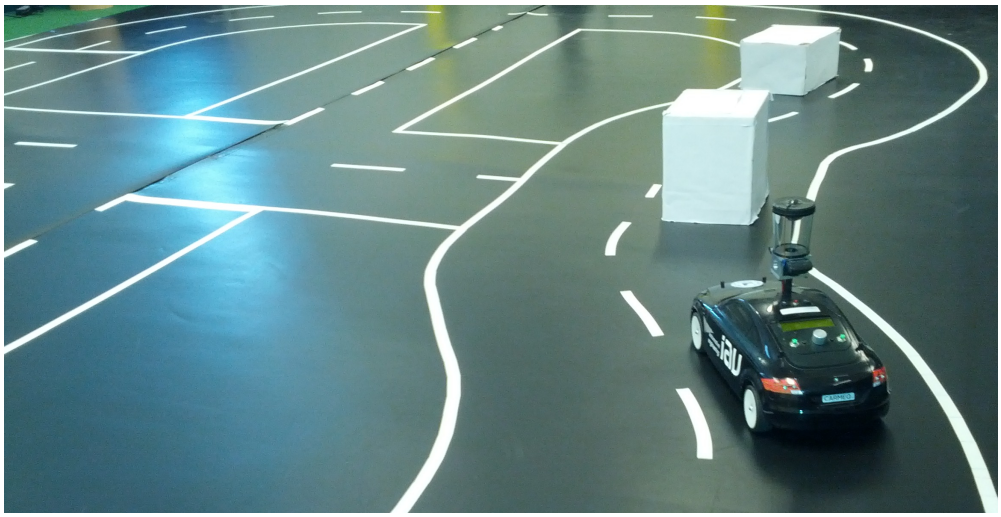


Figure 1.2: A model car with obstacles.

2 The Platform

The project started in 2012, and when I joined the team in late 2013, a working model car already existed, with an identical second one being assembled. It had been used in a previous race track competition[1] and partially working software was available. The following sections contain a general description of the hardware in use and the software running it.

2.1 Hardware

Because the hardware has already been described elsewhere [Hardware], only a summary is given here.

To sense its surroundings, our car primarily uses images from a digital camera, which features an omnidirectional mirror so that the car can see the floor in all directions simultaneously.

The design has two programmable components: the primary computer, which at the time of writing is an *ODROID-X2*, and the self-developed *OttoBoard V2* that serves as its interface to most of the hardware.

The *ODROID-X2* is a single board computer (*SBC*) containing a passively cooled 1.7 GHz *ARM Cortex-A9* quad core processor, 2 GB of RAM as well as several external interfaces including 100 MBit/s Ethernet, USB 2.0 and UART. In our car, it runs the main software, which will be described in the next section. A USB WiFi adaptor and the Ethernet port are used to install the software and for remote debugging. The omnidirectional camera is simply connected as a webcam through another USB port.

The UART port is connected to the *OttoBoard*, which contains an *STM32 F4* microcontroller with an *ARM Cortex-M4F* core clocked at 168 MHz. It controls and provides access to the following hardware:

- A **motor controller** that controls the main electric engine and reports the rotor position so that the current motion speed can be inferred. With this engine, the

[1] Carolo-Cup, see `https://wiki.ifr.ing.tu-bs.de/carolocup/`

car can reach speeds of up to $4\frac{m}{s}$.

- A **steering servo** that controls the front wheels' turning position. The minimal curve radius is 42 cm.
- A 9-DoF **inertial measurement unit** (*IMU*) which is included directly on the *OttoBoard* and contains accelerometer, magnetometer and gyroscope.
- A **remote control receiver** using the public 40 MHz band.
- **Lighting** including turn signals, brake signals and an indicator light showing the state of the remote control.
- An external **control panel** containing an LCD showing a menu, and two buttons and a small wheel to interact with it. It can be used to initiate and switch autonomous driving modes and recalibrate some of the hardware when networking is disabled for a competition.
- A **power board** that provides power from the removable lithium-polymer batteries to the other components. Because this type of battery can be damaged by deep discharges, the board also shows the battery level with an RGB LED and cuts off energy when it gets too low. It features external on/off switches both for the main power, and – as an additional safety measure – for the main engine. These are also located on the control panel mentioned above.

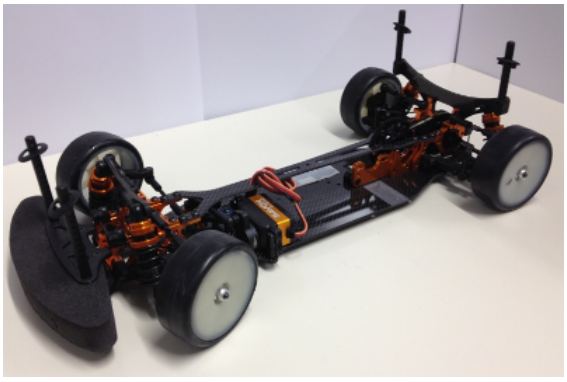
For more details, see [Hardware] and figure 2.1.

2.2 Software

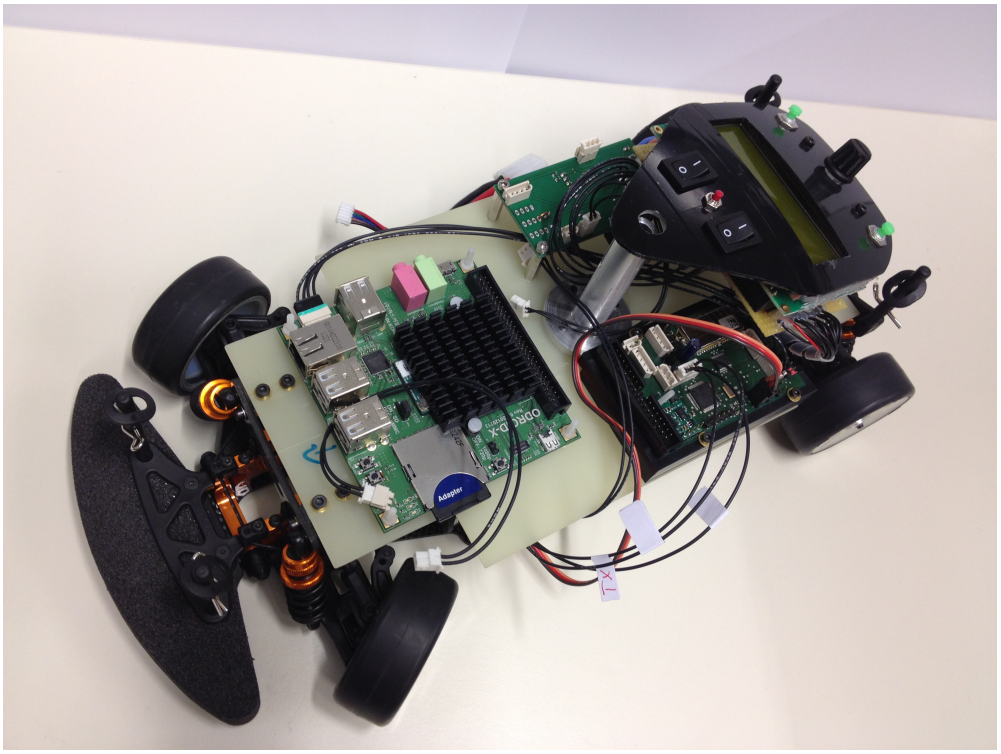
The software for the *OttoBoard* microcontroller is written in C, and it is mainly used as an interface to most of the hardware for the main SBC, whose software can access it through remote procedure calls as well as asynchronously. Another main functionality is the possibility to override the main software's motion commands by remote control. It also contains software for automatic recalibration of the gyroscope.

The main SBC simply runs an ARM Linux, which is useful because it allows inclusion of the large number of off-the-shelf libraries available for that platform. The cognition software, primarily written in C++, is developed on our individual desktop machines, cross-compiled using GCC for the ARM platform and then pushed over via SSH. Fortunately, C++-11 features are also available.

(a)



(b)



(c)

Figure 2.1: (a) shows the assembled car where the omnidirectional lens can be seen at the top with the camera being directly underneath. The chassis with steering servo is shown in (b), upon which the carrier board is placed (c), with the main SBC at the front, the control panel at the back and the *OttoBoard* underneath to the left.

2.2.1 Framework

Since its inception, our car has been using a branch of the *BerlinUnited* robotics framework, which is a collaborative effort between *Humboldt-Universität Berlin* and *Freie Universität Berlin*, and was originally devised for RoboCup soccer robots. It provides an architecture for defining multiple modular execution pipelines as well as utilities like synchronization, file handling and access to the camera. It also features tools for remote debugging with the *FURemote* companion application.

BerlinUnited applications are structured into modules that declare their dependencies, and the framework resolves these dependencies and calculates a suitable execution order.

2.2.2 Architecture

At the time of writing, our model car uses only a single execution pipeline whose execution frequency is bound to the frame rate of the single camera. A high-level overview of the module structure is shown in figure 2.2. It features a classical design

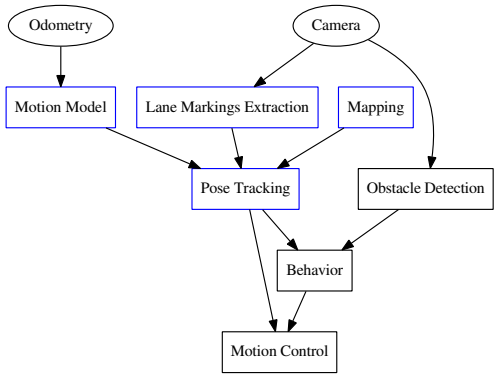


Figure 2.2: A very general overview of the architecture. The module groups marked in blue are the subject of this thesis.

following the *sense-think-act* paradigm:

1. The odometry from the microcontroller and the sensor data from the omnidirectional camera are acquired and preprocessed.

2. This information is refined to provide higher-level information like localization and obstacle detection and used to plan behavior.

3. That behavior is executed, resulting in commands being sent back to the hardware.

The cycle repeats for each frame.

2.3 Weaknesses

Although omnidirectional vision is useful for localization, the mirror causes the image resolution to drop rapidly as distance increases. Because the vision input consists of that omnicam image exclusively, the ability to detect lanes and obstacles in any but the shortest distance is severely restricted. Fortunately, work is already underway to mitigate this problem through installation of an additional front-facing camera that is also supposed to detect obstacles using stereo vision.

Another problem is the fact that there is only a single execution pipeline, because this means that the motion control is limited to the camera's framerate. But the new camera necessitates at least one further pipeline anyway. To this end, a pipeline split into cognition, localization, motion and vision for each camera is being worked on.

Although the aging *BerlinUnited* framework, which was originally written in C, is continuously being developed, using it is not as productive as it could be: it uses and enforces the use of numerous magic preprocessor macros and seems brittle and inflexible, especially about managing the module pipelines at runtime. However, for a beginner, it is also easy to get started on, because, in combination with the *FURemote*, it provides a lot of remote debugging and library functionality – although some of that is covered by the C++ standard library by now.

3 Problem Description

A central question when building autonomous cars is how to model the environment. In our flat-featured model scenario, the environment primarily consists of a black ground with white lane markings and – optionally – white boxes that may or may not move.

An obvious strategy would be to locally model the road by detecting the lane markings on the camera image, approximating them with polynoms, and then constructing a matching trajectory to follow. Other detected features on the track, like obstacles, start and stop lines and crossroads would then potentially trigger special behavior.

This is sufficient a simple lane following challenge, because the expected behavior is simple, and therefore most decisions can be made locally: we have to **follow the right lane**, unless

- **it is blocked** by an obstacle, in which case we **temporarily switch to the left lane** if it is empty, or else wait, or
- we **get to a junction**, where we always wait for some seconds, then potentially wait for the obstacle to the right to pass, then **always go straight**.

We do not have to make any real high-level decisions, like to choose between alternatives based on congestion, or to actually navigate. Therefore, we do not critically need a model of the whole world, and we do not need to keep track of its state. The relevant part of the state can, to a certain degree, be determined locally and without retaining information about the past.

However, it would still be useful to have some sense of where the car actually *is* on the circuit. For example, this information can be used for speed control: as an alternative to detecting the lane curvature and setting the speed accordingly (which might fail when the view is blocked by an obstacle, or, like with our car, the vision range is insufficient), one could fine-tune the optimal speed manually for each track segment before the race. Another application would be to increase accuracy when detecting static track elements by discarding, for example, erroneously detected start lines half-way around the track.

Also, it is desirable that the car can be used in other, more complex situations besides simple lane following, and knowing where the robot is is a necessary prerequisite for many tasks, like navigation.

In the remainder of this chapter, I will first elaborate on the concept of *localization*, but can of course only give a brief overview. Afterwards I will describe the conditions of how to apply it to the flat-featured model setting, and finally introduce our laboratory track.

3.1 Localization

The position and orientation of a mobile robot in relation to some reference coordinate frame are collectively called its **pose**. This is a critical piece of information in many mobile robot applications, because it provides a link between a robot's external state[1] and the state of its environment: it allows the positions of objects and places to be expressed in the robot's own coordinate frame and therefore enables the robot to navigate the environment.

The way it is modeled is application-specific, but for rigid mobile robots it usually has three degrees of freedom (two for position plus one for orientation) in the planar case and six degrees of freedom (three for position plus three for orientation) in the 3-dimensional case.

The problem of determining a mobile robot's pose is called **mobile robot localization**. Because the pose usually cannot be measured directly, at least not without noise, it has to be inferred from sensor data and known information about the environment, also called a **map**.

3.1.1 Taxonomy

There are several related variants of the localization problem that can be classified regarding the following aspects:

- **Online Versus Offline:** Usually, localization needs to happen *online* during normal operation of the robot, because in many cases it is required to plan behavior, like in navigation. Sometimes, however, it is sufficient to just record sensor data at runtime and analyze it later, possibly on a different machine.

[1] A robot's external state also includes, for example, the configuration of the robot's actuators (its *kinematic state*) and the robot's and its actuators' velocity (its *dynamic state*), not all of which actually have to be modeled.

Online localization is a real-time problem, so it is more difficult than offline localization, where the time constraints are not as tight and more computational power is usually available.

- **Local Versus Global:** Depending on how much knowledge about the robot's position is available initially and at runtime, we can distinguish the following cases, each with an increasing degree of difficulty:
  - In the simplest case, the initial pose of the robot is known, hence the only pose uncertainty comes from the inaccuracy in the robot's motion. This variant of the problem is known as *position tracking*.
  - If the initial pose is unknown, *global localization* is necessary. This is obviously more difficult than tracking, because the pose space usually is large. Once the pose is found, it may be possible to switch to a tracking approach to save resources.
  - The *kidnapped robot problem* additionally introduces the possibility that the robot may at any time be picked up and placed somewhere else in the environment ("teleported") at random.[2] The robot now also needs to detect the situation where its pose estimate is suddenly very wrong and then relocalize itself globally.
- **Static Versus Dynamic Environments:** If the only relevant variable of the environment is the robot's own pose, the environment is called *static*. On the other hand, if it contains other stateful objects, it is called *dynamic*, and localization in it obviously becomes more difficult. These objects either have to be modeled explicitly and tracked as part of the localization algorithm at the cost of additional computational and algorithmic complexity, or they need to be filtered as a preliminary step. In some cases, if they only impact single sensor readings, they can be treated as sensor noise.
- **Passive Versus Active:** If the localization algorithm can control the robot's motion directly, it can try to improve localization quality by choosing paths that maximize information gain. This is called *active* localization, and obviously this can conflict with tasks other than localization that the robot might have. If the localization algorithm only observes without interfering, it is called *passive*.

[2] [Thrun] notes that although this might rarely happen in practice, the fact that many (probalistic) "state-of-the-art localization algorithms cannot be guaranteed never to fail" makes the "ability to recover from failures [...] essential for truly autonomous robots."

- **Single-Robot Versus Multi-Robot:** When multiple robots operate in the same environment and they can both communicate with and sense each other, they can combine their knowledge instead of localizing themselves individually, which can greatly improve localization quality.

For more details, see [Thrun, 193ff], where most of this taxonomy is taken from. This book also contains a good overview of the algorithms mentioned in the following sections, and of localization and mapping in general.

3.1.2 Probabilistic View

The localization problem is usually approached using probabilistic methods to control for the inherent uncertainty in the available sources of information. Figure 3.1 shows the involved quantities and their relation to each other. For each time frame t, the goal is to estimate the robot's true pose $\mathbf{x}_t$. The robot's *belief* $\text{bel}(\mathbf{x}_t)$ over where it could be is represented by an arbitrary probability density over the pose space.

If we assume that we already have some sense of where we were at the previous time step $\text{bel}(\mathbf{x}_{t-1})$ (the *prior*), there are two sources of information that we can use to infer the current pose (the *posterior*):

- the **motion control** instructions $\mathbf{u}_t$ that were given to the robot after the previous time step and whose influence on $\mathbf{x}_t$ needs to be modeled in a sensible *motion model* and
- the **sensor readings** $\mathbf{z}_t$ from the beginning of the current time step that are influenced both by the environment and the current pose and need to be modeled in a suitable *sensor model*. This is where the map M comes into play, because when combining it with a good *environment model* we can use Bayes' rule to infer knowledge about $\mathbf{x}_t$ from $\mathbf{z}_t$.

Both of these information sources are noisy (which needs to be accounted for in their respective models), so using either of them on their own would increasingly dilute $\text{bel}(\mathbf{x}_t)$ as time passes. But because they are largely independent, they can be combined to reduce uncertainty.

This is why localization algorithms usually execute the following two steps for each time frame:

1. **Prediction**, where an intermediate representation $\overline{\text{bel}}(\mathbf{x}_t)$ of the robot's knowledge about its pose is constructed using $\mathbf{u}_t$ only. As already indicated, this will usually increase its uncertainty due to the noise inherent in the motion model.

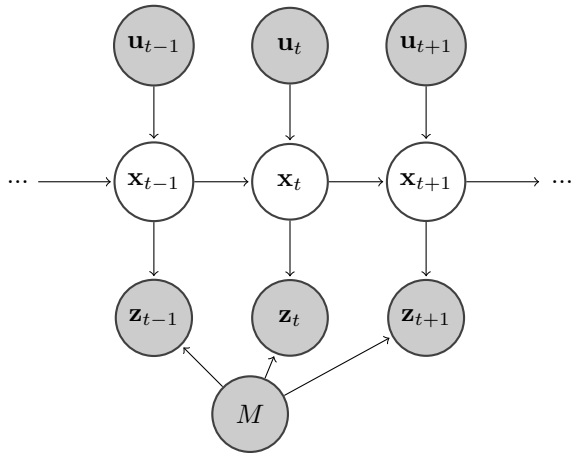


Figure 3.1: A dynamic Bayesian network of the variables involved in localization: for each time step t, the robot's true pose $\mathbf{x}_t$ is determined by the pose at the last time step $\mathbf{x}_{t-1}$ and the motion commands $\mathbf{u}_t$ controlling its movement during the time frame in between. The resulting sensor readings $\mathbf{z}_t$ are determined both by the robot's pose and the environment, represented in the map M. Shaded nodes signify known information.
Note that in this model, $\mathbf{x}_t$ depends on all information from past time frames only indirectly through the immediately previous $\mathbf{x}_{t-1}$, which means that we do not need to retain historical data because all relevant information about the past is included in that $\mathbf{x}_{t-1}$. This is called the *Markov assumption*.

2. **Correction**, where $\overline{\text{bel}}(\mathbf{x}_t)$ is refined using $\mathbf{z}_t$, which can decrease uncertainty depending on the quality of the sensor measurements.

This approach is called a *Bayes filter* and is summarized in algorithm 1.

Algorithm 1 PoseUpdate

Require: previous pose $\text{bel}(\mathbf{x}_{t-1})$, motion data $\mathbf{u}_t$, sensor data $\mathbf{z}_t$, map M
$\overline{\text{bel}}(\mathbf{x}_t) \leftarrow \text{PosePrediction}(\text{bel}(\mathbf{x}_{t-1}), \mathbf{u}_t)$
$\text{bel}(\mathbf{x}_t) \leftarrow \text{PoseCorrection}(\overline{\text{bel}}(\mathbf{x}_t), \mathbf{z}_t, M)$
return new pose estimate $\text{bel}(\mathbf{x}_t)$

When localization starts, $\text{bel}(\mathbf{x}_0)$ needs to be set to a suitable distribution. For example, a tightly bounded gaussian may be used when tracking. For global localization, a uniform distribution could be used.

Repeatedly applying the Bayes filter at each time step then provides localization.

3.1.3 Mapping

A central part of any localization mechanism is how the map of the environment is represented. The choice of mapping strategy depends on the kind of environment and what information about it is available, and also on the availability of computational resources.

The localization problem has been extensively studied since the late 1980s, and solutions to it have long been constrained by limited computational power. Nevertheless, there have been some surprisingly good results from early on. These were achieved by reducing the static part of the environment to few recognizable point features – named *landmarks* – in turn also reducing the complexity of the map. These are examples of *feature-based* maps, which describe an environment by a collection of its features.

Landmarks are not required to be uniquely identifiable among each other due to the availability of probabilistic algorithms for associating sensor readings to landmarks. But of course, the better the landmarks can be distinguished – if possible by unique sensor signatures – the easier and more accurate localization gets.

Depending on the kinds of sensors used, many types of environmental features can be used as landmarks, for example trees, building walls and cliffs when using a laser scanner or sonar system, cell towers and WiFi hotspots for radio, or even colored markers for cameras. In fact, if feasible, artificial landmarks are still a good solution for localization because there is a wide range of efficient, well-understood and easy-to-implement landmark-based localization algorithms available: *EKF localization*[3], *UKF localization*[4] and variants of the *particle filter*, just to name a few.

Another category of maps are the so-called *location-based* maps, which have become more popular recently as computational resources increased. They describe the environment as a function of locations in it. An example of these are *grid maps* that describe some property of the environment in a rasterized fashion, like images of the ground or ceiling, volumetric maps of a building's walls (*occupancy grid maps*), or *likelihood fields* modelling the likelihood of some sensor reading a particular signal for each location. They are usually more memory-intensive, especially in more than two dimensions, and localization algorithms using them tend to need more resources, because they typically require convolution operations between sensor data and the whole map.

There are other representations, for example (graph-like) *topological maps*, but feature-based and location-based maps are the most widely used.

[3] This uses an *extended Kalman filter* over the robot's state vector.
[4] The same, but with an *unscented Kalman filter* instead.

3.1.4 SLAM

It is not always feasible or desirable to acquire a static map of the environment before localization starts. Luckily, it is in many cases possible to use the robot's own sensor data to build the map during localization ("**s**imultaneous **l**ocalization **a**nd **m**apping"). This presents a chicken-and-egg type problem, because the robot's pose is needed to integrate the current sensor data into the map, but the map is in turn needed to determine that pose. The *SLAM problem* is obviously more difficult than either localization or mapping on their own. It can be solved by modeling the map itself as part of the robot's state vector and therefore refining it along with the pose estimate.

If the SLAM approach only updates the current pose in each time step and never revises it again, it is called *online SLAM*, and an early example is the landmark-based *EKF-SLAM* algorithm. A related algorithm is *SEIF-SLAM*, using a *sparse extended information filter*.

SLAM variants that estimate the complete pose trace of the robot at each time step are called *full SLAM*. *GraphSLAM* is an example of this type.

The particle-filter-based *FastSLAM* algorithm can solve both online and full SLAM.

3.2 Application to the Model Environment

In our model lane-following setting, there are two obvious ways the car's pose can be modeled. One possibility is to model $\mathbf{x} = (o, \alpha, t)^T$ with

- the car's offset o to some defined path $p(t)$ on the track – say, along the middle lane markings – orthogonal to its direction,
- the difference in orientation α to that direction, and
- the car's position t along p relative to a predefined point like the start marker (the "global part" of the pose).

For a simple circuit track, this representation is sufficient to completely describe the pose. It is an obvious extension to the "local" view $(o, \alpha)^T$ that is often used for motion control when localization is not implemented, so usually mechanisms are already in place to determine o and α. Therefore, localization could focus on t.[5] However, there are problems with this approach:

[5] This could possibly be done using the track's curvature as a map and the gyroscope's orientation as sensor data. See `https://www.mikrocontroller.net/articles/GhostCarProjekt` for an implementation on a Carrera® track.

- The "true" global cartesian coordinates of the pose depend on (the potentially non-linear) p, so the set of all poses is, in general, not evenly distributed throughout the pose space in this model. Distances between poses are not trivial to compute.
- For the same reason, there can also be multiple representations for any given pose.
- Additional work needs to be done when the track is no longer a simple circuit. So although this model might work well for simple race track challenges, it is not well suited for more complex settings.

For these reasons, the traditional way of modeling the pose

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} \tag{3.1}$$

was chosen, with a 2-dimensional cartesian position x, y and a global orientation θ.

3.2.1 Environment

The used model environment is very different from what real autonomous cars face. The track is much simpler as it can be sufficiently described in two dimensions, and although it is dynamic, its dynamism is not complex. In fact, the form of all dynamic objects is restricted strongly enough that they can be filtered before the localization step and therefore play no role in it, except maybe by blocking part of the vision area.

But for autonomous systems, a simple environment is not always exclusively a good thing: if it is poor in features, there is less information to exploit. On our flat-featured type tracks, the only obviously useful static part of the environment is the way the ground looks.[6] The area around the track is difficult to predict and noisy, so using laser range finders or sonar is probably close to useless. The same is true for the radio spectrum because of the many cell phones, WiFi access points and RCs, so although they do exist, they are difficult to sense reliably and therefore not well suited as features. Also, there are no good natural landmarks (like trees or building walls), and artificial ones may usually not be placed at race track competitions.

[6] ...and, potentially, the ceiling. This could be useful, but would require an additional camera or a more complex lens system.

The situation is further complicated by the fact that many places on the track essentially look the same, which is especially problematic with the limited range of our camera. The only parts that differ from the standard road style of two lanes with markings are start and stop lines and crossroads; the removed sections of lane markings are not known until shortly before the race and can therefore not be mapped offline.

Because driving usually takes place indoors and the environment is small-scale, and different than with real autonomous cars, an accurate satellite-based differential GPS is not available.

3.2.2 Localization Requirements

In most race track scenarios, the starting pose is known, so localization here means a **tracking** problem. This is convenient, because controlling behavior or even driving directly using this information requires the localization to happen **online**, and our computational resources on the model car are limited. However, when driving exclusively using a tracking implementation, losing the correct pose is catastrophic, because unlike when using a local method where we can just use the RC to steer the car back on track, here we have no means of reinitializing the localization.

Global localization could help, but at the cost of a more expensive localization step. Further, because it requires movement due to the small vision range and also because it could potentially fail, it requires a secondary motion strategy using only local information, which further eats on the frame's time budget.

A compromise would be to drive using a local strategy, and use the localization only for high-level behavior. This approach has lower requirements on the accuracy of the localization so it does not have to be executed in every frame, instead tracking the pose using only motion data in between frames and accumulating motion noise along the way. Implementation of this approach is more involved, thus the tracking approach was tried first and is also the focus of this thesis. However, a long term goal should probably be the implementation of this hybrid strategy.

It was already mentioned in the previous subsection we are considering a **dynamic environment**, so localization needs to handle that fact. It should also be obvious that the localization strategy must be **passive**, because the primary motion task is to follow the track all the time. Yet it can assume that the car is generally doing exactly that and staying close to the critically needed lane markings.

At the moment, we are using **single-robot** localization. This could however change in the future, and Severin Junker is already working on car-to-car communication, so

multi-robot localization is not entirely out of the question.

3.3 The Laboratory Track

The track shown in figure 3.2 was used as a testbed. It already existed when I joined the team and has not changed since. This test track is much smaller than most race competition tournament tracks, which is problematic for obvious reasons:

- To contain all relevant tournament situations, it must have a higher feature to space ratio, skewing the results in favor of the localization quality when testing. On tournament tracks, there are long passages of mostly feature-less roads, a case which is undertested on the laboratory track, where symmetry-breaking features like junctions are visible from almost anywhere. This situation is unrealistic on the much larger race competition tracks because of our car's limited vision range.
- The map can be smaller, giving unrealistically high performance when testing.

However, there is only limited space at our laboratory, and no larger rooms were available to us, so this had to suffice for the time being.

The laboratory track also has another problem: the black surface is very reflective, so the ceiling lights can produce "ghost lines" of approximately the same size as the lane markings that have to be filtered out.

All implemented algorithms in this thesis have been tested on this track, and, in my opinion, it works well enough for that purpose. There is of course room for improvement.

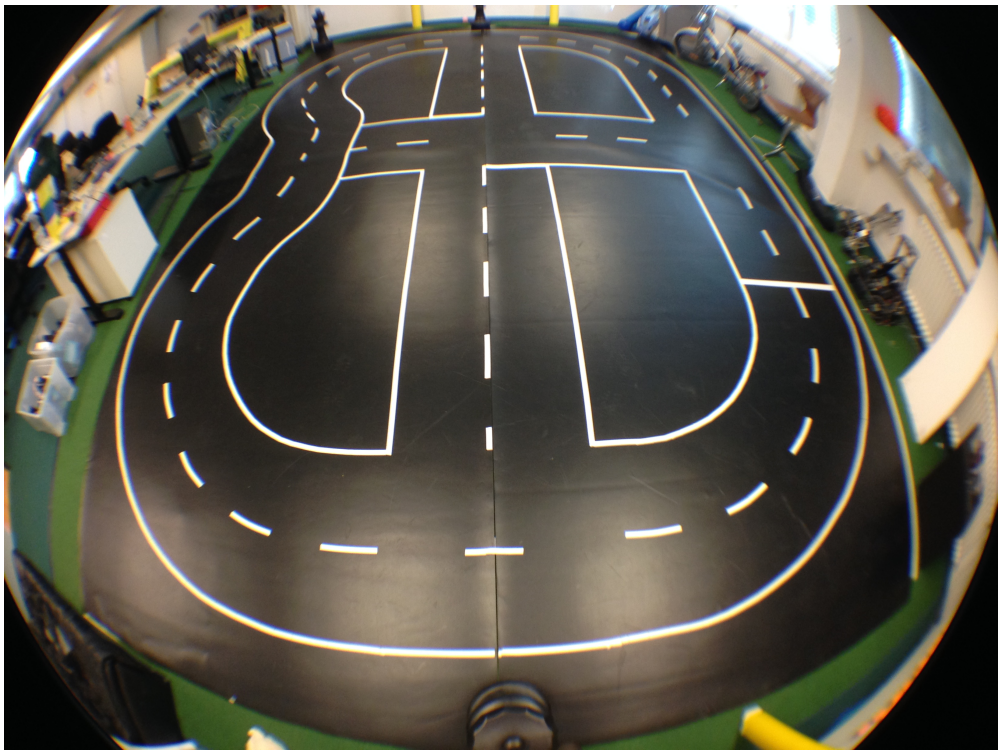


Figure 3.2: This is the test track in our lab, which is approximately 4 by 6 meters large. The reflections appear larger and more diffuse than from the car's perspective because the camera is further from the surface. We also shared this laboratory with the *FUManoids* team, so their test playing field is directly underneath the track. To the right, a line from their field is visible, which can erroneously be detected as a lane marker if not covered.

4 Approach: Tracking with Force Field

We now turn to solving the localization problem for our flat-featured model environment using an omnidirectional camera. As previously mentioned, we primarily focus on tracking the car from a known initial pose and ignore global localization, and we also assume that a map of the track can be acquired offline.

As described in section 3.1.2, we need to iteratively update the car's pose estimate at each time step by first predicting a new pose using motion data and then correcting that intermediary estimate using both our sensor data – the camera image – and our map, and we need to do this in real-time.

Ignoring the prediction step for now, we need to devise a mechanism for pose correction, and the part of the environment most useful for this purpose is the ensemble of white lines on the ground. Obviously, we need a map of where they are supposed to be, and a way of detecting them on the camera image. For a good estimate of our current pose, the position of the lines we see around us should align with the lines on the map at that pose. In other words, after we have applied the camera transformation and the (linear) transformation from our local coordinate frame into the map's (global) coordinate frame to the position of the lines on the image, they should match that part of the map.

In reality, however, they will be slightly misaligned, which we now need to correct. One way to do this is the force field algorithm proposed in [ForceField], wherein it was described for RoboCup soccer robots.

The rest of this chapter explains its basic idea and describes its application to the model environment.

4.1 Idea

The key conceptual decision of the force field algorithm is to model the ensemble of visible lines as a rigid body of point masses[1] $B = \{\mathbf{b}_i\}$. The pose $\mathbf{x} = (x, y, \theta)^T$ we

[1] For simplicity, we assume unit masses. The algorithm would also work for other mass distributions, which could for example be used to devalue lines that are farther away. Similarly, non-point masses

are trying to estimate defines the position x, y and orientation θ of B in the global coordinate frame, and moving B also moves $\mathbf{x}$ along with it. Let $B_{\mathbf{x}}$ be B transformed into global coordinates at the pose $\mathbf{x}$.

We already observed that all of the $\mathbf{b}_i$ should align with the map M. If we now "pull" $B_{\mathbf{x}}$ towards the white lines in M, we are implicitly correcting $\mathbf{x}$. We manage this by modeling a force field $\vec{F}$ over M that is exerted on $B_{\mathbf{x}}$ and for every $\mathbf{x}$ results in a pull

$$\vec{p} = \sum_{\mathbf{b}_i \in B_{\mathbf{x}}} \vec{F}(\mathbf{b}_i) \tag{4.1}$$

and torque

$$\vec{\tau} = \sum_{\mathbf{b}_i \in B_{\mathbf{x}}} (\mathbf{b}_i - \mathbf{c}_{B_{\mathbf{x}}}) \times \vec{F}(\mathbf{b}_i) \tag{4.2}$$

around its center of mass $\mathbf{c}_{B_{\mathbf{x}}}$[2].

How exactly $\vec{F}$ is defined depends on the representation of the map, but it has to be defined in such a way that the forces point "towards" the lines in M, like a gravitational pull. For an example, using a feature-based map, see [ForceField]. Section 4.2.3 implements it for a grid map.

Applying $\vec{p}$ and $\vec{\tau}$ to $B_{\mathbf{x}}$ for some amount of "pseudo"-time[3] hopefully moves it to a position that is better aligned with M, and therefore moves $\mathbf{x}$ closer towards the robot's true pose. When $B_{\mathbf{x}}$ matches M, then $\vec{p}$ and $\vec{\tau}$ are both zero[4] and an equilibrium is reached. We now just iterate until that happens. The full procedure is given in algorithm 2.

A key insight here is that, because we assume that M is static, the force field $\vec{F}$ can be precomputed. This approximation in a "grid of forces" makes the acquisition of $\vec{F}(\mathbf{b}_i)$ a fast memory operation if the raster size is chosen right. Therefore, the algorithm depends only linearly on the number of line points.

---

could be used, especially with feature-based maps of shapes.

[2] The points and vectors in this context need to be extended to the third dimension, so that $\times$ is defined and equation 4.2 makes sense. The torque $\vec{\tau}$ is now a normal vector to the environmental plane defining the axis (and direction) of rotation, and its length $|\vec{\tau}|$, which can be read directly from its third component, defines the torque's absolute strength.

[3] We do not model a real dynamic system here, but instead just move $B_{\mathbf{x}}$ linearly along the resulting $\vec{p}$ for a distance proportional to that force and rotate it for an amount proportional to $\vec{\tau}$.

[4] Note that the reverse is not necessalily true. This is problematic because it means that we can "fall into" false optima.

Algorithm 2 ForceFieldPoseCorrection

Require: predicted pose $\overline{\text{bel}}(\mathbf{x}_t)$ as $\overline{\mathbf{x}}$, sensor data $\mathbf{z}_t$, map M as $\vec{F}$

$B \leftarrow$ line points extracted from $\mathbf{z}_t$ using sensor model {in robot-local coordinates}
$m \leftarrow$ mass of B
$I \leftarrow$ rotational inertia of B
$\mathbf{x} \leftarrow \overline{\mathbf{x}}$
while not too many iterations **do**
  $B_{\mathbf{x}} \leftarrow B$ transformed to global coordinates assuming $\mathbf{x}$ as the robot's pose
  $\vec{p} \leftarrow \sum_{\mathbf{b}_i \in B_{\mathbf{x}}} \vec{F}(\mathbf{b}_i)$
  $\vec{\tau} \leftarrow \sum_{\mathbf{b}_i \in B_{\mathbf{x}}} (\mathbf{b}_i - \mathbf{c}_{B_x}) \times \vec{F}(\mathbf{b}_i)$
  $T \leftarrow$ translation by $\alpha_T \frac{\vec{p}}{m}$
  $R \leftarrow$ rotation by $\alpha_R \frac{\vec{\tau}}{I}$
  if T and R are insignificant **then**
    break
  end if
  apply T and R (around $c_{B_{\mathbf{x}}}$) to $\mathbf{x}$
end while
return bel($\mathbf{x}_t$) as $\mathbf{x}$

The α_T and α_R as well as the abort criteria are implementation constants and $c_{B_{\mathbf{x}}}$ is the center of mass of $B_{\mathbf{x}}$.

4.2 Implementation

An implementation of this force field tracking approach already existed [Kumar], and the car was able to drive a predefined path on the track using it. However, there were some problems with this implementation:

- It was limited to a slow driving speed of approximately 0.6 $\frac{m}{s}$ for risk of losing the pose.
- It was also very dependent on lighting conditions and receptive to the reflections on the ground.
- Further, it could easily be disturbed at the fringe of the track by the environment outside, especially by the edge between the track's black ground and the underlying floor. This required some sections to be covered up by black cloth.
- It could not be used when driving with obstacles, because they confused the algorithm.
- It depended on the *ICP-SLAM* implementation from *MRPT*[5], which was a pain to keep integrated on the target platform, and the resulting map had to be refined in a tedious manual process, making the use of *SLAM* largely superfluous.
- While the author notes that the manner of driving was more stable than when using a local lane-following strategy, the car could not drive for more than a few rounds: the implementation would frequently lose the pose or crash.

For these reasons, the first task was to reimplement the force field tracker.

4.2.1 Motion Model

In the prediction step, we need to find an accurate first estimate of the new pose. The motor controller in our car sends measurements from which the *OttoBoard* can deduce our current speed v_t. For the rotation, we can use the gyroscope from the IMU so that we can estimate the rotation speed ω_t. This provides us with an input

$$\mathbf{u}_t = \begin{pmatrix} v_t \\ \omega_t \end{pmatrix} \tag{4.3}$$

[5] The *Mobile Robot Programming Toolkit*, see `http://www.mrpt.org/`

to our motion model for the current time frame t.[6]

Using the *velocity motion model* from [Thrun, 121ff] transforms our previous pose $\mathbf{x}_{t-1}$ to our estimate for $\mathbf{x}_t$ (see algorithm 3). The idea is that if we assume $\mathbf{u}_t$ to remain constant during the whole time frame, movement will happen along a circle arc. Note that we are ignoring the motion noise (from the measurements v_t and ω_t) here because the force field algorithm does not handle them anyway.

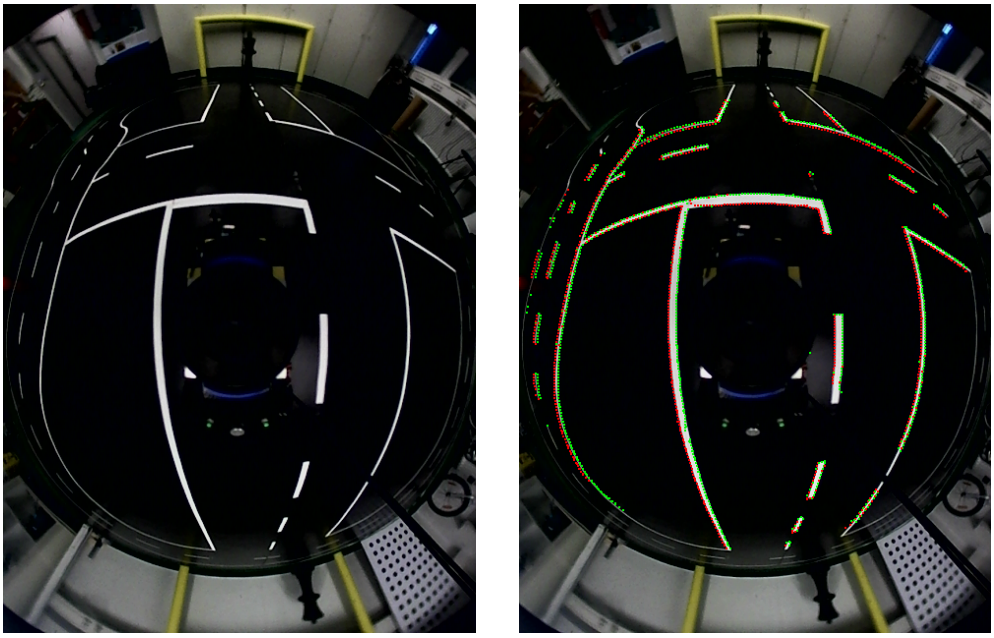
Algorithm 3 NoiselessVelocityMotionModel

Require: previous pose bel($\mathbf{x}_{t-1}$) as $\mathbf{x}_{t-1}$, motion data $\mathbf{u}_t$
  $\Delta t \leftarrow$ length of current time frame
  $(x, y, \theta)^T \leftarrow \mathbf{x}_{t-1}$
  $(v, \omega)^T \leftarrow \mathbf{u}_t$
  if $\omega = 0$ **then** {straight motion}
    $s \leftarrow v\Delta t$
    $\overline{\mathbf{x}_t} \leftarrow \begin{pmatrix} x + s\cos\theta - s\sin\theta \\ y + s\sin\theta + s\cos\theta \\ \theta \end{pmatrix}$
  else {motion on circle arc}
    $r \leftarrow \frac{v}{\omega}$
    $\Delta\theta \leftarrow \omega\Delta t$
    $\overline{\mathbf{x}_t} \leftarrow \begin{pmatrix} x - r\sin\theta + r\sin(\theta + \Delta\theta) \\ y + r\cos\theta - r\cos(\theta + \Delta\theta) \\ \theta + \Delta\theta \end{pmatrix}$
  end if
  return $\overline{\text{bel}}(\mathbf{x}_t)$ as $\overline{\mathbf{x}_t}$

4.2.2 Sensor Model

The sensor model is supposed to extract the white lines' position from the camera image and transform them into robot-local coordinates. A naive approach would be to walk over all white pixels in the image, transform their positions and put them all into B. However, this is inefficient and also adds much unwanted noise. It is better to select a representative set of points so that B resembles the true set of lane markings as much as possible. A skeleton of the white areas would be ideal for that purpose,

[6] Unfortunately, the values handed over by the microcontroller are contaminated by spurious disturbances like occasional zeroes or values that are off by an order of magnitude. Synchronization between motor controller and microcontroller seems to be broken in some way. As a temporary measure, the current implementation of the motion model filters these outliers using an automatic outlier detector and replaces them by a moving average. This means that we are losing information here, which degrades the quality of the motion prediction.



(a) (b)

Figure 4.1: (a) shows an image from the omnidirectional camera, the found edgels are marked in (b).

but is not easy obtain and would still need to be filtered for obstacles, reflections and other noise.

As a first step, an easy way to obtain a representative set would be to take the edge pixels (*edgels*) of the white areas. These can easily be found by walking the camera image on scanlines using – for example – a *Prewitt kernel* [Prewitt] to detect transitions with a high color gradient (see figure 4.1).[7]

These points now need to be tranformed into the robot-local coordinate frame. This is not a trivial task due to the strong distortions of the image introduced by the omnidirectional mirror. On figure 4.1.a we note that all points on any particular circle around the image center also have, among each other, the same distance to the robot's center when they are considered in the undistorted coordinate frame. This is because the mirror has an axis of symmetry right through its center that is also a normal on the camera sensor's image plane. We therefore only need to scale the distance of

[7] An implementation of this already existed in our software, and it also used *non-max suppression* to prevent double entries.

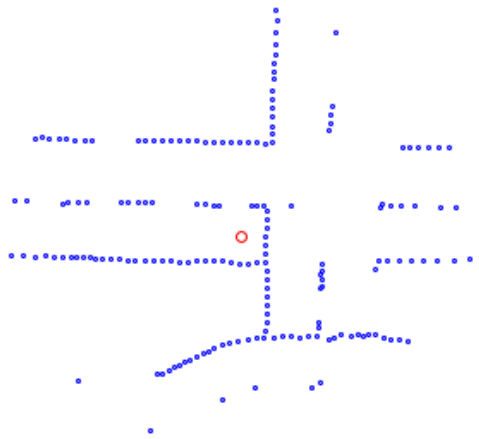


Figure 4.2: The resulting set of points used as B for tracking, where the red circle marks the center of mass. The robot's front points right.

the relevant points from the center with some distance transformation function. This function is specific to the mirror used, and can be approximated by a polynomial using a best-fit method on known distorted/undistorted distance pairs obtained from some calibration image.[8]

The previous implementation of the force field tracking algorithm used the set of transformed edgels directly as B, without filtering obstacles and other noise. A better representation can be found using the following method.

In most parts of the image, the white lines are cut in a steep angle either by the horizontal or the vertical scanlines. This results in edgel pairs on the scanlines whose distance is close to the thickness of the white line they are bordering on. If we now select from these edgel pairs only those whose distance is in an acceptable range and replace them by the point directly at their respective center, we get a set of points that are all directly in the middle of white lane markers.

This point set is now refined by overlaying a grid and replacing all the points in each cell by their centroid, which normalizes their density and reduces the amount of points that have to be considered in B without losing much of the general shape. It turns out that this point set is a good representative of the lines' position and covers most of the visible lines (see figure 4.2).

[8] The implementation of this omnidirectional camera transform and the corresponding calibration process already existed in the software.

4.2.3 Map

As already indicated, the force field in this algorithm plays the role of the map and contains all of the information we need about the static part of the environment. In section 4.1, we deferred its exact definition until this point. Recall that its forces should "point towards" the white lines on the ground.

Assume that we have a grid map

$$G(\mathbf{p}) = \begin{cases} 1 & \text{if there is a white line at the cell which contains } \mathbf{p} \\ 0 & \text{else} \end{cases} \tag{4.4}$$

of the white lines on the ground, where $\mathbf{p} \in \mathbb{R} \times \mathbb{R}$ is a point in the global coordinate frame.

One way to build a suitable force field would be to think of the white lines as if they were exerting a "gravitational pull" on B. To model this, we can assume that the white grid cells are replaced by fixed point masses at their respective centers, and the partial force exerted from any white cell with center $\mathbf{q}$ on any point $\mathbf{p}$ could be defined as

$$\vec{P}(\mathbf{p}, \mathbf{q}) = e^{-\lambda|\vec{d}|}(\vec{d}) \tag{4.5}$$

where $\vec{d} = \mathbf{q} - \mathbf{p}$ and λ is a positive implementation constant controlling the weight difference between near and far points. $\vec{P}$ is chosen such that close white cells exert a greater pull. The resulting combined force at $\mathbf{p}$ is then

$$\vec{F}_{\mathrm{G}}(\mathbf{p}) = \sum_{\mathbf{q} \in \mathbb{R} \times \mathbb{R}} \vec{P}(\mathbf{p}, \mathbf{q}) G(\mathbf{q}). \tag{4.6}$$

This is equivalent to how the force field is defined in [ForceField] for a feature-based map.

An alternative is to simply model the forces as pointing towards the closest white cell:

$$\vec{F}_{\mathrm{NN}}(\mathbf{p}) = \left(\underset{\mathbf{q} \in \mathbb{R} \times \mathbb{R}}{\operatorname{argmin}} \begin{cases} |\mathbf{q} - \mathbf{p}| & \text{if } G(\mathbf{q}) = 1 \\ \infty & \text{else} \end{cases} \right) - \mathbf{p} \tag{4.7}$$

This is the representation [Kumar] used as force field.

G can easily be obtained from an image of the white markings on the track like the one in figure 4.3 by just regarding each pixel as a grid cell with center $\mathbf{q}$ in the global coordinate frame and setting $G(\mathbf{q}) = \lfloor \mathrm{val}(\mathbf{q}) \rceil$ where $\mathrm{val}(\mathbf{q})$ is the greyscale pixel value

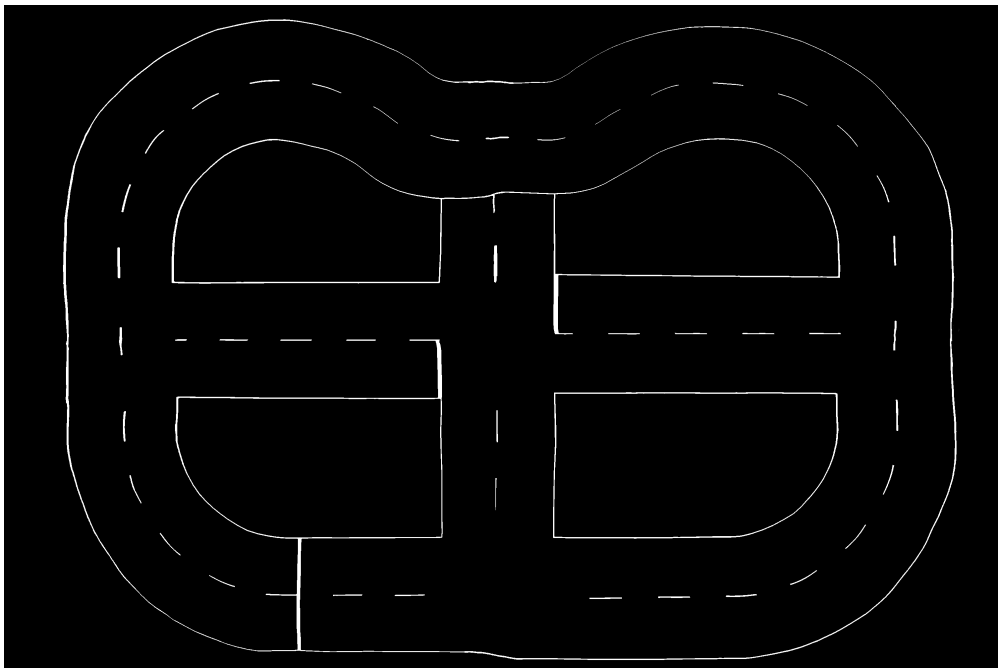
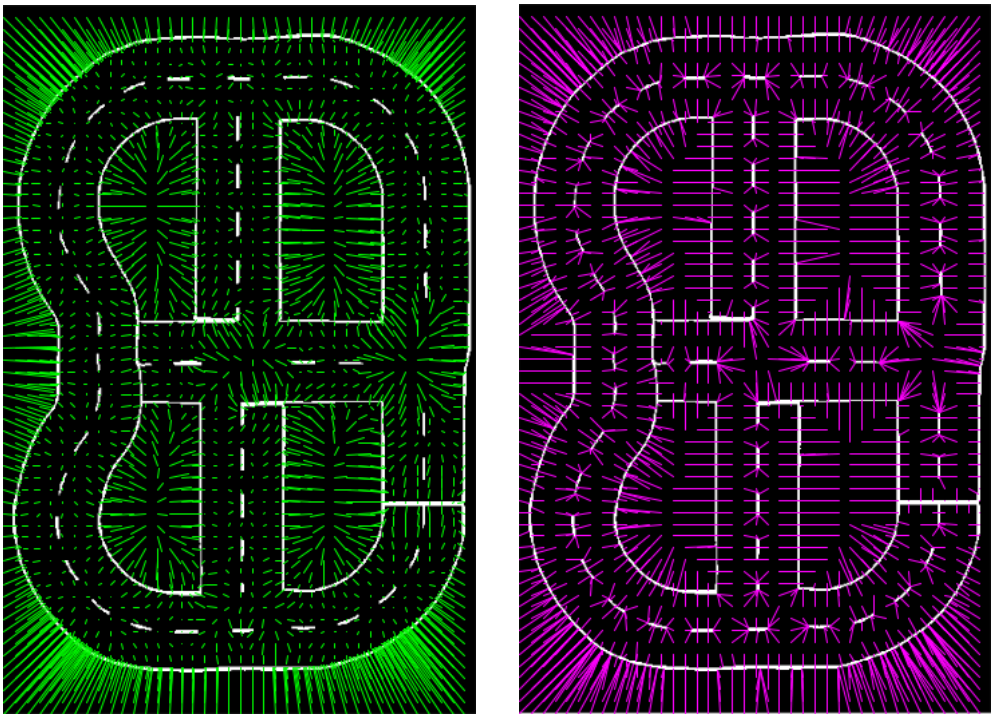


Figure 4.3: An image used to generate the ground map G for our lab track. The size of the mapped region is $4m \times 6m$. This particular image was stitched together by Ercan Kücükkaraca from several pictures of the track and scaled such that each pixel corresponds to a square region of the track approximately $1cm^2$ in size. At the middle some artifacts of the stitching process are visible: some lane markings that should actually be straight are bent here. The localization algorithm can handle this to a certain extent, but it should be fixed nonetheless.



(a) (b)

Figure 4.4: The resulting force fields (a) $\vec{F}_{\mathrm{G}}$ and (b) $\vec{F}_{\mathrm{NN}}$ for the ground map G. The resolution of the force grid has been reduced for this figure.

at that position.[9]

Once G is available, building $\vec{F}_{\mathrm{G}}$ and $\vec{F}_{\mathrm{NN}}$ is only a matter of walking their cells and calculating their resultant forces from equations 4.6 and 4.7. The results for the lab track are shown in figure 4.4.

Obviously, the choice of resolution is a trade-off between accuracy and resource consumption, and that extends to localization time as well: if force grid access regularly blows the cache, performance can degrade significantly, especially for the particle tracker's quality estimator described in the later section 5.2.3.

It should be noted that the representations for $\vec{F}$ do not need to have the same resolution as G, although a force grid cell size of $1cm^2$ works well on our platform.

[9] [Kumar] used *MRPT*'s *ICP-SLAM* implementation to generate G, but because that algorithm does not support *loop closure*, the resulting point cloud quickly "bleeds out" after only a few rounds and needs to be refined manually, and even then is not as accurate as the G generated from the photograph. This defeats the purpose of using SLAM for offline map creation here.

With that resolution, building the maps for our $4m \times 6m$ lab track using the naive approach can take several hours on slower machines.

This could be optimized by walking the inner loop over the cells in G in order of increasing distance to the target cell in $\vec{F}$, and aborting early when it is clear that the target force will not change much anymore. Because this optimization is not necessary for our small-size track, though, it was not implemented yet.

Depending on the image, it might be necessary to handle the border areas by padding $\vec{F}$ by a fixed amount. Also, it might be a good idea to cancel all forces that are too large. That way, parts of B that are too far away from any lane marking, and therefore are most likely sensor noise, will not influence the rest of the measurement. Both extensions have been implemented.

4.2.4 Localization

We now have everything in place to assemble the localization's update step: algorithm 4. Using algorithm 1 as a blueprint, we only need to insert NoiselessVelocityMotionModel for PosePrediction and ForceFieldPoseCorrection for PoseCorrection using either $\vec{F}_{\mathrm{G}}$ or $\vec{F}_{\mathrm{NN}}$[10] as map, and we are done. The only change is to ignore corrections that are unusually far off, because they should not happen in normal operation anyway, and can indicate that the estimate is just falling into a false optimum.

Algorithm 4 ForceFieldPoseUpdate

Require: previous pose $\mathrm{bel}(\mathbf{x}_{t-1})$ as $\mathbf{x}_{t-1}$, motion data $\mathbf{u}_t$, sensor data $\mathbf{z}_t$, map M as $\vec{F}$
  $\overline{\mathrm{bel}}(\mathbf{x}_t) \leftarrow$ NoiselessVelocityMotionModel$(\mathrm{bel}(\mathbf{x}_{t-1}), \mathbf{u}_t)$
  $\mathrm{bel}(\mathbf{x}_t) \leftarrow$ ForceFieldPoseCorrection$(\overline{\mathrm{bel}}(\mathbf{x}_t), \mathbf{z}_t, M)$
  if $\mathrm{bel}(\mathbf{x}_t)$ is unusually far away from $\overline{\mathrm{bel}}(\mathbf{x}_t)$ **then**
    $\mathrm{bel}(\mathbf{x}_t) \leftarrow \overline{\mathrm{bel}}(\mathbf{x}_t)$ {reject corrections that are too aggressive}
  end if
  return new pose estimate $\mathrm{bel}(\mathbf{x}_t)$

Note that we are not modeling any kind of motion and sensor noise with this approach and are only carrying on the best estimate in each step. This means that we can only track a single pose hypothesis, making the belief distribution $\mathrm{bel}(\mathbf{x}_t)$ a probability point mass and therefore unimodal.

[10] $\vec{F}_{\mathrm{NN}}$ tends to work a little better than $\vec{F}_{\mathrm{G}}$ on our track.

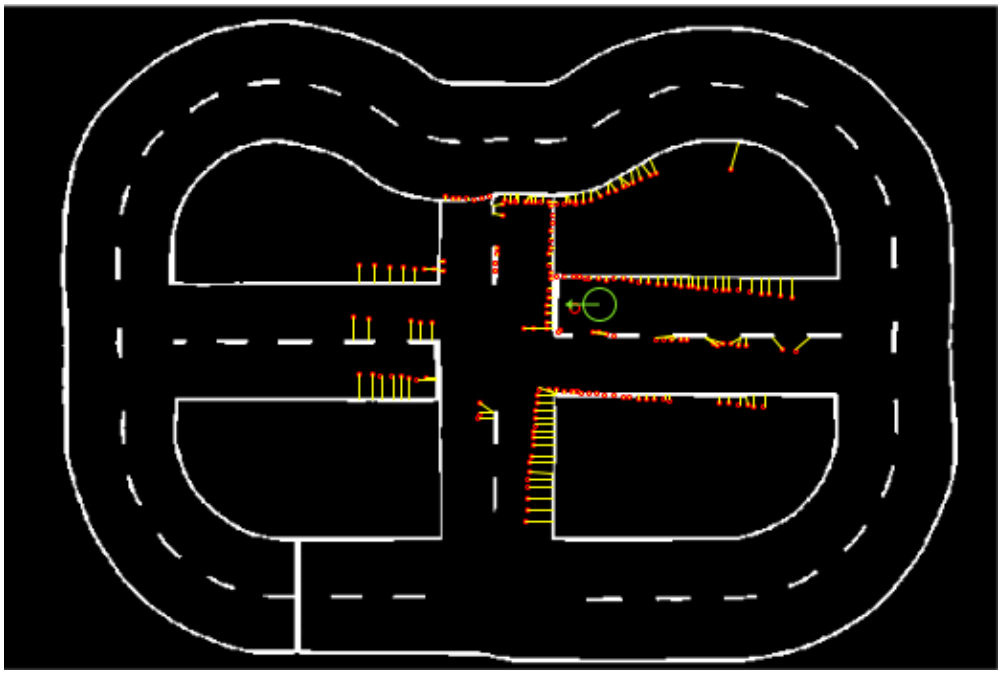


Figure 4.5: This shows an exaggerated example situation right before the correction step. The perception B from figure 4.2 (red) is transformed to the predicted pose estimate $\overline{\mathbf{x}}$ (green), where the force field $\vec{F}_{\mathrm{NN}}$ from figure 4.4.b exerts component forces (yellow) on the visible points from $B_{\overline{\mathbf{x}}}$.

4.3 Navigation

We now turn to the question of how to actually drive the car directly using localization. This is not a part of the localization problem, so it will only be summarized briefly here.

We already observed that simple lane following does not require us to do any complex navigation. All we need to do is to follow a fixed circuit path repeatedly, potentially stopping at predefined points for a short amount of time or changing to a second path temporarily if the main one is blocked.

When the car can localize itself in the global coordinate frame, and the localization is accurate enough, it can also directly steer towards any given point on the map.[11] Following the path is now easy: it just has to be defined as a list of points in global coordinates, and the car has to approach each of these points in turn. Once a point is close enough, it can be dropped for the next one.

Special locations, like junctions, can be mapped as points of interest that trigger special bevior, in this case waiting and watching for dynamic obstacles to the right.

Changing lanes when an obstacle is present is a little bit more difficult, because a good first candidate target point on the second path needs to be selected. But this can easily be achieved by projecting a line orthogonal to the car's moving direction at a fixed distance in front of the car and then dropping points behind it from the target path.[12]

The paths can be recorded simply by driving along the track using the remote control and letting the car record its pose trace, though it may need to be refined manually to improve its quality. See figure 4.6 for a simple test circuit on our lab track.

For more complex scenarios, the navigation concept most likely needs to be extended to a topological map of path segments, especially as the environments get larger and real navigation is required. But for simple lane following, this model is sufficient.

4.4 Evaluation

This new implementation works well enough that the resulting pose estimate can be used for driving as it is accurate to the centimeter range. Usually, its quality should be assessed using a ground truth system, but there was none easily available. As an

[11] This functionality was implemented by Daniel Krakowczyk as part of his forthcoming thesis on motion control.

[12] The main challenge is the actual detection of the obstacles, which is currently being worked on by Jannis Ihrig as part of his thesis.

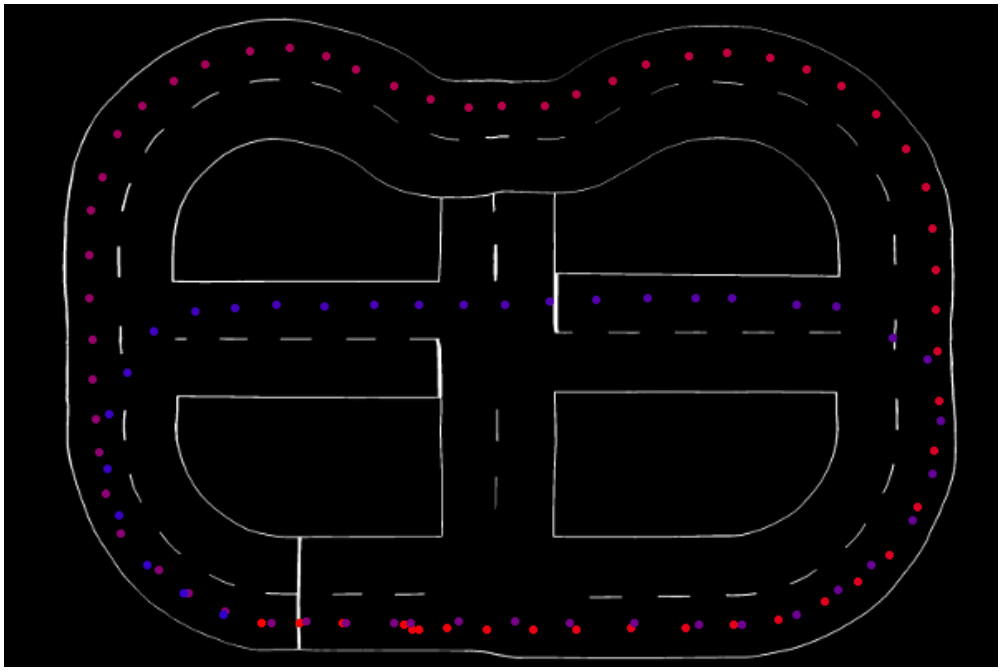


Figure 4.6: A path recorded when driving by remote. Only the first point in each second was picked, but no other changes were made. The path goes from red to blue so that the first and second round can be distinguished.

Speed $[\frac{m}{s}]$	Successful Laps (out of 10)
1.3	10
1.4	8
1.5	0

Table 4.1: The force field tracker stability test. Trials for the speed values lower than $1.3\frac{m}{s}$ are omitted as the tracker never lost the pose there.

alternative, a simple experiment was conducted to assess the stability of the force field algorithm on our lab track.

The car's task was to follow the test circuit from figure 4.6 using the driving approach from section 4.3 with a constant speed. The circuit has a length of approximately 37.25 meters, and it includes several junctions, tight curves and two passes over the dangerous bump visible in figure 3.2 right after the start line. This bump could not easily be removed because it was part of the underlying carpet.

A lap was counted as a success if the car managed to drive the full circuit and pass the start line without losing its pose, which would have led to erratic behavior. The driving speed was increased every 10 laps to determine up to which point localization is safe enough for autonomous driving.

The results are listed in table 4.1, from which we can deduce that the new force field tracking implementation is safe up to $1.3\frac{m}{s}$. It is also significant that the majority of failures at higher speeds happened at the bump or at the first junction, likely due to reflections of the ceiling tube lamps, which have about the width of the lane markings. Also, the car is not in a horizontal position when crossing the bump, which could have disturbed the omnidirectional distance transform.

(Quality of the driving behavior itself is the same as when using the particle tracker, so it is not discussed here. Refer to section 5.3 to see the evaluation.)

In summary, this implementation works and is much more stable than the previous one, which was not safe at any driving speed due to software crashes and was practically unusable for speeds higher than $0.6\frac{m}{s}$.

The improvements on the sensor model were especially useful, because reflections, obstacles and other objects on the track have a much weaker impact – although the mostly non-diffuse reflections from the ceiling lamps can still cause problems, especially if the general level of lighting is low.

Driving with obstacles is now possible using this approach. As with the previous implementation, it is very fast, because the force grid can be precomputed for the

static map. There are, however, still a few problems with this algorithm:

- Like with the previous implementation, speed has to be limited to reduce the chances of losing the pose.
- The fact that we can only track a single pose hypothesis is problematic, because falling into a wrong optimum just once almost certainly leads to a critical failure: it is difficult to recover from such a situation without falling back on an alternative hypothesis.
- Although this implementation is much less susceptible to disturbances in the environment, some problems persist. For example, the border from the underlying RoboCup field (see figure 3.2) is correctly identified as a white line by the sensor model and pushes the pose estimate into a wrong optimum.

The single biggest problem with this algorithm is the unimodal belief distribution, which will be fixed in the next chapter.

5 Approach: Tracking with Particles

To address the weakness of tracking only the best estimate of our current pose, we need to enhance its belief distribution bel($\mathbf{x}$) to support multiple hypotheses. There are basically two options to achieve this. We could model bel($\mathbf{x}$) explicitly, for example using a *mixture of gaussians* or a similar multimodal distribution.

The other alternative would be to represent bel($\mathbf{x}$) implicitly using a set of random samples from that distribution, with the reasoning that the accuracy of its approximation will continually improve with the sample count. A high density of samples (*particles*) in the region of the pose space surrounding any particular pose hypothesis then corresponds to a higher belief in that hypothesis.

Obviously there is a trade-off here, between accuracy on the one side, and speed and memory requirements on the other. But if we have the resources to fill the pose space with samples in the relevant regions then we can represent any belief distribution with an arbitrary precision.

5.1 Idea

The basic idea of the particle tracking algorithm is to use this sample-based distribution model as the representation for bel($\mathbf{x}$), which is now a set of pose hypotheses $\mathcal{X}_\mathbf{t} = \{\mathbf{x}_t^{[i]}\}$ for any given time step t.

In the prediction step, we incorporate the motion noise into $\mathcal{X}_{\mathbf{t-1}}$ by sampling from our motion model to obtain $\overline{\mathcal{X}_\mathbf{t}}$. If we now have a way $q(B_\mathbf{x})$ to assess a particular hypothesis' quality – the degree to which sensor data and map agree at that pose – we can remove some of the prediction's uncertainty and reflect that information in a new sample set $\mathcal{X}_\mathbf{t}$ that has, in comparison, more particles in high-quality regions of the pose space. We do this simply by drawing particles from $\overline{\mathcal{X}_\mathbf{t}}$ using *importance sampling* with q as the importance weight.

In other words, we carry along multiple hypotheses for the true $\mathbf{x}$, apply the motion model to each of them individually, then rate their respective qualities regarding the sensor data and finally use that to build a set of better hypotheses. Over the iterations

of this process, the unlikely hypotheses that have "drifted" away from the robot's true pose increasingly die out.

This idea is called a *particle filter* and is usually applied over the complete pose space, resulting in global localization. It is well known to give good results while at the same time remaining conceptually simple. See [Thrun, 96ff] for an overview.

In the case of our tracking problem, the region of pose space that contains high-quality particles is comparatively small, so that we can potentially get away with much smaller sample sizes than are usually required with this type of algorithm.

But we have to guarantee that the particles do not diverge too much from the previous pose, or else this intuition will not hold anymore. Because large parts of the model track look very self-similar, wrong guesses can easily be rated as high-quality in multiple consecutive iterations, even with a good quality estimator. Consider, for example, a straight road on the track: the only local distinguishing features in forward direction are the middle lane markers. The sensor image at two distinct poses at the beginning of consecutive lane markers look exactly the same.

We can only hope that these "false optima" in the quality function collapse quickly, by more distinguishing features – like a curve or junction for instance – coming into view.

To prevent the belief distribution from diluting too far in this manner, it is truncated to a predefined region around the car. This partially neglects the advantages gained from the particle representation, because it can discard very valid hypotheses. But this price has to be paid to achieve acceptable running times and accuracy on our platform as our resources are limited.

5.2 Implementation

The particle-based algorithm was implemented as an extension to the force field tracker, so much of the code could be reused. The sensor model in this approach is exactly equivalent to the one described in section 4.2.2 for the force field algorithm and is not repeated here. However, we need some extensions to the motion model and the map to implement the particle tracker.

5.2.1 Motion Model

For the prediction step, we need to extend the motion model from section 4.2.1 so that it also includes motion noise; specifically, we need to sample from the resulting

probability distribution. This is achieved by prepending the noiseless velocity motion model with a step that adds random noise to the motion control information $\mathbf{u}$. This is the full version of the velocity motion model from [Thrun, 121ff] that had been stripped of the motion noise for the force field algorithm.

The noise distribution should have a mean of zero and a variance b^2 that depends on the absolute value of $\mathbf{u}$ so that faster motion introduces a stronger uncertainty. The distribution chosen for this implementation is a *symmetric triangular* distribution that can be sampled from using algorithm 5.

Algorithm 5 SAMPLEFROMTRIANGULAR

Require: variance b^2

return $\sqrt{\frac{3}{2}}(\text{SAMPLEFROMUNIFORM}(-b, b) + \text{SAMPLEFROMUNIFORM}(-b, b))$

Here, SAMPLEFROMUNIFORM(l, u) is a library method that generates a pseudorandom number from the interval $[l, u]$.

The variables v_t and ω_t were obtained from seperate measurements, but they are not necessarily independent. For example, the car is unlikely to turn without driving forward. The variance of the noise added to each of the two variables therefore depends on both of them.

But just mutating $\mathbf{u}$ is not enough: because we assume that we move along a circle arc starting in $\mathbf{x}_{t-1}$, we can never reach all possible poses in the target region of the pose space. This is because we are only using two degrees of freedom to change a value that has three, and therefore the image of that operation is a 2-dimensional manifold in the 3-dimensional pose space. Hence we introduce a third source of noise that is applied directly to the final orientation θ_t. For more information on why this is necessary, see [Thrun, 129].

The resulting algorithm 6 now provides a mechanism to sample from our motion model. The α parameters here are robot-specific constants that define the accuracy of the measurements for $\mathbf{u}$ and need to be found by experimentation.

5.2.2 Map

As previously mentioned, we need a way $q(B_{\mathbf{x}})$ to assess the quality of any given pose hypothesis $\mathbf{x}_t^{[i]}$ regarding the current sensor data $\mathbf{z}_t$. In other words, we wish to know how likely a measurement like $\mathbf{z}_t$ is assuming we are at $\mathbf{x}_t^{[i]}$. For this reason, we need a way $l(\mathbf{p})$ to tell how likely the road is to be white at any given position $\mathbf{p}$ – a *likelihood field*.

Algorithm 6 VELOCITYMOTIONMODEL

Require: previous pose $\text{bel}(\mathbf{x}_{t-1})$ as $\mathcal{X}_{\mathbf{t-1}}$, motion data $\mathbf{u}_t$
  $\Delta t \leftarrow$ length of current time frame
  $\overline{\mathcal{X}_{\mathbf{t}}} \leftarrow \emptyset$
  $(v, \omega) \leftarrow \mathbf{u}_t$
  for $\mathbf{x}_{t-1}^{[i]} \in \mathcal{X}_{\mathbf{t-1}}$ **do**
    $\hat{\mathbf{u}} \leftarrow \mathbf{u}_t + \begin{pmatrix} \text{SAMPLEFROMTRIANGULAR}(\alpha_1 v^2 + \alpha_2 \omega^2) \\ \text{SAMPLEFROMTRIANGULAR}(\alpha_3 v^2 + \alpha_4 \omega^2) \end{pmatrix}$
    $\hat{\gamma} \leftarrow \text{SAMPLEFROMTRIANGULAR}(\alpha_5 v^2 + \alpha_6 \omega^2)$
    $\mathbf{x} \leftarrow \text{NOISELESSMOTIONMODEL}(\mathbf{x}_{t-1}^{[i]}, \hat{\mathbf{u}}) + \begin{pmatrix} 0 \\ 0 \\ \hat{\gamma} \Delta t \end{pmatrix}$
    $\overline{\mathcal{X}_{\mathbf{t}}} \leftarrow \overline{\mathcal{X}_{\mathbf{t}}} \cup \{\mathbf{x}\}$
  end for
  return $\overline{\text{bel}}(\mathbf{x}_t)$ as $\overline{\mathcal{X}_{\mathbf{t}}}$

In an ideal world without sensor noise, we could just use $G(\mathbf{p})$ for this purpose, and for this particle filter, this would probably work reasonably well. But if we also want to model sensor noise, we need to incorporate it into l.

Another problem of G as a likelihood field is that it is non-smooth. That means that if $\mathbf{x}_t^{[i]}$ is sufficiently far off, the likelihood distribution in the vicinity is flat, so there is no new information to incorporate into our belief distribution $\overline{\text{bel}}(\mathbf{x}_t)$ and we have little chance of correcting the uncertainty introduced by the motion model.

Therefore, we need a better model for l. A reasonable assumption would be to say that the likelihood of reading a white line at p is higher the closer it is to a white cell in G. Recall that we already have that kind of information in $\vec{F}_{\text{NN}}$, so we can define the likelihood field as

$$l(\mathbf{p}) = e^{-\lambda |\vec{F}_{\text{NN}}(\mathbf{p})|} \tag{5.1}$$

where λ is a positive implementation constant modeling the strength of the sensor noise, the shape of which is described by the wrapping function $e^{-\lambda(\cdot)}$. Other wrapping functions could be used, but usually they should be monotonically decreasing and normalized to the interval $[0, 1]$.

5.2.3 Localization

A pose quality measure q_l can now be obtained by integrating the likelihood of the individual sensor measurements $\mathbf{b}_i \in B$ – which define places where we are seeing

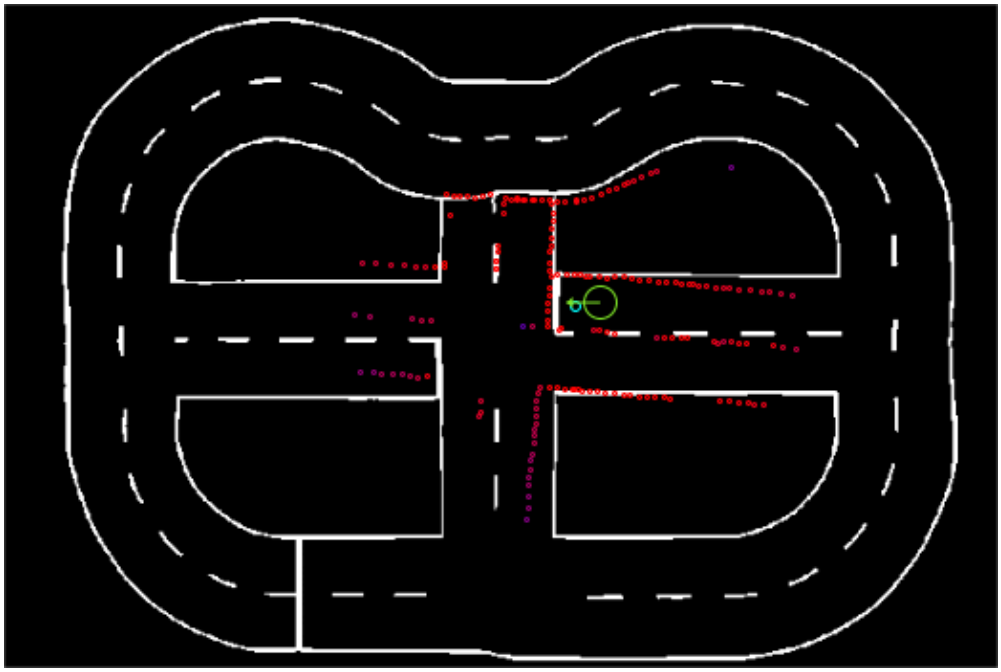


Figure 5.1: This shows the same situation as figure 4.5 right before the correction step. The perception B from figure 4.2 is transformed to the predicted pose estimate $\overline{\mathbf{x}}$ (green), where the visible points from $B_{\overline{\mathbf{x}}}$ are rated regarding l. The points' color indicates their likelihood of being where they are seen, red meaning a high likelihood decreasing towards blue.

white lane markings – into a single likelihood by averaging over them:

$$q_l(B_{\mathbf{x}}) = \frac{\sum_{\mathbf{b}_i \in B_{\mathbf{x}}} l(\mathbf{b}_i)}{|B_{\mathbf{x}}|}. \tag{5.2}$$

Figure 5.1 shows the likelihood of the $\mathbf{b}_i$ at $\mathbf{x}$ right before they are aggregated.

Note that using this quality definition, we are ignoring negative measurements – the places where we *do not* see any white lines. But the usefulness of that information is dubious anyway, because the CaroloCup track might have had some of its lane markings removed since map creation. Using negative sensor information would require a much more complex sensor model that includes the information we have about these missing lane markings.

It is also noteworthy that q_l is still flat in the target range, because $\vec{F}_{\mathrm{NN}}$ is built from G, which does not actually contain white lines but white areas instead. In a future implementation, this could be improved by using a skeleton of G instead of using it directly.

With q defined, we now have everything we need for the correction step, which is given in algorithm 7. Note that the first loop's body – which contains the expensive

Algorithm 7 ParticleFilterPoseCorrection

Require: predicted pose $\overline{\mathrm{bel}}(\mathbf{x}_t)$ as $\overline{\mathcal{X}_\mathbf{t}}$, sensor data $\mathbf{z}_t$, map M as l
  $B \leftarrow$ line points extracted from $\mathbf{z}_t$ using sensor model {in robot-local coordinates}
  for $\mathbf{x}_i \in \overline{\mathcal{X}_\mathbf{t}}$ **do**
    $B_{\mathbf{x}_i} \leftarrow B$ transformed to global coordinates assuming $\mathbf{x}_i$ as the robot's pose
    $\mathbf{w}_i \leftarrow q_l(B_{\mathbf{x}_i})$
  end for
  $\mathcal{X}_\mathbf{t} \leftarrow \emptyset$
  for p times **do**
    $\tilde{\mathbf{x}} \leftarrow$ sample with returning from $\overline{\mathcal{X}_\mathbf{t}}$ weighed by $\mathbf{w}$
    $\mathcal{X}_\mathbf{t} \leftarrow \mathcal{X}_\mathbf{t} \cup \{\tilde{\mathbf{x}}\}$
  end for
  return $\mathrm{bel}(\mathbf{x}_t)$ as $\mathcal{X}_\mathbf{t}$

Here p is the desired number of particles, which can change in any iteration if necessary.

q_l operations – is reentrant, so it can be executed in parallel. Consequently, the implementation uses all four available cores to support an increased number of particles.

The full particle tracker is shown in algorithm 8, where "truncate" is an operation

Algorithm 8 ParticleFilterPoseUpdate

Require: previous pose $\mathrm{bel}(\mathbf{x}_{t-1})$ as $\mathcal{X}_{\mathbf{t-1}}$, motion data $\mathbf{u}_t$, sensor data $\mathbf{z}_t$, map M as l
  $\overline{\mathrm{bel}}(\mathbf{x}_t) \leftarrow$ VelocityMotionModel$(\mathrm{bel}(\mathbf{x}_{t-1}), \mathbf{u}_t)$
  $\mathrm{bel}(\mathbf{x}_t) \leftarrow$ ParticleFilterPoseCorrection$(\overline{\mathrm{bel}}(\mathbf{x}_t), \mathbf{z}_t, M)$
  truncate $\mathrm{bel}(\mathbf{x}_t)$
  return $\mathrm{bel}(\mathbf{x}_t)$ as $\mathcal{X}_\mathbf{t}$

that drops particles from $\mathcal{X}_\mathbf{t}$ which are very far off. It can, for example, be implemented by predicting the new pose of the most likely $\mathbf{x}_{t-1}$ using the noiseless velocity motion model and then define particles that are further than a predefined distance away from the resulting pose as "far off". This is what the current implementation is doing, and the reasoning behind it was explained in section 5.1.

Possibly, the belief over the new pose $\mathrm{bel}(\mathbf{x}_t)$ needs to be converted to some other

Speed $[\frac{m}{s}]$	Successful Laps (out of 10)
1.4	10
1.5	7
1.6	0

Table 5.1: The particle tracker stability test. Trials for the speed values lower than $1.4\frac{m}{s}$ are omitted as the tracker never lost the pose there.

representation before it can be used by other components of the software, because a collection of pose hypotheses is not – for example – very handy for motion control. The current implementation just outputs the most likely pose hypothesis from $\text{bel}(\mathbf{x}_t)$, which can be picked up along the way when calculating the qualities.

Also, the variance in $\text{bel}(\mathbf{x}_t)$ could potentially be used to detect situations where the pose estimate has been lost completely. This, however, has not been implemented.

The current implementation is a vanilla particle filter, which could of course be extended with the usual refinements, such as not resampling in each frame, and using low variance resampling. [Thrun, 109f] This seems especially useful in light of the "flatness" of the likelihood distribution in the target range.

5.3 Evaluation

Compared to the force field tracking implementation from chapter 4, this particle-based tracker works even better. In addition to localizating the car with centimeter-accuracy, it can handle the removal of parts of the lane markings and can also recover from small tracking errors to a certain degree.

For example, it can handle the previously mentioned white field border line of the underlying soccer field. Instead of losing the pose completely, it then just "jumps" to the correct pose; this kind of incontinuity might have to be smoothed for the motion control to achieve better results.

The stability test from section 4.4 was also done for the particle tracker, and the results are shown in table 5.1. From this we can see that driving with the particle tracker is safe up to $1.4\frac{m}{s}$. However, the limiting factor here is the speed in the tight curves, because there the pose hypotheses spread over the largest area in pose space. A similar problem occurs when the car starts meandering at higher speeds, which further complicates localization. When going straight, the car could drive faster, and

Speed $[\frac{m}{s}]$	μ distance travelled $[m]$	μ penalty $[m]$	μ overall score $[m]$
1.0	119	0	119
1.1	130	10	120
1.2	141	11	130
1.3	153	20	133
1.4	164	40	124

Table 5.2: The driving quality test. Trials for the speed values lower than $1.0\frac{m}{s}$ are omitted as the car never left the lane there. All meter values were rounded down to the nearest integer.

incorporating new improved odometry noise parameters[1] into the motion model could further alleviate this problem.

To evaluate the driving behavior itself, another experiment was conducted: the car's task was to follow the circuit track from the previous experiment for two minutes, and its performance was evaluated according to Carolo-Cup rules [CCRules15] for driving without obstacles. (No lane markings were removed.) The score was determined by taking the overall distance travelled on the track minus 5 penalty meters for each instance of leaving the correct lane (which means that at least two tires were outside).

This was done for constant speeds up to $1.4\frac{m}{s}$, with three trials each, and the results are shown in table 5.2. This test shows that localization is precise enough for driving.

Almost all of the driving mistakes were made by overshooting in one of the two tight curves in the second half of the circuit, so they could likely be avoided by passing those curves more slowly. Dynamic speed control was one of the motivations for using localization in the first place and is currently being implemented as part of motion control.

The downside of the particle filter is that the ability to track multiple hypotheses comes at the cost of considerable additional computational load. Even using all cores, the number of particles is limited to around 1000 particles before the framerate starts to deteriorate. This may sound like a lot, but because the odometry readings – especially from the gyroscope – are quite noisy, and we need to track the car in the centimeter range to enable driving directly with localization, we have to pay the price here.

Although this implementation can recover from smaller errors, like the force field approach it is still fundamentally a tracker, so it can neither recover from total failures

[1] Daniel Krakowczyk recently managed to greatly improve the accuracy of the speed measurements, but the impact on motion prediction, and therefore on localization, could not be fully assessed yet. It seems likely that the required number of particles will decrease significantly. This requires the motion model's noise parameters to be readjusted.

nor find its initial pose on its own. Nevertheless, for our flat-featured model track it works quite well.

6 Conclusion

In summary, we have achieved online localization for the autonomous model car in the described model setting. The advantage of using global navigation at all for this application is that it is much more flexible, and easier to define behavior for, than using local methods only.

Two methods for tracking the car were described, each using odometry information for a first estimate and then correcting its inherent uncertainty with with sensor data from an omnidirectional camera:

- An approach based on force fields that is very fast, but can only track a single pose hypothesis, making otherwise small estimation errors potentially critical.
- Another approach using a particle filter to make tracking more robust at the cost of higher computational requirements.

Both methods are accurate enough that the resulting pose estimate can be used directly for driving, even though odometry is extremely noisy and the camera has a very limited range. They are also stable enough that the car can drive autonomously for several hours.

However, although the results look promising, the implementations have only been tested on a laboratory track. Adapting them to the longer and more feature-less tournament track might require some tweaks:

- The recent improvements on the odometry need to be incorporated into the particle filter's motion model by adjusting its noise parameters.
- Image data from the new front-facing stereo camera could be integrated into the sensor model to extend the forward vision range, improving stability especially on long, straight track segments.

Also, some general improvements to the particle tracker are possible. For example, introducing low variance resampling is an obvious extension to any particle filter to improve its robustness, as is not resampling in every time frame. Another possibility

is sacrificing accuracy to achieve global localization, which could then be used for behavior control while an accurate local lane driving strategy keeps the car going.

It would also be interesting to cut out the tedious manual mapping process by using SLAM. This can for example be achieved by recording motion and image data and then building the map using offline SLAM on a different machine. *FastSLAM* with occupancy grid mapping appears promising here, and it also fits well with the current particle filter approach. Because of the good parallelizability of that algorithm, it could even be distributed throughout multiple compute nodes.

In other words, there is still a lot to be done.

List of Algorithms

1 PoseUpdate 16

2 ForceFieldPoseCorrection 25
3 NoiselessVelocityMotionModel 27
4 ForceFieldPoseUpdate 33

5 SampleFromTriangular 41
6 VelocityMotionModel 42
7 ParticleFilterPoseCorrection 44
8 ParticleFilterPoseUpdate 44

Bibliography

[Prewitt] J. M. S. Prewitt
Object Enhancement and Extraction.
Picture Processing and Psychopictorics
January 1970

[Thrun] Sebastian Thrun, Wolfram Burgard and Dieter Fox
Probabilistic Robotics (Intelligent Robotics and Autonomous Agents).
ISBN 0262201623
The MIT Press
2005

[Hardware] Jan Frederik Boldt and Severin Junker
Evaluation von Methoden zur Umfelderkennung mit Hilfe omnidirektionaler Kameras am Beispiel eines Modellfahrzeugs.
Freie Universität Berlin, Institut für Informatik
December 2012

[ForceField] Felix von Hundelshausen, Michael Schreiber, Fabian Wiesel, Achim Liers and Raúl Rojas
MATRIX: A force field pattern matching method for mobile robots.
Technical Report B-08-03
Freie Universität Berlin, Institut für Informatik
August 2003

[Kumar] Nitesh Kumar
Localization and Mapping of Autonomous Car for Carolo Cup.
Thesis Report
International Institute of Information Technology Bangalore
June 2013

[CCRules15] *Carolo-Cup Regelwerk 2015.*
`https://wiki.ifr.ing.tu-bs.de/carolocup/system/files/Hauptwettbewerb2015.pdf`
Technische Universität Braunschweig
03.06.2014