Where am I? What's going on?

—

World Modelling using Multi-Hypothesis Kalman Filters for Humanoid Soccer Robots

Stefan Otte
stefan.otte@gmail.com

Master's Thesis
Freie Univesität Berlin
Fachbereich Mathematik und Informatik

Advisors
Prof. Dr. Raúl Rojas
Dr. Hamid Reza Moballegh

Berlin, December 5th, 2012

Abstract

Humans have an incredible ability to understand the world, to know where they are, to orient themselves, and to predict the actions of their environment. However, to imitate some aspects of these abilities in robots, even in limited environments like the RoboCup, proves to be challenging.

In this thesis, multi-hypothesis Kalman filters are used to build abstractions of the world for humanoid soccer robots to be able to answer the questions "*where am I*" and "*what is going on*". To answer these questions, which enables the robots to play soccer, a *self-localization*, an *local obstacle model*, and a *local ball model* were developed.

Generic extended Kalman filters, generic unscented Kalman filters, and tools to compare and merge Gaussians were implemented to develop an *update-merge-delete-add* cycle which was used in all developed models.

The models were tested in a simulator and on the *FUmanoid* robots. Multi-hypothesis Kalman filters have shown to be computationally highly efficient (especially compared to the previously used particle filter localization). The *update-merge-delete-add* cycle is conceptionally simple, yet powerful, and should be used in the future for the development of further models.

Contents

1
Introduction

> *"The main lesson of thirty-five years of AI research is that the hard problems are easy and the easy problems are hard."*
>
> *– Pinker [21]*

Humans have an incredible ability to understand the world, to know where they are, to orient themselves, and to predict the actions of their environment. They take these abilities for granted and do not even think about them.

These subconscious processes, which seem so easy to us, are incredibly hard to reverse engineer. In fact, Movarec states that

> "It is comparatively easy to make computers exhibit adult level performance on intelligence tests or playing checkers, and difficult or impossible to give them the skills of a one-year-old when it comes to perception and mobility." [20]

As an example: children can easily observe a ball, walk towards it and kick it. As comparison, there are international tournaments for soccer playing robots, such as the *RoboCup*, in which robots sometimes still struggle to find the ball on the field and to kick it (more on the RoboCup in chapter 2).

In this thesis, abstractions which are important for soccer playing robots are developed. These abstractions are called *models*. The goal of this thesis is to make soccer robots, more precisely the *FUmanoid* robots, able to answer the questions *"where am I"* and *"what's going on"*. For this purpose, *models* for the ball, the obstacles on the field, and the location of the robot are developed. In order to complete these tasks, filters for *recursive state estimation*, namely *multi-hypothesis Kalman filters* were applied.

The development of the ball model and obstacle model were necessary because the camera of the FUmanoid robots which had a super-wide-angle-lens was switched with a camera that has a more traditional lens. The old lens allowed the robot to observe everything in front of the robot. The robot did not have to remember where the ball was or where the obstacles were. The new lens, however, made it necessary for the robot to remember these objects because it can no longer perceive everything at the same time.

The replacement of the old localization model which uses a *particle filter* with the new multi-hypothesis Kalman filter was motivated by recent publications [22, 11, 28] which show excellent localization performance, the typical robustness of a particle filter localization, and efficient computation.

1.1 Structure

In chapter 2, the context for the development of the models is given. The RoboCup and the Team FUmanoid are introduced.

Chapter 3 covers some basics such as mathematics and concepts of probabilistic robotics which are then used throughout this thesis.

Chapter 4 explains in detail the algorithms which are typically used for modeling in the context of robotics.

In chapter 5 related work is compared.

Then, starting with chapter 6, the author's contribution to this field of research begins. This chapter analyses the task of developing three different models for the FUmanoid robots and estimates the consequences for their implementation.

The actual implementation of the models is divided into three parts. Chapter 7 describes the implementation of tools, algorithms and extensions of the FUmanoid framework which were necessary for the development of the different models. In chapter 8, the implementation of the local ball and the local obstacle model are described. In chapter 9, the implementation of the self-localization is described.

In chapter 10 the performance of the developed models is evaluated and in chapter 11 a conclusion is drawn.

2
RoboCup and Team FUmanoids

2.1 RoboCup and Humanoid Kid-Size-League

The RoboCup is an international event where research institutes and universities present and compare their research results in the areas of artificial intelligence and robotics. In a competition, teams of robots compete in soccer matches against each other. The first competition of this kind took place in 1997. Since then it has become a yearly event with changing locations. The designated goal of the RoboCup is:

> "By mid-21st century, a team of fully autonomous humanoid robot soccer players shall win the soccer game, complying with the official rule of the FIFA, against the winner of the most recent World Cup."[1]

[1] `http://www.robocup.org/about-robocup/objective/`

SOCCER AS A BENCHMARK

Soccer as a benchmark for artificial intelligence and robotics is quite interesting as compared to the task of chess which was a real challenge in the 90s[2]. The soccer environment is dynamic; there are multiple agents on the field; and data needs to be collected, processed, and put together in a coherent model of the world. The agents need to cooperate, play together as a team, carry out tasks such as scoring goals, and prevent the other team from scoring. Every one of these aspects involves a lot of uncertainty. In comparison, in chess, there is only one agent, the state of the world as well as the actions are perfectly known and deterministic, and there is no uncertainty involved.

[2] The chess computer *Deep Blue* beat the reigning chess world champion *Garry Kasparov* in 1997.

DIFFERENT LEAGUES

Since 1997, the number of soccer leagues in which universities can compete has increased. Some of these leagues are:

Small Sized League The first league of the RoboCup. A team of small driving robots is controlled from a central computer with a central overhead vision system.

Mid Sized League A team of bigger driving robots each equipped with their own vision and processing unit.

Standard Platform League A team of humanoid robots (standardized robot kit provided by *Aldebaran Robotics*) equipped with their own vision and processing unit.

Humanoid League Mostly self made humanoid robots equipped with their own vision and processing unit. Sub leagues are: *Kid Size*, *Teen Size*, and *Adult Size*.

Some non-soccer leagues such as *@Home League* and *RoboCup Rescue* are also included in RoboCup. In @Home, the robots assist humans or perform

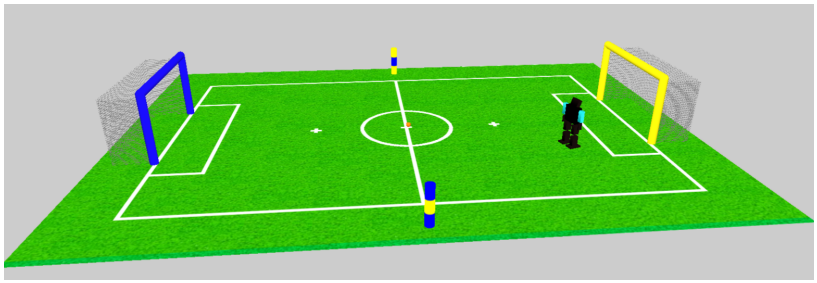


Figure 2.1: The field of play for the Humanoid Kid Size league.

everyday tasks such as preparing breakfast, watering plants etc. RoboCup Rescue involves search and rescue in large scale disaster situations.

CHALLENGES

The humanoid leagues offer the most challenges within the soccer leagues. The robots must be able to perform a wide range of dynamic motions, such as stable walking, standing up after a fall, kicking the ball, and so on. Space and payload on a humanoid robot are very limited because they need to carry all their sensors, computing equipment, actuators, and batteries.

SENSORS

The *sensors* must be human-like. This means that no active sensors such as laser-range finders are allowed. This limits the type of sensors to cameras, inertial measurement units (short *IMU*), and pressure sensors.

THE MATCH

A match takes place on a 6×4 m field. The field lines are similar to the field lines of a regular soccer field: there is a middle line, a circle in the middle, two penalty areas, and penalty points (even though the penalty points are not in the penalty area). The environment is color-coded. One goal is blue, the other is yellow. There are two unique colored side poles, blue-yellow-blue and yellow-blue-yellow (short *BYB* and *YBY*), on the middle line. The ball, a tennis ball, is orange. The robots are black/grayish. They must wear color markers (magenta or cyan) on the arms and legs to make it easier to distinguish between the teams. There are up to three robots on each team usually consisting of one goalie and two field players. There are two periods, each 10 minutes. The robots receive commands from the *game controller* which is a program controlled by a human referee (or his/her assistant). These commands include: start of the match, team magenta scored, half time, robot X is penalized, and so on.

2.2 Team FUmanoids

The *FUmanoids* are the soccer playing robots of the Free University Berlin. They participate since 2005 in the humanoid kid size league. In the following sections the hardware and the software-architecture of the robots will be described.

2.2.1 The Robot Hardware

The FUmanoid robot is about 60 cm high and has a weight of ca. 4 kg. It has five servos in each leg, one servo to bend the torso, three servos per arm, and two for the head movement (pitch and yaw). In the lower part of the body, starting with the torso, the strong *Dynamixel RX64* servos are used. In the rest of the body, the weaker *Dynamixel RX28* are used. This leads to

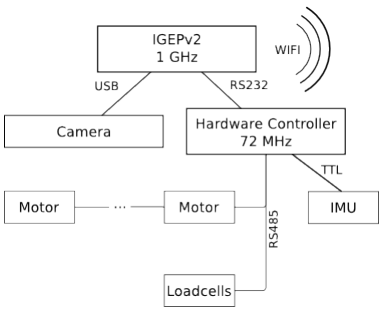


Figure 2.2: Communication within the FUmanoid robots.

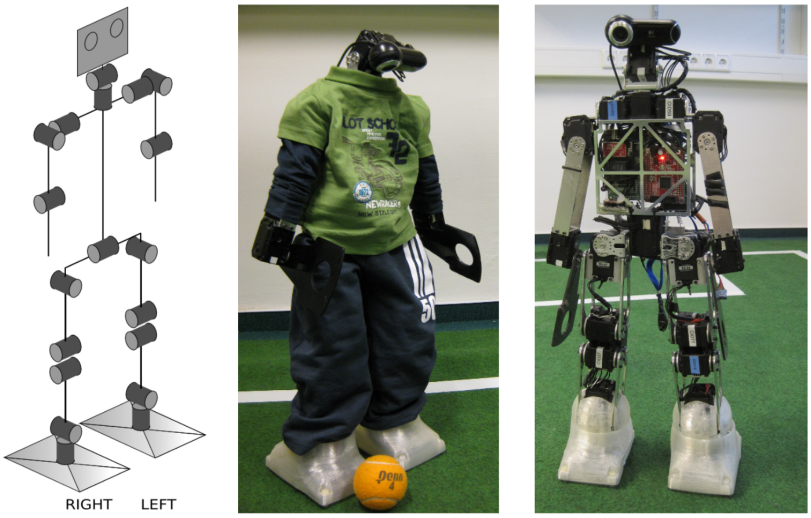


Figure 2.3: The FUmanoid robot: the structure of the kinematic (left), the actual robot with with (middle) and without clothing (right).

a total of 19 *degrees of freedom* (short *dof*).

The main computing unit is an *IGEPv2* board, with an *ARM Cortex A8* CPU (1 GHz) and 512 MB RAM. The board has several extension ports such as USB, LAN, WIFI, serial connection, and more. The WIFI module is used to communicate with teammates, the game controller, and the debugging software.

CAMERA

The main sensor is a camera, which is located in the head of the robot and connected with the IGEP via USB. In previous years, a camera with an ultra-wide-angle lens (179°) was used. This has the advantage of being able to sense everything that is in front of the robot without moving the camera. Therefore, no pitch or yaw servos were used to move the camera.

Using a normal camera, pitch and yaw servos must rotate a lot in order to cover the entire field. This movement introduces noise in the form of position uncertainty of the servos. This noise, in turn, decreases the projection accuracy.

On the other hand, using such an ultra-wide-angle lens is problematic as well. For distances greater than 3 m and on the outer regions of the image, where the distortion squishes objects together, the recognition quality decreased drastically. For this reason, the camera and lens were changed to the conventional off-the-shelf camera *Logitech C910* with a opening angle of 46° vertical and 59° horizontal. As a result, pitch and yaw servos had to be added to enable the robot to move its head again. The camera uses a resolution of 640×480 pixel with a frame rate of about 20 Hz.

The robots have inertial measurement units consisting of accelerometers, gyroscopes and magnetometers. The current configuration offers the angle and the velocity for pitch and yaw.

LOAD CELLS

The robot has four pressure sensors or *load cells* in each foot. In theory, the center of pressure can be calculated using the pressure sensors. In practice, this is not reliable enough to be used. Nevertheless, simple information, such as whether the foot is on the ground, can be used to stabilize the gait of the robot.

```
1 BEGIN_DECLARE_MODULE(BallExtractor)
2     REQUIRE(Image)
3     PROVIDE(BallPercept)
4 END_DECLARE_MODULE(BallExtractor)
5
6 class BallExtractor : public BallExtractorBase {
7     virtual void execute();
8 }
```

Listing 2.1: Creation of a simple module. label

2.2.2 FUmanoid Software Framework

The software framework of the FUmanoids was greatly improved in the years 2011/2012. The architecture of the framework is described in detail in [19]. The framework is used by the FUmanoids and NaoTH[3]. It is a *blackboard system* which separates the calculation (*e.g.,* a module to extract the ball from an image) from the data (*e.g.,* a representation for the position of the ball).

[3] www.naoth.de

BLACKBOARD SYSTEM

The key idea is to have a simple system which makes it easy for new students to join the team and start developing right away. This is achieved by strong modularization. From the users' point of view, there are two main concepts. First, there are *representations* which are dumb data objects. They carry information such as the image from the camera, the position of the ball in the image, or the time left in a match. Representations do not have any complex functionality. Second, there are *modules*, which do the actual calculations. These modules require representations as input and provide representations which yield surplus information. the module BallExtrator requires the representation Image. It searches for the ball in the image and, if found, provides the representation BallPercept which contains the position of the ball.

REPRESENTATIONS

MODULES

Representations are automatically saved on a *blackboard* which makes them accessible to the modules. Specifically, modules which required representations automatically have read-only access to the required representations, modules which provide representations automatically have read-write access to the provided representations. Listing **??** shows how to create a new module.

BLACKBOARD

Modules and representations form a directed acyclic graph. These graphs are stored on so-called *module managers*. Module managers are meant to separate concerns. Currently there are two main module managers in the FUmanoid system, each responsible for their own subtask:

MODULE MANAGERS

Cognition module manager: The cognition module manager is responsible for computer vision, modeling, and the execution of the behavior. It runs at 20 hz/sec, which corresponds to the current frame rate of the camera.

Motion module manager: The motion module manager is responsible for the execution of motions. It runs at 80 hz/sec.

The two module managers communicate by exchanging representations. *E.g.,* the output of the behavior layer is the representation MotionRequest which

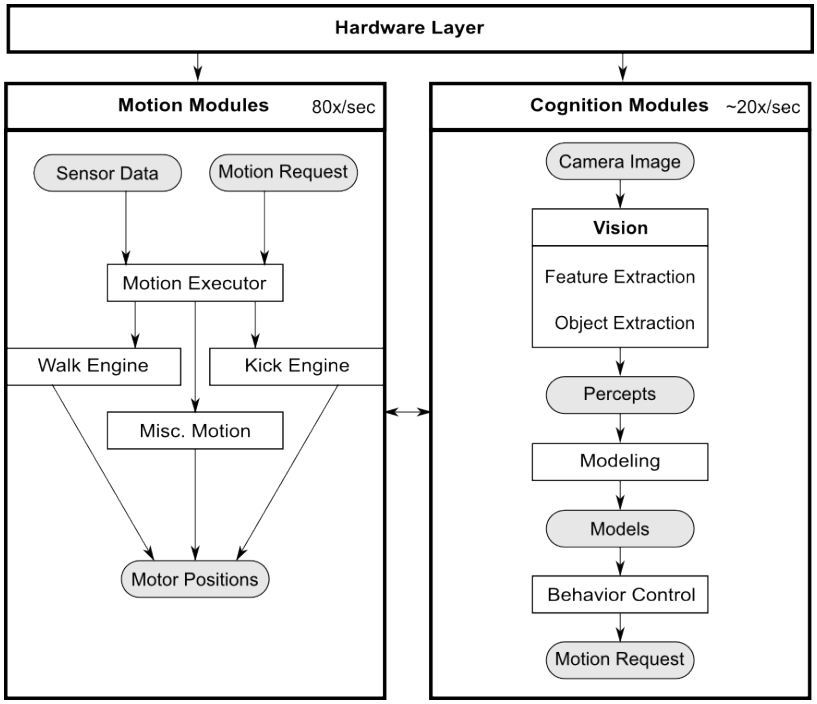


Figure 2.4: Simplified software architecture of the FUmanoids. White rectangles are modules, gray ovals are representations. Arrows represent the flow of information.

is the input for the execution of the motion layer. The overview of the FUmanoid architecture is shown in figure 2.4.

2.2.3 *Additional Tools*

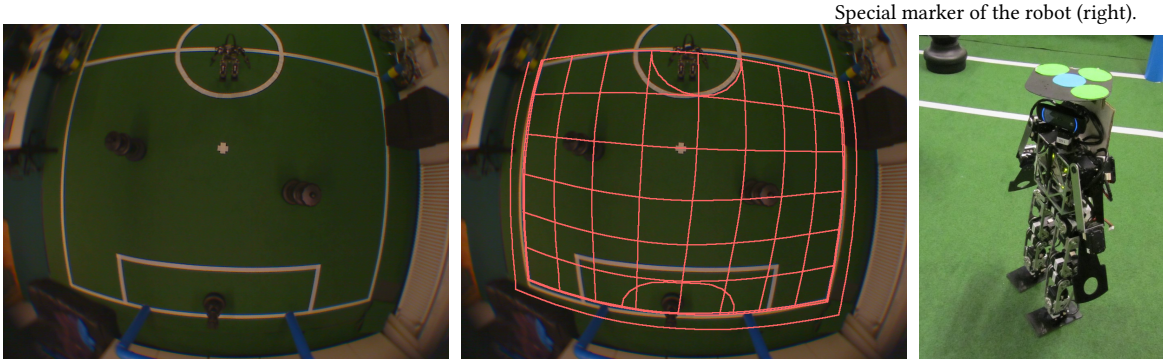
SIMSTAR

To ease developing and testing, the FUmanoids program can be used with the robot simulator *SimStar* [10]. The simulator can provide data for all sensors which includes the data from the IMU and the camera image, but it can also inject ground truth data into the FUmanoid system. This method of providing data to the FUmanoid system is completely transparent. With the simulator, it is possible to test the robot soccer behavior with a perfect knowledge about the world, or to evaluate certain aspects of the system such as the self-localization.

OVERHEAD CAMERA SYSTEM

The newest addition to the FUmanoid tools is an *overhead camera system* which allows the system it to track the robots and the ball. The camera which has a wide-angle lens is mounted on the ceiling and observes the field. The robots must wear special colored markers (see figure 2.5 on the

Figure 2.5: Overhead camera system. View from the over head camera (left and middle). Special marker of the robot (right).



right) to be recognized by the overhead camera system. The provided data (robot pose and ball position) can then be used as ground truth data in the robot. The ground truth data can be compared against certain aspects of the robot such as the self-localization and models. However, the overhead camera system is a new tool and is not very accurate yet.

The debugging and remote control software for the FUmanoid program is called *FUremote*. It enables live visualization of different aspects of the robot such as the vision and the modeling system.

FUREMOTE

3
Basics

This chapter will cover some basics which are required for the rest of this thesis. First, some basic mathematics, specifically the calculus of probability, will be discussed. This will be followed by the topic of probabilistic robotics.

3.1 Mathematics and Probability

The following references were used for this section: [3, 4, 29].

3.1.1 Frequentism vs. Bayesianism

EVENTS

The classical view of statistics is that one has *events*, often called E. Events declare if something is true or not, *e.g.,* the tossed coin is head. The coin toss is a *trial*. The probability is then defined as the ratio of events which are true (the coin was head 50 times), divided by the total number of trials (100 coin tosses): $P(E) = \frac{50}{100} = .5$. This classical approach is called *Frequentism*. Frequentism has its limits when predictions of events which have not yet happened are required.

TRIAL

FREQUENTISM

Bayesianism defines probability as a degree of belief in an event occurring. Therefore, events like the outcome of an election or nuclear catastrophes can be predicted even if they have not yet happened. The groundwork of Bayesianism was laid out by Bayes (1701–1761) (see section 3.1.3).

BAYESIANISM

Up until recently (the last few decades), Bayesianism survived in a niche of statistics, even though it was applied with great success in many fields like economics and cracking the enigma code. It was said to be too subjective because it requires a so-called *prior*, which is some subjective knowledge of the event which can differ from person to person. See McGrayne [17] for more about the history of Bayes' theorem.

3.1.2 Basic Probability and Probability Distributions

RANDOM VARIABLE

A *random variable* (short *RV*) is a variable which can take on different values. The values can be discrete or continuous. *E.g.,* in the discrete case of tossing a coin, the random variable can take on the value *head* or *tail*. The probability for each outcome is $P(X = head) = P(X = tail) = .5$.

MEAN

The *mean* is the *expected value* of a RV. For example, the mean of a fair die is 3.5. The sum of all outcomes multiplied with their individual probability leads to the mean:

$$E[X] = 6 \cdot \tfrac{1}{6} + 5 \cdot \tfrac{1}{6} + 4 \cdot \tfrac{1}{6} + 3 \cdot \tfrac{1}{6} + 2 \cdot \tfrac{1}{6} + 1 \cdot \tfrac{1}{6} = 3.5 \tag{3.1}$$

More generally the formula for the mean is

$$E[X] = x_1 p_1 + x_2 p_2 + \cdots + x_n p_n = \sum_{i \in N} x_i p_i \tag{3.2}$$

where p_i denotes the probability of the outcome x_i. The mean is often abbreviated with μ.

VARIANCE

Often, the mean on its own offers little information. The *variance* is a measure of how far samples are spread out from the mean. More precisely, it is the mean squared deviation:

$$\sigma^2 = \text{Var}(X) = E[(X - \mu)^2] = \frac{1}{n} \sum_i (x_i - \mu)^2 \tag{3.3}$$

STANDARD DEVIATION

The *standard deviation* σ is the square root of the variance. The unit of the standard deviation is the same as the unit of the actual measurements. This allows one to intuitively evaluate the meaning of the standard deviation, whereas this is harder for the variance.

COVARIANCE

The *covariance* is a measure of how two RVs correlate.

$$\text{cov}(X, Y) = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y] \tag{3.4}$$

$E[X]$ is the expected value of x, which is often written as μ_x.

The covariance matrix is the generalization of the covariance to a vector of RVs. Each element in a covariance matrix indicates the covariance of one RV to another RV in a vector of RVs:

$$\Sigma_{i,j} = \text{cov}(X_i, X_j) = E[(X_i - \mu_i)(X_j - \mu_j)] \tag{3.5}$$

In other words

$$\Sigma = \begin{bmatrix} \text{cov}(X_1, X_1) & \text{cov}(X_1, X_2) & \cdots & \text{cov}(X_1, X_n) \\ \text{cov}(X_2, X_1) & \text{cov}(X_2, X_2) & \cdots & \text{cov}(X_2, X_n) \\ \vdots & \vdots & \ddots & \vdots \\ \text{cov}(X_n, X_1) & \text{cov}(X_n, X_2) & \cdots & \text{cov}(X_n, X_n) \end{bmatrix} \tag{3.6}$$

PROBABILITY DISTRIBUTIONS

To be able to express data more fully, one can use *probability distributions* instead of summary statistics such as the mean and the variance. Distributions assign each value a probability. They exist for the discrete and the continuous case. Given the outcomes for a discrete RV, the *probability mass function* (short *PMF*) assigns each outcome of the discrete random variable a probability. *E.g.,* a PMF for a fair die would assign each outcome of $1 \ldots 6$ the probability of $\frac{1}{6}$.

PROBABILITY MASS FUNCTION

PROBABILITY DENSITY FUNCTION

A *probability density function* (short *PDF*) is to continuous random variables what the PMF is to discrete random variables. But to obtain the probability of a continuous random variable one needs to integrate over an interval. The PDF is a function which assigns a random variable a probability of being the given value. PMFs sum up to 1

$$\sum_{x \in X} P(x) = 1 \tag{3.7}$$

whereas PDFs integrate to 1

$$\int P(x)\, dx = 1 \tag{3.8}$$

The *mode* is the value with the most appearances in a set of data. Example: the mode of the set $\{2, 1, 5, 2\} = 2$.

MODE

PDFs and PMFs can be *unimodal* or *multimodal*. Different definitions of unimodal exist. The most common one defines unimodal as a distribution which only has one maximum and therefore only one move. Vice versa, distributions with more than one maximum are called multimodal.

UNIMODAL
MULTIMODAL

The *Gaussian distribution* is a very common probability distribution. It has a bell shaped PDF and is *unimodal*. It is completely defined by its first two *moments*: the *mean* μ and the *covariance* σ^2.

GAUSSIAN DISTRIBUTION

$$P(x) = (2\pi\sigma^2)^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right\} \tag{3.9}$$

A Gaussian Distribution is often simply called *Gaussian* or *Normal Distribution*. As such, it is written as

$$\mathcal{N}(\mu, \sigma^2) \tag{3.10}$$

or when evaluating the PDF as a position x

$$\mathcal{N}(x \mid \mu, \sigma^2) \tag{3.11}$$

Gaussians are often used to approximate (unknown) distributions that cluster around a point. Furthermore, they have some useful properties:

- It is easy to analytically manipulate a Gaussian distribution.
- A Gaussian can be translated and remain a Gaussian.
- A Gaussian can be linearly transformed and remain a Gaussian.
- A Gaussian multiplied with a Gaussian results in a Gaussian.

The *multivariate Gaussian* is the extension of the Gaussian to the multivariate case. In this case, the state x is not a scalar value but a vector, μ is the mean vector, and Σ is a covariance matrix.

MULTIVARIATE GAUSSIAN

$$\mathcal{N}(x \mid \mu, \Sigma) = \det(2\pi\Sigma)^{-\frac{1}{2}} \exp\left\{-\tfrac{1}{2}(x-\mu)^T\Sigma(x-\mu)\right\} \tag{3.12}$$

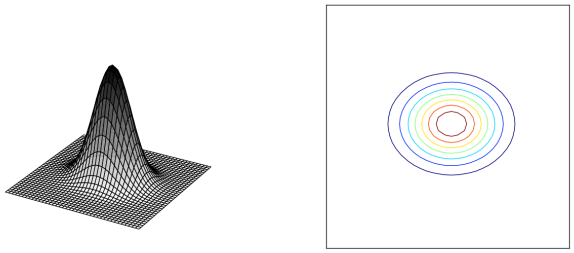


Figure 3.1: Multivariate (bivariate) Gaussian. 3D-view on the left, contour lines on the right.

Sometimes unimodal distributions like the Gaussian are not expressive enough to model certain problems. The *sum of Gaussians* or *Gaussian mixture* is the extensions of Gaussians to the *multi-modal* case. Any distribution can be approximated with a sum of Gaussians.

SUM OF GAUSSIANS
MULTI-MODAL

In addition to the mean μ_i and covariance matrix Σ_i, each Gaussian in a sum of Gaussians has a weight w_i which specifies the influence of the

Gaussian on the overall distribution.

$$\sum_{i \in I} w_i \, \mathcal{N}(\mu_i, \Sigma_i) \tag{3.13}$$

The sum of the weight of the Gaussians must be 1.

$$\sum_{i \in I} w_i = 1 \tag{3.14}$$

Consequently the integral over a sum of Gaussians is 1.

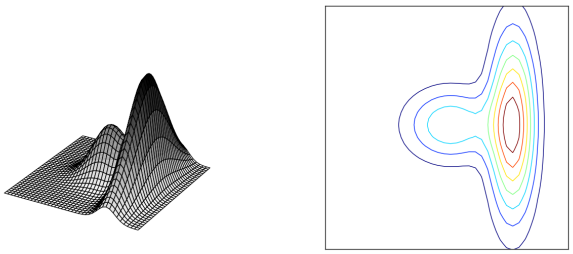


Figure 3.2: Multivariate (bivariate) sum of Gaussians. 3D-view on the left, contour lines on the right.
$w_1 = 0.3$
$w_2 = 0.7$

3.1.3 *Bayes' Theorem*

The probability that two RVs are true at the same time is called *joint probability*.

JOINT PROBABILITY

$$P(a \wedge b) = P(a)P(b \mid a) = P(b)P(a \mid b) \tag{3.15}$$

If, and only if, a and b are conditionally independent, the joint probability can be written as:

$$P(a \wedge b) = P(a)P(b) \tag{3.16}$$

Knowledge about one RV (*e.g.,* it rained) often yields information about another RV (*e.g.,* is the lawn wet). This can be expressed with the concept of *conditional probability*.

CONDITIONAL PROBABILITY

$$P(a \mid b) = \frac{P(a \wedge b)}{P(b)} \tag{3.17}$$

From equation (3.15), *Bayes' theorem* can be inferred easily by solving for one of the two conditional probabilities, *e.g.,* $P(b \mid a)$:

BAYES' THEOREM

$$P(b \mid a) = \frac{P(b)P(a \mid b)}{P(a)} \tag{3.18}$$

Thomas Bayes called his method *inverse probability* [17].

A different interpretation of the Bayes' theorem is known as the *diachronic interpretation*. Diachronic comes from the Greek word *Diahronikos* and means: something happening over time. In this interpretation it states how a body of evidence E affects the hypotheses H over time. New evidence influences the probability of the previous hypothesis:

DIACHRONIC INTERPRETATION

$$P(H \mid E) = P(H)\frac{P(E \mid H)}{P(E)} \tag{3.19}$$

This can be easily expressed by replacing the probabilities with more appropriate terms:

$$\text{posterior} = \text{prior hypotheses} \frac{\text{likelihood of evidence}}{\text{normalization}} \tag{3.20}$$

Even though Bayes' theorem looks relatively simple and insignificant, it forms the core of most modern robotics and machine learning. See section 3.2 and especially chapter 4 for more on this.

3.1.4 Markov Assumption and Markov Chain

MARKOV PROPERTY

The *Markov assumption* or *Markov property* can be summarized as follows: the future is independent of the past given the present. *I.e.,* if the current state describes the world well enough to make the best possible prediction of the future and additional information from the past would not improve the prediction, then the Markov property is valid. Mathematically, the Markov property allows

$$P(x_{t+1} \mid x_{1:t}) \tag{3.21}$$

which uses all previous states to predict the next state to be simplified to

$$P(x_{t+1} \mid x_t) \tag{3.22}$$

which only uses the current state to predict the next state. The Markov property is often used for recursive state estimation, see chapter 4.

COMPLETE STATE

A state that fulfills the Markov property is called a *complete state*. For real world problems, the complete state concept is mostly a theoretical assumption. In practice, it is basically impossible to model the state space as a complete space. But to facilitate calculations, the state is simply assumed to be complete, even though it is actually an *incomplete state*.

INCOMPLETE STATE

MARKOV CHAIN

Figure 4.2 depicts a *Markov chain*. A Markov chain is a temporal process that meets the Markov property. The next state x_{t+1} only depends on the previous state x_t and (if it exists) on the current control u_{t+1}. As one will see later, the Markov chain is used for many modeling tasks (see section 3.2).

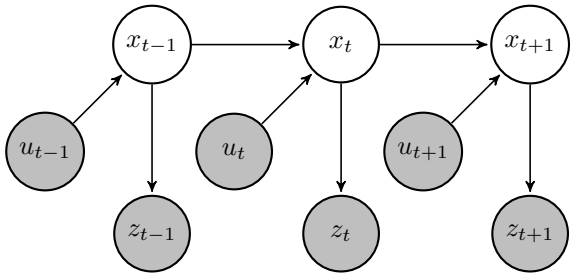
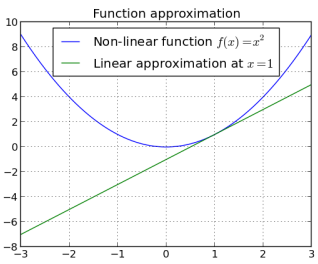


Figure 3.3: The localization task visualized as a Markov chain. The gray circles (control u, and measurements u) are known; the white circles (state x) are unknown.



3.1.5 Local Linear Approximation of Functions

LINEARISATION

Some algorithms are designed to handle linear functions. But in many cases, the relation between RV is not linear. In robotics, it is often necessary to find a linear approximation of non-linear functions. The *linearisation* of function f at point k is defined as

$$f(x) \approx f(k) + \frac{df}{dx}(k)(x - k) \tag{3.23}$$

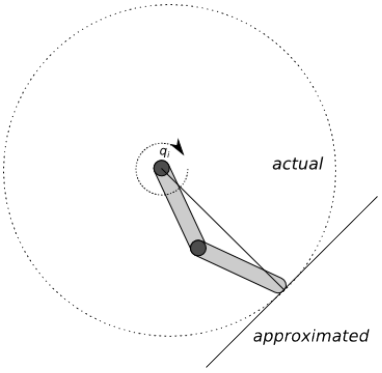
The *Jacobi matrix* or *Jacobian* extends the approach of linearization to the multi-dimensional case. It is a matrix consisting of the first-order partial derivative of a vector or scalar-valued function with respect to another vector.

JACOBIAN

$$J_{i,j} = \left(\frac{\partial F_i(x)}{\partial x_j} \right) \tag{3.24}$$

$$J = \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1} & \cdots & \frac{\partial F_m}{\partial x_n} \end{pmatrix} \tag{3.25}$$

A typical application of a Jacobian is the motion generation of robotic arms (see figure 3.1.5). Inverse kinematics are used to calculate the joint angles. The dotted line indicates the exact motion of the arm when changing q_i. The motion predicted by the Jacobian, the linear approximation, is shown with the black line. Close to the point of the approximation the prediction is normally satisfactory. However, the higher the distance, the less precise the prediction.



Another application of Jacobians is the extended Kalman filter. For more on this see section 4.4.

3.2 Probabilistic Robotics

The main reference for this section is Thrun et al. [29].

Uncertainty is involved in every aspect of robotics and is the cause for many problems in robotics. Sensors and actuators only have a certain degree of accuracy, have a limited field of operation, and are fragile. When measuring the distance to an object, the distance rarely reflects the exact distance. An observed landmark might be incorrectly classified, or the landmark might not even be there. Actuators do not necessarily follow given commands. The world itself can be unpredictable even though it follows physical laws. Other agents and humans can show unpredictable behavior. Aspects of the world which are not modeled are simply not predictable. No matter what the robot perceives and does, it cannot be sure about the world.

UNCERTAINTY

The key idea of *probabilistic robotics* is to incorporate these uncertainties into the models. The aim is not just to create a best guess, but a belief distribution over the entire state space. Ambiguities can be represented and depending on the uncertainty of the state, the robot can act accordingly. In general, probabilistic approaches often outperform traditional approaches. The disadvantage of probabilistic approaches is an increase in running time. Most of the benefits of probabilistic robotics are achieved by applying Bayes' theorem (see section 3.1.3), and the temporal extension of the Bayes' theorem, the *Bayes' filter* (see section 4.1).

PROBABILISTIC ROBOTICS

In the following sections, basic terms of probabilistic robotics are introduced, followed by a discussion of some typical aspects of probabilistic robotics, and then potential problems and solutions are outlined. Afterwards, the reader should have a general understanding of typical approaches of probabilistic robotics.

3.2.1 Basic Terms

AGENT
WORLD

An *agent* is normally a robot which moves around and interacts with its environment. The environment a robot acts in is called the *world*. The world can be a simple board game, it can be part of a room, a floor, or a huge open area. In the world, the agent uses whatever sensors it has to measure the world's properties. The observations are internally represented through data structures called *percepts*. Observations of objects are usually acquired in the form of *local coordinates*. The robot is the origin of the coordinate frame and the measurement is the relative position of the percept in this frame. Percepts are short lived. They only exist if a property is being observed (if the robot looks away it has no percept anymore).

PERCEPTS
LOCAL COORDINATES

MODEL

A *model* is an abstraction of a real world property. Models are usually valid over time and do not cease to exist when there is no percept anymore. Different aspects can be be modelled, *e.g.,* the position of another robot relative to itself. The *pose* of the robot is often modeled. It consists of the position and the orientation (x, y, θ) of the robot in a *global coordinate system*. The variables of a model are called *state space*.

POSE
STATE SPACE

LANDMARKS
STATIC LANDMARKS
DYNAMIC LANDMARKS

For certain tasks, certain properties of the world are more important than others. For the localization task, in which the robot has to estimate where it is in the environment, so-called *landmarks* are very important (see section 3.2.2), because they allow to infer the location of the robot in the world. There are two types of landmarks. *Static landmarks* are properties of the world that do not change their location or appearance. In contrast, *dynamic landmarks* are landmarks that can change their location, and sometimes have the ability to appear and disappear. Dynamic landmarks usually (not always) contain less information than static landmarks.

STATIC VS. DYNAMIC ENVIRONMENTS

In this context, environments are usually divided into *static environments* and *dynamic environments*. As the name suggests, in static environments everything except the robot is static. The world can be described by a few variables, normally by the robot pose. In dynamic environments, the robot and other objects move and influence the world. Therefore, the state space required to describe dynamic environments is much larger.

3.2.2 Models

LOCALIZATION
SLAM
TRACKING OBJECTS

In textbooks, one often differentiates between three types of modeling tasks. On the one hand, there is the pure *localization* task. The world with all its features and landmarks is perceived by the agent. The agent has to localize itself in this world using prior knowledge, the map of the world. On the other hand, there is the other extreme, the so-called *simultaneous localization and mapping* (short *SLAM*). The agent is placed in a completely unknown environment, and needs to create its own map of the environment and localize itself in it at the same time. SLAM is a much harder problem than the pure localization task. The third task involves *tracking objects* in an environment, *e.g.,* tracking a rocket from a ground station or tracking a moving ball on a field.

3.2.3 Localization

The localization task in the previous section can be divided into different categories. There is *position tracking*, in which the initial pose of the agent is known. The position is simply updated by incorporating odometry data. Percepts are used to correct the prediction.

POSITION TRACKING

In contrast, with *global localization*, the initial pose is unknown. The robot must estimate its pose in the world and then track its position using control (odometry) and measurements of landmarks. The problem is that the robot pose cannot be measured directly. In general, unimodal distributions are sufficient for position tracking, but not for global localization.

GLOBAL LOCALIZATION

The *kidnapped robot problem*, an extension of the global localization, is more difficult. In addition to the global localization, the agent can be *teleported*, *i.e.,* moved to a different location without the agent's knowledge. The agent then has to realize that a teleportation has taken place and relocalize. In certain scenarios like the RoboCup (see section 2.1), teleportations happen quite often and are one of the largest challenges. Being able to deal with the kidnapped robot problem also increases the robustness of the localization because the agent never really knows if its pose is correct.

KIDNAPPED ROBOT PROBLEM

3.2.4 Passive vs. Active Approaches

One can divide modeling approaches into active and passive methods. Using a *passive approach* the modeling module only perceives the world. It has no control over the agent and does not actively explore the world or plan the path of the agent in order to maximize perception of objects. In contrast, *active approaches* do exactly this. The modeling module takes actions in order to improve the quantity and/or quality of observations, *i.e.,* it moves the agent towards landmarks or other objects to obtain better information. In general, active approaches yield better results than passive approaches, but active approaches also influence the behavior of the robot. One has to keep in mind that most modeling tasks are only means to an end. Localization in the RoboCup scenario is only necessary for the robots to be able to play soccer, not for the localization itself. Therefore, taking a detour to improve the localization can decrease the actual performance of the robot playing soccer.

PASSIVE APPROACH

ACTIVE APPROACHES

3.2.5 Single-Agent vs. Multi-Agent

A scenario in which one robot models some aspects of the world is called *single-agent* scenario. The agent only uses its own sensors to collect data from the environment.

SINGLE-AGENT

If there are several agents in a world, one speaks of a *multi-agent* scenario. At first glance nothing changes. The modeling task can still be treated as a single-agent modeling task. But the combined knowledge of each agent might lead to improved models of every agent. Having multiple agents means that more of the world can be observed. This knowledge can be communicated between agents.

MULTI-AGENT

More information does not automatically lead to better models. The issues associated with distributed systems must first be addressed. One has to

make sure that modeling errors in one robot do not accumulate in all robots. The information from other robots must not be too old. One has to prioritize the information properly, *i. e.,* local information is more up-to-date and potentially less noisy. Despite the potential new problems, multi-agent scenarios offer the possibility to improve the modeling tremendously.

3.2.6 Symmetric Environments and Localization

Symmetric environments do not offer a way to resolve symmetries (see left side of figure 3.4). Therefore, it is substantially harder to localize in symmetric environments than in asymmetric environments. In the example in figure 3.4, one agent is in a *square world*, and one agent is in a *triangle world*. Each agent perceives two walls which are orthogonal (top row). Due to the nature of the environment the agent in the square world has four potential hypotheses for its poses (bottom row, left). It is impossible (without knowing the initial pose) to determine the correct pose of the agent. The agent in the triangle world has only one hypothesis (bottom row, right). Even though purely symmetric worlds are rare, parts of the world are often symmetric. Also, the parts which might resolve symmetries are not always observable. Therefore, one needs to be able to handle multi-modal belief distributions and resolve symmetries later.

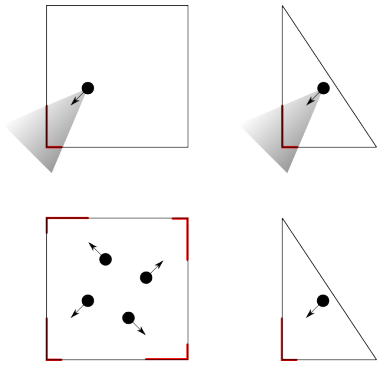


Figure 3.4: Example for a symmetric world (left) and an asymmetric world (right). The top shows the real pose of the robot and what it perceives. The bottom shows possible poses of the robot.

4
Recursive State Estimation

The main reference were for this part is [29].

As mentioned in the previous chapter, states in probabilistic robotics are represented by probability distributions. Control and measurements are noisy and the state cannot usually be measured directly. Recursive state estimation is the main instrument for estimating state distributions by incorporating controls and measurements.

Section 4.1 introduces the basic framework, the *Bayes' filter*, to model state distributions. Following this introduction, three concrete realizations of the Bayes' filter will be explained:

- the *histogram filter* (see section 4.2),
- the *particle filter* (see section 4.3), and
- the *Kalman filter* (see section 4.4) and its extensions the *extended Kalman filter* and the *unscented Kalman filter*.

4.1 Bayes' Filter

The Bayes' filter is a very general filter / framework for calculating belief distributions. A *belief* represents the agent's internal knowledge of the state of the world. This is not necessarily the real state of the world. Mathematically, the belief is the same as a probability distribution over the state. It is really just the *state of knowledge* of the agent. The Bayes' filter assigns every possible state a probability depending on the information the agent has collected so far. Calculating the belief distribution usually happens in two steps: the *prediction step* and the *correction step*. Because of this, the Bayes' filter is often called the *prediction correction filter*. After incorporating all past measurements $z_{1:t}$ and all past controls $u_{1:t}$, the Bayes' filter returns the belief $bel(x_t)$:

BELIEF

$$bel(x_t) = P(x_t \mid z_{1:t}, u_{1:t}) \tag{4.1}$$

There are two important concepts of the Bayes' filter: the *state transition probability* and the *measurement probability*.

The *state transition probability* describes the probability of being in state x_t given all previous states, controls, and measurements:

STATE TRANSITION PROBABILITY

$$P(x_t \mid x_{0:t-1}, z_{1:t}, u_{1:t}) \tag{4.2}$$

Under the assumption that x_{t-1} is a complete state (see section 3.1.4) the equation is equal to:

$$P(x_t \mid x_{t-1}, u_t) \tag{4.3}$$

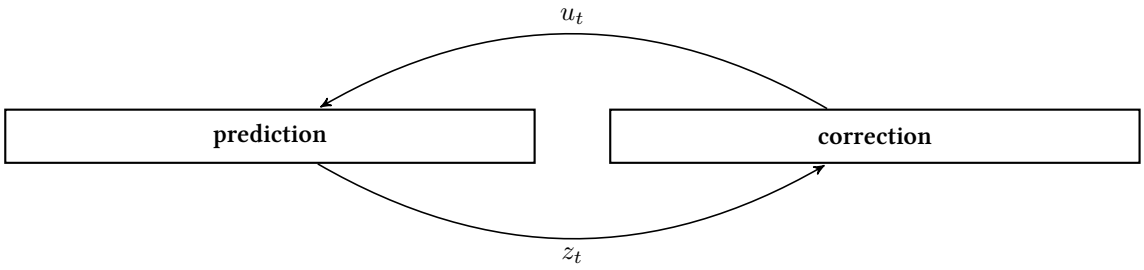


Figure 4.1: The typical structure of a Bayes' filter. Because of the prediction and the correction step it is often called prediction-correction-filter.

This formula makes it easy (and computationally much cheaper) to calculate the probability for each state given the previous state and the control. Equation (4.1) is simplified to represent the state before incorporating the newest measurement z_t:

$$\overline{bel}(x_t) = P(x_t \mid z_{1:t-1}, u_{1:t}) \tag{4.4}$$

Together with equation (4.3), this is later used to perform the prediction step.

MEASUREMENT PROBABILITY

The *measurement probability* describes the probability of obtaining the measurement z_t given all previous states, measurements, and controls.

$$P(z_t \mid x_{0:t}, z_{1:t-1}, u_{1:t}) \tag{4.5}$$

Again, under the assumption that x_t is a complete state, this can be simplified to:

$$P(z_t \mid x_t) \tag{4.6}$$

With these four concepts (the *state transition probability*, the belief before incorporating the measurement $\overline{bel}(x_t)$, *the measurement probability*, and the state after incorporating the measurement $bel(x_t)$), the prediction and the correction step of the Bayes' filter can be stated as follows:

Prediction: The prediction step combines the previous belief $bel(x_{t-1})$ with the current control u_t. *I. e.,* given the belief of the previous state $bel(x_{t-1})$ and the current control u_t, the prediction for the current belief $\overline{bel}(x_t)$ is calculated by integrating over the product of the prior, the previous state $bel(x_{t-1})$, and the state transition probability.

$$\overline{bel}(x_t) = \int P(x_t \mid x_{t-1}, u_t) \;\; bel(x_{t-1}) \; dx_{t-1} \tag{4.7}$$

Correction: In the correction step, the measurement z_t improves the predicted state $\overline{bel}(x_t)$. The final belief $bel(x_t)$ is the distribution of the normalized product of the predicted belief $\overline{bel}(x_t)$ and the measurement probability.

$$bel(x_t) = \eta \; P(z_t \mid x_t) \;\; \overline{bel}(x_t) \tag{4.8}$$

The two steps are repeated for all possible states. The complete algorithm is depicted in listing 4.1.

4.1.1 Example

The rather theoretical discussion of the Bayes' filter in the previous section is illustrated by a concrete application of the Bayes' filter. Figure 4.3 shows a

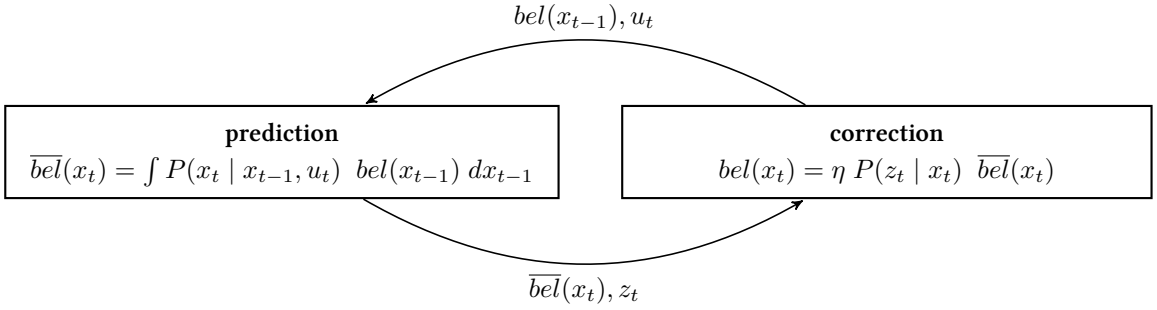


Figure 4.2: Bayes' filter: prediction-correction cycle.

```
1 def bayes_filter(bel(x_{t-1}, u_t, z_t)):
2     for x_t in all X:
3         # prediction
4         bel-bar(x_t) = ∫ P(x_t | x_{t-1}, u_t)  bel(x_{t-1}) dx_{t-1}
                          state transition probability
5         # correction
6         bel(x_t) = η  P(z_t | x_t)  bel-bar(x_t)
                      measurement probability
7     return bel(x_t)
```

Listing 4.1: Bayes' filter algorithm.

Bayes' filter in action. In this section, each step of the example is explained to illustrate how the Bayes' filter works.

The world in this example is a simple 1D world with two gray towers as landmarks (see the first row). The positions of the landmarks are known and the landmarks are indistinguishable from each other. The agent, the black robot, can move left and right in the world. It can only sense landmarks which are at the same position as the agent itself. The right part of the figure represents the sensor input or the control. As explained in the previous section, the Bayes' filter consists of two steps: prediction and correction.

INITIAL STATE

In the *initial state* (row 2), the agent is unaware of its location. The belief is initialized as a uniform distribution because every position in the world has the same (low) probability of being the correct one. In the previous section, the prediction step was followed by the correction step. However, the Bayes' filter does not require this. In practice, there are often multiple correction steps and only one prediction step or vice versa. This also makes it possible to incorporate multiple measurements at one time step.

FIRST SENSOR UPDATE

In the example, the agent knows the world, *i. e.,* it knows where the landmarks are. It can sense a landmark directly at its starting position (row 3). Therefore, it knows that the most likely position is at (or near) one of the two towers. The resulting belief distribution is bivariate. Also, the agent is aware that there are uncertainties in its sensor. Therefore, it cannot be sure that the perceived landmark's position is exact. This explains the Gaussian-like form of the distribution.

FIRST PROCESS UPDATE

In the next step, the agent moves to the right (row 4). Moving increases the uncertainty of the location of the agent. The belief distribution is shifted along the traveled distance according to the odometry and is blurred. In

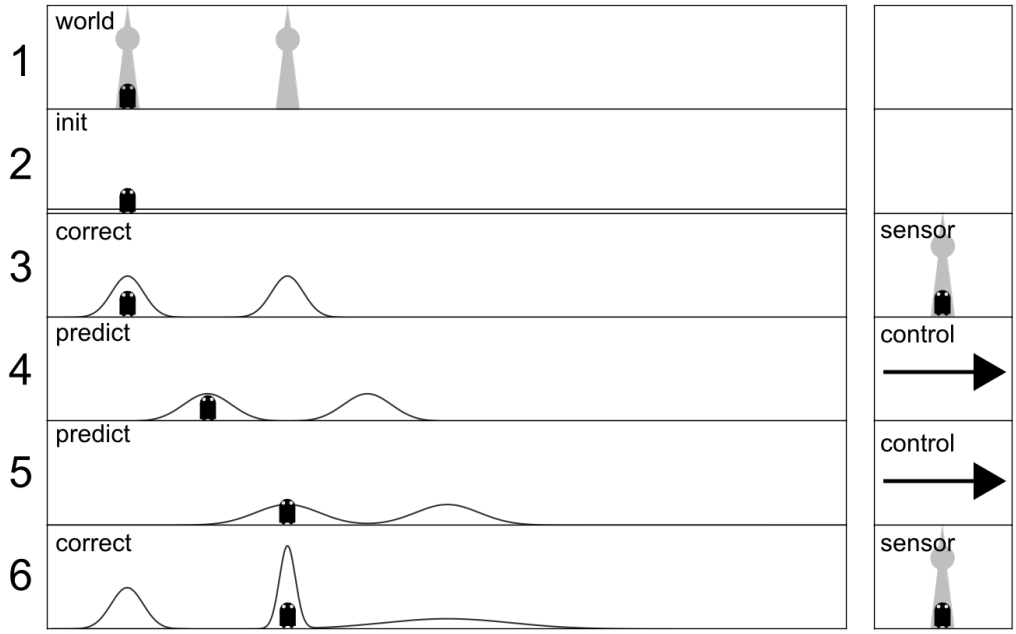


Figure 4.3: An example of a Bayes' filter.

the current state (still row 4) the agent does not sense a landmark. It moves further to the right and the uncertainty increases (row 5). Note that two prediction steps followed without a correction step.

SECOND PROCESS UPDATE

Then, the agent observes a landmark (row 5). Given the prior belief distribution and the measurement, the new belief distribution mostly resolves the ambiguities of possible positions. The most probable position is at the second landmark which reflects the true position.

SECOND SENSOR UPDATE

Even though the example here is simple, it illustrates the basic steps of all Bayes' filters. The key difference of concrete implementations of the Bayes' filter is the way belief distributions are represented and measurements are incorporated.

4.2 Discrete Histogram Filter

The discrete histogram filter is a Bayes' filter for a finite state space. It follows the Bayes' filter step for step and only replaces the integral in the prediction step with a sum.

Listing 4.2 shows the complete algorithm which is very similar to listing 4.1. x denotes the states. There are k states. The probability at time t at each state is $p_{k,t}$ and $\{p_{k,t}\}$ is the probability distribution, a PMF. u_t and z_t are the most recent control and the most recent measurement. Because of the similarity of the histogram filter to the Bayes' filter algorithm the histogram filter is not explained further.

The histogram filter is very useful for making estimations when the number of states is low. It can also be used for continuous state spaces by dis-

```
1 def histogram_filter(\{p_{k,t-1}\}, u_t, z_t):
2     for all k:
3         # prediction
4         \overline{p}_{k,t} = \sum_i \underbrace{P(X = x_k \mid u_t, X_{t-1} = x_i)}_{\text{state transition probability}} p_{i,t-1}
5         # correction
6         p_{k,t} = \eta \underbrace{P(z_t \mid x_t)}_{\text{measurement probability}} \overline{p}_{k,t}
7     return \{p_{k,t}\}
```

Listing 4.2: Discrete histogram filter algorithm.

cretization of the state space.

4.3 Particle Filter

The *particle filter* (short *PF*) approximates the belief distribution as a finite set of particles. The state space can be *univariate* or *multivariate*. Each particle corresponds with a region in the state space. Any distribution, including complex multi-modal distributions, can be approximated with a high number of particles. As such, the particle filter belongs in the category of *non-parametric* filters. Also, the particle filter can handle non-linear transformation.

NON-PARAMETRIC

As with the Bayes' filter, the PF computes the posterior $bel(x_t)$ from the belief $bel(x_{t-1})$ by incorporating the current control u_t and the current measurement z_t. The algorithm requires a set of particles from the previous time step X_{t-1}, the current control, and the current measurement. It then predicts the state of each particle according to the control z_t and calculates the measurement likelihood for the predicted particle. The measurement likelihood is often called *importance factor*. The new particles are saved in a temporary set of particles, $\overline{X}_t$.

IMPORTANCE FACTOR

Next, the importance factor is used in the *resampling step* or *importance sampling*. Each particle in $\overline{X}_t$ is drawn with replacement with the likelihood of the importance factor. This (most likely) eliminates particles with a low importance factor and creates duplicates of particles with a high importance factor.

RESAMPLING STEP

Over time, the resampling step above reduces the variety of particles. This means that after a while, the set of particles X_t consists only of multiple copies of the same particle. A common extension is to replace some particles with randomly seeded particles to increase the variety of the particles. In many cases this also makes the PF more robust.

RANDOM PARTICLES

Another important extension is the so-called *sensor resetting*. Instead of adding a small number of random particles, particles are added by using the most recent measurements. Sensor resetting makes the PF more robust and leads to faster convergence.

SENSOR RESETTING

There are many more extensions of the basic PF which are not discussed here. See [29] for more on this topic.

In general, PF are considered to be very easy to implement and to be quite

```
1  def particle_filter(X_{t-1}, u_t, z_t):
2      \overline{X}_t = X_t = []
3
4      for particle in X_{t-1}:
5          newParticle = predict(particle, u_t)
6          newParticle.weight = p(z_t | newParticle)
7          \overline{X}_t.append(newParticle)
8
9      % resampling
10     X_t = resampleWithReplacement(\overline{X}_t)
11
12     return X_t
```

Listing 4.3: Algorithm of particle filter.

robust. However, they can be computationally expensive. PF have problems covering high dimensional state spaces. They can also be too slow when increasing the number of particles which is necessary in high dimensional state spaces. More specifically, the runtime increases exponentially with the number of dimensions. This is often referred to as the *curse of dimension.*

4.4 Kalman Filter

The *Kalman filter* (short *KF*) was developed by Swerling, Kalman and Bucy around 1960 [12]. The KF is a *Gaussian filter*. Recall that a Gaussian distribution is defined by the first two moments, the mean μ and covariance matrix Σ. GAUSSIAN FILTER

$$bel(x) = \det(2\pi\Sigma)^{-\frac{1}{2}} \exp\left\{-\tfrac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\} \tag{4.9}$$

The KF represents the belief distribution as a Gaussian distribution. The noise of the system and the noise of the measurements are assumed to be zero centered Gaussian noise. Because a Gaussian is used to represent the belief distribution, the KF can only model systems which can be expressed by a unimodal distribution. The systems which are modeled by KFs must be linear. This means that a *linear process model* and a *linear measurement model* must be able to describe the system. LINEAR PROCESS MODEL LINEAR MEASUREMENT MODEL

As a Gaussian filter, the KF is a *parametric filter*. Just by using a KF, one makes strong assumptions about the nature of the system. In practice it means that the KF is only a choice if the system is linear and has Gaussian properties. Gaussian linear systems are very rare in the real world. Nevertheless, the KF is *optimal* for linear systems with known white noise, a perfect process, and a perfect measurement model. Moreover, because of the restriction to normal distributions, the KF is quite *efficient* in terms of computational complexity and is therefore widely used. PARAMETRIC FILTER OPTIMAL EFFICIENT

4.4.1 The Kalman Filter Algorithm

The KF follows the prediction correction steps of the Bayes' filter very closely. In the prediction step, the control u_t changes the mean of the Gaussian, some noise is introduced, and the Gaussian is widened. In the measurement

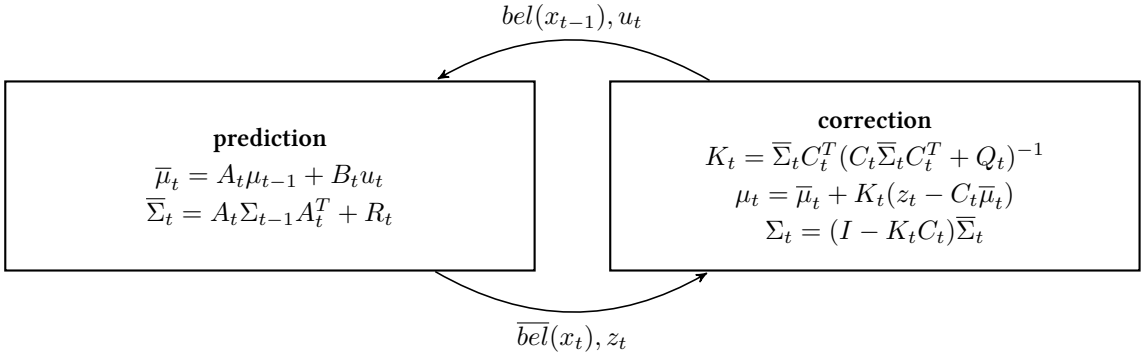


Figure 4.4: The Kalman filter algorithm.

step additional information in the form of a measurement z_t is introduced. The mean is adjusted accordingly and the Gaussian becomes narrower. The complete Kalman filter algorithm is depicted in listing 4.4 and figure 4.2.

The process model must be linear. Therefore, the prediction step can be expressed as:

$$x_t = A_t x_{t-1} + B_t u_t + \varepsilon_t \tag{4.10}$$

The predicted state x_t is the previous state x_{t-1} transformed through the transition matrix A plus the control u_t transformed through the transition matrix B_t. Additionally, some zero centered Gaussian noise ε_t is added which models the uncertainty introduced by the state transition.

The function of the transformation matrices A and B is not apparent at first glance. The state space allows a prediction of the state for the next time step, *e.g.,* if the state consists of the distance to an object (which is assumed to be static) the prediction would not change the state. This means that the transition matrix[1] A_t is an identity matrix. If the state consists of the position and the velocity of an object (in one dimension), a transformation from $t-1$ to t is necessary. The object moves from position p to position $p + v$ where p is the position and v is the velocity of the object. Therefore, A_t would be:

[1] Note that scalars are used in this case. But for the sake of the argument assume that 1×1 matrices are used.

$$A_t = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \tag{4.11}$$

A_t is always a $n \times n$ matrix where n is the dimension of the state vector.

B_t has a similar function to A_t. The transition matrix B_t transforms the control u_t into the state space. The matrix has the form $n \times m$ where m is the dimension of the control.

The measurement model must be linear and can be expressed as:

$$z_t = C_t x_t + \delta_t \tag{4.12}$$

C_t has a similar function as A_t and B_t. It is a transformation matrix with dimension $k \times n$ which transforms the state x_t into the measurement space with the dimension k. δt is again zero-centered Gaussian noise which represents the uncertainty in the measurement.

The KF algorithm is shown in listing 4.4. It uses the previous state x_{t-1}, the previous covariance matrix Σ_{t-1}, the current control u_t, and the current measurement z_t. Then, in the prediction and correction step the mean and the covariance are updated.

```
1  def kalman_filter(μ_{t-1}, Σ_{t-1}, z_t):
2
3      # prediction
4      μ̄_t = A_t μ_{t-1} + B_t u_t
5      Σ̄_t = A_t Σ_{t-1} A_t^T + R_t
6
7      # correction
8      z̄_t = C_t μ̄_t
9      v_t = z_t − z̄_t
10     S_t = C_t Σ̄_t C_t^T + Q_t
11     K_t = Σ̄_t C_t^T S_t^{-1}
12     μ_t = μ̄_t + K_t(v_t)
13     Σ_t = (I − K_t C_t) Σ̄_t
14
15     return μ_t, Σ_t
```

Listing 4.4: Algorithm of the Kalman filter.

In the *prediction step*, the $\overline{bel}(x_t)$ is calculated by incorporating the control u_t, and the related process noise R_t. The update for the mean $\overline{\mu}_t$ follows directly from equation (4.10): PREDICTION STEP

$$\overline{\mu}_t = A_t\mu_{t-1} + B_t u_t \tag{4.13}$$

The new covariance $\overline{\Sigma}_t$ is the sum of the previous covariance matrix Σ_{t-1} transformed by A_t and the noise R_t which is introduced by the control (line 5).

$$\overline{\Sigma}_t = A_t\Sigma_{t-1}A_t^T + R_t \tag{4.14}$$

In the *correction step*, the final belief $bel(x)$ is calculated by incorporating the measurement z_t into the previous belief $\overline{bel}(x_t)$. This takes the covariance matrix of the state Σ, the measurement z_t and its corresponding covariance matrix S_t into consideration. The predicted measurement given the state $\overline{\mu}_t$ is calculated with transition matrix C_t: CORRECTION STEP

$$\overline{z}_t = C_t\overline{\mu}_t \tag{4.15}$$

The *innovation* is the deviation between the real measurement z_t and the predicted measurement $\overline{z}_t$. It would be zero if z_t and $\overline{z}_t$ are the same. INNOVATION

$$v_t = z_t - \overline{z}_t \tag{4.16}$$

The covariance of the measurement S_t is calculated similarly to equation (4.14). The covariance $\overline{\Sigma}$ is transformed into the measurement space with transition matrix C_t, and the measurement noise Q_t is added:

$$S_t = C_t\overline{\Sigma}_t C_t^T + Q_t \tag{4.17}$$

The *Kalman gain* determines how much the measurement is incorporated. KALMAN GAIN

$$K_t = \overline{\Sigma}_t C_t^T S_t^{-1} \tag{4.18}$$

Finally, the Kalman gain is then used to adjust the mean $\mu_t$and the covariance matrix Σ_t.

$$\mu_t = \overline{\mu}_t + K_t(v_t) \tag{4.19}$$

$$\Sigma_t = (I - K_t C_t)\overline{\Sigma}_t \tag{4.20}$$

For a more thorough explanation of the KF see [29].

```
1  def extended_kalman_filter(μ_{t-1}, Σ_{t-1}, z_t):
2
3      # prediction
4      μ̄_t = g(u_t, μ_{t-1})
5      Σ̄_t = G_t Σ_{t-1} G_t^T + R_t
6
7      # correction
8      z̄_t = h_t(μ̄_t)
9      v_t = z_t − z̄_t
10     S_t = H_t Σ̄_t H_t^T + Q_t
11     K_t = Σ̄_t H_t^T S_t^{-1}
12     μ_t = μ̄_t + K_t(v_t)
13     Σ_t = (I − K_t H_t) Σ̄_t
14
15     return μ_t, Σ_t
```

Listing 4.5: Extended Kalman filter algorithm.

4.4.2 Extended Kalman Filter

One of the biggest limitations of the KF is that it cannot handle non-linear systems. The *extended Kalman filter* (short *EKF*) overcomes this limitation.

Instead of the linear transition matrices (A_t, B_t, and C_t) non-linear functions for the process model g and the measurement model h are used:

$$x_t = g(u_t, x_{t-1}) + \varepsilon_t \tag{4.21}$$

$$z_t = h(x_t) + \delta_t \tag{4.22}$$

The EKF uses the Taylor expansion as the linearization technique (see section 3.1.5) to approximate the non-linear functions g and h. The linearization of g and h at μ are the Jacobi matrices G and H.

Compared to the plain KF (see listing 4.4), the EKF algorithm (see listing 4.5) only needs a few adjustments. Namely, all occurrences of A_t are replaced by the Jacobian G_t (line 5), all occurrences of C_t are replaced by the Jacobian H_t (line 10-13) and the non-linear functions g and h are used instead of the transition matrices (line 4 and 8).

To implement an EKF one needs to define the process model g, the measurement model h and calculate the Jacobians G and H.

4.4.3 Unscented Kalman Filter

UNSCENTED KALMAN FILTER

The *unscented Kalman filter* (short *UKF*) is similar to the EKF in that it handles non-linearities by linearization. However, it does not apply the Taylor expansion to linearize the process and measurement model, but it reconstructs the new belief distribution with carefully chosen sample points. The sample points are called *sigma points* and the method to reconstruct the distribution is called *unscented transformation* (see the following section for more). Because the UKF does not use Jacobians, the UKF is often called the *derivative-free KF*.

SIGMA POINTS

UNSCENTED TRANSFORMATION

DERIVATIVE-FREE KF

The UKF has a few advantages over the EKF. In general, the applied linearization technique, the unscented transform, is superior to the Taylor expansion used in the EKF. This often leads to higher accuracy of the UKF.

Calculating the Jacobians of EKFs is an error prone procedure. Because the UKF does not use Jacobians, one can easily experiment with different processes and measurement models.

But there are also some disadvantages of the UKF compared to EKF. The UKF is slower than the EKF by a constant factor. It is also known to be harder to implement.

	KF	EKF	UKF
process	A, B	g, G	g, UT
measurement	C	h, H	h, UT

Table 4.1: Comparison of the Kalman filter, the extended Kalman filter and the unscented Kalman filter.

The Unscented Transform The unscented transform approximates the mean and the covariance matrix with weighted deterministic sample points, so-called *sigma points*. In general, there are $2n + 1$ weighted sigma points, where n is the dimension of the state space. Classical Monte Carlo sampling requires many more samples. There are two sigma points along each dimension and one sigma point at the mean. Let $\mathcal{X}^{[i]}$ denote the ith sigma point.

SIGMA POINTS

For a state space with dimension 3 the UT would use 7 sigma points to approximate the mean and the covariance.

$$\begin{aligned} \mathcal{X}^{[0]} &= \mu \\ \mathcal{X}^{[i]} &= \mu + C_i \qquad\qquad \text{for } i = 1, \ldots, n \\ \mathcal{X}^{[i]} &= \mu - C_{i-n} \qquad \text{for } i = n+1, \ldots, 2n \end{aligned} \tag{4.23}$$

with $C = \sqrt{(n+\lambda)\Sigma}$. C is a matrix square root[2]. C_i denotes the ith column of the matrix. Also, $\lambda = \alpha^2(n + \kappa) - n$ where α and κ are scaling parameters which determine the spread of the sigma points from the mean.

[2] A symmetric, positive definite matrix A can be decomposed as $A = LL^T$. This can be done with the *Cholesky decomposition*.

The weights of the sigma points are different for the covariance matrix and the mean. The weights to reconstruct the mean w_m^i (where i denotes the weight for the sigma point i) are calculated as follows:

$$\begin{aligned} w_m^{[0]} &= \frac{\lambda}{n+\lambda} \\ w_m^{[i]} &= \frac{1}{2(n+\lambda)} \qquad \text{for } i = 1, \ldots, 2n \end{aligned} \tag{4.24}$$

The weights to reconstruct the covariance matrix w_c^i (where i denotes the weight for the sigma point i) are calculated as follows:

$$\begin{aligned} w_c^{[0]} &= \frac{\lambda}{n+\lambda} + (1 - \alpha^2 + \beta) \\ w_c^{[i]} &= \frac{1}{2(n+\lambda)} \qquad\qquad \text{for } i = 1, \ldots, 2n \end{aligned} \tag{4.25}$$

$\beta = 2$ is the best choice for Gaussian distributions.

The sigma points are passed through the function f.

$$\mathcal{Y}^{[i]} = f\left(\mathcal{X}^{[i]}\right) \tag{4.26}$$

f can be any non-linear function. When using the UT in an UKF f is replaced by the non-linear process model g and the non-linear measurement model h.

Then the mean and the covariance are reconstructed from the weighted mean of the transformed sigma points $\mathcal{Y}$ as follows:

$$\overline{\mu} = \sum_{i=0}^{2n} w_m^{[i]} \mathcal{Y}^{[i]} \tag{4.27}$$

$$\overline{\Sigma} = \sum_{i=0}^{2n} w_c^{[i]} \left(\mathcal{Y}^{[i]} - \overline{\mu}\right)\left(\mathcal{Y}^{[i]} - \overline{\mu}\right)^T \tag{4.28}$$

4.5 Multi-Hypothesis Kalman Filter

KF are always unimodal. The *multi-hypothesis Kalman filter* (short *MHKF*) overcomes this limitation. MHKFs approximate arbitrary belief distributions as a *sum of Gaussians*. Gaussians are combined into a sum of Gaussians

MHKF

SUM OF GAUSSIANS

$$\sum_{i=1}^{N} w_i \, \mathcal{N}(\mu_i, \Sigma_i) \tag{4.29}$$

where N is the total number of Gaussians. By using more Gaussians, the accuracy of the approximation can be increased. Each Gaussian has its own mean μ_i and its own covariance matrix Σ_i. Additionally, the weight w_i determines the influence of each Gaussian towards the distribution. The weight of all Gaussians must sum up to 1:

$$\sum_{i=1}^{N} w_i = 1 \tag{4.30}$$

The weights are often chosen to represent measurement likelihood. However, the meaning of the weights can be adjusted according to the problem.

In general, there are quite a few decisions to make when implementing a MHKF:

- how many hypotheses should be in the distribution?
- should the number of hypotheses be constant?
- can the distribution be simplified by removing/merging hypotheses without losing information?
- should every hypothesis execute the measurement update even if the likelihood of the measurement is small?

All these questions are very problem specific and a general answer cannot be given.

MHKFs can use any variant of the KF (plain KF, EKF, or UKF). The MHKFs inherits the properties of the selected KF variant. In general, MHKFs have some similarities to PFs with an extremely low number of particles. They are more robust than single-hypothesis KFs and they experience fewer problems with data association. Linearization problems do not break the entire model but only one hypothesis. These advantages come at the cost of increased computational complexity (and also increased complexity of the implementation).

	PF	**KF**	**EKF, UKF**	**MHKF**
belief distribution	arbitrary	Gaussian	Gaussian	arbitrary*
process model	arbitrary	linear	non-linear*	**
process noise	arbitrary	Gaussian	Gaussian	Gaussian
measurement model	arbitrary	linear	non-linear*	**
measurement noise	arbitrary	Gaussian	Gaussian	Gaussian
high dimensional state	limited	yes	yes	yes
runtime	-	++	++	+
implementation	++	+	+	−

Table 4.2: Comparison of different Bayes' filters.

* means *approximated*
** means *depends on the chosen KF*

5
Related Work

The particle filter is a popular choice in all RoboCup leagues. It is easy to implement and proves to be quite robust. Increasing computing power remedies the disadvantage of higher runtime (compared to the Kalman filter). PFs are used by many of the top teams such as B-Human [23], Darmstadt Dribblers [14], and Team Nimbro [2]. KFs are used more for classical tracking tasks. Fox and Gutmann state in [9] that

> "Markov localization is more robust than Kalman filtering while the latter can be more accurate than the former."

This explains the popularity of particle filter localization in noisy environments such as the RoboCup.

However, in recent publications [22, 11, 28], KF and especially MHKF solutions have gained popularity. Here aspects of PFs are combined with the KFs which results in low runtime, high localization performance, and robustness. KFs are especially interesting when the dimension of the state space exceeds the usual low dimension of the typical localization task (robot position and rotation). This is especially true when incorporating other information like the ball and teammates into the state space [28, 26].

In the following sections, related work concerning the tracking task and the localization task are introduced.

5.1 Local Tracking

B-Human, multiple winner of the SPL league, uses KFs for ball tracking. Position and velocity of the ball are tracked [23]. Twelve independent KFs are used to represent the belief distribution. Prediction and correction are applied to all KFs in the same way. Six KFs are specialized for a rolling ball (reactive process model), and six KFs are specialized for a stationary ball (less reactive process model). The measurement and the covariance matrices are used to determine the quality of each KF. The worst KF is replaced by a new one, and the best KF is used to represent the actual ball.

The Darmstadt Dribblers also apply MHKF to model the ball's position and velocity in a similar fashion [14].

For tracking the position of other robots, B-Human uses UKFs [23]. No motion model for the other robots is assumed and the Mahalanobis distance and/or the Euclidean distance are used to solve the measurement-model correspondence.

The Darmstadt Dribblers follow a different approach and track obstacles on a dynamic occupancy grid map [14, 13]. This is quite uncommon for a team in the RoboCup soccer leagues.

5.2 Localization

B-Human uses a Monte Carlo method with several extensions for self-localization [23, 24, 15]. Random particles and sensor resetting are applied. B-Human filters possible robot poses from the PF localization with a KF and invalidates poses which do not meet certain criteria.

The Darmstadt Dribblers also use a PF localization with several extensions such as a particle maturing step and the Gaussian approximation of the particles to extract the final pose [14, 13].

Quinlan and Middleton [22] describe a multi-model extended Kalman filter approach for self-localization. The belief distribution is approximated with a mixture of Gaussians. The number of Gaussians is variable. The interesting part of their approach is the measurement update. First the unambiguous measurements (like goals) are used. Then, for each ambiguous measurement, each model is split into two models and the observation is only applied to one of the two models. False positives only have influence on one of the two models. Because the splitting step increases the number of models exponentially, similar models are merged afterwards.

In [27] a similar approach is followed. A Gaussian mixture with UKFs instead of EKFs is used. Furthermore, models are not split but the measurements are instead applied to the models according to the maximum likelihood principle. This means that models are only updated if the measurement likelihood is high. Also, new models are created depending on the measurements neglecting the current state. This method is comparable with sensor resetting of particle filters.

In [26] Multi-hypothesis EKF is used for the localization task.

5.3 Global Models and Combined World Models

A global model of all teammates is easy to implement. The robot pose of the localization and its uncertainty can be simply transmitted via WIFI to other robots because the pose is is already in a global coordinate frame. This, for example, is done by B-Human [23].

A common approach to create a global model from a local model is to translate the local models using the robot pose into the global coordinate system. The uncertainty of the localization and the local models can be combined to estimate the uncertainty of the tracked object in the global system. This way, information from other teammates can be incorporated easily. The approach is used to create a ball and/or an obstacle model. As an example, B-Human [23] follows this approach to create a global ball model. The global ball model is the weighted average of the ball information provided by the teammates. The weight depends on certain aspects such as the quality of the localization, the time since last ball percept, etc. The complete global world model in [23] tracks teammates, the ball, and other robots.

In [26] a MHKF is used for a combined world model in the 4-legged

league. The state space has 16 dimensions: the robot pose of the entire team ($4 * 3$ dim) and the ball (4 dim). In the measurement step, the models are split and the measurements are applied to the models similar to [22]. Observations from the teammates are also used. The weight is adjusted according to the quality of the match of the measurement. Similar models are merged and models with low weight are removed. Sushkov and Uther states that preventing false positives is of huge importance [26].

In [28] a single KF is used to model the world state which consists of the pose of all robots in the team, the ball, and the opposing team's robots.

6
Analysis

The goal of this chapter is to choose appropriate algorithms for the local ball model, the local obstacle model, and the self-localization. First, the properties of the FUmanoid robot platform, and the environment with its landmarks and percepts will be examined in section 6.1 and section 6.2. Then, the consequences of these properties for the individual models will be discussed in the sections following.

6.1 Properties of the Robot Platform

The robot platform largely influences the choice and implementation of the algorithms which model certain aspects of the robot and its environment. Therefore, the properties of the FUmanoid robot platform are explained here.

6.1.1 The Camera

In the RoboCup KidSize league the main sensor is the camera. As mentioned in section 2.2.1, the FUmanoid platform of 2012 does not use a camera with a super-wide-angle-lens anymore, but instead uses a camera with an opening of 46° vertical and 59° horizontal. As shown in figure 6.1 and figure 6.2, images taken with the new lens contain less information compared to the previously used super-wide-angle-lens. Because less features are detected per frame, occlusion of objects and false-positives have a more severe impact.

With the super-wide-angle-lens, a robot did not have to move its head to scan the area in front of it. The reduced field of view of the new camera makes it necessary for the robot to move its head. The two servos which are needed for the pitch and yaw movements of the head move relatively fast and the range of movements is large compared to most of the other servos in the robot[1]. Whereas the other servos are (mostly) needed for stable walking, the head servos have a large influence on the camera pose. The servos of the head introduce noise which reduces the quality of measurements.

6.1.2 Odometry

The *control* of the robot is essential for recursive state estimation. The odometry of the robot can be used as control for the filters.

Figure 6.1: A typical image with the previous super-wide-angle-lens.

Figure 6.2: A typical image with the current camera with a more traditional lens. The image it taken with the simulator. The robot is approximately at the same position as in figure 6.1.

[1] The pitch servo of the camera rotates around 90° and the yaw servo of the camera rotates around 200°.

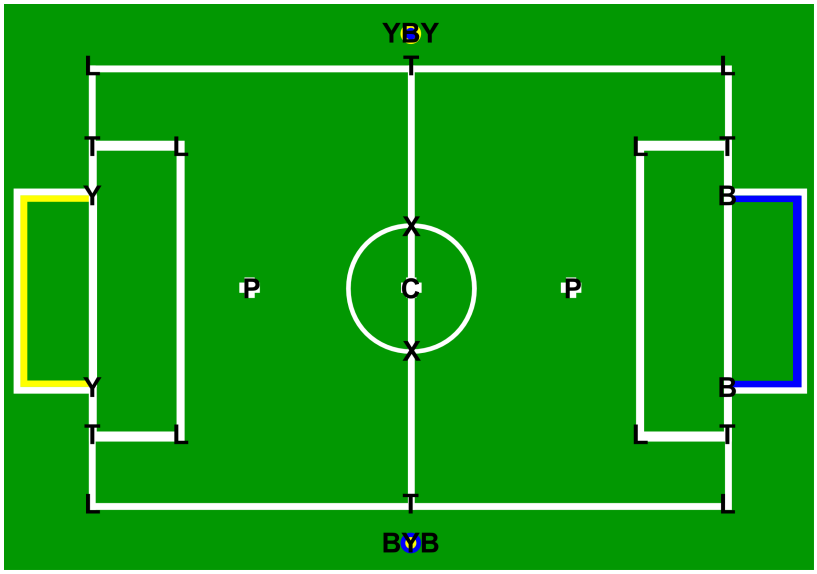


Figure 6.3: All static features of the field which can be represented as point measurement.
B – Blue goal post
Y – Yellow goal post
BYB – Side post
YBY – Side post
C – Center of the field
P – Penalty point
L – L-line crossing
T – T-line crossing
X – X-line crossing

The robots of 2012 use the pose of the feet to calculate the odometry. The end effectors of the walker are the two feet of the robot. The walker commands the end effectors to move in a certain way. This command is then used as odometry.

Another common approach is to use the forward kinematic to calculate the odometry. In every time step the joints are read and the pose of the end effectors is calculated. The pose of the end effectors using this approach should be more precise than the previous approach. However, all joints must be read for this, which puts a load on the communication bus to the servos and also increases the load on the CPU (mostly due to waiting for the read command). However, the difference between both approaches is marginal in the case of the FUmanoid robots.

The 2012 robots have huge odometry errors, mostly due to *slippage*. The degree of the error changes from robot to robot. Little adjustments on the neutral position of the servos have large effects on the robots' walking behavior and therefore also on the error.

SLIPPAGE
A robot which should walk a meter straight ahead often walks a quarter of a circle to one or the other side.

In general, the fact that the error of the odometry is different from robot to robot, that the error changes drastically when adjusting the neutral positions of the servos, and that the error highly depends on the properties of the ground (*i. e.,* the carpet of the field), makes it difficult to determine a good motion model.

6.2 *Properties of the Environment and the Landmarks*

As mentioned in section 2.1 the environment of the kid-size league is well defined. The color-coded environment makes it easier for the vision system to detect certain features. All *percepts* which are important for the modeling tasks will be introduced here. Then, some problematic properties of the percepts will be discussed.

PERCEPTS

Most of the features can be represented with a *point measurement*, *i. e.,* a relative coordinate that fully describes the measurement. When observing

POINT MEASUREMENT

a feature, the agent can be located on a circle around the feature with the given rotation to the feature, see figure 6.4.

Additionally, some features also have an orientation which further restricts possible poses (see figure 6.5).

The following observations are *unique* and can be represented by point-measurements:

Side poles: The two side poles (blue-yellow-blue and blue-yellow-blue) are positioned on the left and right side of the field at the middle line. The module SidePoleExtractor provides the representation SidePoles.

Goal posts: There are two goals on the field: one yellow and one blue. A goal consists of two poles, one left one right, and the crossbar, which are all colored. The net of the goal is gray or white. If the two posts are visible, or one post and the crossbar are visible, the module GoalExtractor provides the unique percepts GoalPostLeft and/or GoalPostRight. If only one post is visible, the GoalExtractor provides the ambiguous percept GoalPostUnknown .

In general, robots orient themselves towards the opposing goal. Therefore, the goals are often observed and they are one of the strongest indicators for the localization. However, when a robot is near the goal the robot perceives only one post at a time (and rarely the crossbar). The robot has to make a correspondence choice: is the GoalPostUnknown a left goal post or a right one?

Field line circle: The module FieldLineExtractor provides the representation FieldLineCircle . The FieldLineExtractor uses curved field lines to calculate the position of the center of the circle. The FieldLineCircle is also calculated when only a part of the actual circle is visible. However, this leads to a noisier measurement of the center of the circle compared to the rest of the features.

In addition to the unique landmarks, there are also ambiguous features in the environment:

Penalty points: There are two penalty points on the field. The module PenaltyPointExtractor provides the representation PenaltyPoint. Even though the two penalty points are not unique, the two points are sufficiently far apart to distinguish them easily/make a correspondence choice.

Field line crossings: The module FieldLineFeatureExtractor uses the representation FieldLines to provide the representation FieldLineCrossings. One can distinguish between three kinds of line crossings: *X-*, *T-* and *L-crossings*. There are two X-crossings, six T-crossings, and eight L-crossings.

One problem with the crossings is the frequent misclassification, *e.g.*, X-crossings are classified as "smaller" crossings (T- or L-crossings) when parts of the field line are obstructed or the line that forms a crossing is simply not recognized properly. Also, sometimes crossings are classified as "bigger" crossings. The less ambiguous a crossing, the more information it yields for the localization task. When crossings are misclassified the ambiguity of a crossing increases and the usefulness decreases.

A crossing landmark does not only have a position but also an orientation. T- and L-crossings have a unique orientation which leads to one possible robot pose when observing the feature, X-crossings have four

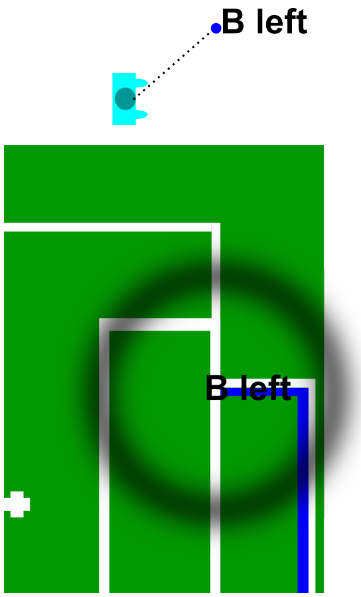


Figure 6.4: Illustration of a point measurement. A robot observes a feature (top). It can be located at any point on the circle around the landmark (bottom).

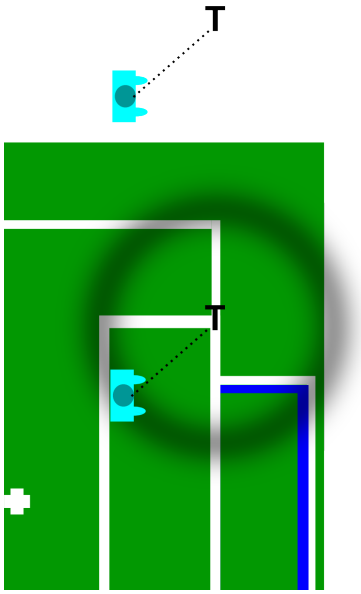


Figure 6.5: Illustration of a measurement (position and orientation). A robot observes a feature (top). It can be located only at one position on the circle (bottom).

indistinguishable orientations which leads to four possible robot poses. However, the current implementation of the FieldLineFeatureExtractor does not provide the direction of the crossings. Thus, simple point measurements are used for all crossings.

Field lines: Field lines are probably the most general landmarks in the environment. Because of the amount and the size of field lines on the field, they are the most observed feature. The advantage of field lines is also their disadvantage. It is hard to find a correspondence for an observed line. For the most part, only a part of a field line is observed which results in an infinite number of robot poses.
The representation FieldLines is provided by the FieldLineExtractor . The representation consists of a list of lines. Each line has a start point, an end point, and points in-between.

DYNAMIC ELEMENTS

In addition to the static features in the RoboCup KidSize environment, there are *dynamic elements*. The number of dynamic elements however is relatively small.

Obstacles: Obstacles are objects on the field which need to be avoided when the robot is walking. This includes opposing robots, teammates, and other objects like the referee. The module ObstacleExtractor recognizes *non-field regions* on the field as obstacles. Parts of the field are classified as non-field regions when there is not enough light or there is a lot of shadow, *e.g.,* when the green field is too dark to be classified as field. Theses regions are often very small. The number of recognized obstacles from the ObstacleExtractor is arbitrary.

Ball: The BallPercept representation is provided by the module BallExtractor . The ball is the fastest moving object on the field. Sometimes there are false-positives: the hands of the referee or the hands of the robot handler are recognized as the ball, the yellow goal posts or the magenta team marker are recognized as the ball. Because of the nature of soccer, false-positives of the ball can have the most serious consequences.

POLE MISCLASSIFICATION

Additionally, there are some problems with side poles and goal posts. Sometimes the side poles are classified as goal posts. This happens if parts of a side pole are occluded or if the colors blue and yellow are not recognized properly because of changing/bad lighting conditions.

All measurements of all features, except the field lines, can be represented as a point measurement. The following section discusses some properties of different representations.

6.3 Representations of Measurements and Linearization Techniques

REPRESENTATIONS OF MEASUREMENTS

There are several coordinate systems available to represent point measurements, *i.e.,* the positions of observed features. More common representations are the *Cartesian coordinate system*, and the *polar coordinate system*. The *spherical coordinate system* is used less often. Figure 6.6 shows the different representations.

CARTESIAN COORDINATE SYSTEM

When using the *Cartesian coordinate system* to represent measurements, the robot is the center of the coordinate system and the distance in x and y

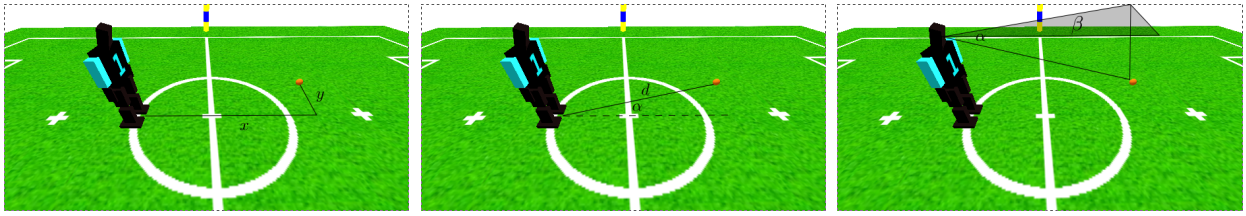


Figure 6.6: From left to right: Cartesian coordinates, polar coordinates and spherical coordinates.

direction of the observed feature are measured:

$$z = (z_x, z_y)^T \tag{6.1}$$

POLAR COORDINATE SYSTEM

The *polar coordinate system* is often used by people who work with laser range finders. A measurement is represented by a distance (the range) and the direction (the bearing) of the observed feature:

$$z = (z_{\text{range}}, z_{\text{bearing}})^T \tag{6.2}$$

SPHERICAL COORDINATE SYSTEM

A less intuitive representation is the *spherical coordinate system*. The measurement is represented by two angles from the point of view of the camera: a vertical angle and a horizontal angle. This representation is very similar to human perception.

$$z = (z_{pitch}, z_{yaw})^T \tag{6.3}$$

The common literature (Thrun et al. [29]) states that the choice of the linearization technique, unscented transform vs. Taylor expansion, has a large influence on the quality of the model. But as suggested in [27], different representations also influence the overall performance of the model. In fact, the choice of the coordinate system for the measurement model is more important than the choice of the linearization technique as shown in figure 6.7.

In the comparison of Tasse et al. in [27], the implementation of the self-localization which uses spherical coordinates outperforms the implementation which uses other coordinate systems for the measurement model. Furthermore, the choice of the linearization technique has less influence than the choice of the coordinate system. When the spherical coordinate

	unique	**static**	**point measurement**
side poles	yes	yes	yes
goals (complete)	yes	yes	yes
goals (part)	no (×2)	yes	yes
field line circle	yes	yes	yes
penalty points	no (×2)	yes	yes
line crossing	no (total ×16)	yes	no*
X-crossing	×2		
T-crossing	×6		
L-crossing	×8		
ball	yes	no	yes
obstacles	no (arbitrary)	no	yes
field lines	no (arbitrary)	no	no

Table 6.1: Summary of the properties of all features in the environment.
Landmarks marked with * are treated as point measurements.

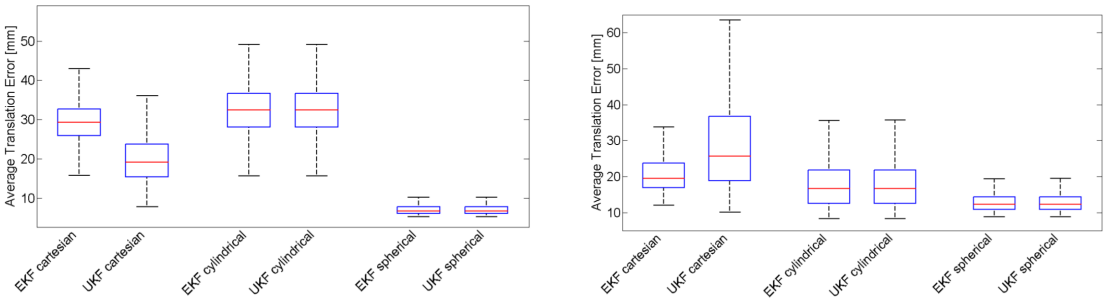


Figure 6.7: Left: "Comparison between localization quality using different linearization approaches and sensor model coordinate systems with simulated perceptions." taken from Tasse et al. [27].
Right: "Comparison between localization quality using different linearization approaches and sensor model coordinate systems with real observations recorded on a Nao." taken from Tasse et al. [27].

system is used, the applied linearization technique has no influence on the performance of the localization.

The reason for the superior performance of the spherical coordinate system might be as follows: small changes of the pitch angle of the camera lead to large changes in the distance of a measured object which increases the further away the object is. However, when using a KF, the noise is supposed to be Gaussian. When using the Cartesian coordinate system for the measurements, one would expect a noise distribution with a long tail rather than a Gaussian distribution. As a reminder, the KF is designed to deal with linear Gaussian systems. This includes Gaussian noise.

The spherical coordinate system has a few interesting properties regarding the problem of representing measurement noise. Figure 6.8 shows how measurements in spherical coordinates with Gaussian noise translate into the relative coordinate system. Only the pitch angle is shown, yaw is neglected for the sake of simplicity. If the robot looks straight down, the angle is $\frac{\pi}{2}$ rad; if the robot looks straight ahead to the horizon, the angle is 0 rad. On the left of figure 6.8, one can see the noise of measurements in spherical coordinates. The noise is normally distributed ($\sigma = 0.04$) as required by the KF. Translating the Gaussian nose from the spherical coordinates (left side of the figure) into the Cartesian coordinate system (right side of the figure) creates new distributions. The resulting distributions are not normalized. This process is shown for four different measurements: the measurements are at a distance of 1 m or 1.55 rad, 2 m or 1.53 rad, 3 m or 1.51 rad, and 4 m or 1.49 rad. Intuitively, the resulting distributions in the Cartesian coordinate system better represent the expected noise model.

In general, one can say that the spherical coordinate system better captures the noise of measurements. Because of this advantage, the spherical coordinate system is used exclusively in all implementations of the local models (see chapter 8) and the self-localization (see chapter 9).

6.4 Consequences for the Local Models

When only the relationship between the robot and certain objects in the world are of interest, local models are fully sufficient and under certain circumstances are even better than global models [18].

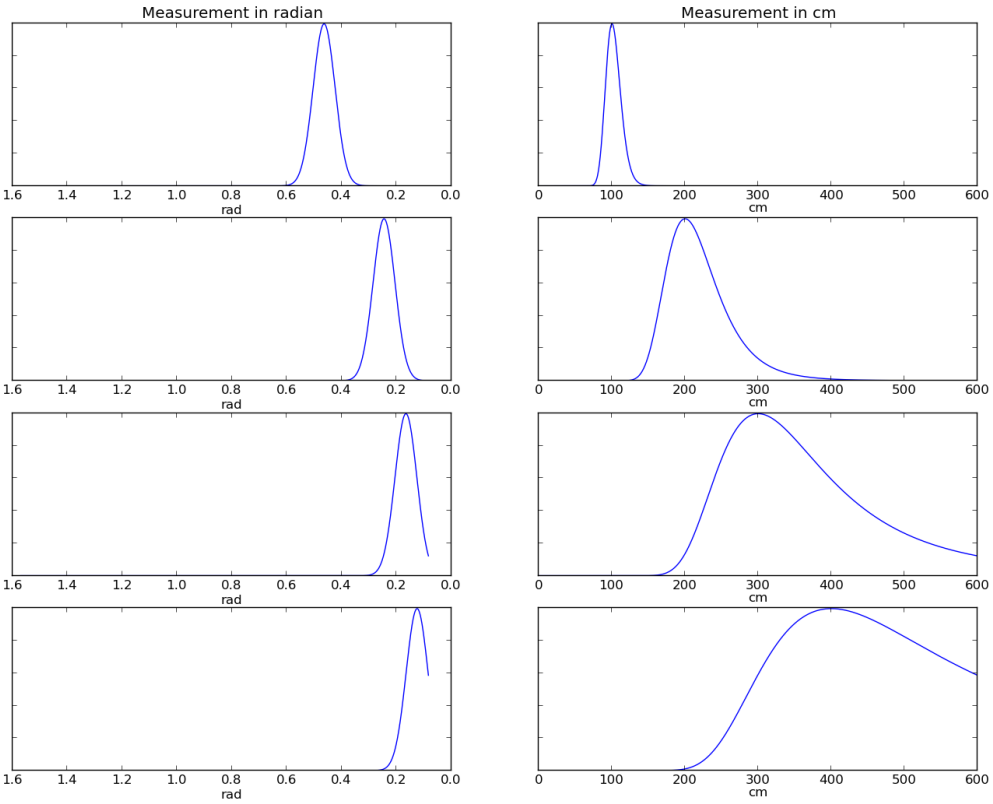


Figure 6.8: Transforming a Gaussian from the Cartesian coordinate system into the spherical coordinate system (not normalized).

6.4.1 *The Local Ball Model*

The ball follows the physical laws of motion. Friction decreases the speed of the ball. The speed of the ball can be quite high and lead to large positional changes. Hence, modeling the *speed* of the ball in addition to its *position* is a better representation of the actual ball and increases the accuracy of the model. Changes of direction caused by collision with other robots, the referee, or other objects, are not modeled explicitly.

STATE SPACE

In general, an EKF/UKF is a good choice for tracking problems such as tracking the ball. However, frequent false-positives can be problematic. This is especially the case when the covariance is very small and the false-positive is observed at a completely different position (which is normally the case for false-positives). This can completely derail the KF.

FALSE-POSITIVES

To handle false-positives, a *multi hypotheses tracking* approach will be used. The observations will only be used by a hypothesis if the likelihood for the measurement is high. Otherwise a new hypothesis will be created. As a result of this, false-positives spawn new hypotheses but do not destroy the other hypotheses. Hypotheses which do not receive enough confirmation in the form of measurements die out.

HANDLING FALSE-POSITIVES

The implementation of the ball model is described in section 8.1.

6.4.2 Consequences for the Obstacles Model

The obstacle model is used by the behavior engine to avoid collisions with objects on the field, *i.e.,* opponents, teammates, the referee, and the robot handlers.

In contrast to the ball, one cannot determine a simple rule for the motion of the obstacles. One could assume the tendency of robots to move in the same direction from which they came, but they could change their direction at any moment. Furthermore, the ObstacleExtractor often provides false-positives, *i.e.,* observes obstacles where there are none. Also, the form of the recognized obstacles in the image is not very accurate and changes from frame to frame. The referee and the two robot handlers also act differently than the robots. These characteristics make it difficult to derive a sophisticated process model. Therefore, using a *state space* which models more than the position of the obstacles does not make sense.

Keep in mind that obstacles can be robots as well as the referee and the robot handlers.

STATE SPACE

In an ideal case there are six robots on the field, plus maybe a referee and up to two robot handlers. The maximum number of obstacles could be used to improve the modeling. In practice, the ObstacleExtractor provides many obstacle percepts. A *multi hypotheses tracking* approach will be used to allow an arbitrary number of obstacles to be modeled. A percept is only used by the most likely hypothesis. If the percept does not match any hypotheses a new hypothesis is created.

NUMBER OF OBSERVATIONS
MULTI HYPOTHESES TRACKING

Additionally, to filter out the high number of false-positives, a *percept history* is introduced which invalidates percepts which were only observed for a very short amount of time.

PERCEPT HISTORY

The implementation of the local obstacle model is described in section 8.2.

6.5 Consequences for the Self-Localization

The self-localization is by far the most complex system. The self-localization is implemented using *multi hypotheses tracking* to increase the robustness of the self-localization against false-positives and wrong correspondence choices. In order to solve the kidnapped robot problem an approach very similar to the sensor resetting is applied.

MULTI HYPOTHESES TRACKING

The following *static landmarks* can be directly used as point measurements for the measurement update:

STATIC LANDMARKS

- the side poles,
- the goals,
- the field line circle,
- the penalty points, and
- the line crossings (X-, T-and L-crossings).

Because the observations are of the same type, *i.e.,* point measurements in the form of spherical coordinates, only one measurement model must be implemented.

To use the *field lines as landmarks* without the need to introduce another measurement model a trick is applied. A graph matching/constraint based algorithm which is independent from the current states is applied to find *measurement-landmark correspondences.* The algorithm calculates for the

FIELD LINES AS LANDMARKS

start and the end point of each line observation corresponding points on the field lines. The details are explained in section 9.4.

Histogram filters are used to reduce the negative effects of the *pole misclassification* and the *crossing misclassification* (see section 9.2 and section 9.2).

HISTOGRAM FILTERS

POLE MISCLASSIFICATION

CROSSING MISCLASSIFICATION

7
Implementation: Prerequisites

In order to ease implementation, debugging, and evaluation of the different models, the FUmanoids framework was extended. This section will explain the changes.

7.1 Log Recorder and Player

Data analysis and debugging is an integral part of developing complex systems. This is especially true for systems that produce and evaluate lots of data in a short span of time, such as a localization. The current FUmanoids framework already distinguishes between modules (algorithms) and representation (data). The representations in the module chain contain (almost) all the information at any given time.

The goal of the log recorder and log player is to be able to to save all relevant information and to replay this information in order to evaluate the performance of algorithms. The user should be able to change the algorithm, use the recorded representation as input for the algorithm, and see the results immediately. This allows a repeatable offline analysis of the algorithm. The data could then be recorded with the simulator or with the real robot. Also, with such a system, one has the potential to automatically analyse the performance of modules over time. A continuous interaction system like Jenkins[1] could run modules with prerecorded log files every night to properly quantify the latest changes.

[1] `http://jenkins-ci.org/`

RECORDING

Because all information is already in the module chain, more precisely in the representations, recording the data is relatively straightforward: all representations must be serialized at each time step. The serialized data must be written to file for later use.

PROTOBUF

In the FUmanoid framework, Google's *protobuf* is used for most tasks which involve exchanging and storing data. It has proven itself in this domain and is therefore used to serialize the representations.

"Protocol Buffers are a way of encoding structured data in an efficient yet extensible format. Google uses Protocol Buffers for almost all of its internal RPC protocols and file formats." Taken from [8].

A protobuf message LogFrame is defined. It contains all representations. Each representation implements a serializer which puts all data of the representation in the LogFrame and a deserializer that populates the representation when given a LogFrame.

The actual process of serialization is integrated into the cognition module. At the end of a cognition cycle the serializers of the representations are executed. Executing the serialization at the end of the cycle assures that all modules were executed and that the representations are filled.

Because protobuf messages must be read completely and log files can quickly become large, the LogFrames are not saved as a collection of protobuf messages. Instead single LogFrames are saved in a file with the following file format:

```
framenumber                 uint32   4bit
size of protobuf message    uint32   4bit
protobuf message
```

The implementation of the *playback* of a log file is straightforward as well. The user can specify the modules which are to be analyzed per command line. All modules are deactivated. The modules which are to be analyzed are reactivated again. PLAYBACK

The following steps are repeated for each frame: a key press triggers the execution of the next frame. The log file is read and all representations are deserialized with their according deserializer. Then, all active modules are executed. They have access to the required representations and overwrite the representation they provide.

The provided representation can be analyzed by the user. Because LogPlayer is completely transparent and does not change anything within the system, the debugging framework and FUremote works as expected.

7.2 Linear Algebra Library

Like many frameworks, the FUmanoid framework grew and evolved over time. This is especially problematic, because parts of the code show their history, *i. e.,* early versions of the FUmanoid framework were running on a micro processor and the hardware platforms did not offer a floating point unit (PFU). Therefore, there were many different math and linear algebra libraries used in the system such as:

- a self written fixed point math library,
- a self written matrix library,
- a self written matrix library with fix point arithmetic,
- *OpenCV*[5] math functionality is used, and
- some routines were taken from other frameworks and adjusted to the requirements.

During the course of this thesis, the old math libraries were removed, possible replacements were evaluated and finally, a new linear algebra library was introduced. Special emphasis was put on ease of use[2], the overall performance, the availability of certain matrix decompositions (like singular value decomposition (short SVD) and Cholesky decomposition), and the quality of the documentation.

[2] *MATLAB*[16]-like syntax and semantic are preferred because most students are familiar with MATLAB.

The following libraries were evaluated:

- *GNU Scientific Library* (GSL) [6],
- *OpenCV2 math* [5],
- *Armadillo* [25], and
- *Eigen* [1].

GSL has an unintuitive syntax and therefore was not further evaluated. The syntax and semantic of Eigen and Armadillo is very close to MATLAB

code. The speed of most of the common matrix operations is about the same for OpenCV, Armadillo and Eigen. Depending on the operation, OpenCV and Armadillo are a bit faster than Eigen. The common matrix decompositions are available in all libraries. Armadillo has a very good documentation and is easy to integrate. Overall, Armadillo was found to be the best linear algebra library and therefore it was integrated into the FUmanoid system.

7.3 Generic UKF Implementation

A generic UKF (see section 4.4.3) including the unscented transform (see section 4.4.3) was implemented in the class GenericUKF. Because the UKF is a derivative free KF, the user only needs to specify the process model f and the measurement model g to implement an UKF. Furthermore, the noise matrices Q and R must be provided by the user.

Using an UKF makes it easy to experiment with different measurement and process models because one does not need to derive the Jacobians. After conducting the experiments with different measurement and process models, one can switch to the computationally more efficient EKF if needed.

```
1 ukf.predict(g(control), getQ())
2 ukf.correct(h(measurement), getR())
```

Listing 7.1: How to use the GenericUKF.

7.4 Generic EKF Implementation

A generic class for all EKF implementation was written: GenericEKF. The class implements the basic math which is common to all EKFs. To realize a concrete EKF implementation the user needs to implement the process and the measurement model. This means implementing g, its Jacobi matrix G, h and its Jacobi matrix H. Furthermore, the noise matrices Q and R must be provided.

```
1 ekf.predict(g(control), getG(), getQ())
2 ekf.correct(h(measurement), getH(), getR())
```

Listing 7.2: How to use the GenericEKF.

7.5 Gaussian Tools

To ease the discussion of Gaussian distributions in the following chapters, similarity metrics and merging algorithms are introduced here.

If the number of Gaussians in a Gaussians mixture is too big or the distribution is simple enough to approximate it with fewer Gaussians, one can try to merge similar Gaussians in a way that the overall properties of the distribution do not change or change only minimally. There are two questions that need to be answered to achieve this:

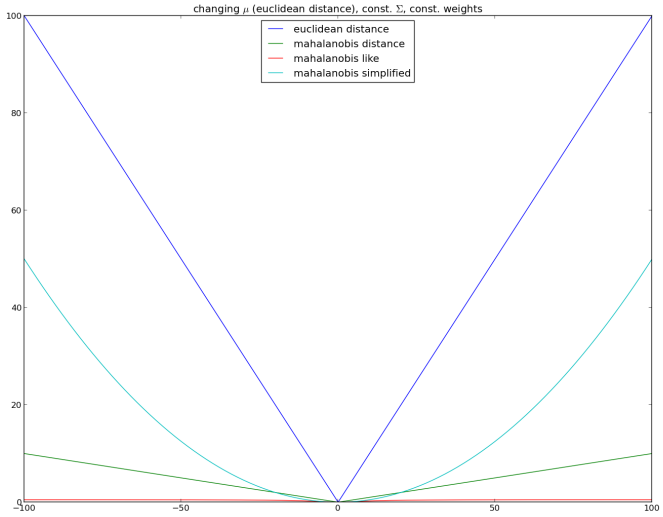


Figure 7.1: Different distance metrics. The euclidean distance of the two Gaussians changes, the covariance matrics and weights are constant.

- What is similar? (see section 7.5.1)
- How are Gaussians of a Gaussian Mixture merged without losing (too much) information? (see section 7.5.2)

The following distance metrics and merge algorithms were implemented.

7.5.1 Distance Metrics

Euclidean Distance If the state space is the Euclidean space, *e.g.,* when tracking an object in Euclidean coordinates, the simplest distance metric to compare two Gaussians i and j is the euclidean distance:

$$d_{ij} = |\mu_i - \mu_j| \tag{7.1}$$

The closer the two means, the more similar the two distributions are. The obvious disadvantage is that the covariance matrices, which are an essential part of Gaussians, and the weights, which are an essential part of sum of Gaussians, are neglected. Nevertheless, this is a very easy and computationally inexpensive method to compare the distance of two Gaussians. Also, if there is a huge number of Gaussians one can use this (cheap) metric to filter out highly dissimilar Gaussians and then use more sophisticated metrics for the rest.

The Mahalanobis Distance The *Mahalanobis distance* is a measure which estimates the similarity of an unknown sample x to a known distribution given by the mean μ and the covariance matrix S.

$$D_M(x) = \sqrt{(x-\mu)S^{-1}(x-\mu)} \tag{7.2}$$

It can also be used to compute the similarity of two random vectors of the same distribution. This makes it possible to apply the Mahalanobis distance as a measure of similarity of two Gaussian distributions:

$$d(\mu_1, \mu_2) = \sqrt{(\mu_1 - \mu_2)\Sigma ij^{-1}(\mu_1 - \mu_2)} \tag{7.3}$$

where Σ_{ij} is the covariance matrix of the two joined distributions and μ_i and μ_j are the means of the two distributions. A disadvantage of this metric is that the weights are still neglected.

Mahalanobis-like Distance Williams [30] proposes a Mahalanobis-like distance metric to compare Gaussians from a sum of Gaussians. The method uses the mean, the covariance matrix and the weight of each distribution.

$$d_{ij} = \frac{w_i w_j}{w_i + w_j}(\mu_i - \mu_j)^T \Sigma_{ij}^{-1}(\mu_i - \mu_j) \tag{7.4}$$

where i and j are the indices of the Gaussians, w the weight of the distributions, and Σ_{ij} is the covariance of the joined distributions.

$$\Sigma_{ij} = \frac{w_i}{w_i + w_j}\Sigma_i + \frac{w_j}{w_i + w_j}\Sigma_j + \frac{2w_i w_j}{w_i + w_j}((\mu_i - \mu_j)(\mu_i - \mu_j)^T) \tag{7.5}$$

Simplified Mahalanobis-like Distance Quinlan and Middleton [22] propose a simplified version of this metric to avoid the matrix inverse in Equation (7.4) which is computationally expensive if the state space is higher. They approximate the distance metric (7.4) with:

$$D_{ij} = (w_i + w_j)(\mu_i + \mu_j)^T (w_i \operatorname{diag}(\Sigma_i) + w_j \operatorname{diag}(\Sigma_j))^{-1}(\mu_i + \mu_j) \tag{7.6}$$

Note that the matrix inverse in this equation is the matrix inverse of a diagonal matrix which is computationally cheap.

7.5.2 Merge Distributions

The goal when merging two Gaussians[3] is to keep the properties of the original distribution or to lose as little information as possible. Williams [30] formulates this problem as an optimization problem and proposes several possible solutions. However, most of his suggested algorithms would be computationally too expensive.

[3] Of course there might be better algorithms that do not simply merge two Gaussians but more. However, due to performance restrictions and to keep the algorithms simple, only algorithms which merge two Gaussians are introduced and implemented.

Simple Merge Algorithm A very simple and computationally very inexpensive merge algorithm which mostly neglects the covariance is the following. The new weight is the sum of the two weights:

$$w_{new} = w_1 + w_2 \tag{7.7}$$

The new mean is the weighted average of the two means of the Gaussians:

$$\mu_{new} = \frac{1}{w_{new}}(w_1\mu_1 + w_2\mu_2) \tag{7.8}$$

The new covariance matrix simply uses the covariance matrix of the Gaussian with the higher weight.

$$\mu_{new} = \begin{cases} \Sigma_1 & \text{if } w_1 > w_2 \\ \Sigma_2 & \text{if } w_1 \leq w_2 \end{cases} \tag{7.9}$$

This algorithm does not keep the properties of the overall distribution because it simply neglects the properties of the covariance matrices.

Merge According to Williams A more sophisticated approach is presented in [30]. The weight of the new Gaussian should be equal to the sum of the two weights:

$$w_{new} = w_1 + w_2 \tag{7.10}$$

The new mean is the weighted average of the two means

$$\mu_{new} = \frac{1}{w_{new}}(w_1\mu_1 + w_2\mu_2) \tag{7.11}$$

The new covariance is given by

$$\Sigma_{new} = \frac{1}{w_{new}}\left\{w_1\Sigma_1 + w_2\Sigma_2 + \frac{w_1 w_j}{w_{new}}(\mu_1 - \mu_2)(\mu_1 - \mu_2)^T\right\} \tag{7.12}$$

Merge According to Quinlan and Middleton Quinlan and Middleton [22] propose a similar version to the merge algorithm from Williams. They try to avoid drifting of the mean when the weights are sufficiently different. The weight is calculated with equation (7.10). The weighted average of the means as in equation (7.11) is only used if the two weights are not too far apart. Otherwise, Quinlan and Middleton use the mean of the Gaussians with the higher weight:

$$\mu_{new} = \begin{cases} \mu_1 & \text{if } w_1 > 10 \cdot w_2 \\ \mu_2 & \text{if } w_2 > 10 \cdot w_1 \\ \frac{1}{w_{new}}(w_1\mu_1 + w_2\mu_2) & \text{otherwise} \end{cases} \tag{7.13}$$

The new covariance is given by:

$$\Sigma_{new} = \frac{w_1}{w_{new}}(\Sigma_1 + (\mu_1 - \mu_{new})(\mu_1 - \mu_{new})^T) + \tag{7.14}$$

$$\frac{w_2}{w_{new}}(\Sigma_2 + (\mu_2 - \mu_{new})(\mu_2 - \mu_{new})^T) \tag{7.15}$$

7.6 Notation

Some notations and abbreviations which will be used in the following chapters are introduced here.

Let $\Omega(\theta)$ denote the *rotation matrix* around θ: ROTATION MATRIX

$$\Omega(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \tag{7.16}$$

The control u_t of the robot is given by the *odometry*. More precisely, u_t is the translational and rotational difference between the odometry at timestep t and $t-1$: ODOMETRY

$$u_t = \begin{pmatrix} u_x \\ u_y \\ u_\theta \end{pmatrix} \tag{7.17}$$

The odometry is available at every time step.

As mentioned section 6.3, let z denote a *measurement* of an observed feature in spherical coordinates: MEASUREMENT

$$z = \begin{pmatrix} z_{pitch} \\ z_{yaw} \end{pmatrix} \tag{7.18}$$

The number of measurements is problem specific. There might be only one measurement (*e.g.,* the ball), multiple measurements (*e.g.,* different landmarks), or none is nothing was observed.

8

Implementation: The Local Models

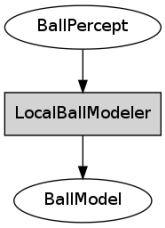


Figure 8.1: Local ball model architecture.

The sections for the local ball and the local obstacle model have a similar structure. First, the general problem is summarized, followed by the explanation of the process and the measurement model, then the multi-hypothesis management and task specific adjustments are illustrated.

All local models were implemented using the GenericEKF (and the local ball model was prototyped with the GenericUKF).

General Remarks The process and the measurement models of the local models closely follow the process and measurement models described in [28].

The local models are only updated when the robot is in a stable position, *i.e.,* it is standing or walking. The models are reset when a robot falls.

Note that the subscripts for the time index have been left out in the following sections for the sake of readability.

8.1 The Local Ball Model

The LocalBallModel is implemented using the GenericEKF described in section 7.4. The process model including g, G, and Q and the measurement model including h, H, and R need to be provided for the GenericEKF. For the mathematical model of the ball, a few terms need to be defined. The *state space* for the local ball model is defined as

EKF

STATE SPACE

$$\mu = \begin{pmatrix} p_x \\ p_y \\ v_x \\ v_y \end{pmatrix} \tag{8.1}$$

where p_x, p_y are the relative Cartesian coordinates of the ball, and v_x and v_y are the velocity of the ball in x and y direction respectively. p denotes the *positional part* of the state μ

POSITIONAL PART

$$p = \begin{pmatrix} p_x \\ p_y \end{pmatrix} \tag{8.2}$$

and v denotes the *velocity part* of the state μ

VELOCITY PART

$$v = \begin{pmatrix} v_x \\ v_y \end{pmatrix} \tag{8.3}$$

8.1.1 Process Model

This section will describe the process model. First, the influence of the velocity on the positional part of the state will be explained. Then, the change of the velocity will be explained. Finally, the full process model which combines the previous aspects will be shown.

INFLUENCE ON THE POSITION

The state μ of the ball can be predicted using the odometry of the robot and the velocity of the ball. Using only the velocity of the ball, the following equation predicts the position of the ball

$$p_t = p_{t-1} + v_{t-1}\Delta t \tag{8.4}$$

When incorporating the odometry, one has to rotate the state around u_θ and add the translational part of the odometry. This yields the new position of the ball

$$p_t = \Omega(-u_\theta)\, p_{t-1} + \Delta t\, \Omega(-u_\theta)\, v_{t-1} + (u_x, u_y)^T \tag{8.5}$$

INFLUENCE ON THE VELOCITY

The velocity part of the state is influenced in a different manner. As mentioned, the ball is governed by the laws of motion. Let $k = [\frac{m}{s^2}]$ denote the friction which slows down the ball. The velocity without the odometry is simply the old velocity reduced by the friction k. When incorporating the odometry, one also has to rotate the velocity vector around u_θ. Furthermore, one has to take into consideration that the friction stops the ball eventually, but it cannot lead to a negative ball speed. This makes is necessary to handle two cases: one for the rolling ball, and one for the stationary ball, *i.e.,* a ball with a very small velocity $|v| \geq |k\ \Delta t|$. Taking this into consideration, this results in the following equation for the prediction of the velocity:

$$v_t = \begin{cases} \overbrace{\left(1 + \dfrac{k\ \Delta t}{|v_{t-1}|}\right) \Omega(-u_\theta)}^{:=V} & v_{t-1} \quad \text{if } |v_{t-1}| \geq |k\ \Delta t| \\ 0_{2\times 2} & v_{t-1} \quad \text{otherwise} \end{cases} \tag{8.6}$$

Note that the matrix V will be used later.

Combining equations (8.5) and (8.6), the prediction of the new full state $\overline{\mu_t}$ is computed:

$$\overline{\mu_t} = g(\mu_{t-1}, u) = \begin{pmatrix} \Omega(-u_\theta) & \Delta t\, \Omega(-u_\theta) \\ 0_{2\times 2} & V \end{pmatrix} \mu_{t-1} - \begin{pmatrix} u_x \\ u_y \\ 0 \\ 0 \end{pmatrix} \tag{8.7}$$

Given $g(\mu_{t-1}, u)$, the Jacobian G can be computed as follows:

$$G = \begin{pmatrix} \Omega(-u_\theta) & \Delta t\, \Omega(-u_\theta) \\ 0_{2\times 2} & M \end{pmatrix} \tag{8.8}$$

$$M = \begin{cases} \begin{pmatrix} \frac{\partial g_{v_x}}{\partial v_x} & \frac{\partial g_{v_x}}{\partial v_y} \\ \frac{\partial g_{v_y}}{\partial v_x} & \frac{\partial g_{v_y}}{\partial v_y} \end{pmatrix} & \text{if } |v_{t-1}| \geq |k\Delta t| \\ \Omega(-u_\theta) & \text{otherwise} \end{cases} \tag{8.9}$$

$$\frac{\partial g_{v_x}}{\partial v_x} = c\cos(-u_\theta) - \frac{k\ \Delta t\ v_x(\cos(-u_\theta)\ v_x - \sin(-u_\theta)\ v_y)}{|v|^3} \tag{8.10}$$

$$\frac{\partial g_{v_x}}{\partial v_y} = -c\sin(-u_\theta) - \frac{k\ \Delta t\ v_y(\cos(-u_\theta)\ v_x - \sin(-u_\theta)\ v_y)}{|v|^3} \tag{8.11}$$

$$\frac{\partial g_{v_y}}{\partial v_x} = c\sin(-u_\theta) - \frac{k\ \Delta t\ v_x(\sin(-u_\theta)\ v_x - \cos(-u_\theta)\ v_y)}{|v|^3} \tag{8.12}$$

$$\frac{\partial g_{v_y}}{\partial v_y} = c\cos(-u_\theta) - \frac{k\ \Delta t\ v_y(\sin(-u_\theta)\ v_x - \cos(-u_\theta)\ v_y)}{|v|^3} \tag{8.13}$$

with

$$c = \left(1 + \frac{k\ \Delta t}{|v|}\right) \tag{8.14}$$

The noise matrix R is highly simplified. It should incorporate the noise of the motion of the robot as well as the noise of the ball which is mosty the product of collisions with other robots. The change of the position of the ball through a collision is asuumed to change the position not more than 100 cm/s. The velocity is assumed to change in a similar fashion. The process noise matrix R is

$$R = \begin{pmatrix} 100^2\ \Delta t & 0 & 0 & 0 \\ 0 & 100^2\ \Delta t & 0 & 0 \\ 0 & 0 & 100^2\ \Delta t & 0 \\ 0 & 0 & 0 & 100^2\ \Delta t \end{pmatrix} \tag{8.15}$$

Pure additive process nosie would lead to a ever growing uncertainty for the velocity. The uncertainty of the velocity however is limited by the maximal ball velocity. Therefore, the velocity part of the covariance matrix is limited to the assumed maximal ball speed of 100 cm/s.

$$\overline{\Sigma}_{vel} = \begin{pmatrix} 100 & 0 \\ 0 & 100 \end{pmatrix} \tag{8.16}$$

8.1.2 Measurement Model

The measurement model h is relatively simple given the current state μ (more precisely the position of the ball p_x and p_y) and the height of the camera h_{camera}. The measurement model can be inferred directly from the spherical measurement representation from figure 6.6:

$$h(\mu) = \begin{pmatrix} \text{atan2}(h_{camera}, |p|) \\ \text{atan2}(p_y, p_x) \end{pmatrix} \tag{8.17}$$

Note that the speed of the ball cannot be observed by one percept. The Jacobian H looks like this

$$H = \begin{pmatrix} -\frac{h_{camera}\ p_x}{|p|^3 + h_{camera}^2 |p|} & -\frac{h_{camera}\ p_y}{|p|^3 + h_{camera}^2 |p|} & 0 & 0 \\ -\frac{p_y}{p_x^2 + p_y^2} & \frac{p_x}{p_x^2 + p_y^2} & 0 & 0 \end{pmatrix} \tag{8.18}$$

Measurements of the ball show that the noise of measurements can be approximated with a *constant* noise in the pitch and the yaw directions. The standard deviation was 0.014 radian in pitch and 0.03 radian in yaw direction which leads to the following noise matrix Q:

$$Q = \begin{pmatrix} 0.014^2 & 0 \\ 0 & 0.03^2 \end{pmatrix} \tag{8.19}$$

8.1.3 Robot-Ball Interaction

It is difficult to model the influence of other robots (teammates and opposing robots) on the ball. However, the influence of the robot that has the ball model can be described. The modeling agent interacts with the ball only when it is very close to the ball, *e.g.,* when the agent dribbles the ball. If no interaction with the dribbling robot is modeled, the ball would simply roll through the robot.

The ball state is reset if the ball was in front of the robot in the previous time step, *i.e.,* $p_x > 0$ and $-w < p_y < w$ where w is the width of a foot, and would be behind the robot in the current time step, *i.e.,* $p_x \leq 0$. Here, reset means that the ball is simply placed in front of the robot, the velocity is set to zero and the covariance matrix is set to high values for the position and the velocity. The state is corrected with the next percept.

This model is simple, but modeling the angle of impact as the exit angle does not improve the model. Because the ball is spinning and the front of the feet of the robot is not an ideal plane (especially when the robot is walking), the ball does not change the direction as assumed under ideal conditions.

The robot usually interacts with the ball when it is facing the ball (dribbling and kicking). Therefore, only the interaction with the ball in front of the robot is modeled, other interactions are neglected.

This reset process of the ball is executed during the prediction step.

8.1.4 The Weight Update

The weight should represent the likelihood that a given hypothesis represents the real ball. The lower the weight, the more likely it is that the hypothesis represents a false-positive. Here, false-positive means that the measurement does not fit the given model. A measurement which is false-positive for one model can be a valid measurement for another model. An intuitive approach is to model the weight as likelihood of the measurement:

$$\mathcal{N}(v_i, S_i) = \det(2\pi\Sigma)^{-\frac{1}{2}} \exp\left\{-\tfrac{1}{2} v_i^T S_i^{-1} v_i\right\} \tag{8.20}$$

where $v_i = z - \overline{z_i}$ is the innovation of the measurement for the given hypothesis i, and S_i is the covariance of the measurement. The current weight $w_{i,t}$ is calculated as the product of the previous weight $w_{i,t-1}$ and the measurement likelihood:

$$w_{i,t} := w_{i,t-1}\, \mathcal{N}(v_i, S_i) \tag{8.21}$$

However, a false-positive z which was perceived far away from the expected measurement $\overline{z}$ has a likelihood close to 0 and therefore would lead to the immediate removal of the hypothesis. To incorporate the possibility of false-positives, the equation for the measurement likelihood is adjusted:

$$w_t := w_{t-1} \left((1-\varepsilon) \det(2\pi\Sigma)^{-\frac{1}{2}} \exp\left\{-\tfrac{1}{2} v_i^T S_i^{-1} v_i\right\} + \varepsilon\right) \tag{8.22}$$

ε can be interpreted as the likelihood for a false-positive. It reduces the negative effect of a measurement which does not fit the model.

This weight update step is executed for all hypotheses. Afterwards, the weights are normalized.

```
def updateBallModel():
    # predict
    for hypothesis in hypotheses:
        hypothesis.predict(control)

    # correct
    noCorrection = true
    for hypothesis in hypotheses:
        if likelihood(percept, given=hypothesis) > threshold:
            hypothesis.correct(percet)
            noCorrection = false

    # add new hypothesis
    if noCorrection:
        hypothesis.addHyothesis(percept)

    # merge
    for hypA, hypB in pairwise(hypotheses):
        if similarity(hypB, hypA) > threshold:
            merge(hypB, hypA)
            hypB.weight = 0

    # delete
    for hypothesis in hypotheses:
        if hypothesis.weight < threshold:
            delete(hypothesis)
```

Listing 8.1: Logic of the local ball model.

In addition to the described weight update there are some criteria which influence the weights. The weights are reduced if the position of the ball is too far away from the robot, if the speed is unrealistically large, or if the ball was not seen within a certain time span.

8.1.5 Multi-Hypotheses Management

The entire algorithm for the ball modeling is shown in listing 8.1. The first ball model is initialized with the first percept. When *initializing* a new hypothesis, the spacial component is set to the relative position of the percept and the velocity component is set to 0. The uncertainty for the velocity part is set to a high value.

INITIALIZING

The control is applied to each hypotheses. As mentioned in section 6.4.1, multi-hypotheses tracking is applied to make the ball model more robust against false-positives. Therefore, ball observations are only applied to models if the likelihood for the measurement for the given model is high enough. Otherwise a new hypothesis is created. This results in models which are only updated with measurements which make sense for the given model. False-positives have their own model.

Hypotheses with a low weight are removed, and similar tracks are merged. In general, tracks have the tendency to cluster around the real ball and are subsequently merged.

8.2 The Local Obstacle Model

The local obstacle model is implemented using *multi-hypothesis tracking*. However, a sum of Gaussians in the strict sense is not used because the Gaussians which model obstacles do not have a weight. Multiple targets are tracked with multiple Gaussians. Nevertheless, the update procedure of the local obstacle model is very similar to the procedure of the ball model.

MULTI-HYPOTHESIS TRACKING

As mentioned in section 6.4.2, is is hard to predict the state of an obstacle by incorporating its speed. Therefore, the *state space* of each obstacle was chosen to only consist of the relative position of the obstacle:

STATE SPACE

$$\mu = \begin{pmatrix} p_x \\ p_y \end{pmatrix} \tag{8.23}$$

The *measurement input* for the LocalObstacleModel is

MEASUREMENT INPUT

$$Z = \{z_1, \cdots, z_n\} \tag{8.24}$$

with $z_i = (z_{pitch}, z_{yaw})^T$ the spherical coordinates of a potential obstacle.

Each obstacle of the LocalObstacleModel is implemented using the GenericEKF and g, G, Q, h, H, and R must be provided.

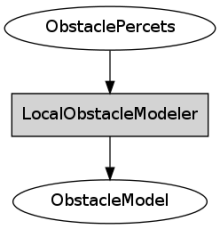


Figure 8.2: Obstacle model architecture.

8.2.1 Process Model

The process model is similar to the process model of the ball. The difference is that the velocity component does not exist. This simplifies the process model dramatically.

In the prediction step, only the odometry of the robot changes the state. The state is rotated around u_θ and translated by u_x and u_y:

$$\overline{\mu_t} = g(\mu_{t-1}, u) = \Omega(-u_\theta)\, \mu_{t-1} - \begin{pmatrix} u_x \\ u_y \end{pmatrix} \tag{8.25}$$

Respectively, the Jacobian is much simpler:

$$G = \Omega(-u_\theta) \tag{8.26}$$

Because the motion of an obstacle cannot be predicted, noise is added. In the Kid Size league, the robots move at less than 30 cm/s. This leads to the following process noise matrix R:

$$R = \begin{pmatrix} 30^2 \Delta t & 0 \\ 0 & 30^2 \Delta t \end{pmatrix} \tag{8.27}$$

Most teams are much slower, so a noise matrix with smaller entries on the diagonal could be chosen instead. However, this model works quite well.

8.2.2 Measurement Model

The measurement model of the ball (see equation (8.1.2)) is very similar to the measurement model for obstacles:

$$h(\mu) = \begin{pmatrix} \text{atan2}(h_{camera}, |p|) \\ \text{atan2}(p_y, p_x) \end{pmatrix} \tag{8.28}$$

Because the speed of an obstacle is not modeled, the Jacobian H for the obstacle model is also simpler than the Jacobian for the ball model.

$$H = \begin{pmatrix} -\frac{r\ p_x}{|p|^3 + r^2|p|} & -\frac{r\ p_y}{|p|^3 + r^2|p|} \\ -\frac{p_y}{p_x^2 + p_y^2} & \frac{p_x}{p_x^2 + p_y^2} \end{pmatrix} \tag{8.29}$$

The measurement of the obstacles is less reliable than the measurements of the ball, especially in the yaw direction (the width of the obstacle sometimes changes). The noise is approximated to be *constant* in pitch and yaw directions. This leads to the following noise matrix Q:

$$Q = \begin{pmatrix} 0.014^2 & 0 \\ 0 & 0.06^2 \end{pmatrix} \tag{8.30}$$

8.2.3 Pre-Filtering of False-Positives

The ObstaclePercept contains a list of potential obstacles. This list includes a high number of false-positives. To avoid spawning many new hypotheses the history of a percept is saved in the so-called *percept history*. A percept from the percept history is only valid if the percept was seen for a minimum of three frames out of four, and if the percept is big enough. In the current implementation only obstacles which are relatively close (3 m) are modeled. Only close obstacles are of interest for the behavior layer because the obstacle model is mostly used for obstacle avoidance. Only modeling close obstacles reduces the number of hypotheses and improves the performance.

PERCEPT HISTORY

8.2.4 Multi-Hypotheses Management

The largest difference between the LocalObstacleModel and the LocalBallModel is that there are multiple measurements and each measurement has to be assigned to its corresponding model. If no corresponding model exists, a new one has to be created. Also, creating a new model for false-positives must be prevented. The algorithm for the obstacle modeling is shown in listing 8.2.

MEASUREMENT-MODEL CORRESPONDENCE

First, all existing models are updated with the odometry of the robot (line 2-5). This shifts and blurs each model according to the odometry.

PREDICTION

Then, the correction step follows (line 6-10). For each percept, one needs to determine the likelihood of the correspondence to every model (line 8). If the highest likelihood is above a threshold, the corresponding model executes the actual correction step (line 10). If there is no corresponding model, the percept is added to the percept history, which is a list for potential new models (line 11-12). New models are only added for valid percepts from the percept history (line 14-17). False-positives do not immediately create a new model which leads to a reduction of the number of models.

Despite the careful method of creating new models, there are sometimes multiple models for one actual object. This necessitates merging similar models (line 22-26). The similarity of models is compared pairwise. If the models are similar they are merged and the weight of the untouched model is set to zero.

MERGE

Another step is to *delete* models which are unlikely to be true (line 28-32). In contrast to the ball model, no weight is used here. The likelihood for

DELETE

```
def obstcale_model():
    # predict
    for model in models:
        model.predict(odometry)

    # correct existing models
    for percept in percepts:
        model = argmax_model likelihood(models, percept)
        if model.likelihood > threshold:
            model.correct(percept)
        else:
            perceptHistory.add(percept)

    # add new models
    for percept in perceptHistory.getConfirmed():
        models.addNew(percept)
        perceptHistory.remove(percept)

    # cleanup
    perceptHistory.removeBad()

    # merge similar models
    for modelA, modelsB in pairwise(models):
        if similarity(modelA, modelB) > threshold:
            merge(modelA, modelB)
            models.remove(modelB)

    # remove bad models
    for model in models:
        if model.covariance > threshold or
           model.lastCorrection > TTL:
            models.remove(model)
```

Listing 8.2: Local obstacle model algorithm.

a model representing a real robot is simply represented by the covariance of the robot. When the covariance ellipse is too big the model is removed. Also, if the last measurement update was not conducted within a certain time span the model is removed. A *time to live* counter (short *TTL*) is used for this.

There are other heuristics for deleting models which are currently not used. *E.g.,* one could remove models which should be in the field of view of the robots, but are not (there is no percept). The problem with this approach is that the blur of the (often) fast moving camera decreases the accuracy of the vision and obstacles are not recognized, even though they are there. The more conservative approach of not deleting the models yields better results in this case.

9

Implementation: The Self-Localization

The self-localization computes the global pose of the robot on the field using the odometry and all percepts. The old particle filter localization, the module PFLocalization, is replaced by the new module MHKFLocalization.

As with the PFLocalization, the robot pose or *state space* is modeled as

STATE SPACE

$$\mu = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} \tag{9.1}$$

where x and y denote the global position, and θ is the rotation of the robot.

The localization is implemented as a *multi-hypotheses extended Kalman filter* (short *MHEKF*). It was first prototyped with UKFs, then implemented with EKFs.

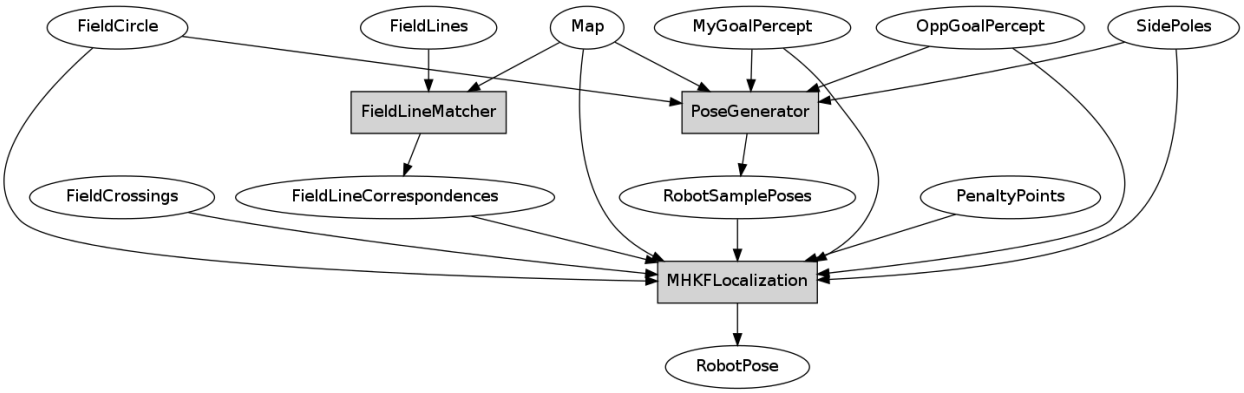
MHEKF

Figure 9.1 shows the architecture of the localization module. Section 9.4 explains the FieldLineMatcher that transforms field line percepts into point measurements which can be easily used by the localization. The PoseGenerator (see section 9.8.2) creates poses purely on the measurements. Two modules which are not shown in figure 9.1, the PoleFilter and the CrossingFilter are described in section 9.1 and section 9.2.

9.1 Histogram Filter for the Poles

Side poles are sometimes incorrectly classified as goal poles (see section 6.2). A histogram filter was implemented to make the classification more reliable and prevent misclassified percepts from being used. The histogram filter is

Figure 9.1: The architecture of the complete localization module.



```
1 def pole_filter():
2     for percept in polePercepts:
3         filter = histogramFilter.getNear(percept)
4         if filter is None:
5             histogramFilters.create(percept)
6         else:
7             filter.update(percept)
```

Listing 9.1: Discrete histogram filter algorithm for the poles.

independent of the localization and it only uses the percepts to determine the most likely state of the poles. An observed pole can have one of the following states:

- *unknown blue goal post*,
- *unknown yellow goal post*,
- *BYB side post*, or
- *YBY side post*.

If a pole is recognized as a left or a right goal post, the classification is always correct and the pole is not filtered. Only poles which have one of the four states are sometimes misclassified.

In contrast to the histogram filter shown in section 4.2, the state here is static, *i.e.*, it cannot be changed by any control. In fact, there is no control. The prediction step of the histogram filter is simply removed. The resulting algorithm resembles the Bayes' belief update.

The pole filter algorithm is shown in listing 9.1. There can be multiple pole percepts (side poles and goal posts) which must be filtered. This means that a histogram filter is associated with the position of the percept. For each pole percept, a new histogram filter is created (line 6) if there is not already a histogram filter near the position of the percept (line 3 - 6). If there is a histogram filter near the position of a percept, the histogram filter updates the belief with the measurement. If the probability of a state for a histogram filter is above a *threshold*, the percept is marked as *valid*.

THRESHOLD

Listing 9.1 shows the actual *belief update* for one histogram filter. This

BELIEF UPDATE

```
1 def update(percept):
2     position = percept.position
3
4     # belief update
5     for state in states:
6         state.prob = P(percept, given=state.type) * state.prob
7
8     # normalize
9     totalProb = sum[state.prob for state in states]
10    for state in states:
11        state.prob = state.prob / totalProb
```

Listing 9.2: Belief update of the histogram filter.

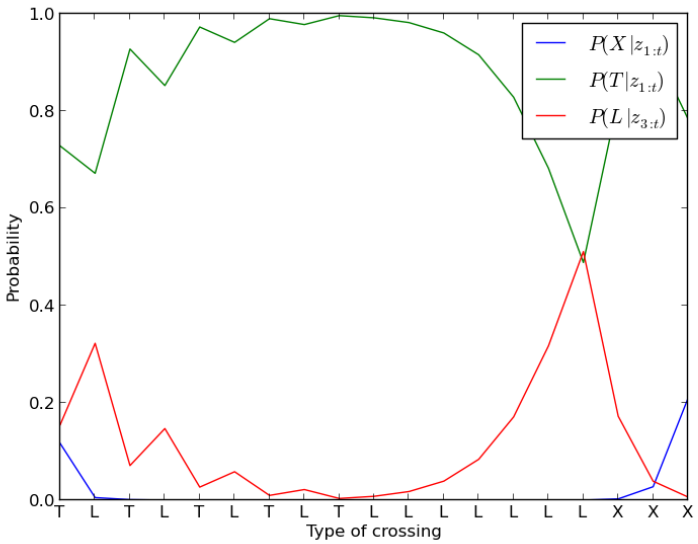


Figure 9.2: Probability for a series of crossing measurements. The x-axis shows the type of the measurement.

Blue: $P(X \mid z_{1:t})$
Green: $P(T \mid z_{1:t})$
Red: $P(L \mid z_{1:t})$

follows the histogram filter in section 4.2 very closely. The probability for every state is multiplied with the measurement likelihood (line 6) and then normalized (line 9-11). The measurement likelihood `P(percept, given=state.type)` is implemented as a simple lookup in a table.

In general histogram filters work quite well for these kinds of problems. The assumption is that the noise of the sensor can be modeled via the measurement likelihood. However, the misclassification does not appear randomly but is reproducible. Under certain conditions the side pole is always classified as a blue/yellow goal post. A histogram filter simply cannot solve this problem.

In the end, the histogram filter for the poles was not used because depending on the choice of measurement likelihood, the histogram filter reduced the number of valid percepts too much or it did not prevent false-positives. The pole classification of the vision should be improved instead.

9.2 Histogram Filter for the Line Crossings

To improve line classification, a histogram filter like the one described in the previous section was implemented. The state space consists of *X-crossing*, *T-crossing*, and *L-crossing*. The algorithm is the same as in the previous section and will therefore not be explained further.

During the course of the implementation it was discovered that the crossings are rarely recognized and that the classification is often wrong. The histogram filter improved the classification, but also reduced the number of valid percepts. Figure 9.2 shows a run of the histogram filter with a series of pole measurements.

In the end, the use of crossings features were discarded in favour of field lines which model the crossings implicitly and also solve the misclassification problem. See section 9.4 for more on how the field lines are used for the localization task.

```
def getCorrespondence(μ, Σ, percepts):
    for i, landmark in enumerate(landmarks):
        z_i = h(μ, percept, landmark)
        H_i = getH()
        S_i = H_i Σ H_i^T + Q
        likelihood_i = N(percept_i | z_i, S_i)
    best = arg max_i likelihood_i
    return landmarks[best]
```

Listing 9.3: Unknown correspondence of a measurement and a landmark.

9.3 The Percept-Landmark Correspondence Problem

In the case of ambiguous landmarks the measurement-landmark correspondence is not clear. This section describes how to select the best landmark on a map for a given observation. Once a correspondence choice is made, the correction step is executed.

For unambiguous landmarks the correspondence is clear, *e.g.,* the BYB side pole percept corresponds with the BYB side pole landmark. For ambiguous landmarks, such as line crossings, one needs to determine the correspondence first. The correspondence choice is implemented as a *maximum likelihood* choice. The likelihood of the measurement z given the state (μ and Σ) is calculated for each landmark l_i

MAXIMUM LIKELIHOOD

$$P(z \mid \mu, \Sigma, l_i) \tag{9.2}$$

Then, the landmark with the highest likelihood is selected for the correction step:

$$l_{best} = \arg\max_i P(z \mid \mu, \Sigma, l_i) \tag{9.3}$$

The likelihood of a measurement-landmark correspondence can be modeled with a Gaussian distribution. The expected measurement $\overline{z_i}$ in the state μ for a given landmark l_i is calculated. The corresponding uncertainty of the measurement S_i is also calculated. Note that this happens in the measurement space. The likelihood is then calculated with a normal distribution

$$\mathcal{N}(percept \mid \overline{z_i}, S_i) \tag{9.4}$$

In listing 9.3 the selection of the best correspondence is shown in detail.

Depending on the number of percepts and the number of potential corresponding landmarks, the approach above can be quite expensive[1].

The described approach selects the best landmark for a measurement, but false-positives and correspondences with a low likelihood are still used in the correction step. Therefore, in addition to the maximum likelihood correspondence criterion, a *threshold* is used. Only percepts that are sufficiently explainable, *i.e.,* the likelihood is above the threshold, are used for the correction step. Most false-positives are filtered out by the threshold because their likelihood is too low. If they are close to the actual landmark (which normally does not happen) the negative effect of false-positives is not as drastic. Preventing false-positives to be used in the correction step is extremely important because false-positives do not have a Gaussian noise property, which the KF assumes.

[1] Example:
Assume the field has 16 crossings and four crossings (without a type) were observed. The calculation to find the best correspondence must be executed $4 \cdot 16 = 64$ times.

FALSE-POSITIVES

THRESHOLD

9.4 Constraint-based Field Line Matching

Field lines are the most visible features. But field lines are ambiguous landmarks and they are also the only landmarks which cannot simply be represented as a point in the environment. This section describes how the correspondence problem for field line percepts and landmarks can be solved and how point updates can be used for the correction step.

The problem with field line percepts is that a point correspondence cannot be easily determined. Assume a simple world which consists of only a single field line. In most cases, the agent only observes a part of a line. The length of the observed line is not a sufficiently strong constraint to determine the robot pose. The *percept-line point correspondence* cannot be determined. Percept-line point correspondence means that the start and end point of the line percept corresponds to two points on the line on the field. If the agent observes the whole line, the start and end point of the percept will have a correspondence with the start and end point of the line on the field. The correspondence must be unique in order to use it for the measurement update of the Kalman filter (this is the correspondence problem described in section 9.3). In this section, the percept-line point correspondence is simply called *line correspondence*.

Given one line segment percept, the robot's pose must be on a parallel line of the field line. In this section this line is called *pose line*. However, the exact pose is unknown and therefore the exact line correspondence is unknown as well. Figure 9.7 illustrates the problem. Potential poses and line correspondences are shown for a perceived line. The gray line is the line percept which is mapped to the field lines on the field. The black dotted line represents the possible pose lines. The red dots represent concrete correspondences. So far there are unlimited line correspondences because the poses are on a (continuous) line.

The paragraph above does not only illustrate the problem of field lines as landmarks, but also outlines a possible algorithm to find line correspondences. Even though one line does not restrict possible poses and their line correspondences, several percepts of line segments might do this. Note that long lines yield more information than short ones. Line percepts which are normal to each other constrain the space the most. Each line percept creates a pose line (the black dotted line in figure 9.7). The intersection of the two pose lines restricts the possible poses (continuous) to a set of poses (discrete). Figure 9.3 and figure 9.4 show possible positions after restricting the pose space with two line segment percepts.

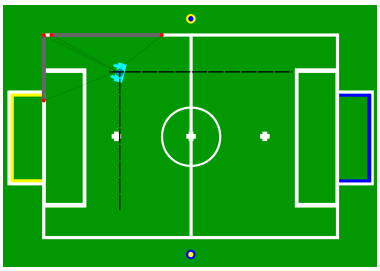


Figure 9.3: One potential pose for two line segment percepts. The pose is at the intersection of two pose lines.

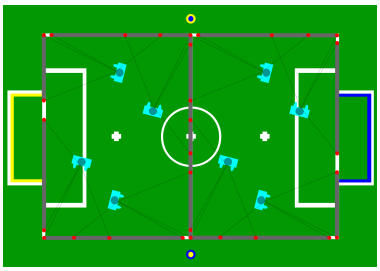


Figure 9.4: Potential pose (all shown) for two line segment percepts.

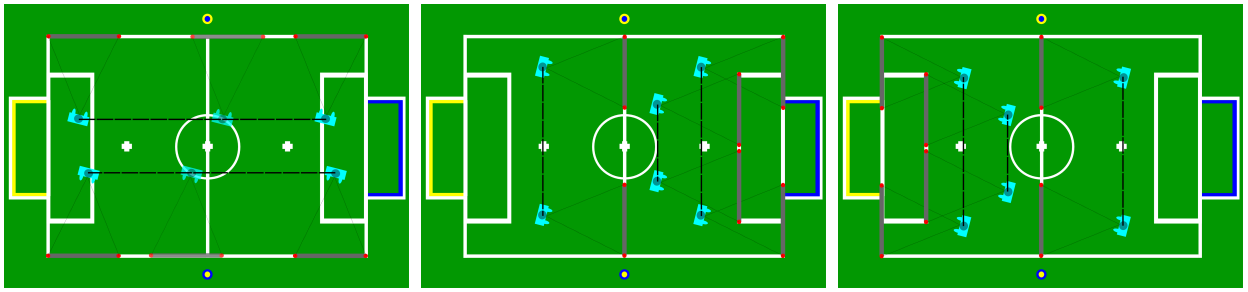


Figure 9.5: Pose lines and line correspondences for a perceived line segment.

```
1  def findLinePerceptCorrespondences():
2    perceptsH, perceptsV = groupLinesByDirection(allLines)
3
4    if not perceptsH or not perceptsV:
5        return
6
7    # Sort lines by length. Long lines yield more information
         than short ones.
8    perceptsH, perceptsV = sorted(perceptsH), sorted(perceptsV)
9
10   # Use the first normal lines to find initial poses.
11   poses = getInitialPoses(perceptsH[0], perceptsV[0])
12
13   # Shrow away the first line
14   perceptsH, perceptsV = perceptsH[1:], perceptsV[1:]
15
16   for perceptH, perceptV in zip(perceptsH, perceptsV):
17       if perceptH:
18           poses = restrict(poses, perceptH)
19       if perceptV:
20           poses = restrict(poses, perceptV)
```

Listing 9.4: Algorithm to determine correspondences of field line percepts and field line landmarks.

```
1  def restrict(poses, percept):
2    result = []
3    for pose in poses:
4        if P(percept, given=pose) > threshold:
5            pose.addCorrespondence(percept)
6            result.append(pose)
7    return result
```

Listing 9.5: Restrict possible poses by the given line percept.

A histogram of the angle of the percepts is used to cluster the lines by direction as shown in figure 9.6. The clustered lines are called perceptsH and perceptsV for horizontal and vertical lines. The initial set of poses (and their line correspondences) is then calculated with the first percepts in perceptsH and perceptsV (see figure 9.3 and figure 9.4). The poses are at an intersection of two pose lines. Then, each additional line percept reduces the number of poses (and their line correspondences), see listing 9.5. In line 4 the likelihood of the measurement is calculated for the given pose (for the sake of simplicity the euclidean distance is currently used). If the likelihood is above a threshold, the pose is kept, otherwise the pose is rejected. If the pose is kept, the line correspondences are added.

In the end, one is left with only a few poses and their line correspondences. The poses are just the byproduct of the algorithm, the line correspondences are the important information which is used for the correction step. Nevertheless, the poses could be used to spawn new hypotheses. In the current implementation this is not done.

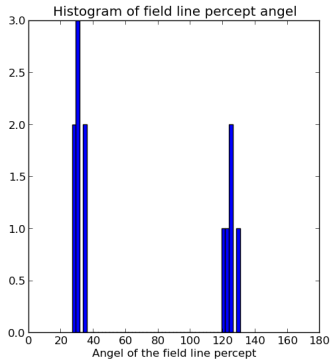


Figure 9.6: Histogram of the angles of the field lines which allows it to determine the two main directions of the field lines.

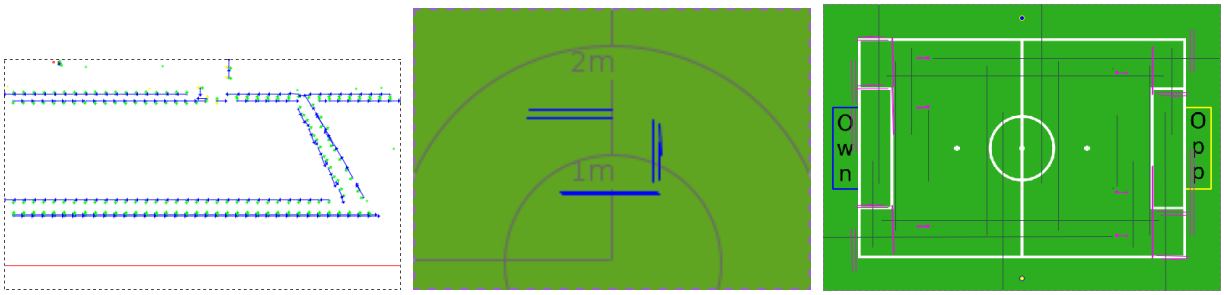


Figure 9.7: Implemented line point matching.

The field line percepts which are used by this algorithm must be straight. Currently the FieldLineProvider connects lines when they are adjacent and the angle of the tow lines is below a threshold. The result is, that two orthogonal lines are sometimes connected. If only the start and the end point of the percept lines are considered, as in the current implementation, field line matches cannot be found. This is shown in figure 9.8. The two long orthogonal lines are connected to one an no match can be found.

Because of this limitation the field point matching is deactivated. However, as soon as the FieldLineProvider is improved, the feature can be activated again.

Constraint-based vs Maximum Likelihood The advantage of this algorithm is that it considers the dependency between lines. When there is a false-positive, the algorithm does not find any matches. This is desirable because false-positives should not be used for the correction step. The simple maximum likelihood criteria for the correspondence problem does not consider correlations between different measurements.

Example of the advantages of constraint-based methods: Two X-crossings are observed and they are 3 m apart. They can only be 1.2 m apart. One of the two percepts is definitely a false-positive. The maximum likelihood decision might assign each percept the more likely X-crossing on the field. A constraint-based algorithm like the one one above, would simply reject the percepts.

9.5 EKF Process Model

As mentioned in section 6.1, the odometry is not reliable. The overhead camera system did not allow extensive measurement series to determine the noise properties of the odometry (see section 2.2.3). Therefore, the cho-

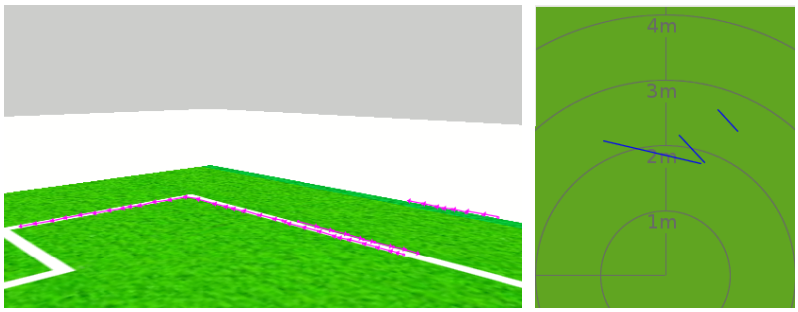


Figure 9.8: Orthogonal field lines are not separated correctly. The field line matching algorithm does not work.

sen process model is very simple. The previous state μ_{t-1} is rotated and translated according to the odometry.

$$\overline{\mu_t} = g(\mu_{t-1}, u_t) = \underbrace{\begin{pmatrix} x \\ y \\ \theta \end{pmatrix}}_{\mu_{t-1}} + \begin{pmatrix} u_x \cos\theta - u_y \sin\theta \\ u_x \sin\theta - u_y \cos\theta \\ u_\theta \end{pmatrix} \tag{9.5}$$

The Jacobi matrix G is given as

$$G = \frac{\partial g(u_t, x_{t_1})}{\partial x_{t-1}} = \begin{pmatrix} \frac{\partial x'}{\partial \mu_{t-1,x}} & \frac{\partial x'}{\partial \mu_{t-1,y}} & \frac{\partial x'}{\partial \mu_{t-1,\theta}} \\ \frac{\partial y'}{\partial \mu_{t-1,x}} & \frac{\partial y'}{\partial \mu_{t-1,y}} & \frac{\partial y'}{\partial \mu_{t-1,\theta}} \\ \frac{\partial \theta'}{\partial \mu_{t-1,x}} & \frac{\partial \theta'}{\partial \mu_{t-1,y}} & \frac{\partial \theta'}{\partial \mu_{t-1,\theta}} \end{pmatrix} \tag{9.6}$$

$$G = \begin{pmatrix} 1 & 0 & -u_x \sin\theta - u_y \cos\theta \\ 0 & 1 & u_x \cos\theta + u_y \sin\theta \\ 0 & 0 & 1 \end{pmatrix} \tag{9.7}$$

A more sophisticated process model should be implemented as soon as the noise properties can be more reliably measured. Each control/odometry component has an influence on the other components which should be modeled, *e.g.,* walking forward introduces rotational noise. See [29] for more on the modeling of the odometry noise.

9.6 EKF Measurement Model

The measurement model with h, H, and R will be specified in this section. It is assumed that the landmark-percept association is correct and that there is a single percept z (in spherical coordinates) which is used for the correction step.

Let l denote a landmark with l_x and l_y as the absolute position on the field. Also, let

$$d = \sqrt{(l_x - \mu_x)^2 + (l_y - \mu_y)^2} \tag{9.8}$$

be the euclidean distance of the robot to the landmark. As in the previous sections, h_{camera} denotes the height of the camera. Then the measurement model can be given by

$$\overline{z} = h(\overline{\mu_t}, l, h_{camera}) = \begin{pmatrix} \text{atan2}(h_{camera}, d) \\ \text{atan2}(l_y - \mu_y, l_x - \mu_x) - \mu_\theta \end{pmatrix} \tag{9.9}$$

and its corresponding Jacobian

$$H = \frac{\partial h(\mu, l)}{\partial \mu} = \begin{pmatrix} \frac{\partial z_{pitch}}{\partial \mu_x} & \frac{\partial z_{pitch}}{\partial \mu_y} & \frac{\partial z_{pitch}}{\partial \mu_\theta} \\ \frac{\partial z_{yaw}}{\partial \mu_x} & \frac{\partial z_{yaw}}{\partial \mu_y} & \frac{\partial z_{yaw}}{\partial \mu_\theta} \end{pmatrix} \tag{9.10}$$

$$\begin{array}{rcl} \frac{\partial z_{pitch}}{\partial \mu_x} & = & \frac{h_{camera}(l_x - \mu_x)}{d(h_{camera}^2 + d^2)} \\ \frac{\partial z_{pitch}}{\partial \mu_y} & = & \frac{h_{camera}(l_y - \mu_y)}{d(h_{camera}^2 + d^2)} \\ \frac{\partial z_{pitch}}{\partial \mu_\theta} & = & 0 \\ \frac{\partial z_{yaw}}{\partial \mu_x} & = & \frac{(l_y - \mu_y)}{d^2} \\ \frac{\partial z_{yaw}}{\partial \mu_y} & = & \frac{(-l_x + \mu_x)}{d^2} \\ \frac{\partial z_{yaw}}{\partial \mu_\theta} & = & -1 \end{array} \tag{9.11}$$

```
1 def ekf_update(μ, Σ, u, percepts, map):
2   μ̄, Σ̄ = predict(μ, Σ, u)
3
4   for percept in percepts:
5       landmark = map.getCorrespondence(percept)
6       μ̄, Σ̄ = correct(μ̄, Σ̄, percept, landmark)
7
8   return μ̄, Σ̄
```

Listing 9.6: Applying multiple measurements with known correspondence.

The measurement noise for the landmarks is the same as the noise for the ball (see section 8.1.2) and therefore the noise matrix R is the same:

$$Q = \begin{pmatrix} 0.014^2 & 0 \\ 0 & 0.03^2 \end{pmatrix} \tag{9.12}$$

9.7 Multiple Measurement Updates

In contrast to the other models, there can be multiple percepts which are used for the correction step, *e.g.*, the robot observes two goal posts, a penalty point and field line crossings. Multiple observations yield more information than a single one and should be used. There are two ways to conduct the correction step with n updates:

- n iterative updates, and
- one high dimensional update with n observations.

9.7.1 Iterative Updates

ITERATIVE UPDATES

A common approach is to apply the measurements iteratively as shown in listing 9.6. Executing the correction step with different measurements successively is possible, because the measurements are assumed to be independent of each other. First, the corresponding landmark of the map for the given percept is selected[2]. getCorrespondence() returns the corresponding landmark of the map for the given percept. The landmark is needed in the correction step to calculate the deviation between the measurement and the landmark. Then, once the landmarks are given, the correction step can be executed. This is repeated for every measurement.

[2] Here, a known correspondence between the percept and the landmark is assumed.

9.7.2 Single High Dimensional Update

SINGLE HIGH DIMENSINAL UPDATE

An alternative approach is to execute the measurement update as a single high dimensional update as proposed in [27]. If multiple measurements originate from the same source, the camera of the robot, then the percepts and their noise are not actually independent. The error of the measurement results from the inaccuracy of the camera pose.

Let C denote the covariance of a measurement in spherical coordinates

$$C = \begin{pmatrix} c_{pitch}^2 & 0 \\ 0 & c_{yaw}^2 \end{pmatrix} \tag{9.13}$$

where c_{pitch} and c_{yaw} are the standard deviation of the error in pitch and yaw directions. Assuming that the error of all measurements in a frame is a result of the camera, the combined covariance for all measurements is the same, as shown in equation (9.14). λ is the factor for the dependence of the measurements. When $\lambda = 0$ there is no dependence between measurements.

$$C_{multi} = \begin{pmatrix} C & & \lambda\, \mathrm{C} \\ & \ddots & \\ \lambda\, \mathrm{C} & & C \end{pmatrix}$$

Figure 9.9 shows the superior performance of one high dimensional measurement update with full covariance compared to one high dimensional measurement update with no inter-feature covariance ($\lambda = 0$) and low dimensional iterative updates.

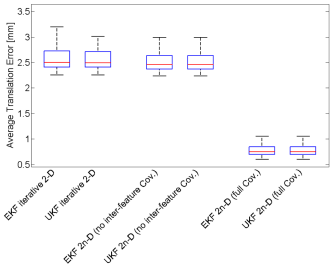


Figure 9.9: "Comparison between different methods to handle multiple measurements at one time step." Taken from [27].

To implement the high dimensional measurement update the measurement model needs to be modified accordingly. Equation (9.9) is used to create the expected measurements $\overline{z}_{multi}$ for the given state and the given landmarks

$$\overline{z}_{multi} = h_{multi}(\overline{\mu}, h_{camera}, l_1, \ldots, l_n) = \begin{pmatrix} h(\overline{\mu}, l_1, h_{camera}) \\ h(\overline{\mu}, l_2, h_{camera}) \\ \vdots \\ h(\overline{\mu}, l_n, h_{camera}) \end{pmatrix} \tag{9.15}$$

The vector $\overline{z}_{multi}$ has $2n$ rows where n is the number of percepts. The Jacobian H_{multi} is an extension of Equations (9.9). Each H uses a specific landmark. Hence it is called H_{l_i} where l_i is landmark i. H_{multi} has the dimension of $2n \times 3$.

$$H_{multi} = \begin{pmatrix} H_{l_1} \\ H_{l_2} \\ \vdots \\ H_{l_n} \end{pmatrix} \tag{9.16}$$

The measurement noise matrix R_{multi} is set up like equation (9.14)

$$R_{multi} = \begin{pmatrix} R & & \lambda\, \mathrm{R} \\ & \ddots & \\ \lambda\, \mathrm{R} & & R \end{pmatrix} \tag{9.17}$$

R is constant for all measurements.

The update algorithm of listing 9.6 changes only slightly. Of course, h, H and R are implemented as shown above. Then, the landmark-percept correspondences are determined. In the end correction () is called once with the two vectors of percepts and landmarks (each with $2n$ rows). The single high dimensional update is used in the current implementation.

9.8 *Multi-Hypothesis Handling*

This section will explain the additional steps that are necessary to implement a MHEKF localization. Most of it is a straightforward extension of the previous algorithms.

The localization can be divided into several steps, as shown in listing 9.8. Each step is explained in detail in the following sections.

```
def ekf_update(μ, Σ, u, percepts, map):
  μ̄, Σ̄ = predict(μ, Σ, u)

  for percept in percepts:
      correspondences.add(map.getCorrespondence(percept))

  μ̄, Σ̄ = correct(μ̄, Σ̄, percepts, correspondences)

  return μ̄, Σ̄
```

Listing 9.7: Applying multiple measurements with one single update.

```
def execute():
    updateHypotheses()
    addNewHypotheses()
    mergeSimilarHypotheses()
    removeBadHypotheses()
    selectBestHypotheses()
```

Listing 9.8: The basic structure of the localization algorithm.

9.8.1 Updating Existing Hypotheses

The update step of the MH localization is the extension of the usual prediction and correction step of a single KF. Each KF executes the process update with the odometry separately.

The measurement-landmark correspondence problem is solved as described in section 9.3 with the maximum likelihood criterion for each hypothesis. The goal is to create hypotheses which are the product of concrete correspondence choices and subsequently an individual series of measurement updates. Not every observation is applied to every hypothesis, instead, each EKF only uses observations which are likely for the given state[3].

[3] If the observation is not unique the maximum likelihood correspondence is selected first.

A percept has to pass a number of requirements in order to be used for the measurement update for a given hypothesis. First, the innovation $v = \overline{z} - z$ of a percept must be smaller than a threshold. This filters out measurements which do not fit at all. This check is mostly applied to improve the runtime performance. Then, the measurement likelihood must be smaller than a threshold. If the percept fulfills both criteria it is used in the measurement update. False-positives which decrease the localization performance drastically are reliably filtered out by this method.

When the selection of the measurements is implemented as above, the number of measurements used by a hypothesis is an indicator for the quality of the hypothesis. Every observation which is used for the measurement update increases the weight of a hypothesis, and every observation which is not used decreases the weight.

The type of landmark also has an influence on the weight of the hypotheses. Unique landmarks are a stronger indicator for the quality of a hypothesis than ambiguous landmarks.

```
def updateHypotheses():
    for hypothesis in hypotheses:
        hypothesis.predict(odometry)

        # unique landmarks
        for percept in uniquePercepts:
            found = false
            if hypothesis.likelihood(percept) > threshold:
                hypothesis.correct(percept)
                found = true
            hypothesis.updateWeight(found, influenceUnique)

        # ambiguous landmarks
        for percept in ambiguousPercepts:
            found = false
            landmark = hypothesis.getMostLikeLandmark(percept)
            if hypothesis.likelihood(percept, landmark) >
                threshold:
                hypothesis.correct(percept)
                found = true
            hypothesis.updateWeight(found, influenceAmbiguous)
```

Listing 9.9: The basic structure of the update of the hypotheses.

The weights are updated according to the following:

$$w_{i,t} = \begin{cases} w_{i,t-1} \cdot (1-\alpha) + \alpha & \text{if percept fits model} \\ w_{i,t-1} \cdot (1-\alpha) & \text{otherwise} \end{cases} \tag{9.18}$$

where i is the index of the hypothesis, and α is the influence factor of the measurement with $0 < \alpha < 1$. Also, the weights of hypotheses which were not updated in a while are set to zero to be deleted in the deletion step.

9.8.2 Adding Additional Hypotheses

Sometimes it is possible to deduce the current pose directly from observations, *e.g.,* when when there are two (or more) measurements of unique landmarks. A single measurement limits possible robot poses to a circle around the landmark with the diameter d, the distance to the landmark. Combining two (or more) measurements of this kind reduces the number of possible poses to one or two, as shown in figure 9.10. This process, which is similar to *sensor resetting* of PF, is used to create additional hypotheses. It results in faster re-localization in case of a kidnapped robot.

SENSOR RESETTING

The *goal posts*, the *side poles*, and the *circle* in the middle of the field are used for the triangulation process. For each pair of the percepts of these landmarks, the triangulation process is executed and a new hypothesis with a low weight is added. The triangulation can create multiple similar new hypotheses. In general, this is not a problem, because similar hypotheses are merged later (see section 9.8.3).

Figure 9.10 shows some results of the triangulation process. The robot pose is not changed during the process. The filled blue dots mark the observed features. The blue circles indicate the distance of the observation. An intersection of two circles is a possible robot pose. Note that the results of the triangulation are spread over the entire field due to the measurement

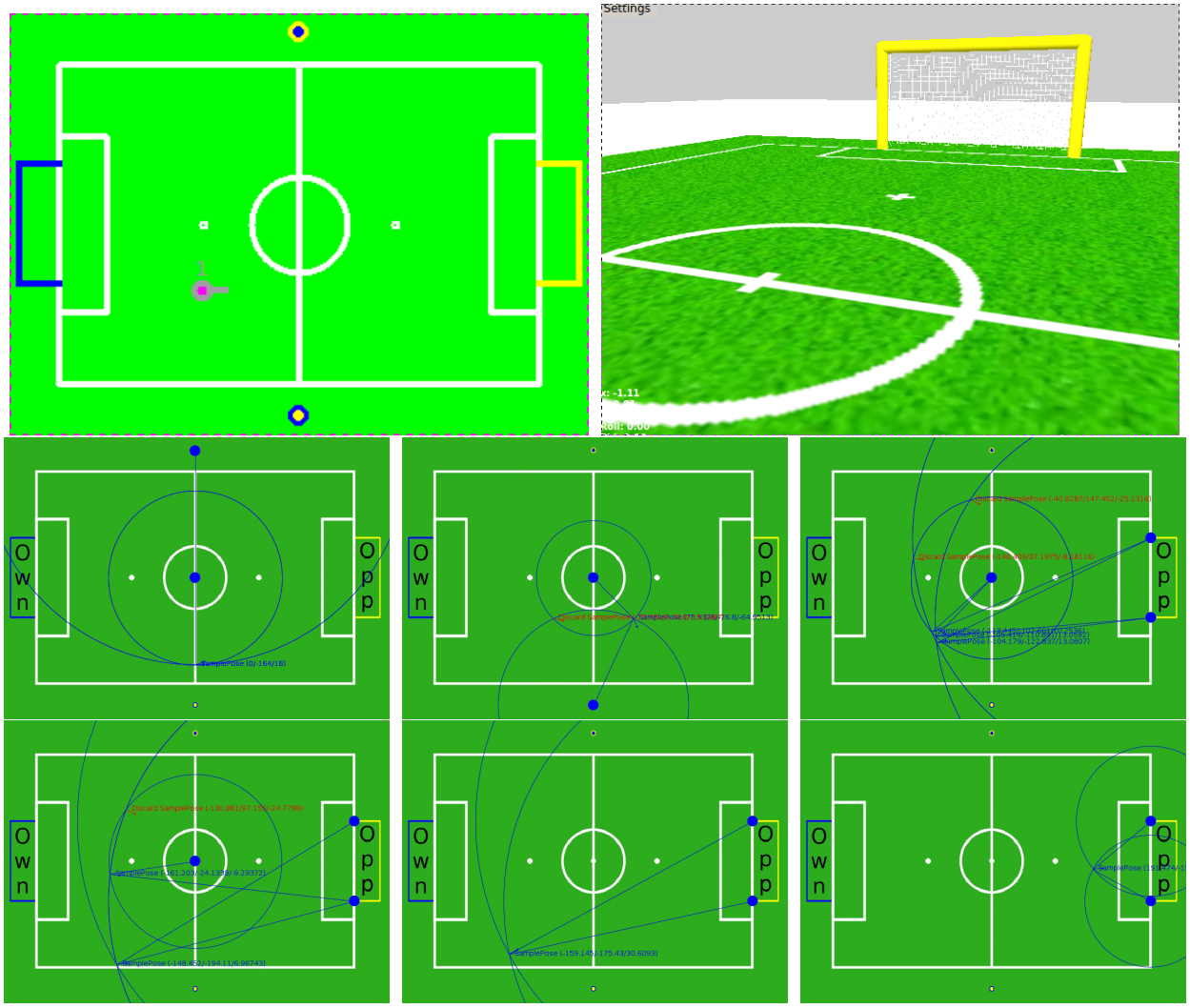


Figure 9.10: Results of the triangulation for one robot pose (top left). The top right is an example of the view of the robot. Because of the measurement noise the triangulation process is not very accurate.

noise. The image on the bottom right demonstrates the influence of the noise: goal posts which are about 4.5 m away are observed at a distance of about 1 m.

9.8.3 Merging of Hypotheses

Performance constraints necessitate the merging of hypotheses. Merging simplifies the belief distribution by expressing the Gaussian mixture with fewer Gaussians. Ideally, hypotheses of the localization converge at the same pose: the actual robot pose. The Gaussian mixture at this pose can be easily expressed by one Gaussian.

The implemented merge algorithm is shown in listing 9.10. It compares the similarity of the tracks (line 6). A similarity metric, the simplified Mahalanobis distance, is used (see section 7.5.2). If the similarity is above a threshold the tracks are merged (line 7) using the merge algorithm proposed by Quinlan and Middleton [22]. After the merge, HypB has the combined properties of the two distributions. The weight of HypA is set to 0 which leads to the removal of HypA in the deletion step.

```
def mergeSimilarHypotheses():
    size = len(hypotheses)
    for i in range(size - 1):
        for j in range(i, size):
            hypA, hypB = hypotheses[i], hypotheses[j]
            if similarity(hypA, hypB) > threshold:
                hypB.merge(hypA)
                hypA.weight = 0
```

Listing 9.10: Hypotheses which are similar are merged.

9.8.4 Removing Hypotheses

Some Gaussians have little to no influence on the overall distribution and their weight is very low. In the deletion step, hypotheses with a weight below a threshold are removed.

9.8.5 Selecting the Final Robot Pose

Determining the global maximum of a Gaussian mixture distribution is not trivial and is computationally demanding. However, a simple approximation which is sufficiently appropriate for the given task is to use the hypothesis with the highest weight.

10 Evaluation

Figure 10.1: The evaluation setup: the robot is wearing the special markers and is tracked by the overhead camera system.

The overhead camera system (see section 2.2.3) is used to evaluate the developed models. However, the system is relatively new and the accuracy is not high enough to use the data as ground truth data. Also the robots' feet were modified before the evaluation. By removing the loadcells the closed-loop gait was turned into an open-loop gait. The current gait is not stable yet and the robot falls after just a few cm of walking. During the short time in which the robot is walking without falling it shows a completely different walking behavior and also completely different characteristics of the odometry error. This made it necessary to conduct the evaluation without walking robot and without any odometry information. To simulate a moving robot it was moved by hand. That means that in the prediction step no control was used. Instead a constant Gaussian noise was added in each time step to simulate the uncertainty which is introduced by moving the robot by hand.

Because of the inaccuracy of the ground truth system and the changed walking behavior too much emphasis cannot be placed on the quantitative analysis.

10.1 Runtime

The runtime of the models was evaluated during match-like conditions. Several obstacles were on the field and one ball.

The implemented models are fast and only put a minimal load on the system. This is especially important for the self-localization. The previous PF localization was a runtime bottleneck with a runtime between 50 and 100 ms.

Note that no explicit performance optimizations were done for the implemented models. The KF methods have proven to be very efficient and should be used for additional models.

runtime	**average**	**min**	**max**
local ball model	3 ms	< 1 ms	20 ms
local obstacle model	< 1 ms	< 1 ms	20 ms
self-localization	5 ms	1 ms	24 ms

Table 10.1: The runtime of the three different models. Note that the initialization of the modules takes a bit longer which explains the max runtime.

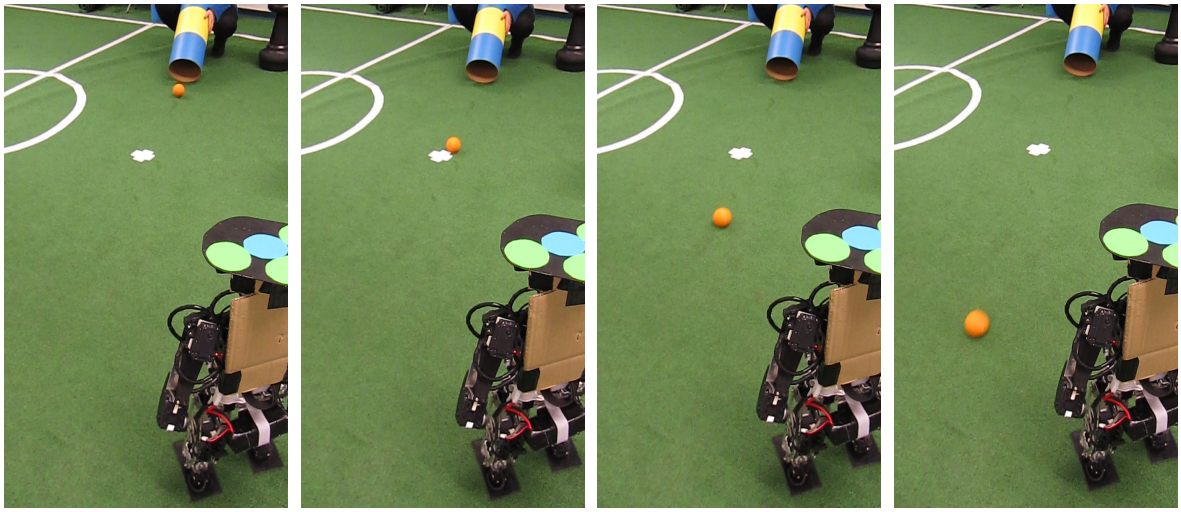


Figure 10.2: Conduction of the ball measurements. The ball was rolled through a pipe to simulate a kick of the ball.

10.2 The Ball Model

For the quantitative evaluation of the ball model the ball information from the ground truth system and the information from the robot were compared for different scenarios. The ball on the field is rolled through a pipe to simulate a kick (see figure 10.2). The robot observes the ball when it is in the robot's field of view. The real trajectory of the ball as recorded by the ground truth system is compared to the trajectory of the ball model. Three scenarios were tested:

Scenario I: The standing robot observes an area in front of it without moving the head. This is similar to the behavior of the goalie. The ball was always observed in this scenario.

Scenario II: The standing robot executes the standard behavior of the *GazeControl*. The ball was not seen in 8 out of 20 cases.

Scenario III: The walking robot which was stabilized by a human to avoid falling executes the standard behavior of the *GazeControl*. The ball was not seen in 9 out of 20 cases.

Figure 10.3 showcases some of the measurements. The results are satisfactory. However, in some cases the ball model has suboptimal performance as shown in figure 10.4. Theses problems look rather like a bug in the implementation than a general problem of the model.

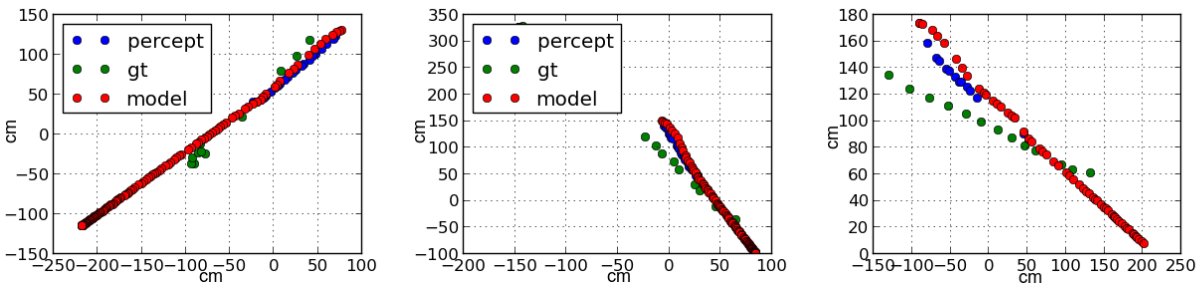


Figure 10.3: Examples of the ball model. The robot is the center of the coordinate system (0, 0). Scenario I: left; Scenario II: middle; Scenario III: right.

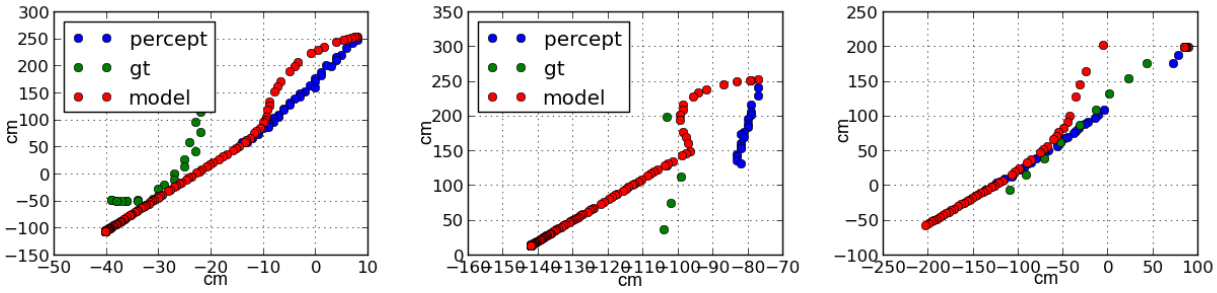


Figure 10.4: Problems of the ball model. Probably caused by a bug in the implementation.

10.3 The Obstacle Model

There is no quantitative evaluation of the local obstacle model. The overhead camera cannot track obstacles and is currently limited to only one robot.

Figure 10.5 shows an exemplary obstacle model. The upper two rows show clippings of the camera image. The recognition of the obstacles (red boxes) fluctuates strongly. The bottom row shows the initial model (left) and the model after a few percepts (right) As one can easily see, the model represents the real situation reasonably well. The two obstacles on the left are modeled as one.

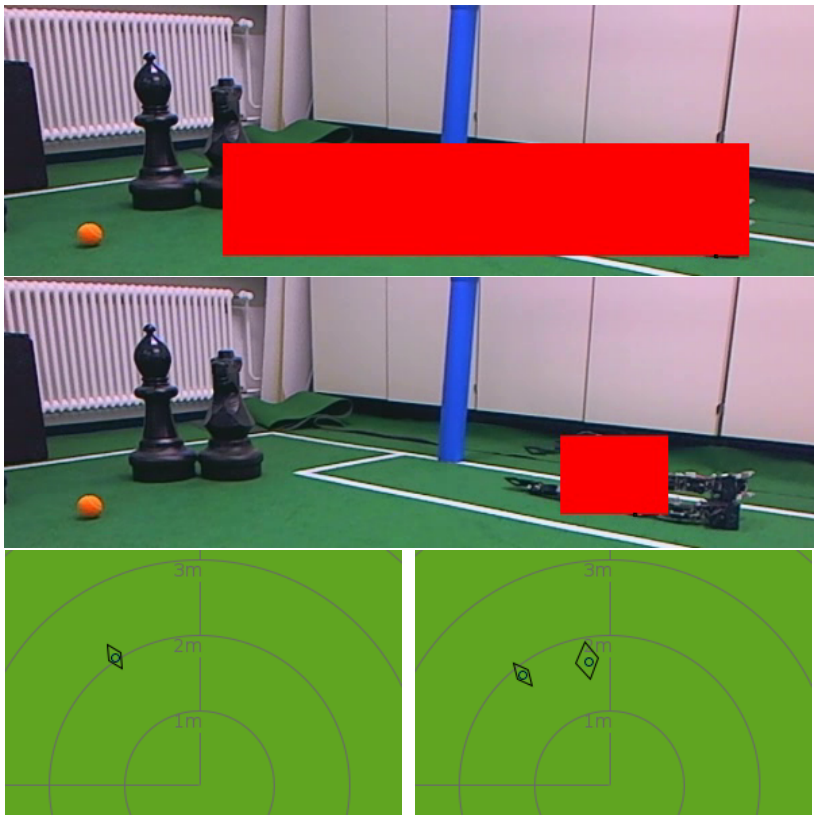


Figure 10.5: Top and middle: two of many changing obstacle percepts for the same scene. The red boxes are the recognized obstacles.
Bottom: initial model (left), and model after a few percepts (right).

10.4 The Localization

To evaluate the quality of the self-localization the robot pose from the ground truth system and the robot pose from the localization are compared and some quality statistics are calculated.

Table 10.2 shows the results of some localization runs. The runs and their evaluation are shown in figure 10.8 and figure 10.9.

	error min	error max	error mean	error std
run #1	19.2 cm	125.5 cm	59.7 cm	31.5 cm
run #2	16.4 cm	92.4 cm	33.34 cm	16.4 cm
run #3	15.3 cm	72.2 cm	35.5 cm	11.9 cm
run #4	5.1 cm	145.1 cm	43.3 cm	36.7 cm

Table 10.2: Localization performance. Note that no control was used. A run was between 2 and 3.5 m long.

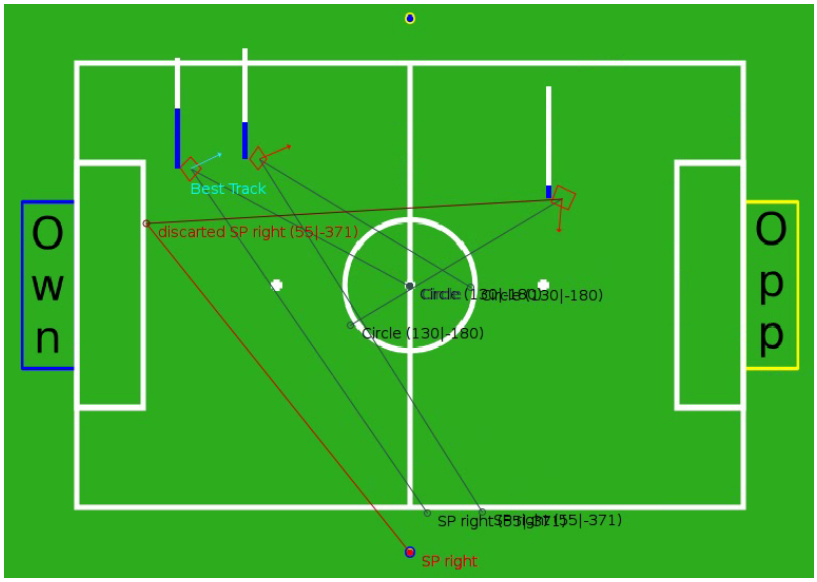


Figure 10.6: Typical scene of the localization.

Figure 10.6 shows a typical scene of the localization[1]. The robot has a few hypotheses. A hypothesis is shown as arrow with its corresponding uncertainty (red polygon). The blue-white bars represent the weights of the hypotheses. The more blue, the higher the weight. The gray lines represent the measurements for a given hypothesis.

Two hypotheses are close to the correct pose and have a high weight. The other hypothesis on the right was created by the triangulation process of noisy measurements. It has a low weight.

In general, there is almost always one hypothesis or more hypotheses near the correct pose. However, a common problem is that the selection of the best pose is to reactive, *i.e.,* the selected best pose jumps from one to another hypothesis close to the real pose. To counteract this, different weighting methods should be evaluated. A weighting method similar to the one of the ball model would probably increase the quality of the localization because the measurement likelihood has a more direct influence on the weight.

Another problem is the localization performance directly in front of the

[1] More pictures of the localization process are in the appendix.

goal. There are relatively few features and the unpredictable walking behavior leads to wrong correspondence choices for the goal posts. However, once the field line recognition is improved, the field-line matcher can be applied and the localization performance should improve.

10.4.1 Rejection of False-Positives

The false-positive rejection is shown in figure 10.6 and figure 10.3. The red lines connect rejected measurements with the corresponding landmarks. This process works quite well as the following example demonstrates. During the evaluation the two side poles were standing on the wrong side. The hypotheses which were close to the real robot pose rejected the side pole percepts because their measurement likelihood was too low. The other observations were used as usually.

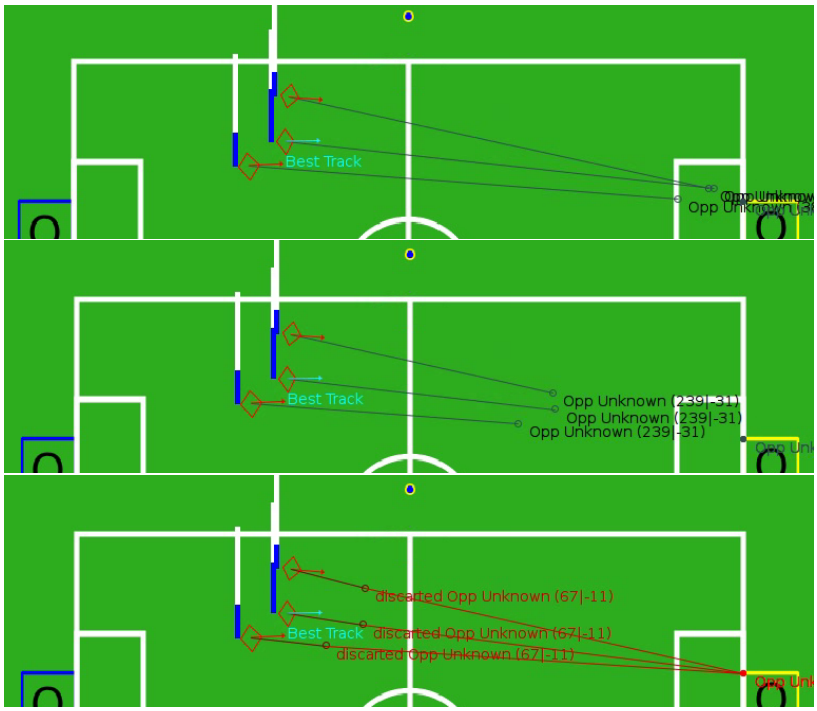


Figure 10.7: False-positive rejection. The robot moves its head and measures the goal at different positions. The third measurement is rejected because the measurement likelihood is too low.

10.4.2 Re-localization

Here the ability for re-localization is evaluated. The standing robot is teleported/moced from a known pose to an unknown pose. The camera is covered during the teleportation. This was repeated 20 times. A re-localization trial was aborted after 30 s if the robot was not yet re-localized.

	average	**min**	**max**
time for re-localization	4.1 s	1 s	8 s

Table 10.3: Time needed for re-localization after teleporting. Only the 15 successful out of 20 total re-localization trials were used.

The re-localization worked in 15 out of 20 cases. The re-localization works really well when the robot observes features which are used for the

sensor-resetting process (15 times). If no such landmarks are observed, the re-localization does not work (5 times) Because the hypotheses only use observations which have a high measurement likelihood most measurements are rejected and do not correct the hypotheses. A relaxation of the rejection criteria of measurements would improve the re-localization abilities. The potential poses created by the field line matcher would improve the re-localization abilities as well.

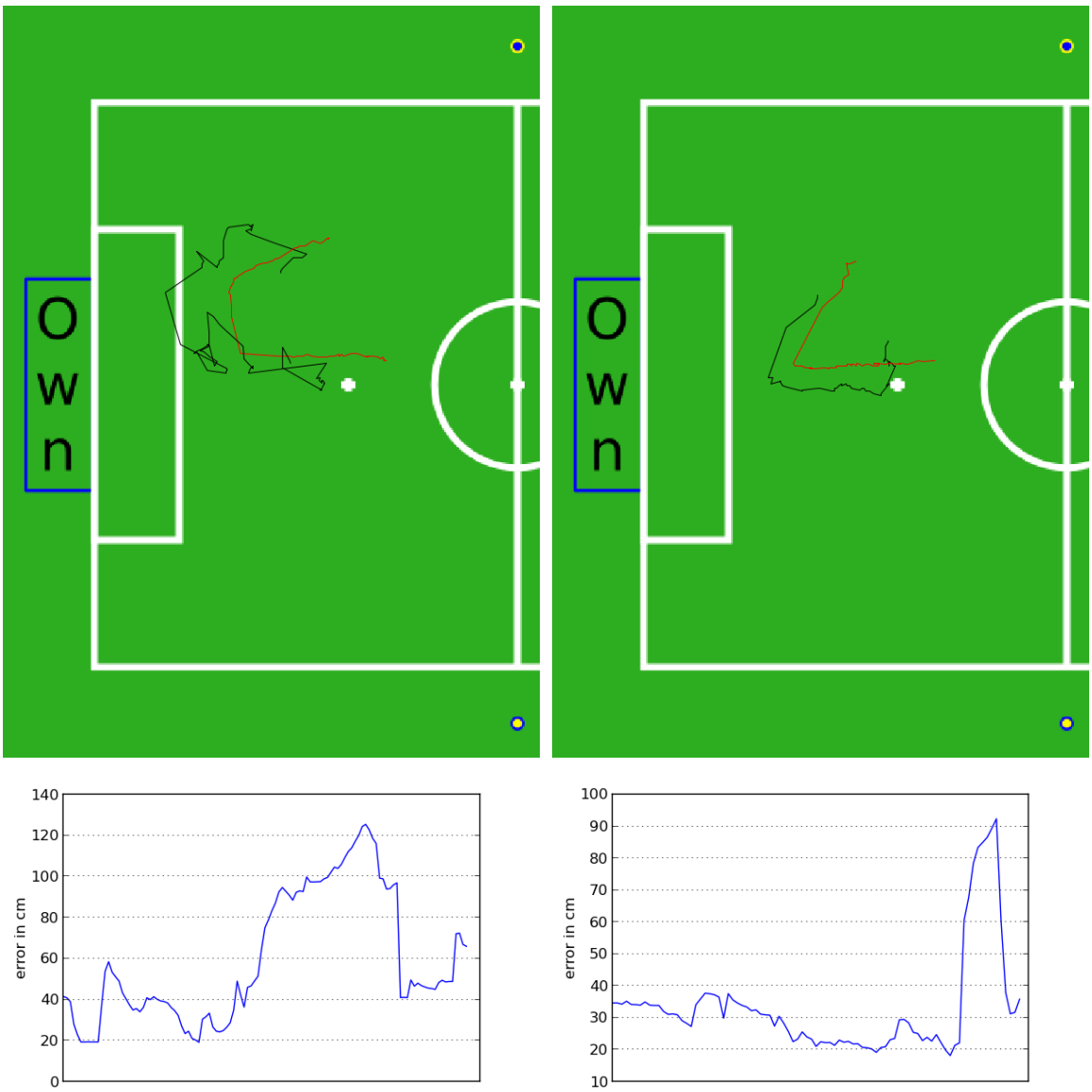


Figure 10.8: Localization: run 1 and run 2.
Red is the ground truth. Black is the belief of the robot.
Bottom: error in cm per time step.

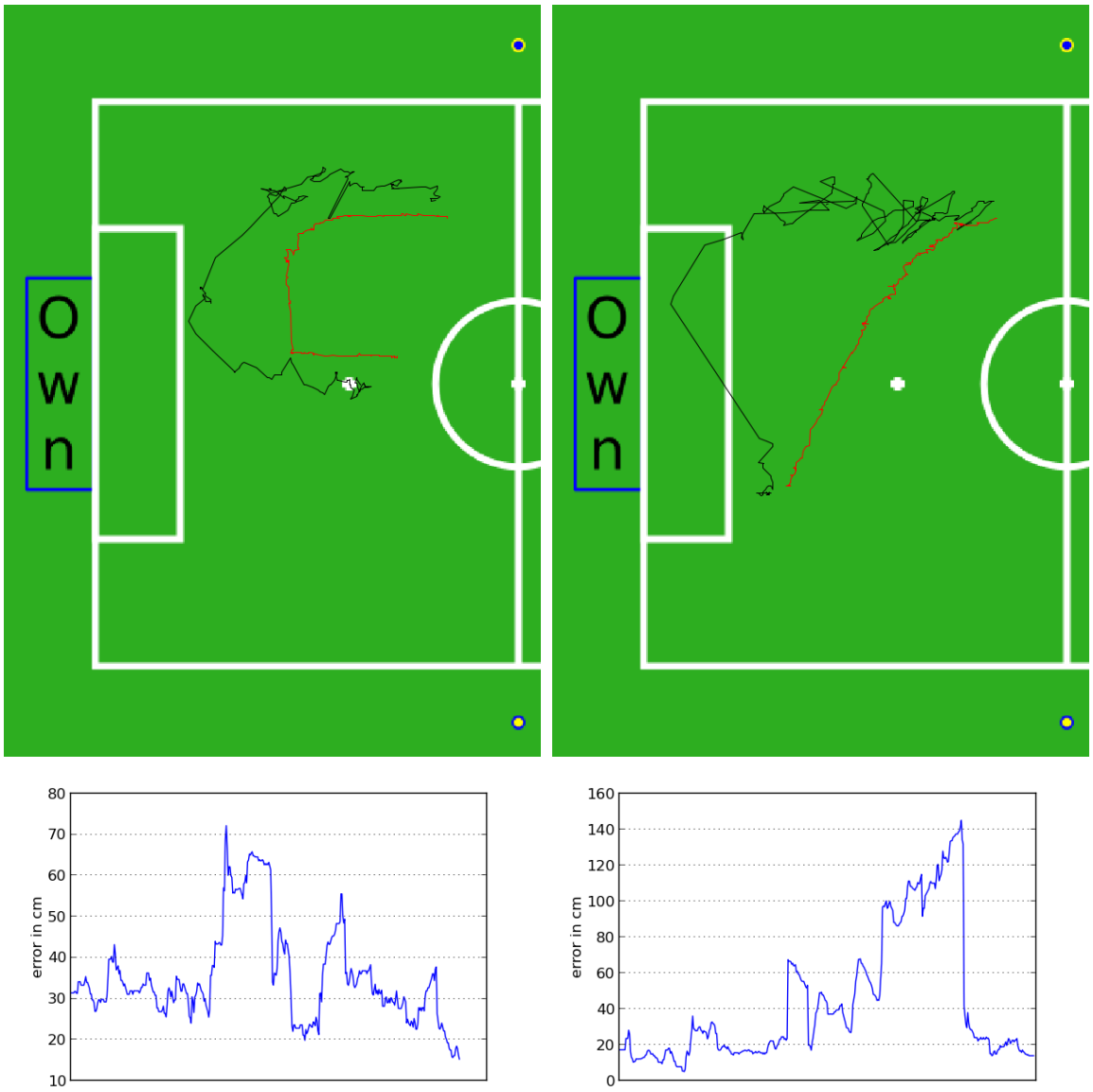


Figure 10.9: Localization: run 3 and run 4.
Red is the ground truth. Black is the belief of the robot.
Bottom: error in cm per time step.

11
Conclusion and Outlook

"When the facts change, I change my mind. What do you do, sir?"
– *John Maynard Keynes*

The goal of this thesis was to enable the FUmanoid robots to have understanding of the world in terms of the questions *"where am I"* and *"what's going on"*. Local models for the ball and for the obstacles were developed as well as a module for global self-localization. For all modeling tasks, *multi-hypothesis Kalman filters* were successfully used.

The MHKF framework has proven to be a viable alternative to the PF. Once a number of tools were in place (similarity metrics, merge algorithms, etc.), the additional expenses were moderate compared to the development of particle filters. The *update-merge-delete-add* cycle has proven to be particularly effective. The developed mechanism can be easily extended and applied to other models.

The previous PF localization created a runtime bottleneck within the system, whereas the current implementations of the filters are computationally inexpensive.

11.1 Future Work

Even though the developed models work well, there are many areas in which they can be improved upon.

11.1.1 General

COVARIANCE MATRICES

The *covariance matrices* are a central part of all implemented models. It is always difficult to measure and estimate the covariances. A better ground truth system would result in better estimates. Also, more sophisticated models for the noise or different representations might yield performance improvements for the models. The speed of the head movement was completely neglected but has an influence on the noise which should be considered in the future. The *autocovariance least-square* technique could be applied to better estimate the covariance matrices.

Bad odometry is a huge problem for all models. The gait of the robots should be improved with the criterion of less slippage. More accurate measurements of the noise properties should also be conducted.

NEGATIVE INFORMATION

Negative information, i.e., not observing a feature which should be ob-

served, could be applied throughout the models. It is a strong indicator for the quality of a hypothesis.

The field line matching algorithm has shown that the line detection should be improved. Also, the orientation of the projected field lines would allow the robot to judge the quality of the projection. Field lines which are orthogonal to each other on the field should also be orthogonal to each other after projecting them from the camera image into the Cartesian space. This feedback could be used to improve the projection.

The focus of this thesis was on local models and localization. The logical next step is to include more information from the team and create a global model similar to [28].

PARAMETER OPTIMIZATION

The implemented models have a huge number of parameters which are set by expert knowledge or sometimes trial and error. Optimization algorithms should be applied to optimize these parameters.

LOGPLAYER EXTENSION

The LogPlayer should be extended and the performance of modules should be checked automatically. Automatic regression tests should be used in the future.

The process model of the ball and the obstacles was simplified and should be improved in the future to incorporate the movement of the robot itself as well as the ball/the obstacles.

11.1.2 Localization

The line matching algorithm in section 9.4 has the useful property of mutual exclusion, *i.e.,* contradicting percepts are automatically filtered out. This algorithm should be extended to include the other landmarks. Another byproduct of the algorithm is the creation of potential robot poses. These could be used to spawn new hypotheses which would also improve re-localization abilities. See [7] for more.

ACTIVE LOCALIZATION

In some cases the agent has two symmetric hypotheses. It would be interesting to pursue an *active localization* approach here. The localization could command the camera to look at a position which resolves the symmetries. Also, the robot could stop walking and scan the environment if it is not localized properly.

RULE CHANGES 2013

The rules for the RoboCup 2013 change the environment in a few but important aspects. The side poles will be removed, and both goals will have the same color. This turns the field into a completely symmetric environment. The orientation of the robot pose is of utmost importance in such an environment. Possible adjustments for this scenario include:

- The localization must focus more on position tracking of the robot.
- New hypotheses must be created more conservatively, *i.e.,* new tracks must be created only around the best hypothesis.
- To resolve the symmetry, additional features such as a stationary goalie, and the ball position, can be used.

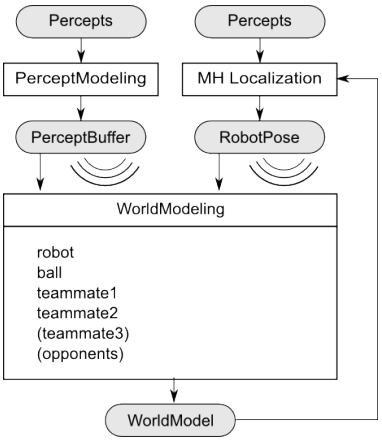


Figure 11.1: Draft of the architecture of the global world model. Arrows represent the local data flow. The radio waves represent the data which is transfered via WIFI.

11.1.3 Global Models

The next step should be to implement global models for obstacles and the ball. Also a global world model as suggested in [28] would make as as most of the building blocks are already in place.

Bibliography

[1] Eigen. URL `http://eigen.tuxfamily.org`.

[2] Sven Behnke, Marcell Missura, and Hannes Schulz. Nimbro teensize 2012 team description. Technical report, Rheinische Friedrich-Wilhelms-Universitt Bonn, 2012.

[3] I.N. Bronštejn, K.A. Semendjaev, G. Grosche, and E. Zeidler. *Teubner-Taschenbuch der Mathematik. [1].* Number v. 1 in Teubner-Taschenbuch der Mathematik. Teubner, 2003. ISBN 9783519200123.

[4] Allen B. Downey. *Think Stats.* O'Reilly Media, July 2011. ISBN 1449307116.

[5] Willow Garage. Opencv. URL `http://opencv.willowgarage.com`.

[6] GNU. Gnu scientific library. URL `http://www.gnu.org/software/gsl/`.

[7] Daniel Göhring, Heinrich Mellmann, Kataryna Gerasymova, and Hans-Dieter Burkhard. Constraint based world modeling. *Fundam. Inf.*, 85(1-4):123–137, January 2008. ISSN 0169-2968.

[8] Google. Protobuf. URL `http://code.google.com/p/protobuf/`.

[9] J.-S. Gutmann and D. Fox. An experimental comparison of localization methods continued. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2002.

[10] Steffen Heinrich. Development of a multi-level sensor emulator for humanoid robots. Master's thesis, Free University Berlin, 2012.

[11] Gregor Jochmann, Sören Kerner, Stefan Tasse, and Oliver Urbann. Efficient multi-hypotheses unscented kalman filtering for robust localization. In *RoboCup 2011: Robot Soccer World Cup XV*. Springer Berlin / Heidelberg, 2012.

[12] R. Kalman. On the general theory of control systems. *IRE Transactions on Automatic Control*, 4(3):110–110, 1959.

[13] S. Kohlbrecher, A. Stumpf, and O. von Stryk. Grid-based occupancy mapping and automatic gaze control for soccer playing humanoid robots. In *Proc. 6th Workshop on Humanoid Soccer Robots at the 2011 IEEE-RAS Int. Conf. on Humanoid Robots*, Bled, Oct. 26th - Oct. 28th 2011.

[14] J. Kuhn, S. Kohlbrecher, K. Petersen, D. Scholz, J. Wojtusch, and O. von Stryk. Team description for humanoid kidsize league of robocup 2012. Technical report, Technische Universität Darmstadt, 2012.

[15] Tim Laue and Thomas Röfer. Particle filter-based state estimation in a competitive and uncertain environment. In *Proceedings of the 6th International Workshop on Embedded Systems*. Vaasa, Finland, 2007.

[16] MathWorks. Matlab. URL `http://www.mathworks.de/`.

[17] Sharon B. McGrayne. *The Theory That Would Not Die: How Bayes' Rule Cracked the Enigma Code, Hunted Down Russian Submarines, and Emerged Triumphant from Two Centuries of Controversy*. Yale University Press, 2011. ISBN 0300169698.

[18] Heinrich Mellmann and Marcus Scheunemann. Local goal model for a humanoid soccer robot. In *Proceedings of the Workshop on Concurrency, Specification, and Programming CS&P 2011*, pages 353–360. Bialystok University of Technology, 2011.

[19] Heinrich Mellmann, Yuan Xu, Thomas Krause, and Florian Holzhauer. Naoth software architecture for an autonomous agent. In *Proceedings of the International Workshop on Standards and Common Platforms for Robotics (SCPR 2010)*, Darmstadt, November 2010.

[20] Hans P. Moravec. *Mind Children: The Future of Robot and Human Intelligence*. Harvard University Press, 1988.

[21] S Pinker. *The Language Instinct*. Harper Perennial Modern Classics, New York, 1994.

[22] Michael J. Quinlan and Richard H. Middleton. Multiple model kalman filters: a localization technique for robocup soccer. pages 276–287. Springer-Verlag, Berlin, Heidelberg, 2010. ISBN 3-642-11875-5, 978-3-642-11875-3.

[23] Thomas Röfer, Tim Laue, Judith Müller, Alexander Fabisch, Fynn Feldpausch, Katharina Gillmann, Colin Graf, Thijs Jeffry de Haas, Alexander Härtl, Arne Humann, Daniel Honsel, Philipp Kastner, Tobias Kastner, Carsten Könemann, Benjamin Markowsky, Ole Jan Lars Riemann, and Felix Wenk. B-human team report and code release 2011, 2011. Only available online: `http://www.b-human.de/downloads/bhuman11_coderelease.pdf`.

[24] Thomas Röfer, Tim Laue, and Dirk Thomas. Particle-filter-based self-localization using landmarks and directed lines. In *In RoboCup 2005: Robot Soccer World Cup IX, Lecture Notes in Artificial Intelligence*. Springer, 2005.

[25] Conrad Sanderson. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, 2010.

[26] Oleg Sushkov and William Uther. Robot localisation using a distributed multi-modal kalman filter.

[27] Stefan Tasse, Matthias Hofmann, and Oliver Urbann. On sensor model design choices for humanoid robot localization. *RoboCup 2012: Robot Soccer World Cup*, XVI, 2013.

[28] Stefan Tasse, Matthias Hofmann, and Oliver Urbann. Slam in the dynamic context of robot soccer games. *RoboCup 2012: Robot Soccer World Cup*, XVI, 2013.

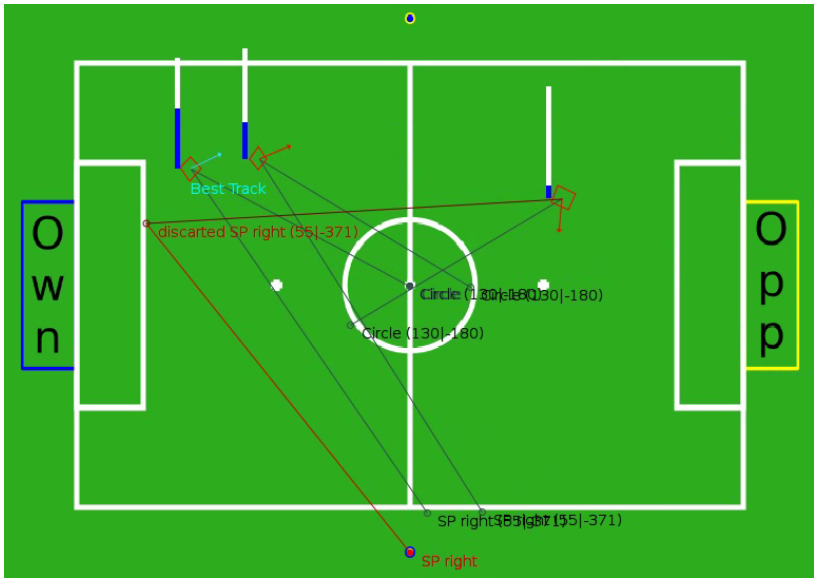
[29] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents series)*. Intelligent robotics and autonomous agents. The MIT Press, August 2005. ISBN 0262201623.

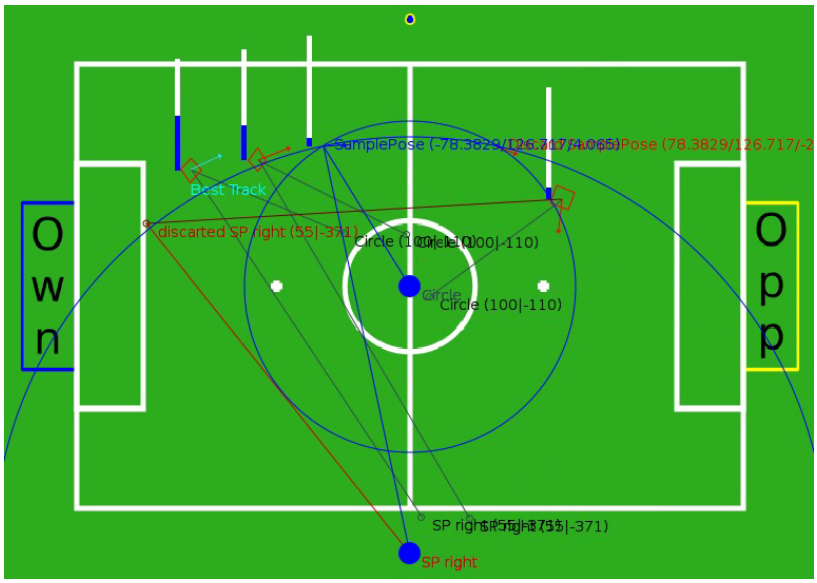
[30] J.L. Williams. Gaussian mixture reduction for tracking multiple maneuvering targets in clutter. Technical report, DTIC Document, 2003.

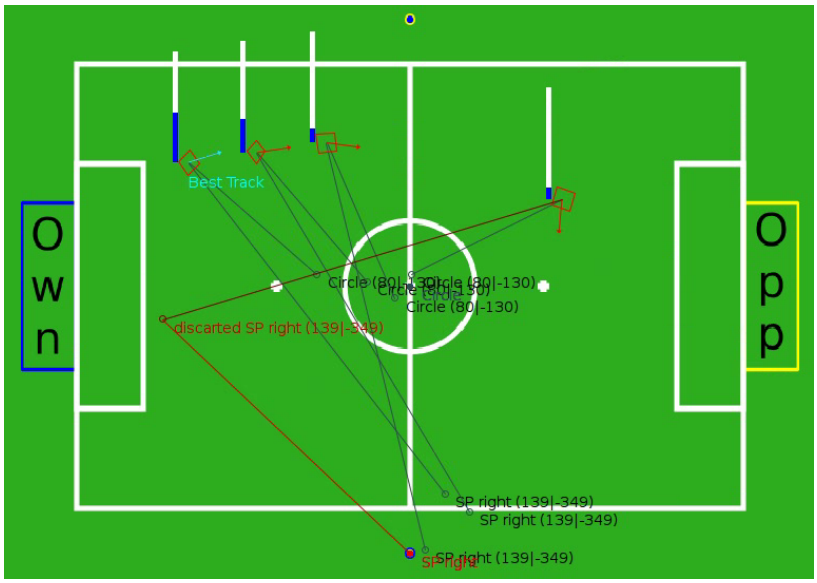
APPENDIX

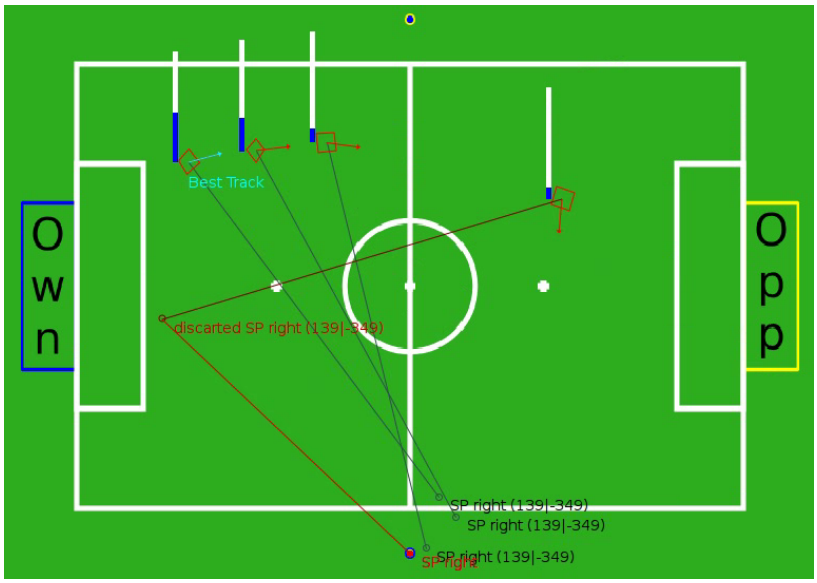
Localization Graphics

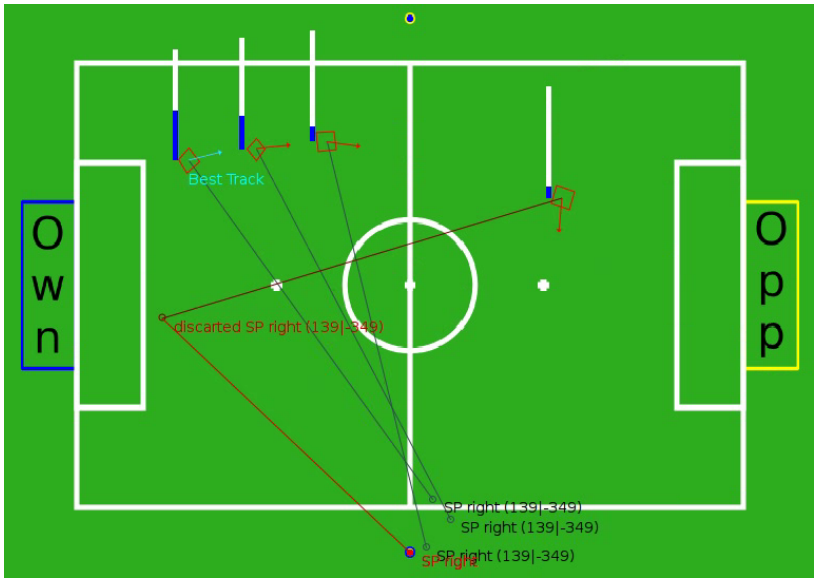
A short series of graphics that show the localization process is shown here.

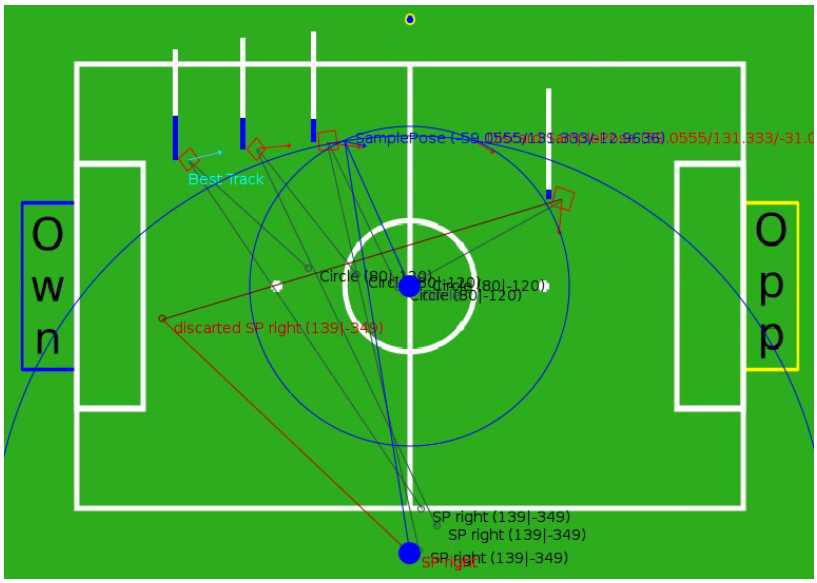


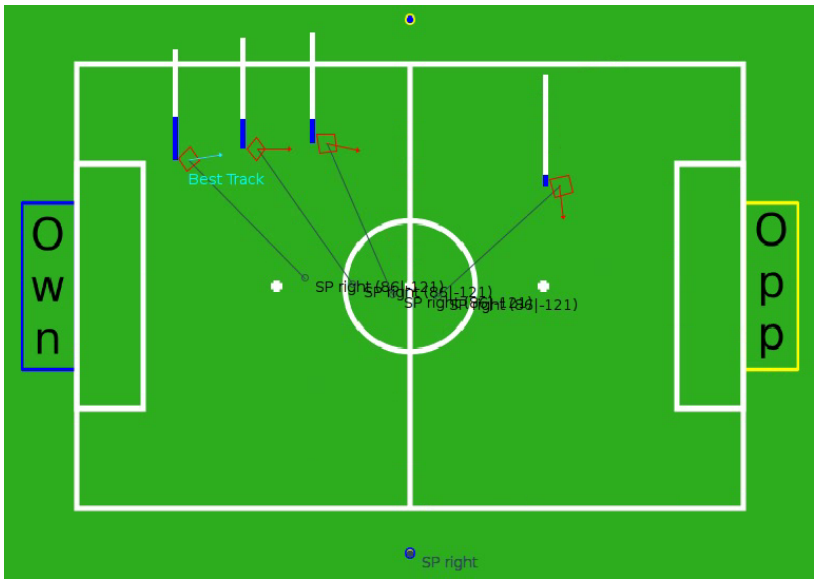


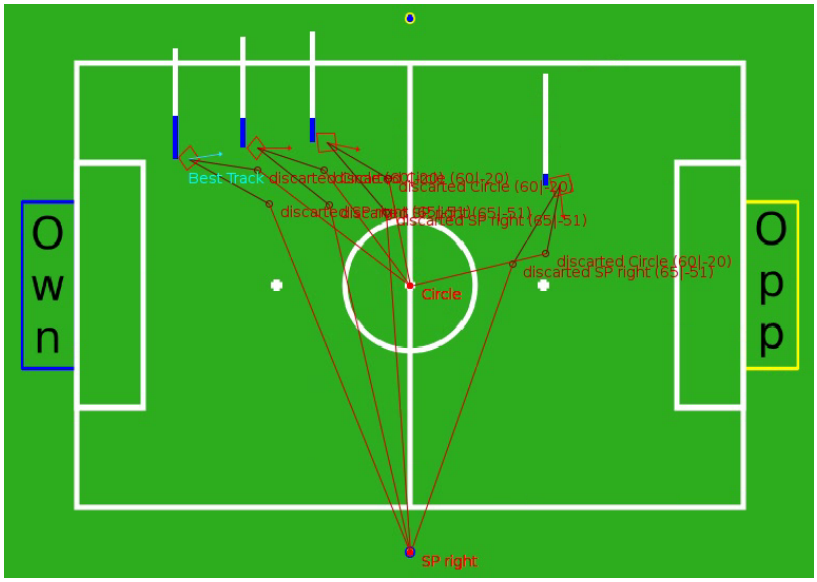

Own
Opp
Best Track
Circle (80|-130)
Circle (80|-130)
Circle (80|-130)
discarded SP right (139|-349)
SP right (139|-349)
SP right (139|-349)
SP right (139|-349)


Own
Opp
Best Track
discarded SP right (139|-349)
SP right (139|-349)
SP right (139|-349)
SP right (139|-349)


Own
Opp
Best Track
discarded SP right (139|-349)
SP right (139|-349)
SP right (139|-349)
SP right (139|-349)


Own
Opp
Best Track
discarded SP right (139|-349)
SP right (139|-349)
SP right (139|-349)
SP right (139|-349)
SP right


Own
Opp
Best Track
SP right


Own
Opp
Best Track
discarded Circle (60|-20)
discarded SP right (65|-51)
discarded Circle (60|-20)
discarded SP right (65|-51)
Circle
SP right

Declaration of Academic Honesty

I hereby declare that this master's thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given in the bibliography.

Stefan Otte Berlin, December 5th, 2012

Acknowledgement

I hereby want to thank the team FUmanoid for their help, their feedback, and their criticism. It was a lot of fun!