UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science

Priit Kallas

Probabilistic Localization of a Soccer Robot

Master's thesis (30 ECTS)

Supervisor: Konstantin Tretyakov, M.Sc.

Autor: ……. "….." mai 2013
Juhendaja: …… "….." mai 2013

Lubada kaitsmisele
Professor: …… "….." mai 2013

TARTU 2013

Contents

Introduction

Robot localization techniques attempt to solve the problem of estimating mobile robot position and orientation (pose) relative to an external frame of reference. Localizing the robot is a prerequisite step to smart decision making: before answering “Where am I going” and “How I should get there”, one needs to first have at least an estimation of “Where am I?” [1].

As robots are given ever more complicated tasks in natural and dynamically changing environments, the control algorithms must cope with the inherent uncertainty of such tasks. For example, the Google self-driving cars [20] must be able to navigate various streets and motorways in changing weather, road conditions and lighting, with other cars and pedestrians on the road that often behave unpredictably. It is impossible to model and predict such complex systems precisely, so statistical algorithms have been developed that explicitly account for the uncertainties and, instead of providing a single “best guess”, define *probability distributions* over all the possible states. These methods perform *recursive state estimation*, where data from previous states is taken into account along with control inputs and sensory data. This information is fused together in time so the robot can incrementally learn its location over several steps of movement and measurements.

The uncertainty of such complex systems arises from the fact that all models of the world are approximate. For example, one might express the kinematics of a car using a simple “bicycle model” where the two rear wheels are fixed and the two front wheels are turning and abide to *Ackermann steering geometry* [2], but this is only a crude approximation of the complex interplay of forces involved in the suspension, tires, etc. Statistical models explicitly acknowledge and take these model imperfections into account. Movement actions generally increase the uncertainty of the system about its location. Measurements, on the contrary, provide information and reduce uncertainty. Another major source of uncertainty comes from the imperfections of the various sensors used to get information about the environment. Most sensors have limited resolution, range and are susceptible to noise that needs to be filtered out. Popular sensors often used for localization purposes include ultrasonic sonars and lasers for distance measurement, RFID and barcode readers, as well as cameras employing *computer vision* algorithms. Additional useful information is often extracted from *odometer systems*, where actual motion of the robot is calculated from its actuators feedback. For example, given a two-wheeled *differential drive* [3] robot, the

relative motion of the robot can be calculated from measured wheel speeds. Of course, all measurements include noise and effects such as wheel slip are difficult to take into account. Since this error is cumulative, the location estimate obtained using odometry measurements alone quickly drifts from the real location and needs to be corrected with measurements.

For most complex robotics problems, the pose of the robot cannot be directly observed but instead needs to be inferred from various indirect sensor measurements. Often a global map of the environment is provided that is used as the reference frame for localization. Robot builds local maps of its surroundings and tries to match these to the global map to figure out its location. The map must contain some features that the robot can detect. For example, one might use special *geometric beacons* that are reliably and uniquely detected by the sensors [1]. While such an approach utilizing artificial beacons makes solving the problem easier, it requires modifying the environment. Another similar approach is to use naturally occurring, but still easily distinguishable *landmarks*, such as doors and corners of an office building [4]. This technique is more flexible, but makes the localization problem harder as it might be tricky to, for example, tell two similar doors or room corners apart. Such problems are handled by algorithms that support *multi-modal* beliefs, i.e. the ability to track several distinct hypotheses simultaneously, hoping to eventually converge and resolve the ambiguities. Probabilistic methods for mobile robot localization have proven to work well for complex, dynamically changing systems. They are robust in the face of sensor limitations and model approximations and are often capable of real-time tracking without being prohibitively computationally expensive.

The goal of this thesis is to explore and compare various techniques for real-time autonomous robot localization using a camera as the main source of information, and to develop a working positioning solution for a soccer-playing robot. The main focus lies on two techniques: the *Kalman filter* and the *particle filter*. In the first part of this thesis, general statistical background is introduced, employing the ideas of Bayes filtering. Then the dynamics of the autonomous soccer-playing robot and its environment are described and used as a practical example of robot localization. Several different positioning algorithms are presented, implemented, tested and compared in the simulator as well as on a physical robot.

1 Bayesian filtering

The theoretical treatment of the approaches and algorithms presented in this section is generic for all localization problems. Later chapters will investigate in depth the special case of a soccer-playing autonomous robot that uses its camera and feedback from wheels, keeping track of landmarks on the field to try and guess its pose on the field. The following statistics background and Bayes filter chapters are largely based on *Probabilistic robotics* by S. Thrun, W. Burgard and D. Fox [5].

1.1 Statistics background

The following section is meant to provide the basic statistical ideas and vocabulary to help in understanding the material that will follow.

In probabilistic robotics, quantities such as sensor measurements, control inputs and the state of the robot are represented as *random variables*, which can take multiple values according to specific probabilistic laws. We say there is a probability p for a random variable X to take the value *x* and write it as p(X = x).

For example, consider a coin flip. There is an equal chance of getting either heads or tails, $\mathrm{p}(\mathrm{X} = \mathrm{heads}) = \mathrm{p}(\mathrm{X} = \mathrm{tails}) = \frac{1}{2}$. Probabilities for all possible outcomes must sum to one: $\Sigma \mathrm{p}(\mathrm{X} = \mathrm{x}) = 1$.

Techniques described in this thesis deal with continuous random variables, in which case, the probability for a random variable to take any particular value is actually zero, hence it is customary to instead speak about *probability density* at a given value and the corresponding *probability density function* (PDF). One of the most common probability distributions is the one-dimensional Gaussian normal distribution with mean μ and variance σ^2 usually denoted as $N(x; \mu, \sigma^2)$ and given by:

$$\mathrm{p}(\mathrm{x}) = (2\pi\sigma^2)^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}\frac{(\mathrm{x}-\mu)^2}{\sigma^2}\right\}$$ (see Figure 1).

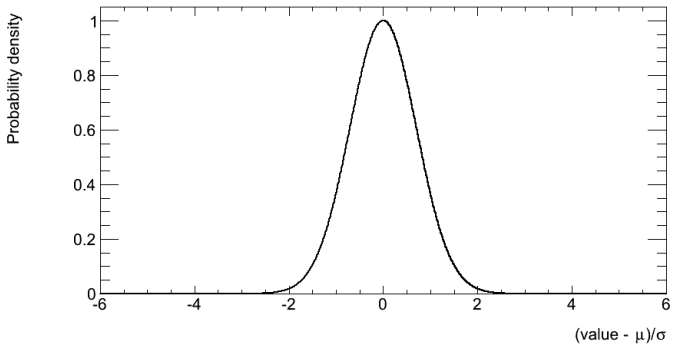


Figure 1. Gaussian normal distribution.

When working with multidimensional data, it makes sense to speak of *random vectors*. The distribution of a continuous random vector is specified as a multidimensional density function. Again, one of the most common distributions here is multivariate normal distribution:

$$p(x) = \det(2\pi\Sigma)^{-\frac{1}{2}}\exp\left\{-\frac{1}{2}(x-\mu)^{T}\Sigma^{-1}(x-\mu)\right\} \quad (1)$$

An example multivariate distribution is shown in Figure 2. A probability density function must always integrate to one: $\int p(x)dx = 1$.

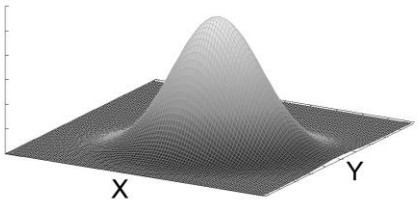


Figure 2. Example multivariate normal distribution visualization.

An important notion is the *conditional (in)dependence* of random variables. Let (X, Y) be a two-dimensional random vector. Its components, X and Y are, of course, themselves random variables. However, as they are related to each other, the probability distribution of X may depend on whether we know something about Y or not. In particular, if we know that Y = y, our knowledge of X is represented by a *conditional probability distribution* given by:

$$p(x \mid y) = p(x, y) / p(y) \quad (2)$$

If p(x | y) is the same for all y, we say that X and Y are *independent random variables*. It is possible to show that this implies p(x, y) = p(x)p(y) [5].

1.1.1 Bayes' rule

By modifying Equation (2) slightly, we obtain Equation (3), known as the Bayes' rule [6].

$$p(x \mid y) = \frac{p(y \mid x)\, p(x)}{p(y)} \quad (3)$$

The Bayes rule provides a mechanism for inferring the robot's position (X) from measurements (Y). The probability p(x) is the *prior probability distribution*, usually the last state of the robot for our purpose, and y is the *data*, usually acquired from a sensor measurement. The probability p(x | y) is called the *posterior probability distribution* over X that is derived from the previous knowledge of X incorporating data y. The "inverse" conditional probability p(y | x) specifies the likelihood of observing data y given state x and thus describes how state variables X cause sensor measurements Y.

The importance of properly taking into account prior beliefs is illustrated well in the famous *false positive paradox*. Consider a case of testing for cancer [7]. Say we know that 1% of the population has a specific cancer and there is a test that correctly detects it 80% of the times and incorrectly results positive test result 10% of the times. This means that 20% of the times, it does not detect cancer when the person really is sick and 90% of the times correctly finds that a person does not have cancer. Bayes rule provides a way for calculating the actual odds of having cancer when the test comes back positive.

The chance of true positive is the chance that you really have cancer times the odds that the test catches it which in our example calculates as 1% * 80% = 0.008. Chance of false positive is similarly 99% * 10% = 0.099. The chance of an event is the number of ways it could happen given all possible outcomes. Chances of getting a true positive is 0.008 and chance of getting any positive result is the true positive plus the false positive 0.008 + 0.099 = 0.107. Thus the chance of actually having cancer when getting a positive test result is desired event probability divided by all possibilities which

calculates to only 0.008 / 0.107 = 7.5%. Even though the accuracy of the test seems to be 80%, the actual chance of having cancer given example statistics is quite low even when getting positive test results.

1.1.2 Expectation and covariance

The *expectation* (also known as expected value, mean or first moment) of a random variable X is calculated as the weighted average of all possible values the random variable can take on. This is calculated for discrete and continues as given by Equations (4), (5) respectively.

$$E(X) = \Sigma\, x\, p(x) \tag{4}$$

$$E(X) = \int x\, p(x)\, dx \tag{5}$$

Expectation can be interpreted as the long-run average of many independent repetitions of an experiment, such as dice-rolling. For example a standard fair dice with face values 1-6, the expected value is 3.5 as that is the value one is likely to get when averaging the face values for many rolls of the dice:

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

From above, we can define *covariance* as $\mathrm{Cov}[X] = E[X^2] - E[X]^2$, which measures the squared expected deviation from the mean. A multivariate normal distribution $N(x; \mu, \sum)$ has a mean of μ and its covariance is defined using covariance matrix $\sum_{ij} = Cov[x_i, x_j]$ which expresses the pairwise covariance of the variables in the random vector.

1.2 Bayes filters

1.2.1 Intuitive explanation

The following section uses an example from *Probabilistic robotics* [5]. Imagine a robot in a one dimensional hallway with three indistinguishable doors. The robot can move forwards or backwards and can sense with some certainty whether is currently next to a door or not. Initially it does not know its location in the hallway but it knows that it is heading in the positive direction and it also has a map of the hallway with door

positions (the map). The goal of the robot is to find out where it is in the hallway using its map, door sensor and movement commands. Since the robot motors are not perfect, moving introduces some uncertainty into the system.

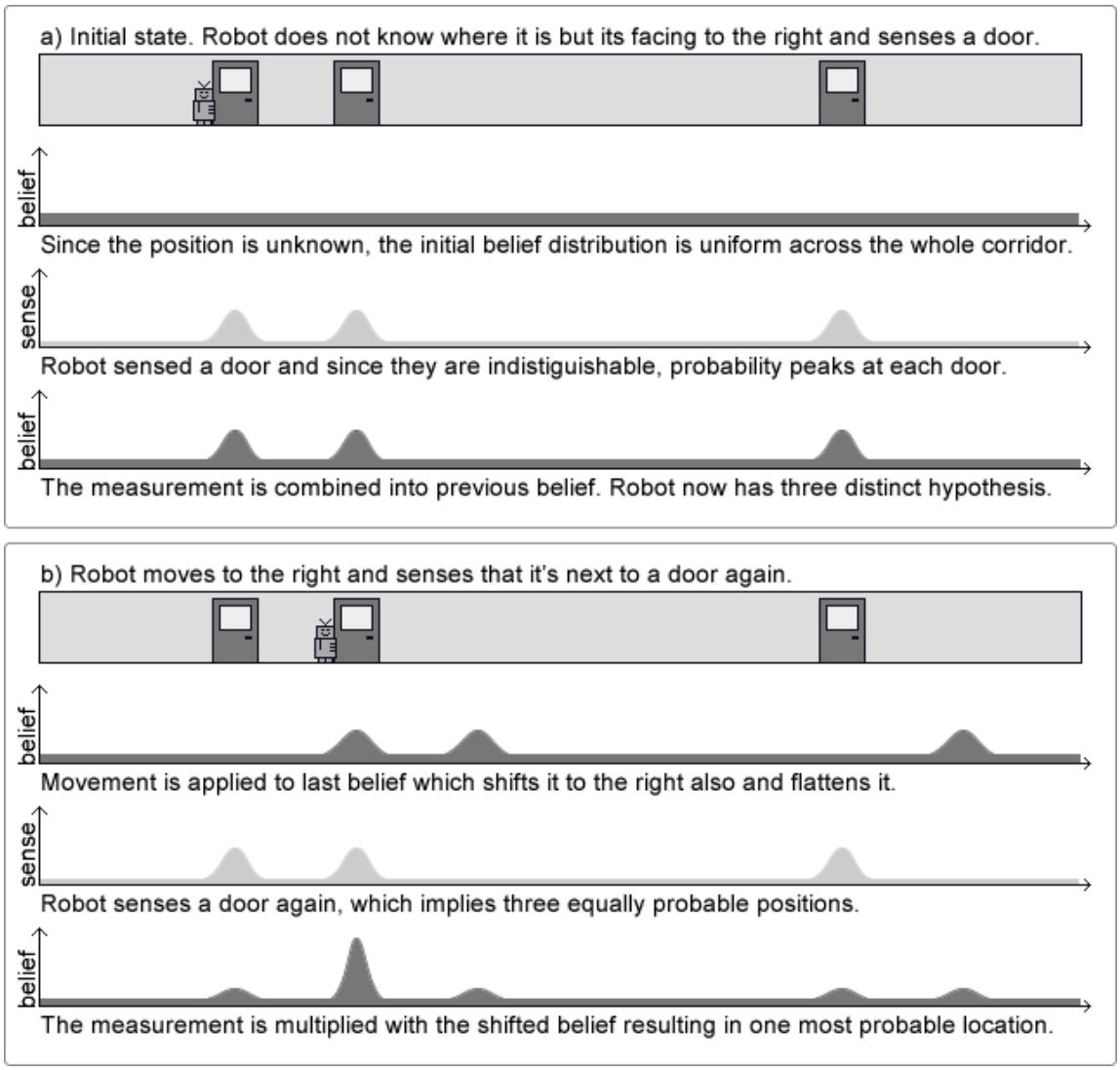


Figure 3. Example of probabilistic localization [5].

See Figure 3 for visualization of the example. To begin with in step a), the robot does not know anything about its location so the initial belief is uniform across the whole map of its environment represented by the first darker flat belief graph. The real position of the robot is indicated in the image. It correctly detects that it is next to a door. This observation is shown on the second (light gray) sense plot. Since the doors are indistinguishable, detecting a door implies that with equal probability, the robot may be

next to any of the three doors in the corridor. This sensory information is then fused into the existing flat prior belief resulting in a distribution with three bumps indicating that the robot knows it is most likely next to one of the doors but it is unsure which one.

Next in step b), the robot executes a command to move forwards, say, four meters, which happens to be the same as the gap between the first two doors. The same movement command is applied to the previous belief, shifting the distribution forward by the same amount. Since the movement is expected not to be perfect - the motors take some time to accelerate, wheels might be slipping etc., this operation reduces the confidence of the previous belief and thus the “hills” become a bit flatter. The robot now detects that it is again next to a door. Since it still doesn’t know which door it is observing, the sense plot is the same as in the first step. But when we multiply this sensory data with the prior distribution according to the Bayes rule, we can see that one of the measurements coincides with one of the prior estimates. Combining the prior and measurement distributions gives us one high peak, which correctly corresponds to the real location of the robot.

Different algorithms discussed later will use slightly different dynamics and ways to represent probability distributions, but the general idea of fusing information from previous steps with measurement data and control inputs to incrementally improve location estimate is the same for all of them and form the foundation of probabilistic localization.

1.2.2 Controls and measurements

The two main interactions between the robot and its environment are the robot’s control actions and its measurements of the surroundings via the sensors. While these are often performed simultaneously, it helps to treat them separately algorithmically.

Through perception, the robot gets information about the state of its surroundings which is often noisy and limited to a small region in the vicinity of the robot. There are many ways of sensing the environment, ranging from a simple tactile switches on the perimeter of the robot, triggering of which would indicate being close to a wall, to range sensors, radar and laser systems, and computer vision. As sensors usually have some delay required for acquiring and processing of information, they provide information about the state of a few moments ago. The result of perception (observation) is a *measurement* or, more often, a set of measurements. This measurement data at time t is denoted as z_t.

Control data carries information about the change in state of the robot in relation to the environment and is denoted by u_t at time t. For mobile robots, a typical example would be setting the robot velocity in certain direction. For example, if we command the robot to move forward at 2 meters per second then we expect it to change its pose by 6 meters in 3 seconds. The problem with this approach is that the powertrains of robots are not perfect and are often subject to complex dynamics that is hard to model accurately. For example, one cannot expect the robot to instantly accelerate to the requested speed and keep it at exactly that.

For this reason data is often extracted from odometry, where the actual rotational velocities of the wheels are measured using encoders and combining this data and the robot dynamics model, one can calculate the relative pose changes at each step. While this sort of input is actually sensory measurement data, it is often used as control data in localization algorithms. This approach is also subject to noise and unpredictable events, such as wheel slip, but it is usually more accurate and simpler to implement than predicting state transitions relying only on control inputs. As the odometry error is cumulative, it will eventually drift from real world state so some measurement data is required to get the localizer back on track. The perception step usually extracts new information about the environment and improves the robot's estimate of its whereabouts, whereas motion tends to lose precision due to the inherent noise.

1.2.3 Markov assumption

State x_t is called *complete* if the knowledge of any other past states, measurements and controls would not help us make any better predictions about the future than would just knowing x_t. Thus, state x_t is a sufficient summary about everything that has happened up to time t regarding the robot. This concept is called the *Markov assumption* and, while it is generally not feasible in the real world, it is a useful approximation to employ in order to make the algorithms computationally viable. This way, only the last state and the current control and measurement information needs to be considered for predicting future states.

Assuming that the robot state is complete and exploiting the conditional independence assumption (which states that certain variables are independent of others knowing a third set of conditioning variables), we can denote the *state transition*

probability $p(x_t \mid x_{t-1}, u_t)$ which means that the probability distribution of the robot state at time t depends on the state at last timestep x_{t-1} and the control input u_t.

The *measurement probability* $p(z_t \mid x_t)$ defines which measurements z_t are generated from the state x_t, meaning that measurements can be thought of as noisy projections of the state.

Remember that we denoted the state of a robot as x_t, the control data as u_t and measurements with z_t at time t. Figure 4 visualizes the evolution of states driven by control data and measurements.

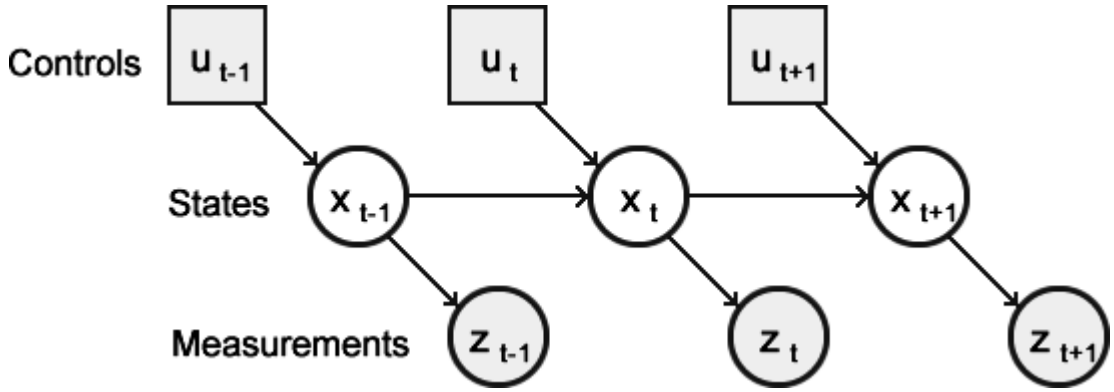


Figure 4. Dynamic Bayesian network characterizing the evolution of state through controls and measurements [5].

1.2.4 Belief, prediction and correction

Belief is an important concept of probabilistic robotics which represents the robot's internal knowledge about the state of the environment. Often the pose of a robot cannot be measured directly but rather needs to be inferred from the data that can be observed such as the distance and angles to goals on a soccer field. A belief *bel* is a conditional probability distribution that assigns a probability $bel(x_t)$ to each possible hypothesis x_t of the real state, conditioned on all past controls and measurements:

$$bel(x_t) = p(x_t \mid z_{1:t}, u_{1:t}) \tag{6}$$

Sometimes it is useful to calculate the posterior after incorporating current command u_t but before incorporating measurements from current time z_t. This is called a *prediction* as denoted by Equation (7). Calculating $bel(x_t)$ from $\underline{bel}(x_t)$ is called *correction* or the *measurement update*.

$$\underline{bel}(x_t) = p(x_t \mid z_{1:t-1}, u_{1:t}) \quad (7)$$

1.2.5 Bayes filter

The Bayes filter update rule presented in Algorithm 1 is the most general algorithm for calculating beliefs. As input, it takes the belief of the last step bel(x_{t-1}), the control input u_t and measurement data z_t and returns the new belief bel(x_t) at time t.

1. bayes_filter (bel(x_{t-1}), u_t, z_t):
2. for (every x_t) {
3. $\underline{bel}(x_t) = \int p(x_t \mid u_t, x_{t-1}) bel(x_{t-1})\, dx_{t-1}$
4. $bel(x_t) = \eta p(z_t \mid x_t) \underline{bel}(x_t)$
5. }
6. return bel(x_t)

Algorithm 1. Bayes filter pseudo code implementation [5].

The algorithm has two main steps. First, on line 3, it uses the control input u_t to transition the state probability distribution from x_{t-1} to x_t. As this does not take measurement into account yet, this is called the *control update* or *prediction*. The transitioned probability distribution is then multiplied with the belief of last step and integral sum of this product is found, resulting in prediction of the state based on last state belief and control input.

The second part on line 4 is called the *measurement update* where the prediction calculated on the previous line is multiplied by the probability that the measurement z_t may have been observed given state x_t. Requirement of a probability distribution was that it has to integrate to one. To achieve that, the product at line 4 needs to be normalized with the normalization constant η, which is chosen so that the resulting distribution is normalized.

After processing each posterior state x_t, the modified belief bel(x_t) is returned. To get started, the algorithm requires initial belief bel(x_0) which is usually initiated as a point mass centered around known value of x_0, or as a uniform distribution over the entire domain of x_0 if the initial state is unknown. This generic Bayes filter implementation can generally only be used for simple estimation problems as one needs to either be able to perform the integration on line 3 and multiplication in line 4 in closed form or be limited to finite state spaces so the integral in line 3 is a finite sum.

Methods discussed later will use slightly different approach to get around these limitations and are more computationally effective, but do so by losing generality and rely on different assumptions regarding measurement and state transition probabilities, initial belief and approximations used. Also the implementation complexity of various algorithms varies. For example, the dynamics of state transition probability $p(x_t \mid u_t, z_t)$ and measurement probability $p(z_t, x_t)$ can be hard to accurately model. This is one of the reasons why the *particle filter* approach described in section 1.4 has become popular.

1.3 Kalman filter

1.3.1 Gaussian filters

Gaussian filters are a popular family of recursive state estimators implementing the Bayes filter for continuous spaces. The main idea of this group of filters is to represent probability distributions using multivariate normal distributions given by the following equation:

$$p(x) = \det(2\pi\Sigma)^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\} \quad (8)$$

Using this approach enables characterizing the distribution using just two parameters, the mean μ and covariance matrix Σ. The mean has the same dimensionality n as the state x, while the covariance is a symmetric quadratic matrix of size $n \times n$.

While this representation is efficient to calculate, it has important limitations in that the shape of the distribution can only be that of a normal distribution. Simple Gaussian filter implementations are *unimodal* meaning they can represent a single maximum. This is suitable for a wide range of localization tasks where usually the starting position is known and the filter focuses around a single true state with relatively small uncertainty but will not work that well in complex *global localization* problems where it might be necessary to pursue several different hypotheses. The most basic implementation also works well only for *linear systems*. Extensions to the basic Gaussian filter combine several normal distributions to represent multimodal posteriors and can handle nonlinear systems at the expense of increased implementation and computational complexity.

1.3.2 Introduction

The Kalman filter (KF) is a popular Gaussian filter invented by Swerling (1958), Kalman (1960) and Bucy (1962). It represents beliefs using the mean and covariance of their Gaussian probability functions.

In addition to Markov assumption (state contains everything to make the best possible predictions), it must also hold that the state transition probability $p(x_t \mid u_t, z_t)$ and measurement probability $p(z_t \mid x_t)$ are both linear functions of their arguments with added Gaussian noise where x is the state and u the control input, both of which are vertical vectors:

$$x_t = A_t x_{t-1} + B_t u_t + \varepsilon_t \tag{9}$$

$$z_t = C_t x_{t-1} + \delta_t \tag{10}$$

The ε represents the Gaussian noise associated with uncertain state transition and has the same size as the state vector. Systems that meet these assumptions are called *linear Gaussian systems*.

In the equations above, A_t and B_t are matrices where A_t is a square matrix with dimensions n × n, where n is the size of the state vector x_t, B_t is of size n × m where m is the size of control vector u_t and C_t is of size k × n where k is the dimension of the measurement vector z_t. Mean of movement noise ε_t is zero and its covariance is denoted R_t. Vector δ_t describes measurement noise with also a mean of zero and covariance Q_t. When the transition and measurement functions are defined this way and the initial belief $bel(x_0)$ is normally distributed, it holds that the posterior $bel(x_t)$ is also always a Gaussian [5].

1.3.3 Algorithm

1. kalman_filter (μ_{t-1}, Σ_{t-1} u_t, z_t):
2. $\bar{\mu}_t = A_t \mu_{t-1} + B_t u_t$
3. $\bar{\Sigma}_t = A_t \Sigma_{t-1} A_t^T + R_t$
4. $K_t = \bar{\Sigma}_t C_t^T (C_t \bar{\Sigma}_t C_t^T + Q_t)^{-1}$
5. $\mu_t = \bar{\mu}_t + K_t(z_t - C_t \bar{\mu}_t)$
6. $\Sigma_t = (I - K_t C_t)\bar{\Sigma}_t$
7. return μ_t,Σ_t

Algorithm 2. Kalman filter implementation [5].

The linear Kalman filter is given in Algorithm 2. Formally, the algorithm can be derived by substituting the appropriate distributions into the generic Bayes filter algorithm (Algorithm 1). We do not provide all the formal details (see [5] for in-depth discussion) and instead give a brief intuitive explanation.

Being recursive, it requires the mean μ_{t-1} and covariance Σ_{t-1} of the last step as input. In addition, it uses the current step command input u_t and measurement data z_t. Lines 2, 3 calculate the predicted new mean and covariance based on just the control input. Line 2 applies the state transition step to the previous mean along with the control input resulting in belief $\underline{bel}(x_t)$ a step later but before incorporating measurement. On line 3, the state transition function is also applied to the covariance from the last step where A_t is multiplied into the covariance twice as covariance is a quadratic matrix. These two lines represent the *prediction step*.

Line 4 calculates matrix K, referred to as the *Kalman gain* which specifies how reliable the measurement is thought to be and thus at which degree should it be incorporated to the posterior result. A high gain mean that measurement data is followed more precisely which makes the filter more responsive but also more subject to noise in the measurement data. Low gain on the other hand relies more on the model and random fluctuations in the sensor data affect it less at the expense of being less responsive.

Lines 5 and 6 calculates the new mean and covariance, deriving them from the predicted mean, measurement data and Kalman gain. The predicted mean is adjusted by the factor of how much the current measurement z_t differs from expected measurement $C_t\bar{\mu}_t$ multiplied by the Kalman gain. This difference between the actual and expected measurement is called *innovation*. Line 6 adjust the covariance with the information gain received from measurement data. These lines represent the *update step* of a Kalman filter.

Kalman filter is computationally quite efficient. The performance of it is mainly dictated by the inversion operation on line 4 and the multiplications on lines 3, 6. The inversion has complexity of $O(k^{2.4})$ where *k* is the dimension of the measurement vector z_t [8]. Often the measurement space is lower dimensional than the state space in which case the multiplication complexity starts to dominate.

1.3.4 Visualization

Figure 5 illustrates how the Kalman filter incrementally applies movement commands and sensor measurement to produce a new estimation at each step for a simple one dimensional problem such as the robot in the corridor example explained in section 1.2.1.

We start with an initial prior belief shown in Figure 5.a (dark plot) and a measurement received from the sensors in lighter distribution. These are combined in Figure 5.b with a new mean around the center of the prior and sensed data but the resulting Gaussian has less variance (is narrower and taller) which is the result of information integration is Kalman filters. Figure 5.c visualizes the robot moving to the right which results in the belief also shifting to the right and having larger variance (wider and shorter) resulting of Kalman filter lines 2 and 3 and the fact that movement action is associated with some noise and uncertainty. Figure 5.c shows another measurement which largely coincides with the existing belief resulting in even more confident posterior with rather small variance.

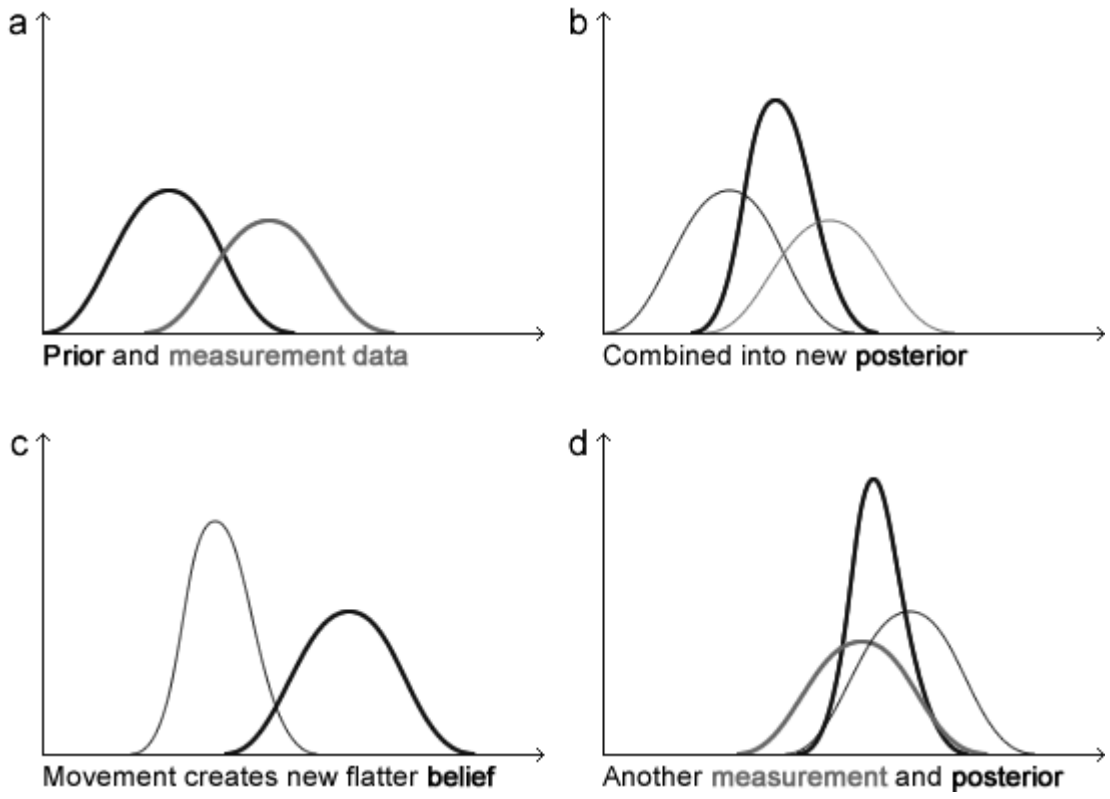


Figure 5. Illustration of Kalman filter [5]. Distributions are not to exact scale.

1.3.5 Interactive example

As part of this thesis, a web browser based implementation of Kalman filter was developed using JavaScript and HTML5 technologies that can be used to experiment with the filter and as a working reference implementation for others to incorporate into their project starting with actual code rather than mathematical formulas.

The first example shows the simplest use of a Kalman filter, guessing a one-dimensional constant from a stream of noisy measurements as seen on Figure 6. A practical use for this might be the voltage measurement in a digital multimeter. As can be seen, even with large amount of sensor noise (green is the sensor data, red the actual value and blue the Kalman filter value), the filter quickly converges on the right value. The small graphs on the side show the various properties of the filter and the user can play with input and filter parameters using the sliders below.

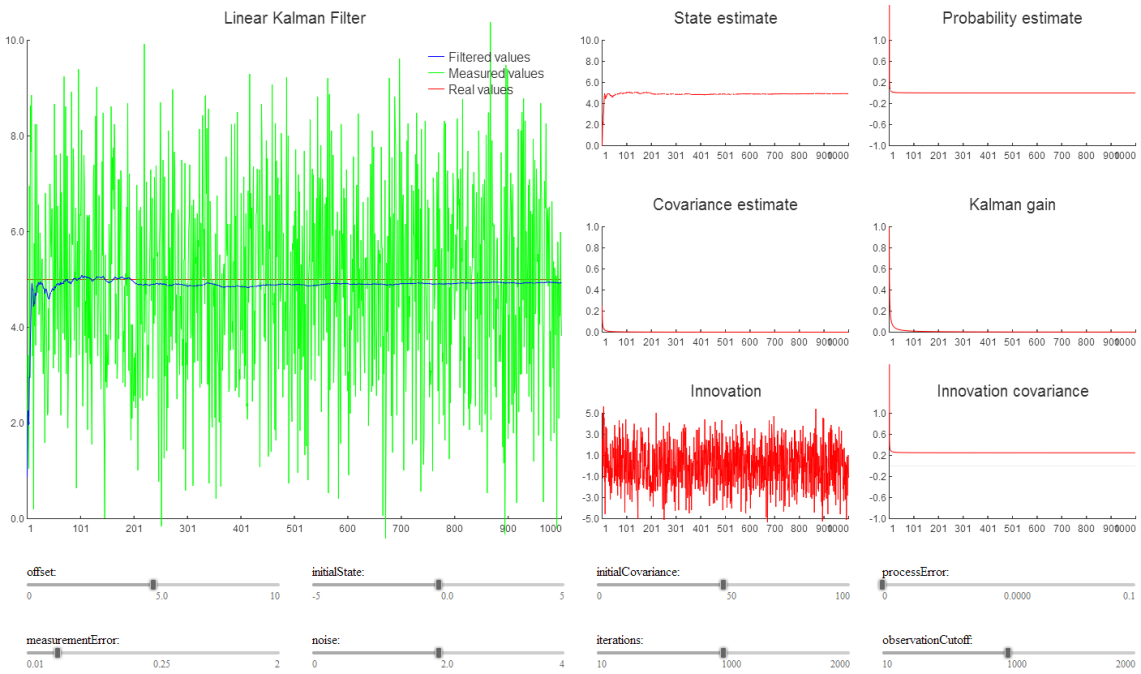


Figure 6. One dimensional Kalman filter guessing a constant value.

The second example in Figure 7 shows a more complex case of guessing the flight path of a cannonball. Imagine a projectile is shot out of a cannon and there is a camera following its path that is not very accurate at determining the cannonball's real position at any time. But since the dynamics of a flying projectile are well known, this can be modeled in the Kalman filter. This sort of task with concise model is well suited

for the Kalman filter as it can follow the model to produce accurate results in spite of lots of noise in the sensor input.

As can be seen from the smaller Kalman gain plot on the right, initially it is quite high. As more data comes in and the estimate stabilizes, the Kalman gain is reduced and the filter starts to rely more on the model than the noisy sensor data. The covariance is also reduced.

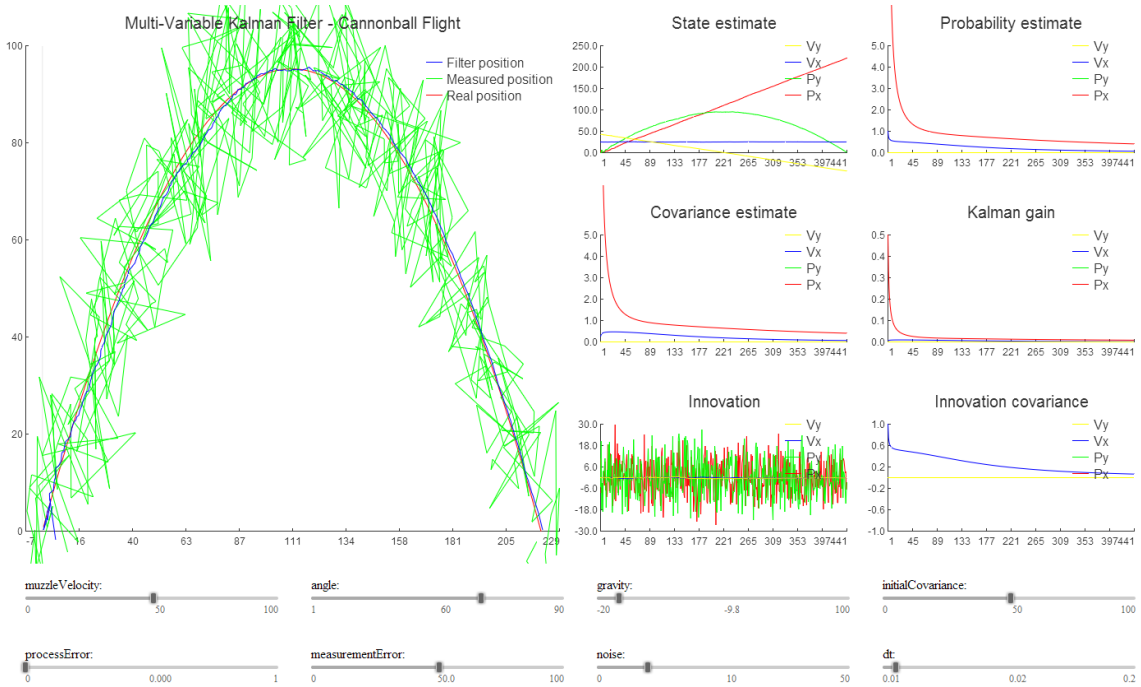


Figure 7. Kalman filter predicting the flight of a cannonball.

This JavaScript implementation is an open source project on Github [16] with live interactive examples that run directly in the browser. It is meant as both an example for understanding Kalman filters as it allows to see the underlying dynamics and play with input data as well as a working reference implementation that is easy to port into any other language. The same implementation is used for the soccer robot simulation experiments performed is Section 2.3.1.

1.3.6 Extensions

An alternative dual method of using canonical parameterization exists that uses an *information matrix* and *information vector* with some trade-offs of computational characteristics. Without going into details, when using the canonical parameterization to

represent the posterior, the resulting implementation of the Bayes filter is called the *information filter* [9].

The main limitation of the Kalman filter is that it is only applicable to linear systems, but extensions to the general idea exist that can handle nonlinear problems.

The *extended Kalman filter* (EKF) approximates a potentially non-linear function locally by calculating a tangent to the nonlinear function which is linear, making the general case of the filter applicable. The tangent is calculated using *Taylor expansion* which involves calculating the first derivative to the function in question, resulting in *Jacobian matrix,* and evaluating it at a specific point. While the result of such operation is an approximation, extended Kalman filters have become very popular for state estimation in robotics due to their relative simplicity and computational efficiency, having the same $O(k^{2,4} + n^2)$ complexity. EKF works best in situations of low uncertainty and local nonlinearity, making it perform the best in local localization problems (cases where the initial location is known and the algorithm needs to track the change of pose). EKF retains the limitation of a single unimodal guess, another extension to Kalman filter exists called *multi-hypothesis extended Kalman filter* (MHEKF) [10] which supports multimodal beliefs.

Taylor expansion is not the only method for linearization the transformation of Gaussians. The *unscented Kalman filter* (UKF) [11] probes the function to be linearized at selected points and calculates a linearized approximation based on the outcomes of these probes. It has the same asymptotic complexity and is shown to produce the same or better results as EKF [5]. It also has the advantage that it does not require computing the Jacobians, which can be difficult to determine for some problems. For this reason, it is often referred to as a *derivative-free filter*.

1.4 Particle filter

1.4.1 Introduction

Particle filter (PF) is a non-parametric implementation of the Bayes filter that, instead of representing probability distributions using parametric functions such as Gaussians, uses a finite set of random state samples to represent the posterior belief $bel(x_t)$.

Although this representation is approximate, it can represent a wide range of probability distributions, which is not possible when using Gaussians or other

parameterized models. Another advantage to particle filters is the ability to model nonlinear systems without having to linearize the transformation function. Finally, particle filter is often found to be the easiest to implement, especially for complex systems where the Jacobians can be hard to find for EKF. For the reasons above, particle filters have become popular in the robotics community and are used extensively for both simple and more complex estimation tasks.

The random samples of a particle filter are called *particles*, each of which is a concrete instance of the state, representing a separate hypothesis of the possible state space. The set of particles is denoted by χ_t as given by:

$$\chi_t := x_t^{[1]}, x_t^{[2]}, \ldots, x_t^{[M]} \tag{11}$$

Usually the number of particles M needs to be relatively high to sufficiently accurately represent the probability distribution. A thousand may be a good starting point for simpler low-dimensionality problems but depending on the complexity, the number of particles required for adequate probability distribution representation may grow exponentially with state dimensionality. Also the number of particles may sometimes depend on some properties of the belief such as algorithm run time t.

1.4.2 Intuitive explanation

The basic idea of the algorithm is to first generate a set of particles, each representing a possible state of the problem at hand, usually initiated at random. For the example of a soccer-playing robot, the state of each particle would include its x, y position on the field and orientation of the robot. If we do not know the position of the robot in the beginning, we would just randomly initiate the position of the particles uniformly over the playing field with arbitrary orientations. If we do know that the robot starts at a particular corner, we can initiate the particles with a set position and orientation, making it the simpler case of local localization problem.

The next step is *movement update*. This can be done by using a model-based approach where the robot is given a command to move in certain direction and this command is modeled to result in certain movement. Often a more accurate and popular solution is to use the odometry system mentioned earlier, where feedback from the actuators (wheels for our example) is used to calculate the heading of the robot with more accuracy. In either case, this movement information is then applied to all of the particles so that they make the same maneuver as the robot is believed to have made.

The filter accounts for uncertainty in robot movement by introducing some artificial noise to this modelled motion so that each particle moves slightly differently both in distance and in change or orientation. For example, let us examine a situation where the cluster of particles representing the probability distribution has lagged behind the robot as shown in Figure 8. If we were to apply the odometry movement to these particles directly, they would likely not be able to catch up with the robot, but with noise, some of the particles would move quicker than the odometry information suggested, get closer to the robot and would more likely "survive" (get resampled) in the following steps (notice that in a real implementation, one would use several magnitudes larger number of particles).

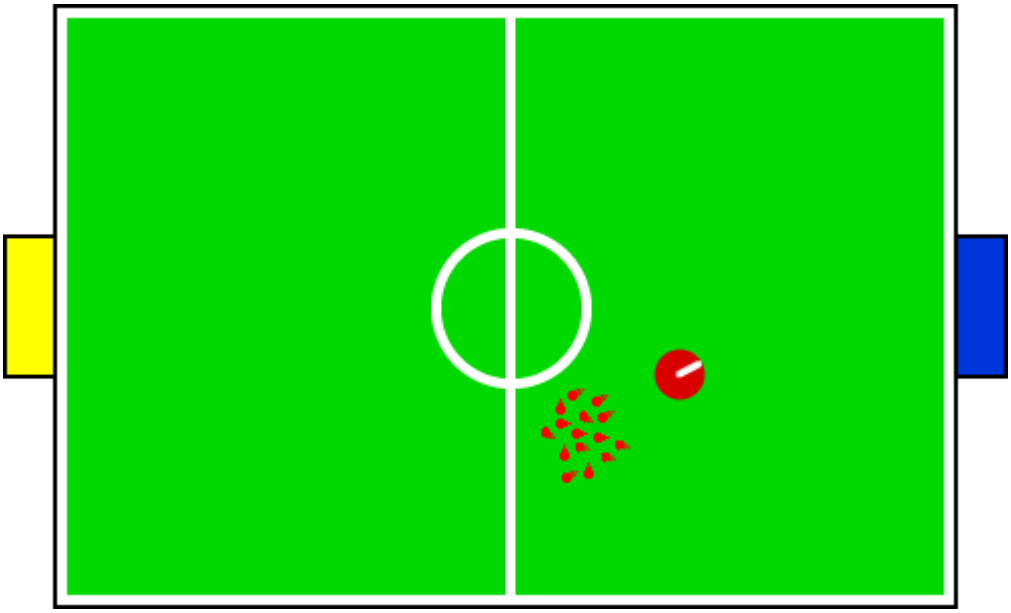


Figure 8. Particles are lagging behind the robot.

After the movement phase comes the *update step* in which each particle is evaluated based on its fitness to the measurement data. For example let us consider Figure 9. After the movement step, the particles have spread out slightly due to having different starting orientations and the applied movement noise. The camera takes a measurement which could be as simple as distances to both goals. Robot observes the blue goal at a distance of 1.6 meters and the yellow one at 3.4 meters. The algorithm then computes the *likelihood* of each particle given the observations. We picked a random particle and calculated that given its state, the blue goal should have been observed 2.3 meters away and the yellow one at around 2.8 meters. As this expectation

does not match the observed information very well and there are other particles which would rank better, this particle will have a low probability of getting resampled for the next step.

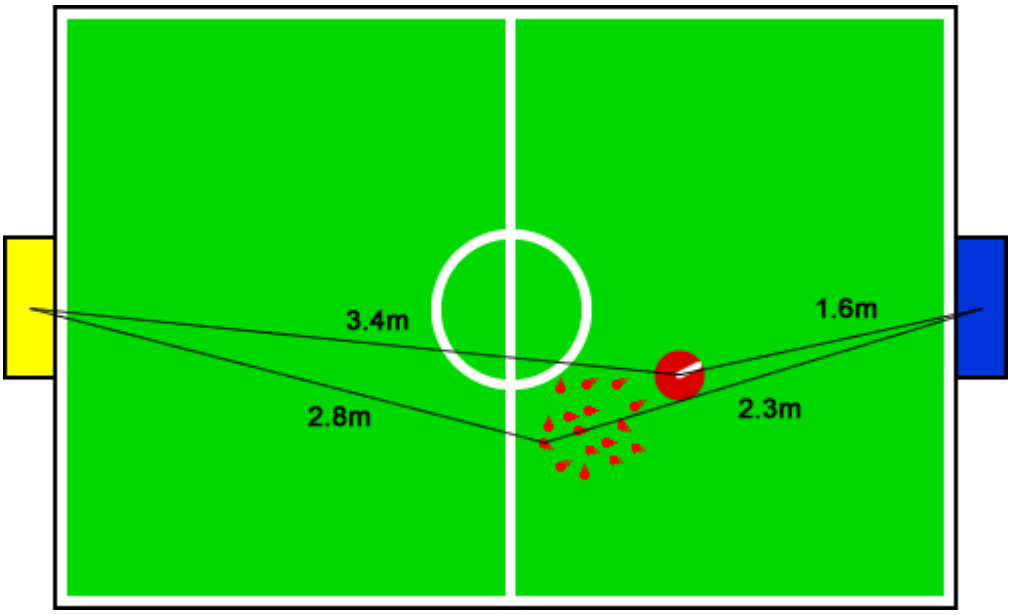


Figure 9. Update phase evaluation for a specific particle.

The last step of the particle filter is to *resample* the particle set based on the likelihood metric of each one. The better the particle state matched the observation, the more likely it is to get picked into the new set and any one particle may be selected several times. Since movement step of the following rounds will evolve each of them in a slightly new direction, picking any of the particles multiple times is not an issue. In a few iterations of the algorithm, the particles will have likely converged at the correct location of the robot given adequately accurate model and sensor information as shown in Figure 10.

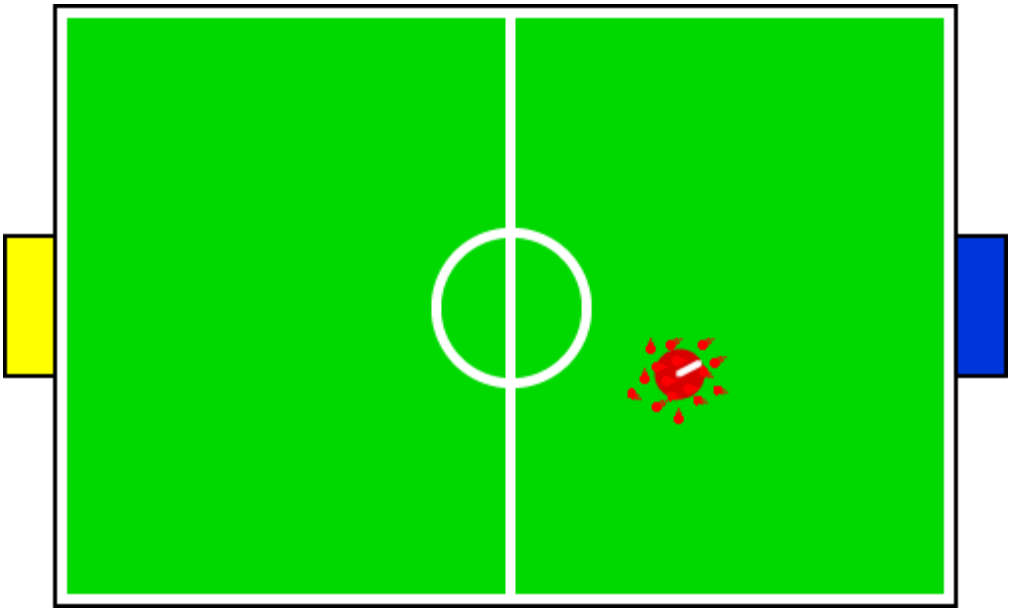


Figure 10. Particles have converged on the correct position of the robot.

What remains is to extract the location of the robot from the set of particles. Assuming a unimodal distribution (which is acceptable for simpler problems), one can simply average the positions and orientations of the individual particles. This would result in a single best guess. For more complex global localization problems, the resulting distributions are often multimodal due to large initial uncertainty and symmetries of the environment. Particle filters can represent such multimodal and arbitrarily shaped beliefs well, but extracting concrete positions from it can require the use of clustering algorithms such as k-means [19].

1.4.3 Algorithm

1. particle_filter (χ_{t-1}, u_t, z_t):
2. $\underline{\chi}_t = \chi_t = \Theta$
3. for (m = 1 to M) {
4. sample $x_t^{[m]}$ from $p(x_t \mid u_t, x_{t-1}^{[m]})$
5. $\omega_t^{[m]} = p(z_t \mid x_t^{[m]})$
6. $\bar{\chi}_t = \bar{\chi}_t + \langle x_t^{[m]}, \omega_t^{[m]} \rangle$
7. }
8. normalize $\omega_t^{[m]}$
9. for (m = 1 to M) {
10. draw i with probability proportional to $\omega_t^{[i]}$
11. $\chi_t = \chi_t + x_t^{[i]}$
12. }
13. return χ_t

Algorithm 3. Particle filter implementation [5].

Basic implementation of the particle filter is given in Algorithm 3. As input, it takes the particle set from previous step χ_{t-1} along with the latest control u_t and measurement info z_t.

On line 2 it creates two empty sets to hold the new particles. Lines 3 to 7 iterate over the set of particles and for each of them, line 4 generates the hypothetical state $x_t^{[m]}$ based on the particle state from previous step $x_{t-1}^{[m]}$ and current control input u_t. Notice that one needs to be able to sample from the state transition distribution $p(x_t \mid u_t, x_{t-1})$. Line 5 calculates the *importance factor* or *fitness* of each particle $x_t^{[m]}$ denoted by $\omega_t^{[m]}$ which is the probability of observing measurement z_t for particle $x_t^{[m]}$ given $\omega_t^{[m]} = p(z_t \mid x_t^{[m]})$. This is used to later incorporate measurement information to the filter. This loop results in the particle filter representation of $\underline{bel}(x_t)$.

The importance factors are normalized on line 8 so the largest probability would equal 1. Lines 9 to 12 implement the importance factor based resampling of the particles. The algorithm draws replacement M particles from the temporary set $\underline{\chi}_t$ where the probability of drawing each particle is given by the importance factor $\omega_t^{[m]}$ (the same

particle can be drawn several times). This results in a new set of particles χ_t which are distributed approximately according to the posterior $\mathrm{bel}(x_t) = \eta p(z_t \mid x_t^{[m]}) \underline{\mathrm{bel}}(x_t)$. The resampling phase focuses the particles to regions in state space with high posterior probability and convey the Darwinian idea of *survival of the fittest*. The results are returned in line 13, containing the set of particles that are transformed by the control input and best match the measurement data.

1.4.4 Density estimation

Particle filter represents the posterior probability distribution using discrete approximation but some applications require having an estimate at any point of the state space, rather than just at the states represented by the finite number of particles. Process of extracting continuous estimates from a set of particles is called *density estimation*. Without going into much detail regarding various density estimation techniques, this section names a few of them.

A popular technique for unimodal problems is transforming the set of particles into a Gaussian. Efficient approximation techniques exist that convert a set of particles to a Gaussian normal distribution with mean and variance [5] (see Figure 11.b). While this approach is effective for many simpler local localization problems, the Gaussians capture only basic properties of a density. One could apply clustering algorithms such as k-means to support multimodal hypothesis using a mixture of Gaussians.

Alternative approach would be to use histograms as shown on Figure 11.c. which support multi-modal distributions. Histograms are efficient to compute by summing the weights of particles falling into a particular range and density at any state can be extracted in time independent of the number of particles, but the state complexity is exponential in the number of dimensions. This issue can be alleviated by using density tree approach although this makes extracting the density at any point of the state space more costly.

Another method would be to use each particle as the center of a Gaussian *kernel* and combining these mixtures of kernels to represent the overall density (see Figure 11.c). This method is called *kernel density estimation* [21]. Advantage of such approach is its algorithmic simplicity and smooth resulting density but the complexity of computing density at any point is linear in the number of points.

Choosing the method to use depends on the problem at hand and the available computational resources. For autonomous robots, the processing power is often the

limiting factor and a simple mean will suffice. More complex global and *active localization* problems (changing robot behavior to improve localization performance by for example keeping close to landmarks) might require using one of the more computationally demanding but accurate approaches.

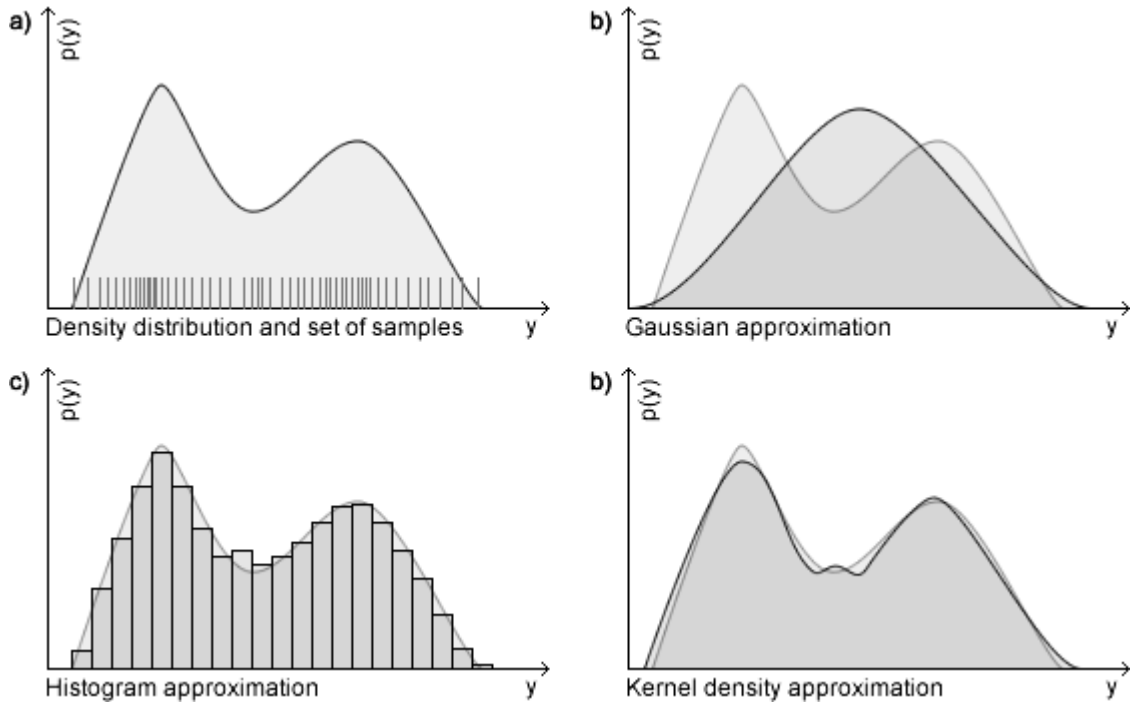


Figure 11. Example of density estimation approaches [5]. Plots are not to exact scale.

1.4.5 Resampling

Consider an extreme case of a rather useless robot without any sensors or motors, incapable of learning anything about its environment or movement [5]. As such a robot never moves or senses its surroundings, its state estimate should not change in time.

Unfortunately this is not the case for the simplistic implementation considered above. With each iteration of the algorithm, the resampling step will slightly change the statistics of the original probability density. With each step, more and more particles are erased from the set simply due to the random nature of the resampling step without creating any new particles which will eventually result in all of the particles being

identical copies of one-another. It would seem that the robot has uniquely determined its state which contradicts the fact that it has not sensors to improve on its estimate.

A simple solution to this problem would be to never resample if the robot state is known to be static ($x_t = x_{t-1}$). Even if the state changes, it can help reduce the variance of the particle set as an estimator by reducing the frequency of resampling. There is a balance where resampling too often can lead to loss of diversity and doing so too seldom causes particles to be wasted in regions of low probability. Standard approach for deciding whether to resample or not is to base it on the variance of the importance weight which relates to the efficiency of the sample based representation [5]. If the variance is zero (all weights are identical) then no resampling should be performed and vice versa.

An alternative strategy would be to use a low-variance sampling algorithm [12], where instead of selecting samples independently of each other, the selection involves a sequential stochastic process, cycling through all particles systematically. This approach covers the space of samples in a more systematic fashion and if all particles have the same importance factor, the resulting sample set is the same as input set so that no samples get lost when resampling without accounting for movement and observation data. Also the complexity or low variance sampling is linear to the number of particles O(M) when independent samplers have complexity of O(M logM) [5].

2 Localization of a soccer robot model

2.1 The Robotex soccer robot and its environment

While the methods described in this thesis apply for various localization problems, a specific case of a soccer-playing robot is researched in depth in an effort to find the best way for an autonomous *omnidirectional* robot with two cameras to localize itself. This section describes the robot, its environment and game rules for the practical implementation.

The environment and rules described in this section correspond to the *Robotex* [12] 2012 professional robotic football league competition held in Estonia. The rules are a simplified version of the popular international robotics competition *Robocup* small and middle-size leagues held since 1997 promoting robotics and artificial intelligence (AI) research.

2.1.1 The environment

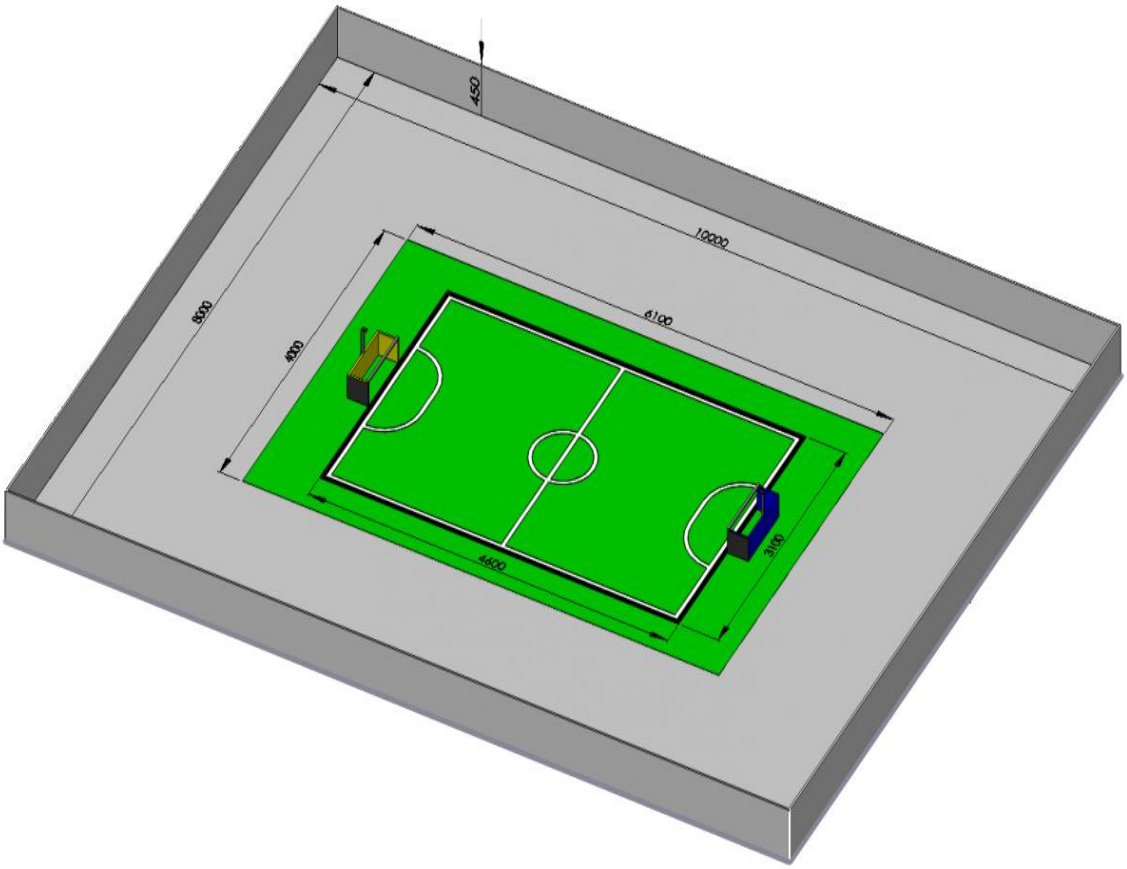


Figure 12. Soccer field [17].

Figure 12 shows the layout of the playing field. The game is played one-on-one with two competing robots starting in opposite corners. Eleven orange golf-balls are placed on the field in random but symmetrical positions with respect to the center of the field. The round is won by the robot that scores the most balls into the opponent's goal in 90 seconds. Balls fetched from outside the black border line do not count and robot ramming the goal or leaving the green carpet gets removed from the field while the other robot can keep playing until the match time runs out or all the balls are kicked off the field.

This setup has two obvious objects usable for localization, which all of the robots generally have to be able to detect anyway – the blue and yellow goals. As they are quite large and uniquely colored, the goals are not hard to find with relatively simple computer vision algorithms by looking for blobs of certain color range. Calculating an angle to a detected object in the video frame is relatively easy as well, and, as cameras are usually mounted at a fixed angle on the robot (see Figure 13), one can match the pixel row of the object in the picture to an approximate real-world distance. This approach is quite accurate near the robot but becomes inaccurate at larger distance as for a camera quite close to the ground, the difference between an object at, let us say, 3 and 4 meters away might be only a few pixels.

2.1.2 The robot

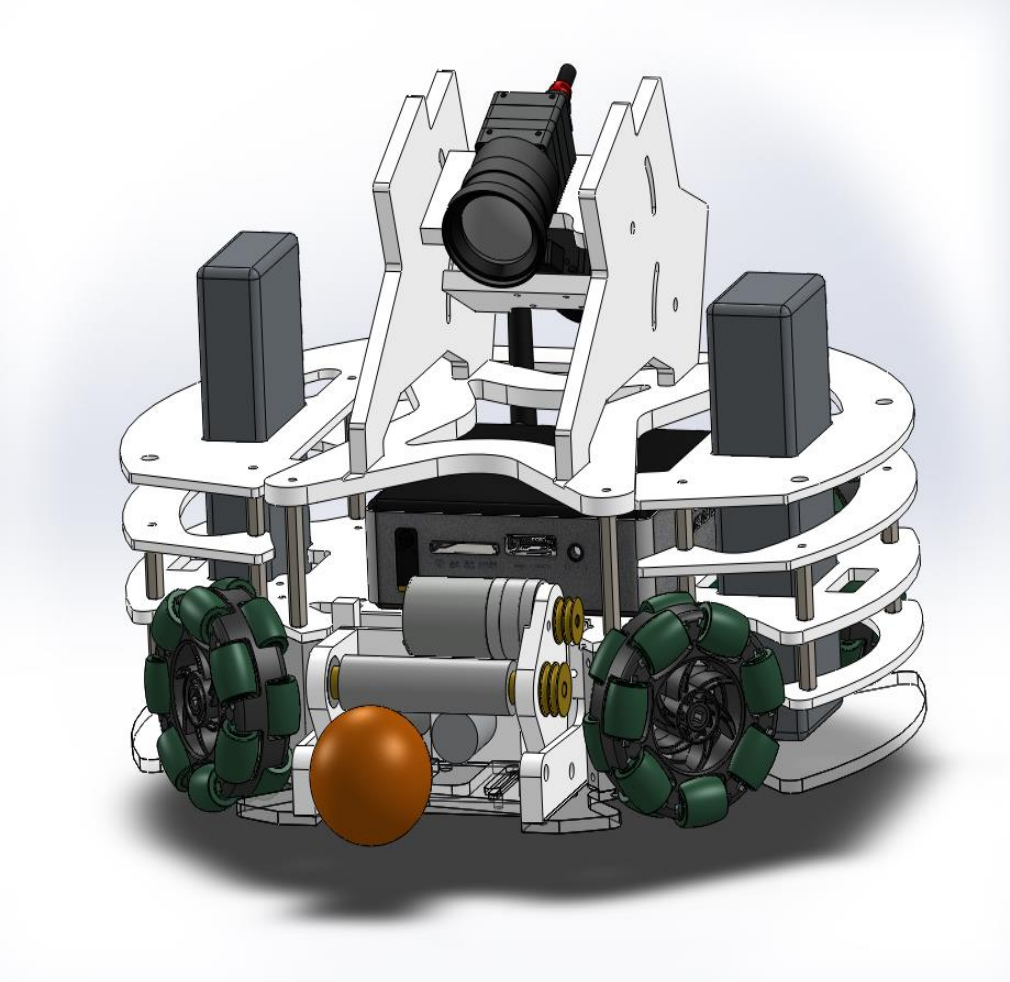


Figure 13. Soccer robot “Telliskivi II”.

In general, the rules of Robotex do not specify the actual mechanical solution to be used in the competition and every team designs their own. In this treatment we shall focus on the robot "Telliskivi II" that was designed and built for Robotex 2012 by the author of this thesis together with Reiko Randoja, Mark Laane and Taavi Põri. The robot successfully competed in it, achieving second place.

The robot has a diameter of approximately 250 mm and employs four *omnidirectional wheels*, which allow it to move in an arbitrary direction while rotating about its axis at the same time. Being able to move in any direction enables it to maneuver efficiently and control its heading separately from its motion vector. The wheels include *rotary encoders*, which allow gathering odometry data and calculate the robot’s relative heading and rotational velocity.

The platform is equipped with two cameras positioned back-to-back. Each camera provides about 60 degrees horizontal field of view. This arrangement has several advantages over the common “single front camera” approach. Firstly, it allows the robot to see more of the field at once, detecting balls and goals without having to rotate about its axis as much. Secondly, the robot is capable of observing the two goals simultaneously, which enables calculating the approximate location of the robot. However, depending on the position and orientation of the robot, it may sometimes only see a single goal or no goals at all. The localization algorithms must gracefully handle such situations.

The robot state having to do with localization consists of two main components – the position (x, y) on the plane of the field and its orientation denoted by Θ in the range of 0..2π. Together, these form the *pose* of the robot given by the vector $(x, y, \Theta)^T$. Other potentially interesting state information in our model includes robot speed Vx, Vy and its rotational velocity ω (called omega). These parameters comprise the *control input vector* $(Vx, Vy, \omega)^T$ generated by the algorithm guiding the robot. These inputs are converted to individual wheel speeds and the motor hardware then does its best to maintain those speeds relying on the feedback from the wheel encoders.

Thus, the main inputs to the localization algorithm are the distances and relative angles to the two goals acquired by the camera, the odometer information extracted from the wheel encoders and the command signals generated by the control algorithm.

2.2 Motion model and odometry

As different robots utilize various means of navigating their environment, this section will not go into much details trying to cover them all. A popular omnidirectional movement model is implemented using omni-wheels shown in Figure 14. What makes these wheels special is the fact that they only have considerable grip in the longitudinal axis while the lateral grip is minimized by special rollers. This allows them to push forward in the longitudinal direction while slipping freely at the same time in lateral direction.

Figure 14. Omni-wheel used for omni-directional movement.

These wheels are arranged in a circular pattern on the perimeter of the robot as shown in Figure 15 for 4-wheel configuration. Such configuration allows the robot to move in any direction while simultaneously turning about its axis. This makes the robot very maneuverable and allows controlling the direction of the robot separately from its orientation, which is useful for minimizing the amount of turning the robot needs to perform with the ball to aim for the goal (which often needs to be performed slowly not to lose the ball).

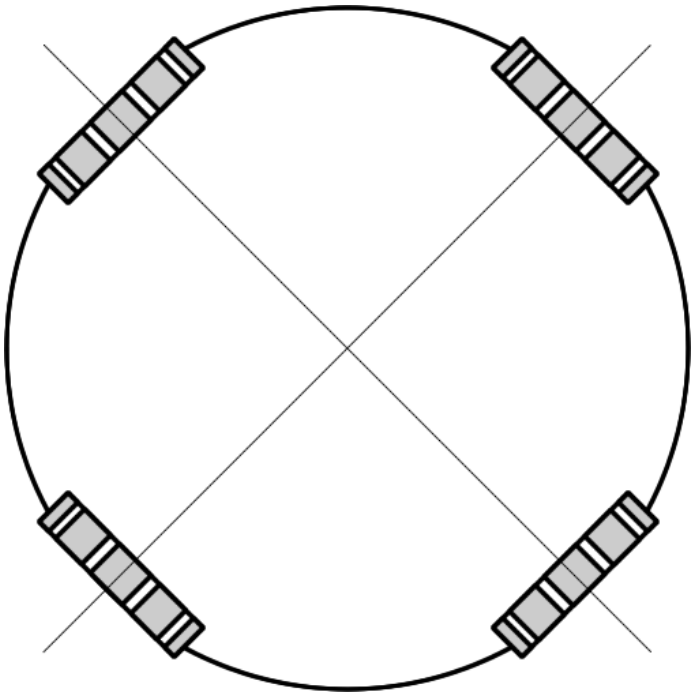


Figure 15. Omni-wheels configuration.

Depending on the individual wheel speeds, the robot can move in any direction and rotate simultaneously. Figure 16.a shows the robot moving forwards without any rotational speed and 16.b shows the robot rotating about its center. To move diagonally for example, two opposing wheels would be run at the same speed while the two other ones would stand still.

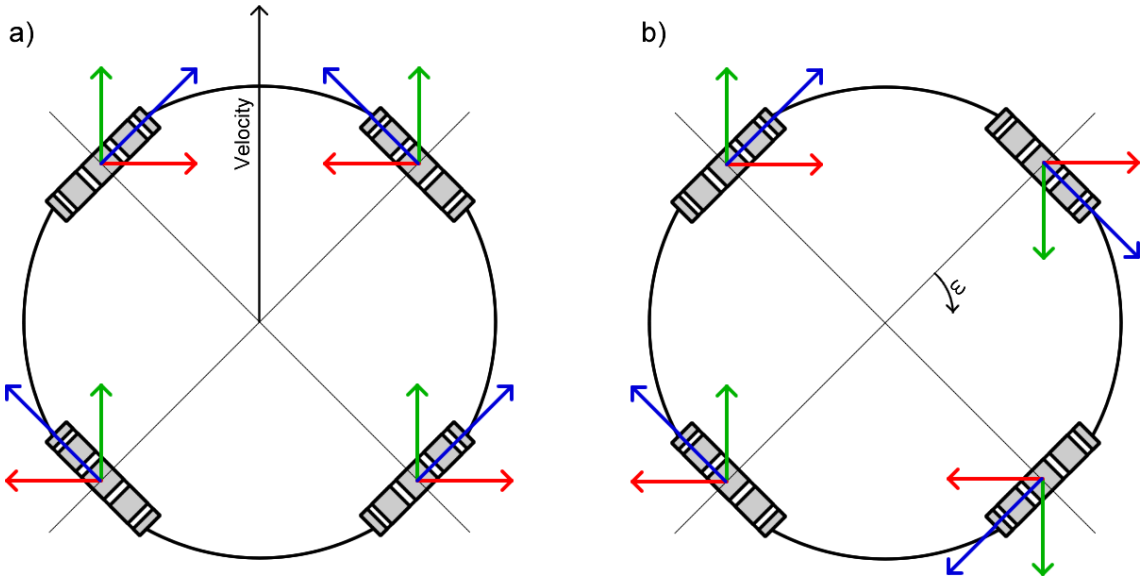


Figure 16. Omnidirectional movement.

For our example, we are using a four-wheel setup with the wheels labelled as shown on Figure 17. The numbering is generally a matter of choice but it defines the “forward” direction of the robot.

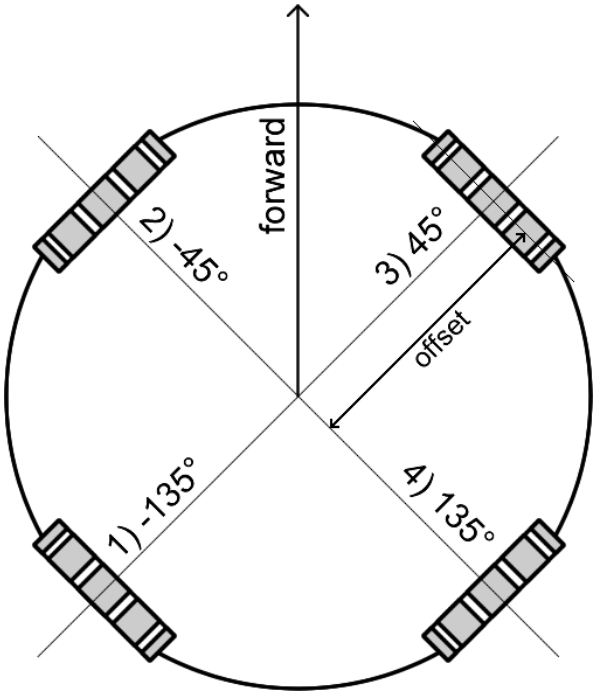


Figure 17. Wheel numbers and angles.

Since the velocities of three wheels are enough to define the motion model of the robot, we deduce it from four wheels by generating four sets of three wheels, calculating each set separately and averaging the results to produce the final motion vector and rotational velocity ω.

2.3 Robotex simulator

This section provides practical implementations of the discussed algorithms. A full-featured simulator [13] was created as a part of this thesis that enables testing the various localization and control algorithms. It is implemented using *HTML5* technologies, written in *JavaScript* and rendered using *SVG*. The simulator is open-source and accessible from *Github*. All of the code discussed in future sections is referenced from this simulator literately licensed to be used in any way.

2.3.1 JavaScript implementation

This thesis provides an example implementation for the main algorithms discussed, written in JavaScript within the simulator framework. JavaScript was chosen because it is widely known, runs in the browser without any additional dependencies, does not require compiling and can be understood by people with various programming

language backgrounds. The whole codebase is available in the open-source simulation platform [13]. The code below will reference to specific files in the simulator source for complete code, the main source tree layout is presented in Figure 18.

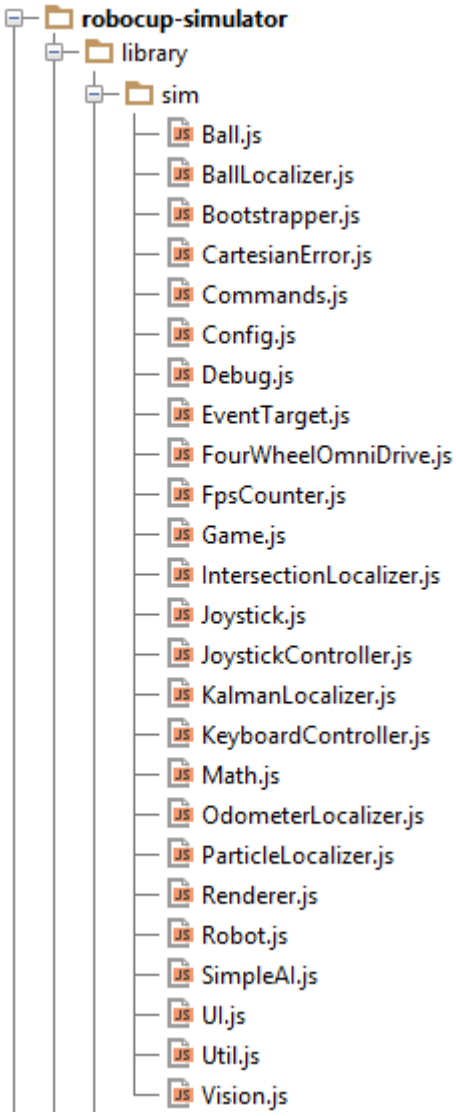


Figure 18. Simulator source tree layout.

The robot model setup step is given in Code Listing 1. As input, it requires the wheel radius, an array of wheel angles and the wheel offset from robot's center in meters. The first omega matrix defined in line 6 is used to calculate the wheel speeds from input target heading and omega in the following step. The other matrices are the four combinations of three wheels that will later be used to calculate the relative

movement of the robot based on wheel speeds as input. Inverse of these matrices are also pre-calculated.

```
1   Sim.FourWheelOmniDrive = function(wheelRadius, wheelAngles, wheelOffset) {
2       this.wheelRadius = wheelRadius;
3       this.wheelAngles = wheelAngles;
4       this.wheelOffset = wheelOffset;
5
6       this.omegaMatrix = new Sim.Math.Matrix4x3(
7           -Math.sin(this.wheelAngles[0]), Math.cos(this.wheelAngles[0]), this.wheelOffset,
8           -Math.sin(this.wheelAngles[1]), Math.cos(this.wheelAngles[1]), this.wheelOffset,
9           -Math.sin(this.wheelAngles[2]), Math.cos(this.wheelAngles[2]), this.wheelOffset,
10          -Math.sin(this.wheelAngles[3]), Math.cos(this.wheelAngles[3]), this.wheelOffset
11      );
12      this.omegaMatrixA = new Sim.Math.Matrix3x3(
13          -Math.sin(this.wheelAngles[0]), Math.cos(this.wheelAngles[0]), this.wheelOffset,
14          -Math.sin(this.wheelAngles[1]), Math.cos(this.wheelAngles[1]), this.wheelOffset,
15          -Math.sin(this.wheelAngles[2]), Math.cos(this.wheelAngles[2]), this.wheelOffset
16      );
17      this.omegaMatrixB = new Sim.Math.Matrix3x3(
18          -Math.sin(this.wheelAngles[0]), Math.cos(this.wheelAngles[0]), this.wheelOffset,
19          -Math.sin(this.wheelAngles[1]), Math.cos(this.wheelAngles[1]), this.wheelOffset,
20          -Math.sin(this.wheelAngles[3]), Math.cos(this.wheelAngles[3]), this.wheelOffset
21      );
22      this.omegaMatrixC = new Sim.Math.Matrix3x3(
23          -Math.sin(this.wheelAngles[0]), Math.cos(this.wheelAngles[0]), this.wheelOffset,
24          -Math.sin(this.wheelAngles[2]), Math.cos(this.wheelAngles[2]), this.wheelOffset,
25          -Math.sin(this.wheelAngles[3]), Math.cos(this.wheelAngles[3]), this.wheelOffset
26      );
27      this.omegaMatrixD = new Sim.Math.Matrix3x3(
28          -Math.sin(this.wheelAngles[1]), Math.cos(this.wheelAngles[1]), this.wheelOffset,
29          -Math.sin(this.wheelAngles[2]), Math.cos(this.wheelAngles[2]), this.wheelOffset,
30          -Math.sin(this.wheelAngles[3]), Math.cos(this.wheelAngles[3]), this.wheelOffset
31      );
32
33      this.omegaMatrixInvA = this.omegaMatrixA.getInversed();
34      this.omegaMatrixInvB = this.omegaMatrixB.getInversed();
35      this.omegaMatrixInvC = this.omegaMatrixC.getInversed();
36      this.omegaMatrixInvD = this.omegaMatrixD.getInversed();
37  };
```

Code Listing 1. Four-wheel omnidirectional motion model setup.

The motion model defines two main operations. Firstly, it needs to calculate the individual wheel speeds required for the robot to move at a certain direction and change orientation with given speed. This calculation is given in Code Listing 2. As input, it takes the target heading and rotational velocity and as output, the four wheel speeds are returned. The target direction has components x and y which combined with target omega are used to build the target 3x1 matrix on line 40. The wheel omegas are then calculated by scaling the omega matrix by the inverse of wheel radius and then multiplying the result with target matrix. This results in a 4x1 matrix containing the

individual wheel speeds. The operation could be optimized slightly by including the wheel radius inverse multiplication in the original *omegaMatrix*.

```
39  Sim.FourWheelOmniDrive.prototype.getWheelOmegas = function(targetDir, targetOmega) {
40      var targetMatrix = new Sim.Math.Matrix3x1(
41              targetDir.x,
42              targetDir.y,
43              targetOmega
44          ),
45          wheelOmegas = this.omegaMatrix
46              .getMultiplied(1.0 / this.wheelRadius)
47              .getMultiplied(targetMatrix);
48
49      return [
50          wheelOmegas.a11,
51          wheelOmegas.a21,
52          wheelOmegas.a31,
53          wheelOmegas.a41
54      ];
55  };
```

Code Listing 2. Wheel omega calculation from target heading and rotational velocity.

The second operation involves calculating the relative robot velocity and omega from wheel speeds as given by Code Listing 3. Four combinations of three wheel speeds are calculated matching the setup step. The inverse of the omega matrices created in setup step are then scaled by wheel radius and multiplied by the wheel omega matrices, resulting in four different hypotheses. These are then averaged to produce the final x, y velocities and omega. Since the data from all wheels are averaged, should any of them slip slightly, it will have less effect on the calculated movement vector then that would be the case for three wheels. An alternative could be eliminating the outlier (set of wheels that is most different from the average). Data extracted from this last step is the odometry information that is used in the localization algorithms following in later sections.

```
Sim.FourWheelOmniDrive.prototype.getMovement = function(omegas) {
    var wheelMatrixA = new Sim.Math.Matrix3x1(
            omegas[0],
            omegas[1],
            omegas[2]
        ),
        wheelMatrixB = new Sim.Math.Matrix3x1(
            omegas[0],
            omegas[1],
            omegas[3]
        ),
        wheelMatrixC = new Sim.Math.Matrix3x1(
            omegas[0],
            omegas[2],
            omegas[3]
        ),
        wheelMatrixD = new Sim.Math.Matrix3x1(
            omegas[1],
            omegas[2],
            omegas[3]
        ),
        movementA = this.omegaMatrixInvA
            .getMultiplied(wheelMatrixA)
            .getMultiplied(this.wheelRadius),
        movementB = this.omegaMatrixInvB
            .getMultiplied(wheelMatrixB)
            .getMultiplied(this.wheelRadius),
        movementC = this.omegaMatrixInvC
            .getMultiplied(wheelMatrixC)
            .getMultiplied(this.wheelRadius),
        movementD = this.omegaMatrixInvD
            .getMultiplied(wheelMatrixD)
            .getMultiplied(this.wheelRadius);

    return {
        velocityX: (movementA.a11 + movementB.a11 + movementC.a11 + movementD.a11) / 4.0,
        velocityY: (movementA.a21 + movementB.a21 + movementC.a21 + movementD.a21) / 4.0,
        omega: (movementA.a31 + movementB.a31 + movementC.a31 + movementD.a31) / 4.0
    };
};
```

Code Listing 3. Calculating robot movement from wheel omegas – the odometer.

2.3.2 Odometry localizer

Using the odometry information directly, we can implement the simplest localization system that uses no additional feedback. This has little practical value as the odometry-only approach will eventually drift away from the true state. Imagine that the orientation of the robot has drifted just a few degrees, when the robot now moves a larger distance, this will accumulate a relatively large error as shown on Figure 19.

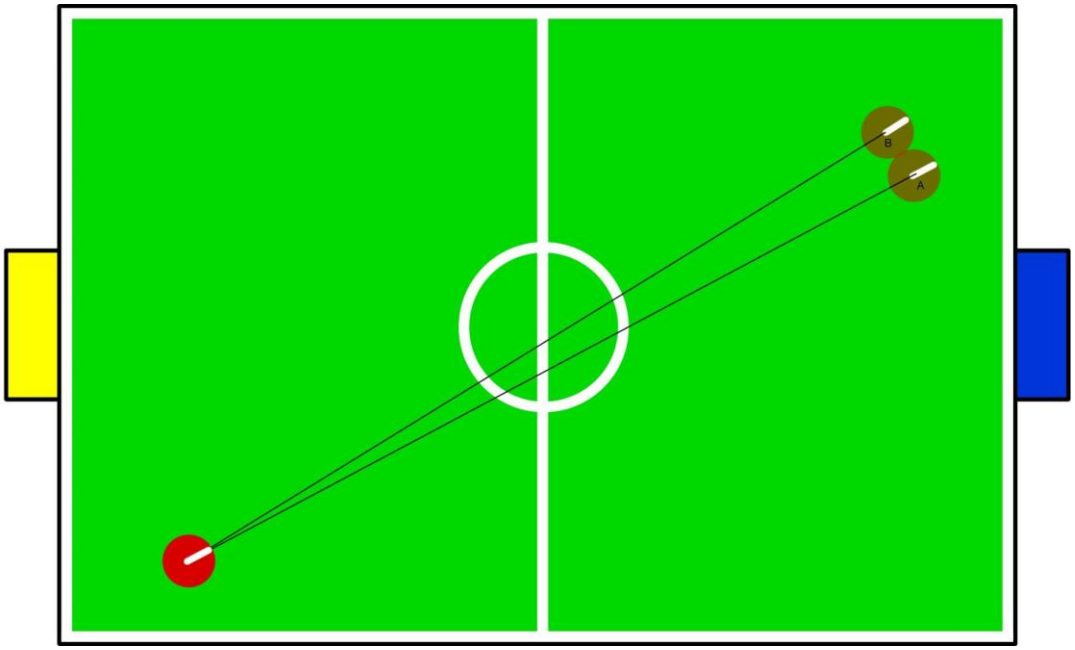


Figure 19. Odometry drift from small orientation difference.

The *OdometerLocalizer* is implemented in *library/sim/OdometerLocalizer.js* and given by Code Listing 4. While this localization algorithm is not very useful in this form for the reason described above, it sets a template for the other algorithms. In the constructor, the initial state is set up. As the position and orientation of the robot is known at the start of the round based on which side its set to play against and that the robot is always oriented the same way (let us say at a 45 degree angle), the localizer should be initiated to this known pose.

The main method of the localizer is the move command. This is called at every iteration with the robot's local x, y and rotational velocities. The *dt* is short for delta-time and this is the time in seconds that has passed since the last iteration, used for calculating the actual velocities. Line 8 calculates the new orientation by adding rotational velocity *omega * dt* to the previous value and also limits it between 0 and 360 degrees (or 2π in radians).

The updated orientation is then used in lines 10 to 13 to update the actual position. The local x, y velocities are transformed to global coordinate system space by rotating the vector by robot orientation. The *getPosition* method simply returns the pose of the robot as predicted by this localization algorithm.

```
Sim.OdometerLocalizer = function() {
    this.x = 0.0;
    this.y = 0.0;
    this.orientation = 0.0;
};

Sim.OdometerLocalizer.prototype.move = function(velocityX, velocityY, omega, dt) {
    this.orientation = (this.orientation + omega * dt) % Sim.Math.TWO_PI;

    this.x += (velocityX * Math.cos(this.orientation)
        - velocityY * Math.sin(this.orientation)) * dt;
    this.y += (velocityX * Math.sin(this.orientation)
        + velocityY * Math.cos(this.orientation)) * dt;
};

Sim.OdometerLocalizer.prototype.getPosition = function() {
    return {
        x: this.x,
        y: this.y,
        orientation: this.orientation
    };
};
```

Code Listing 4. Odometry-only localizer implementation.

2.3.3 Direct measurement model

Having two cameras back to back enables the robot to calculate its approximate position directly given that it sees both of goals at the same time as shown on Figure 20. As the positions of the goals are fixed, observing the distances *d1* and *d2* to both goals, we can see that the robot must be located in one of the intersection points *P1* or *P2* of two circles drawn from the center of the goals with a radius matching the distance of the goals respectively.

Even if the robot only sees a single goal, it can still use it to improve its position estimate as shown in Figure 21. Imagine the robot's last position estimate was 1) and it now observes the yellow goal at distance *d1*. It can now improve its location estimate by switching to position 2) as the robot needs to be somewhere on the circle drawn from the center of the goal with diameter *d1*. This functionality is implemented in Code Listing 5.

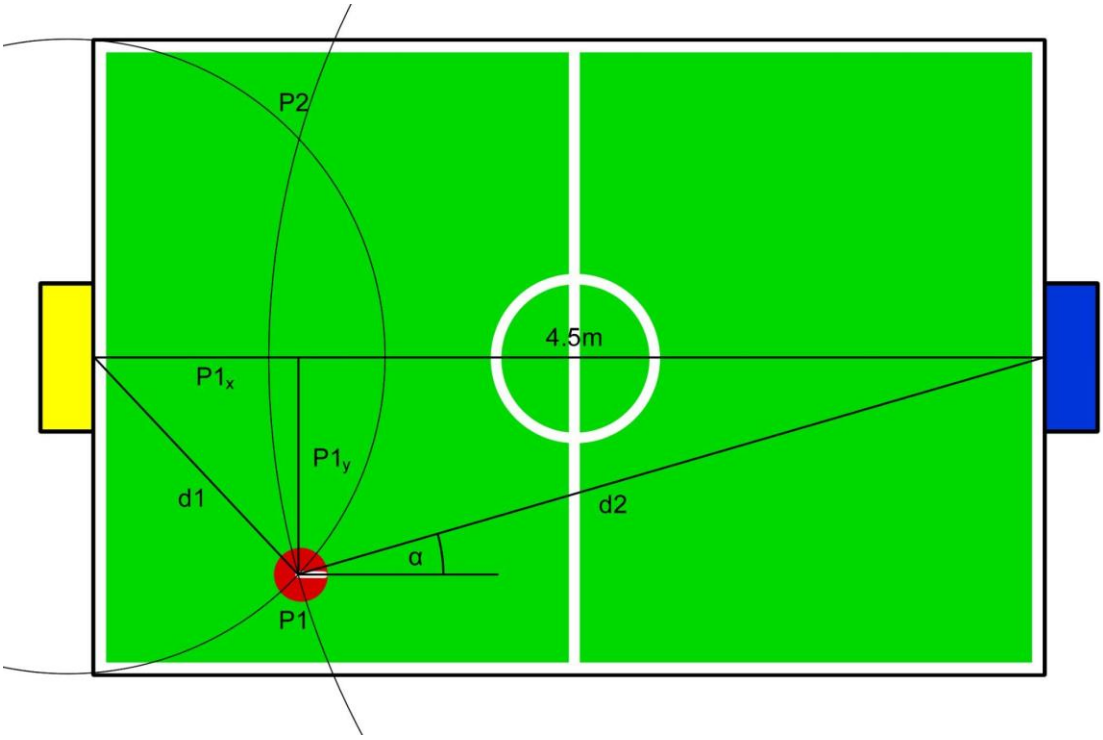


Figure 20. Calculating robot position from observing distance to both goals.

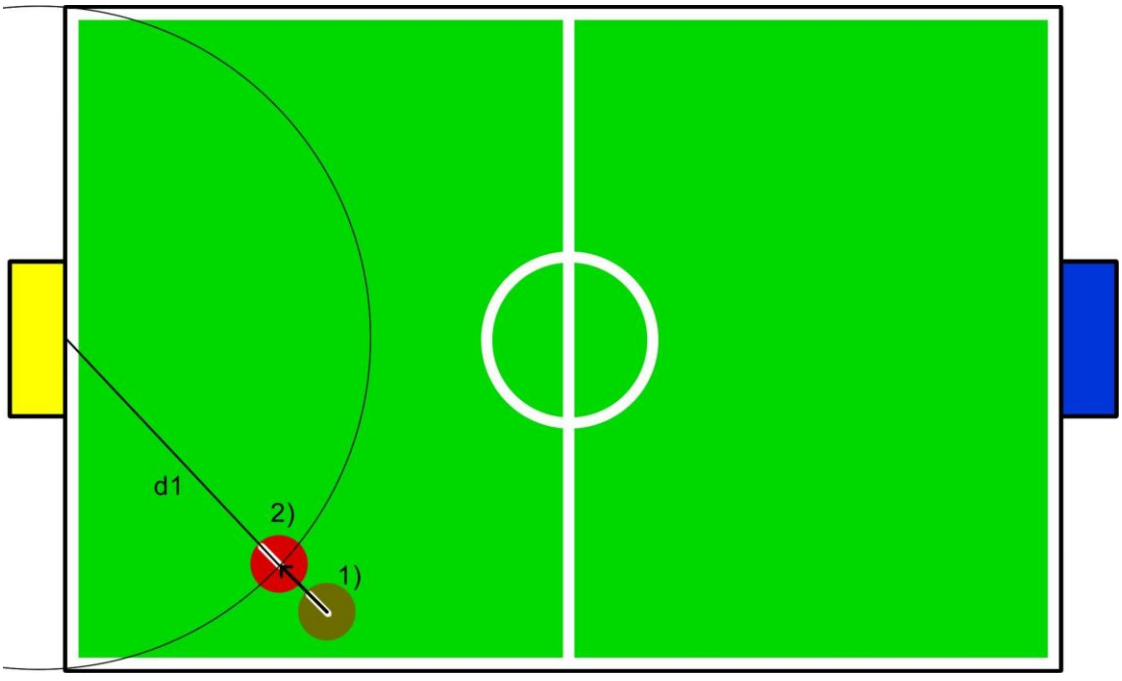


Figure 21. Improving position estimate from observing a single goal.

```
37      var currentPos = {
38              x: this.x,
39              y: this.y
40          },
41          dirVector,
42          scaledDir,
43          newPos;
44
45      if (yellowDistance !== null) {
46          dirVector = Sim.Math.createDirVector(currentPos, yellowGoalPos);
47          scaledDir = Sim.Math.createMultipliedVector(dirVector, yellowDistance);
48          newPos = Sim.Math.createVectorSum(yellowGoalPos, scaledDir);
49
50          this.x = newPos.x;
51          this.y = newPos.y;
52      } else if (blueDistance !== null) {
53          dirVector = Sim.Math.createDirVector(currentPos, blueGoalPos);
54          scaledDir = Sim.Math.createMultipliedVector(dirVector, blueDistance);
55          newPos = Sim.Math.createVectorSum(blueGoalPos, scaledDir);
56
57          this.x = newPos.x;
58          this.y = newPos.y;
59      }
```

Code Listing 5. Position adjustment from observing a single goal.

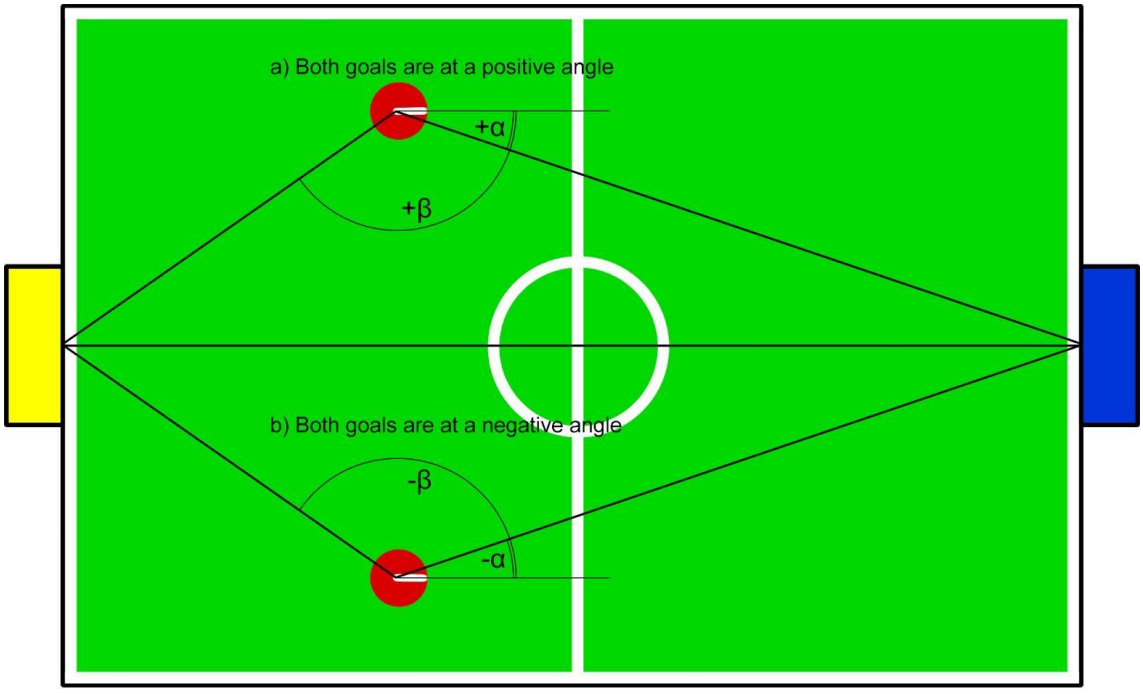
2.3.4 Intersection localizer

The second simplistic localization algorithm based directly on the observation of goal distances and angles is implemented in *library/sim/IntersectionLocalizer.js* of the simulator. The basic idea of this method is to find the intersection points of two circles drawn from the center of the goals with a radius of observed distances to them respectively. Given that the two goals are currently visible and distance measurement is sufficiently accurate for them to overlap, this gives us coordinates to two points P1, P2 on the field as was demonstrated in Figure 20.

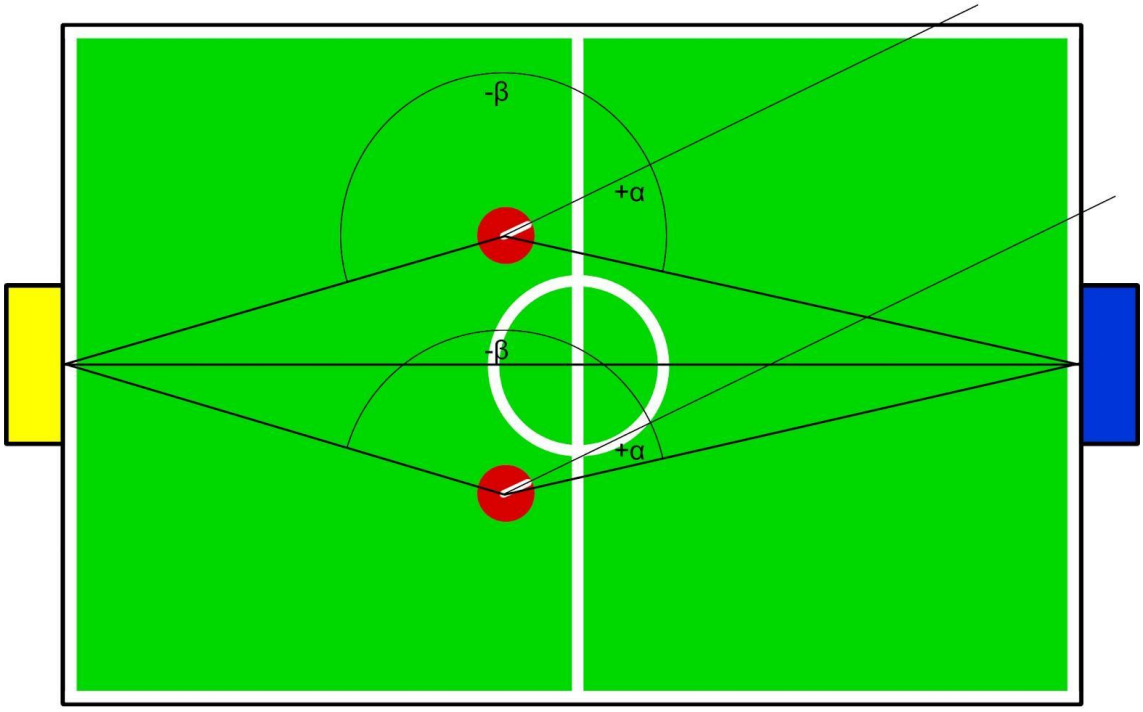
One now needs to decide in which of these the robot resides in. This can be done by comparing the last position of the robot to the new measurements and picking the one closest to previous. While this generally works, it can fail when the robot is near the horizontal centerline of the field as the intersections move very close together. The correct intersection can be uniquely detected if the observed angles to the two goals have the same sign meaning that both of the goals are either to the right or to the left of the robot. This is visualized in Figure 22.1. If on the other hand one of the goals is to the left of the robot and the other to the right from its centerline, the correct intersection point cannot be detected from just the distances and signs of angles to the goals as the same situation can occur for both intersection points as visualized in Figure 22.2. We

can see that the robot on both sides of the centerline observe the blue goal to be on its right (positive angle) and the yellow one on the left (negative angle). In such situations, the simulated intersection localizer algorithm picks the point closest to last position which can in some cases fail near the centerline as discussed above. This functionality is implemented in Code Listing 6.

In a physical robot implementation using a camera image for perception, one could extract the y-coordinates of the top or lower edges of the observed goals and from these extract whether the goal is viewed from the left or right. For example consider Figure 23 where the robot sees the blue goal with left top corner higher than the right implying that it is looking at it from the left. Extracting this information from the camera image is not trivial and this would again not work well around the centerline of the field where the goal would appear not skewed in the image.



1) The correct intersection point is uniquely determinable from the observed angles of the goals.



2) Both scenarios result in same angle signs, intersection is not identifiable from goal angles.

Figure 22. Detecting correct intersection point from observed goal angles.

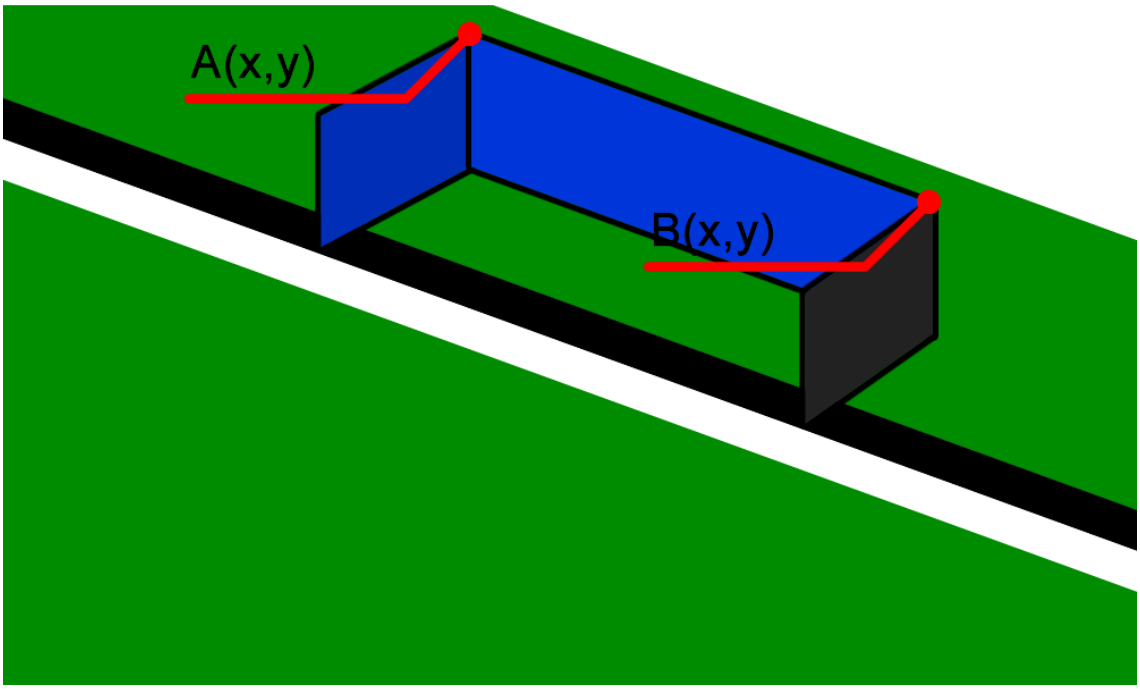


Figure 23. Deciding intersection field side from camera image.

```
64      var correctIntersection = 'unsure';
65
66      if (blueAngle > 0 && yellowAngle > 0) {
67          correctIntersection = frontGoal === 'blue' ? 'top' : 'bottom';
68      } else if (blueAngle < 0 && yellowAngle < 0) {
69          correctIntersection = frontGoal === 'blue' ? 'bottom' : 'top';
70      } else {
71          var distance1 = Sim.Math.getDistanceBetween(
72                  {x: intersections.x1, y: intersections.y1},
73                  {x: this.x, y: this.y}
74              ),
75              distance2 = Sim.Math.getDistanceBetween(
76                  {x: intersections.x2, y: intersections.y2},
77                  {x: this.x, y: this.y}
78              );
79
80          if (distance1 < distance2) {
81              this.x = intersections.x1;
82              this.y = intersections.y1;
83          } else {
84              this.x = intersections.x2;
85              this.y = intersections.y2;
86          }
87      }
88
89      if (correctIntersection === 'top') {
90          this.x = intersections.x2;
91          this.y = intersections.y2;
92      } else if (correctIntersection === 'bottom') {
93          this.x = intersections.x1;
94          this.y = intersections.y1;
95      }
```

Code Listing 6. Deciding which circle intersection to use.

The last step is calculating the robot orientation from the observed goal distances and angles which is given in Code Listing 7. Seeing both goals gives us two guesses and an average of them is used as the final orientation guess. Care must be taken when averaging angles taking the rollover over zero into account.

```
97      var verticalOffset = this.y - sim.config.field.height / 2.0,
98          zeroAngleBlue = Math.asin(verticalOffset / blueDistance),
99          zeroAngleYellow = Math.asin(verticalOffset / yellowDistance),
100         posYellowAngle = yellowAngle < 0 ? yellowAngle + Sim.Math.TWO_PI : yellowAngle,
101         posBlueAngle = blueAngle < 0 ? blueAngle + Sim.Math.TWO_PI : blueAngle,
102         yellowGuess = (Math.PI - (posYellowAngle - zeroAngleYellow)) % Sim.Math.TWO_PI,
103         blueGuess = (-zeroAngleBlue - posBlueAngle) % Sim.Math.TWO_PI;
104
105     while (yellowGuess < 0) {
106         yellowGuess += Sim.Math.TWO_PI;
107     }
108
109     while (blueGuess < 0) {
110         blueGuess += Sim.Math.TWO_PI;
111     }
112
113     this.orientation = Sim.Math.getAngleAvg(yellowGuess, blueGuess);
114
115     if (this.orientation < 0) {
116         this.orientation += Sim.Math.TWO_PI;
117     }
```

Code Listing 7. Extracting orientation.

2.3.5 Kalman filter localizer

The main issue with the intersection localizer explained in the previous section is that it is highly susceptible to the noise in the distance measurement sensor data making the guessed location of the robot jump around considerably. Any control algorithms depending on such variable location estimate would have problems maintaining stable state. We can alleviate this problem by passing the data through the Kalman filter which allows us to choose a balance between the odometer and visual sensory data. Relying more on the odometer makes the filter more stable but less responsive while the opposite makes it behave more like the intersection localizer alone. Kalman filtering can give us better results than simple odometer or intersection localizer because it fuses localization information from two independent sources.

The linear Kalman filter is implemented in *library/kalman/LinearKalmanFilter.js* of the simulator. The main prediction and observation procedures are given in Code Listing 8 which corresponds to the mathematical definition given in Section 1.3.3.

```
29  LinearKalmanFilter.prototype.predict = function(controlVector) {
30      this.predictedStateEstimate = this.stateTransitionMatrix
31          .multiply(this.stateEstimate)
32          .add(this.controlMatrix.multiply(controlVector));
33
34      this.predictedProbabilityEstimate = this.stateTransitionMatrix
35          .multiply(this.covarianceEstimate)
36          .multiply(this.stateTransitionMatrix.transpose())
37          .add(this.processErrorEstimate);
38
39      this.stateEstimate = this.predictedStateEstimate.dup();
40  };
41
42  LinearKalmanFilter.prototype.observe = function(measurementVector) {
43      this.innovation = measurementVector
44          .subtract(this.observationMatrix.multiply(
45              this.predictedStateEstimate
46          ));
47
48      this.innovationCovariance = this.observationMatrix
49          .multiply(this.predictedProbabilityEstimate)
50          .multiply(this.observationMatrix.transpose())
51          .add(this.measurementErrorEstimate);
52
53      this.kalmanGain = this.predictedProbabilityEstimate
54          .multiply(this.observationMatrix.transpose())
55          .multiply(this.innovationCovariance.inverse());
56
57      this.stateEstimate = this.predictedStateEstimate
58          .add(this.kalmanGain.multiply(this.innovation));
59
60      this.covarianceEstimate = Matrix.I(this.covarianceEstimate.dimensions().rows)
61          .subtract(this.kalmanGain.multiply(this.observationMatrix))
62          .multiply(this.predictedProbabilityEstimate);
63
64      this.predictedStateEstimate = this.stateEstimate.dup();
65  };
```

Code Listing 8. Kalman filter algorithm implementation.

The Kalman localizer is implemented in *lib/sim/KalmanFilterLocalizer.js.* First we need to decide which data to use and define the various matrices of the filter that control how it reacts to this input data. The first part of this initialization is given in Code Listing 9. The localization algorithm models the pose (x, y coordinates and orientation) of the robot along with its global velocity. This means that we need to use 5x5 matrices for our Kalman filter setup.

The state transition matrix defines the following rules that correspond to lines 29-33:

- $x(n+1) = x(n) + Vx(n)$
- $y(n+1) = y(n) + Vy(n)$
- $Vx(n+1) = velocityPreserve * Vx(n)$
- $Vy(n+1) = velocityPreserve * Vy(n)$
- $O(n+1) = O(n)$

The *velocityPreserve* is a parameter in range 0..1 which defines how much of the existing velocity information is trusted and in which amount will it be replaced by the input odometry data. The orientation is conserved by state transition. The control matrix defines how the control input vector is integrated into the estimate. Here we extract a portion of the control velocities and rotational velocity. The observation matrix is an identity matrix as we can observe all the parameters directly. The initial state estimate is initiated to known robot start pose.

```
28      this.stateTransitionMatrix = $M([
29          [1.00, 0.00, 1.00, 0.00, 0.00], // x
30          [0.00, 1.00, 0.00, 1.00, 0.00], // y
31          [0.00, 0.00, this.velocityPreserve, 0.00, 0.00], // Vx
32          [0.00, 0.00, 0.00, this.velocityPreserve, 0.00], // Vy
33          [0.00, 0.00, 0.00, 0.00, 1.00]  // orientation
34      ]);
35
36      this.controlMatrix = $M([
37          [0.00, 0.00, 0.00, 0.00, 0.00],
38          [0.00, 0.00, 0.00, 0.00, 0.00],
39          [0.00, 0.00, 1.0 - this.velocityPreserve, 0.00, 0.00],
40          [0.00, 0.00, 0.00, 1.0 - this.velocityPreserve, 0.00],
41          [0.00, 0.00, 0.00, 0.00, 1.00]
42      ]);
43
44      this.observationMatrix = $M([
45          [1, 0, 0, 0, 0],
46          [0, 1, 0, 0, 0],
47          [0, 0, 1, 0, 0],
48          [0, 0, 0, 1, 0],
49          [0, 0, 0, 0, 1]
50      ]);
51
52      this.initialStateEstimate = $M([
53          [x],
54          [y],
55          [0], // start velocity x
56          [0], // start velocity y
57          [orientation]
58      ]);
```

Code Listing 9. Kalman filter configuration.

The second part of the initialization is given by Code Listing 10. Here the initial covariance is defined along with the process and measurement error estimates. These are currently defined as constant values for all parameters but could be individually tuned for optimal results. Finally, an instance of the Kalman filter algorithm is created with all of the configuration parameters.

```
60      this.initialCovarianceEstimate = $M([
61          [this.initialCovariance, 0, 0, 0, 0],
62          [0, this.initialCovariance, 0, 0, 0],
63          [0, 0, this.initialCovariance, 0, 0],
64          [0, 0, 0, this.initialCovariance, 0],
65          [0, 0, 0, 0, this.initialCovariance]
66      ]);
67
68      this.processErrorEstimate = $M([
69          [this.processError, 0, 0, 0, 0],
70          [0, this.processError, 0, 0, 0],
71          [0, 0, this.processError, 0, 0],
72          [0, 0, 0, this.processError, 0],
73          [0, 0, 0, 0, this.processError]
74      ]);
75
76      this.measurementErrorEstimate = $M([
77          [this.measurementError, 0, 0, 0, 0],
78          [0, this.measurementError, 0, 0, 0],
79          [0, 0, this.measurementError, 0, 0],
80          [0, 0, 0, this.measurementError, 0],
81          [0, 0, 0, 0, this.measurementError]
82      ]);
83
84      this.filter = new LinearKalmanFilter(
85          this.stateTransitionMatrix,
86          this.controlMatrix,
87          this.observationMatrix,
88          this.initialStateEstimate,
89          this.initialCovarianceEstimate,
90          this.processErrorEstimate,
91          this.measurementErrorEstimate
92      );
```

Code Listing 10. Kalman filter initialization.

The main update method of it is given by Code Listing 11. As input, it requires the calculated pose of the robot from the intersection localizer, the commanded and sensed velocities and current timestep.

The first part of the update routine on lines 130 to 143 deal with the fact that orientation is sensed in the range from zero to 360°, which creates problems for the algorithm at the wrapping point around 0°. Imagine that the current orientation estimate is 350° and the next step, orientation of 10° is calculated. Even though the actual angle difference is only 20 degrees, the Kalman filter will start to incorporate this measurement by gradually reducing the orientation estimate, resulting in the estimate failing at each rotation of the robot. To deal with this issue, the rollovers are detected

and the algorithm keeps track of the number of rotations the robot has performed, adding these to the calculated orientation. This ensures that the orientation remains continuous at all times.

Lines 147 to 168 calculate the control and measurement vectors used as inputs to the Kalman filter. This step requires calculating the global command and odometer velocities using trigonometry from currently estimated orientation. The control vector includes the commanded velocities and omega while the measurement vector includes information from the odometer and calculated noisy pose. Lines 171 to 177 update the filter using the input data calculated above and extract the estimated pose.

```
Sim.KalmanLocalizer.prototype.move = function(
    x, y, orientation, cmdVelocityX, cmdVelocityY, cmdOmega,
    odoVelocityX, odoVelocityY, odoOmega, dt
) {
    var originalOrientation = orientation,
        jumpThreshold = 0.1;

    if (
        orientation < jumpThreshold
        && this.lastInputOrientation > Sim.Math.TWO_PI - jumpThreshold
    ) {
        this.rotationCounter++;
    } else if (
        orientation > Sim.Math.TWO_PI - jumpThreshold
        && this.lastInputOrientation < jumpThreshold
    ) {
        this.rotationCounter--;
    }

    orientation = orientation + this.rotationCounter * Sim.Math.TWO_PI;

    var globalCmdVelocityX = (cmdVelocityX * Math.cos(this.orientation)
            - cmdVelocityY * Math.sin(this.orientation)) * dt,
        globalCmdVelocityY = (cmdVelocityX * Math.sin(this.orientation)
            + cmdVelocityY * Math.cos(this.orientation)) * dt,
        globalOdoVelocityX = (odoVelocityX * Math.cos(this.orientation)
            - odoVelocityY * Math.sin(this.orientation)) * dt,
        globalOdoVelocityY = (odoVelocityX * Math.sin(this.orientation)
            + odoVelocityY * Math.cos(this.orientation)) * dt,
        controlVector = $M([
            [0],
            [0],
            [globalCmdVelocityX],
            [globalCmdVelocityY],
            [cmdOmega * dt]
        ]),
        measurementVector = $M([
            [x],
            [y],
            [globalOdoVelocityX],
            [globalOdoVelocityY],
            [orientation]
        ]),
        state;

    this.filter.predict(controlVector);
    this.filter.observe(measurementVector);
    state = this.filter.getStateEstimate();
    this.x = state.e(1, 1);
    this.y = state.e(2, 1); // matrix second row first column value
    this.orientation = state.e(5, 1);
    this.lastInputOrientation = originalOrientation;
};
```

Code Listing 11. Kalman filter localizer main update step.

There are several ways one could choose to set up the Kalman filter for the same task, for example, it is possible to work at lower level of abstraction such as individual wheel speeds or even the pixels in the image, however the linear version of the Kalman filter would generally not be applicable in these cases and the implementation would be more complex. Our implementation uses another simpler intersection-based filter output as its input to merge the directly observable noisy location data with more stable but eventually drifting odometer data producing smooth and accurate results in the simulator

2.3.6 Particle filter localizer

The particle filter is an alternative localization technique discussed in this thesis which does not rely on any other input but the distances and azimuth angles to the landmarks on the field.

The general idea is to create a set of particles, each with the pose initialized at the known starting point of the robot. Then a set of landmarks is defined, each having a name and global position x, y. For current implementation, just the distances and angles to the two goals are used but this list could be extended for better localization if we could detect more landmarks such as the field center, corners and intersections. At each iteration of the algorithm (that generally corresponds to the new frame of information extracted from the cameras), all of the particles are moved by the relative motion vector extracted from the odometer plus some additional movement and orientation noise individual to each particle. This evolves the set of particles to a new state where they deviate from each other and spread out a little, enabling them to represent states that the odometer data alone would not take and it is likely that some of the particles will move closer to the real position of the robot.

Next, measurement data from the cameras is considered in the form of distances and angles to landmarks. For each particle, the probability of observing given measurements is calculated. For example if we measured that both goals are at approximately equal distance from the robot, a particle in the corner of a field would not match this observation data well as we'd expect the robot to be somewhere in the middle of the field. Based on the calculated measurement probabilities, the set of particles are resampled where the ones which matched the measurement data more precisely are more likely to appear in the resampled set.

The particle filter is implemented in the simulator in JavaScript file library/sim/ParticleLocalizer.js. The main setup code for the algorithm is given in Code Listing 12. In the constructor, one can see five input parameters – how many particles to use and the various noise parameters for translation and orientation, sensing the distance and angle. These are tunable to change the characteristics of the filter. One of the strengths of the particle filter is that the trade-off between performance and required computational resources is easily configurable by changing the number of used particles.

Lines 17-22 define the particle that is nothing more but the pose combined with probability of the particle getting resampled in the update phase. Lines 24-33 implement initiating the set of particles uniformly distributed across the playing field. Alternatively, the particles could be initialized at a specific position and orientation using the *setPosition* method defined on lines 42-49, as generally we know the starting position of the robot at the start of each round. Lines 35-40 define the method for registering new landmarks, each having a name and a position.

```
Sim.ParticleLocalizer = function(
    particleCount, forwardNoise, turnNoise,
    distanceSenseNoise, angleSenseNoise
) {
    this.particleCount      = particleCount || 1000;
    this.forwardNoise       = forwardNoise || 0.1;
    this.turnNoise          = turnNoise || 0.1;
    this.distanceSenseNoise = distanceSenseNoise || 0.1;
    this.angleSenseNoise    = angleSenseNoise || Sim.Math.degToRad(5);
    this.landmarks          = {};
    this.particles          = [];
    this.x                  = 0;
    this.y                  = 0;
    this.orientation        = 0;
};

Sim.ParticleLocalizer.Particle = function(x, y, orientation, probability) {
    this.x = x;
    this.y = y;
    this.orientation = orientation;
    this.probability = probability;
};

Sim.ParticleLocalizer.prototype.init = function() {
    for (var i = 0; i < this.particleCount; i++) {
        this.particles.push(new Sim.ParticleLocalizer.Particle(
            Sim.Util.random(0, sim.config.field.width * 1000) / 1000.0,
            Sim.Util.random(0, sim.config.field.height * 1000) / 1000.0,
            Sim.Util.random(0, Sim.Math.TWO_PI * 1000) / 1000.0,
            1
        ));
    }
};

Sim.ParticleLocalizer.prototype.addLandmark = function(name, x, y) {
    this.landmarks[name] = {
        x: x,
        y: y
    };
};

Sim.ParticleLocalizer.prototype.setPosition = function(x, y, orientation) {
    for (var i = 0; i < this.particles.length; i++) {
        this.particles[i].x = x;
        this.particles[i].y = y;
        this.particles[i].orientation = orientation;
        this.particles[i].probability = 1;
    }
};
```

Code Listing 12. Particle filter setup.

Code Listing 13 shows the two main methods of the particle filter that implement the move and update steps of the algorithm. The move method gets the relative velocity x, y components and the omega as input that are extracted from the odometry. There are two additional parameters: *dt* that marks how long the last step took and *exact* which, when true, indicates that robot cameras failed to spot any landmarks. This information is used in lines 58-66 so that in case no landmarks were observed, no noise is added to the odometry information that could cause the particles to diverge without the update step resampling them according to the measurement data. The update method simply updates the position and orientation of each of the particles by applying the odometry movement combined with some artificial noise evolving the particles to new states.

The update method is passed a map of measurements as input where the keys are the names of the landmarks and the values include the distance and angle to given landmark. For each particle, the probability of observing such measurements given the particle pose is calculated. The update loop keeps track of the largest probability observed and uses this in lines 96-98 to normalize the probability to the range between zero and one. Finally in line 100 the set of particles are resampled based on their measurement probabilities.

```
Sim.ParticleLocalizer.prototype.move = function(velocityX, velocityY, omega, dt, exact) {
    var particleVelocityX,
        particleVelocityY,
        particleOrientationNoise,
        i;

    for (i = 0; i < this.particles.length; i++) {
        if (exact !== true) {
            particleVelocityX = velocityX + Sim.Util.randomGaussian(this.forwardNoise);
            particleVelocityY = velocityY + Sim.Util.randomGaussian(this.forwardNoise);
            particleOrientationNoise = Sim.Util.randomGaussian(this.turnNoise) * dt;
        } else {
            particleVelocityX = velocityX;
            particleVelocityY = velocityY;
            particleOrientationNoise = 0;
        }

        this.particles[i].orientation = this.particles[i].orientation
            + omega * dt + particleOrientationNoise;
        this.particles[i].x += (particleVelocityX * Math.cos(this.particles[i].orientation)
            - particleVelocityY * Math.sin(this.particles[i].orientation)) * dt;
        this.particles[i].y += (particleVelocityX * Math.sin(this.particles[i].orientation)
            + particleVelocityY * Math.cos(this.particles[i].orientation)) * dt;
    }
};

Sim.ParticleLocalizer.prototype.update = function(measurements) {
    if (Sim.Util.isEmpty(measurements)) {
        return;
    }

    var particle,
        maxProbability = null,
        i;

    for (i = 0; i < this.particles.length; i++) {
        particle = this.particles[i];

        this.particles[i].probability = this.getMeasurementProbability(particle, measurements);

        if (maxProbability == null || this.particles[i].probability > maxProbability) {
            maxProbability = this.particles[i].probability;
        }
    }

    for (i = 0; i < this.particles.length; i++) {
        this.particles[i].probability /= maxProbability;
    }

    this.particles = this.resample(this.particles);
};
```

Code Listing 13. Move and update methods of the particle filter implementation.

Code Listing 14 starts with the measurement probability calculation method. As input, it receives the particle under consideration and the map of measurements extracted from the camera image. The probability is initiated with a value of one. The algorithm then cycles through all of the measurements available for given step. As the landmarks are uniquely distinguishable in our case (we can detect whether the goal is

yellow or blue), we can directly map the measurements to landmarks. The algorithm then extracts the observed distance and angle to the landmark and calculates the expected distance and angle given the particle's current position and orientation. Sum of Gaussians from the differences of these expected and observed measurements are then multiplied to form the probability metric. Two configuration variables for expected distance and angle noise are included in this calculation. As the probabilities from multiple landmark measurements are multiplied, several strongly correlating measurements fortify the overall probability and vice versa.

```
Sim.ParticleLocalizer.prototype.getMeasurementProbability = function(
    particle,
    measurements
) {
    var probability = 1.0,
        landmarkName,
        landmark,
        measuredDistance,
        measuredAngle,
        expectedDistance,
        expectedAngle;

    for (landmarkName in measurements) {
        landmark = this.landmarks[landmarkName];
        measuredDistance = measurements[landmarkName].distance;
        measuredAngle = measurements[landmarkName].angle;
        expectedDistance = Sim.Math.getDistanceBetween(particle, landmark);
        expectedAngle = Sim.Math.getAngleBetween(landmark, particle, particle.orientation);

        probability *= Sim.Math.getGaussian(expectedAngle, this.angleSenseNoise, measuredAngle)
            + Sim.Math.getGaussian(expectedDistance, this.distanceSenseNoise, measuredDistance);
    }

    return probability;
};

Sim.ParticleLocalizer.prototype.resample = function(particles) {
    var resampledParticles = [],
        particleCount = particles.length,
        index = Sim.Util.random(0, particleCount - 1),
        beta = 0.0,
        i;

    for (i = 0; i < particles.length; i++) {
        beta += Math.random() * 2.0;

        while (beta > particles[index].probability) {
            beta -= particles[index].probability;
            index = (index + 1) % particleCount;
        }

        resampledParticles.push(new Sim.ParticleLocalizer.Particle(
            particles[index].x,
            particles[index].y,
            particles[index].orientation,
            particles[index].probability
        ));
    }

    return resampledParticles;
};
```

Code Listing 14. Measurement probability calculation and resampling procedure.

The second method resamples the set of particles based on their measurement probabilities. While the probability value returned by the measurement probability function does not have a fixed range, remember that the update method discussed above normalized the probability values to 0..1 range. The resample method implements low-variance sampler algorithm introduced in section 1.4.5 and requires

linear time in the number of particles. The basic idea is that we divide the particles on a ring where the size of the "slice" each particle occupies is proportional to its measurement probability. We then start with a random particle index from the set and in a loop, resample a new set of the same number of particles. To do so, beta is increased by a random uniform value between zero and twice the largest probability in the set (note that since in our particle filter implementation, the probabilities are normalized, the largest probability value is always one so this is omitted). Now for as long as the probability of particle at given index is smaller than this beta, the current index particle probability is subtracted from beta and index is increased. Eventually we will find a particle that matches the beta value and this is resampled into the new set. Notice that the uniform random value added to beta can be small enough so that the same particle can be picked several times. This method of resampling is illustrated in Figure 24.

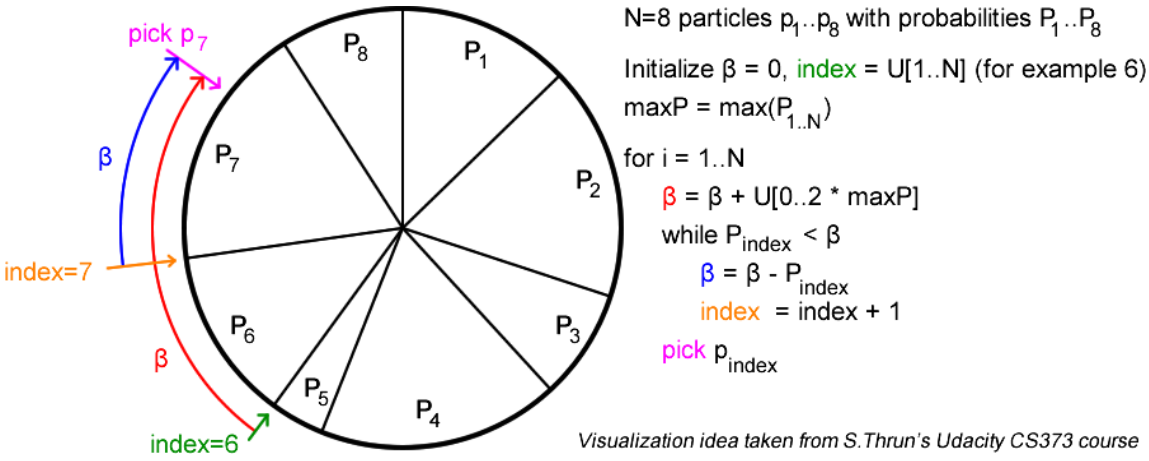


Figure 24. Particle filter efficient resampling algorithm example [18].

The last step of the particle filter is extracting a single pose from the set of particles. As we are dealing with local localization problem with a single position estimate, this implementation simply averages the positions and orientations of the entire particle set as shown in Code Listing 15. One needs to be careful when working with orientations as generally angles are kept in the range from 0 to 360 degrees (2π). If the algorithm just averaged the angles, some of which are for example just above zero and some just short of 360, one would get values around 180 degrees, which is not correct. Thus the algorithm does not limit the range of angles during its execution, but rather the orientation is truncated to the normal range only in the pose extraction method.

```
184  Sim.ParticleLocalizer.prototype.getPosition = function(robot) {
185      var evaluation = this.evaluate(robot, this.particles),
186          xSum = 0,
187          ySum = 0,
188          orientationSum = 0,
189          i;
190
191      for (i = 0; i < this.particles.length; i++) {
192          xSum += this.particles[i].x;
193          ySum += this.particles[i].y;
194          orientationSum += this.particles[i].orientation;
195      }
196
197      this.x = xSum / this.particles.length;
198      this.y = ySum / this.particles.length;
199      this.orientation = (orientationSum / this.particles.length) % Sim.Math.TWO_PI;
200
201      while (this.orientation < 0) {
202          this.orientation += Sim.Math.TWO_PI;
203      }
204
205      return {
206          x: this.x,
207          y: this.y,
208          orientation: this.orientation,
209          evaluation: evaluation
210      };
211  };
```

Code Listing 15. Extracting position and orientation from set of particles.

3 Algorithm performance comparison

In the following sections we tune the parameters of the localization algorithms to work best in the simulation environment and then compare their performance. As the simulator works in modern browsers without the need for compilation, the reader is encouraged to clone the open-source repository at https://github.com/kallaspriit/Robocup-Simulator and play around with it and the various algorithms.

3.1 Simulator setup

As part of the thesis, an application was developed that simulates the main dynamics of a soccer-playing robot, enabling the development and testing of more complex algorithms such as localization methods described in this thesis. Using a simulator is useful as complex algorithms tend to be hard to test on a physical robot. Once the robot is started, it is hard to understand why it is behaving the way it behaves and what is the internal state of the robot. The simulator is written in JavaScript using HTML5 web-technologies making experimenting efficient as there is no need to re-compile the code on every iteration. Figure 25 shows it in action with a legend of data it visualizes. Simulation also makes the code easier to debug as modern browsers include powerful developer tools enabling setting breakpoints on live code, setting up watch expressions and stepping through code. At any point during development and testing, there is a clear overview of what the robot sees and thinks of the environment around it.

The simulator accurately models the omnidirectional motion model as described in the motion model and odometry Section 2.2. The same code was ported to C++ for the real robot and it worked without modifications, showing the power of simulator-based prototyping. The motion model also specifies the way of calculating motion from wheel-speeds used for odometry. The virtual odometer includes a Gaussian noise model proportional to the rotational velocity of individual wheels. This makes the odometry data drift from true position in time as is the case for the real robot.

The camera is modeled as a polygon defining a field of view denoted by the lighter area in Figure 25. The camera reports back the distance and relative angle from robot's orientation to the balls and goals that are located in camera view. Gaussian noise is also added to the distance and angle readings with the error increasing with

distance as is the case with the real robot. The playing field, balls and robot have accurate dimensions and approximately realistic attributes such as maximum velocity and turn rate. The simulator includes basic physics simulation of the balls bouncing off walls, the robots and each other.

The robot is controlled by setting the relative direction to move at (the robot utilizes omnidirectional wheels enabling moving in any direction), how fast to move and how fast to turn about its axis at the same time. In the front of the robot there is an area in which balls get attached to the robot when approached, simulating the dribbler mechanism holding the ball. The robot also has a kicker mechanism enabling it to propel the ball in the same direction that it is facing. The robot itself does not have any behavioral logic, it is commanded by a controller. The robot can be controlled by different controllers such as different control algorithms or people using keyboard or gamepad. This separation makes it easy to test different controller algorithms on a single robot model.

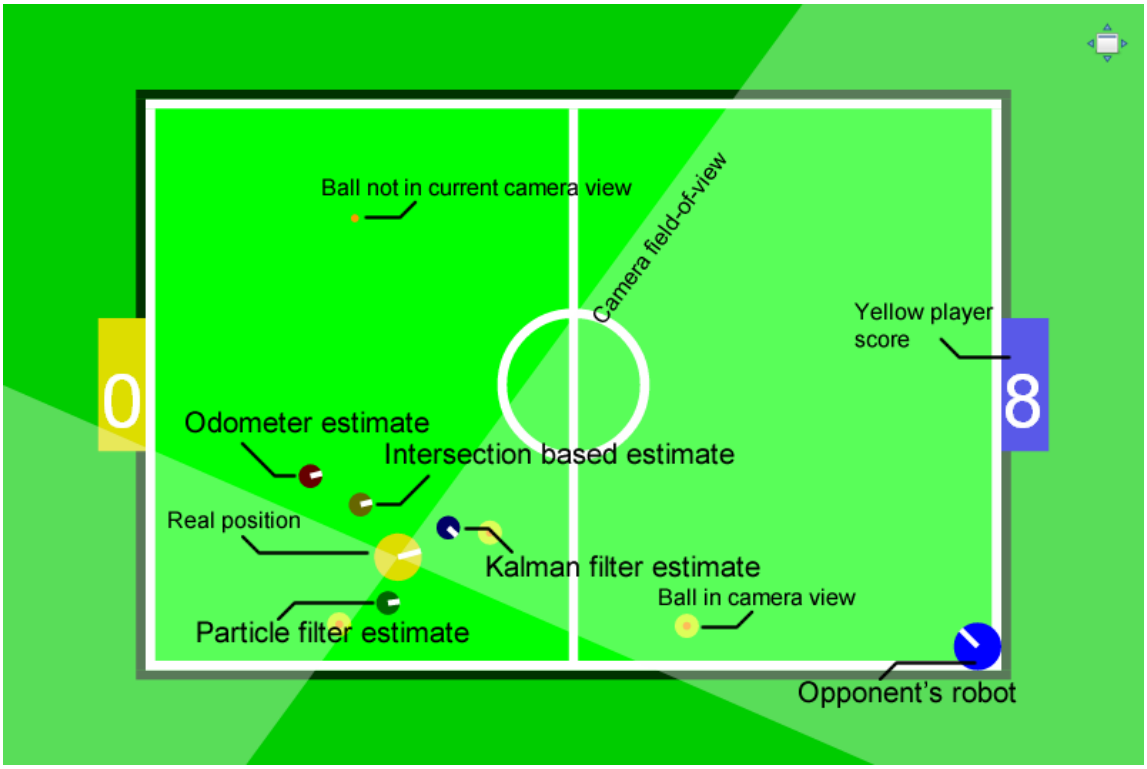


Figure 25. Simulator screenshot with legend.

3.2 Test methodology

The simulator includes a simple control algorithm that can win the game (i.e. clear the field of the balls) in about 27 seconds on average. For general simulation purpose, the ball positions are generated randomly, symmetrical to the centerline, but for localization algorithm testing, the positions of the balls were fixed so the path that the robot takes was always the same. This reduced the variation of localizer performance. Small variation remained as the noise in the odometer and filters are generated randomly.

In the simulator, we always precisely know the actual position and orientation of the robot so we can directly compare it to the pose predicted by the localization algorithms. At each iteration of the simulator, the distance error from the real location of the robot to the one predicted by the localization algorithms is calculated and stored. From this information set, the average error (average difference between real and predicted position) is calculated. The same metric was calculated for the difference in orientation.

3.3 Simulator parameters

The simulation environment and localization algorithms include a number of user-tunable parameters. While it's hard to create a simulator that would very accurately match the real world, the environments behave sufficiently similarly to test out various ideas and map the strengths and weaknesses of algorithms. Below is the table of defaults for the most important simulation parameters.

Parameter name	Value	Comment
Timestep	1/60th seconds.	Simulation runs at 60 FPS.
Vision distance noise	Gaussian noise 0.1m multiplied by distance to object.	Objects further away are sensed with more noise.
Vision angle noise	5 degrees Gaussian noise.	Error on the real robot is likely not Gaussian, but works for simulation.
Wheel angular velocity noise	Gaussian noise 0.3 multiplied by the angular velocity of given wheel.	Faster spinning wheels are noisier.

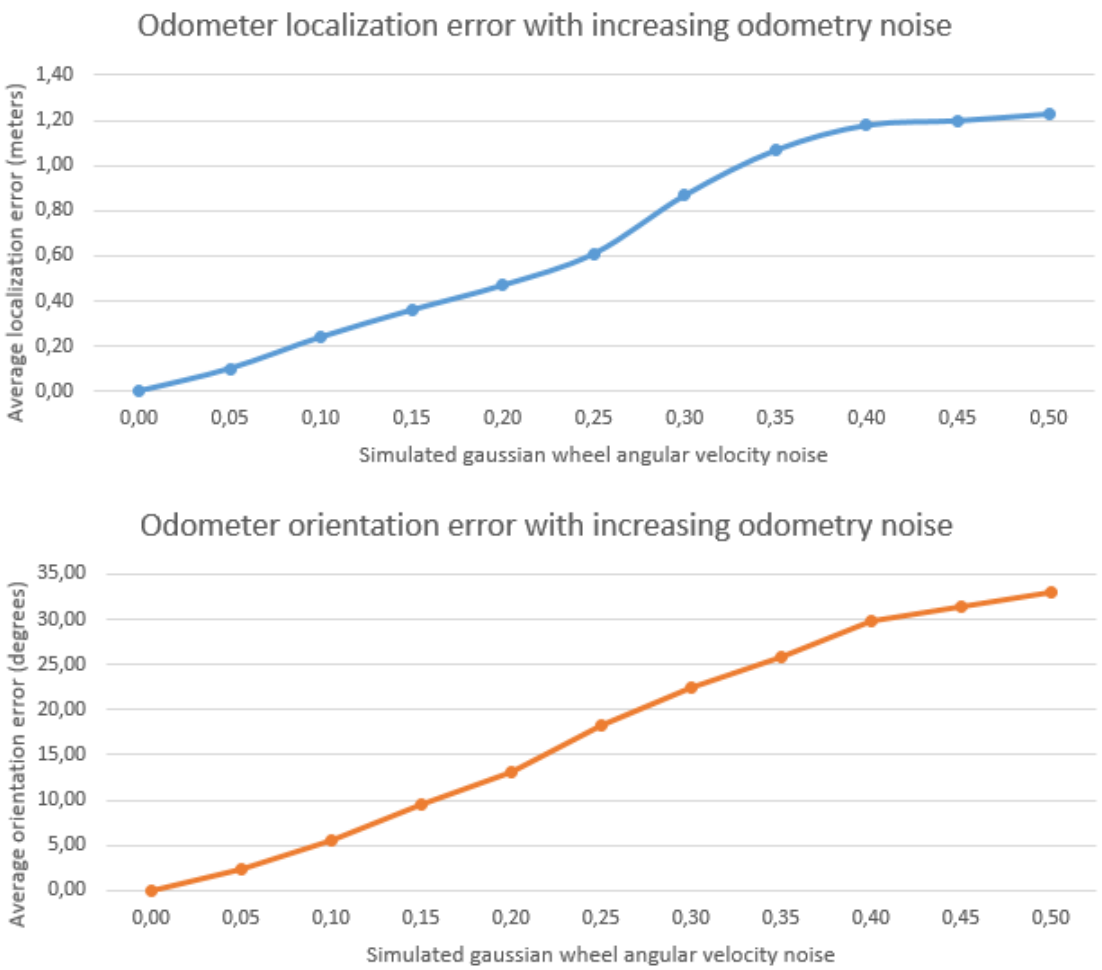
3.4 Experiments

The following section tests the localization algorithms and attempts to tune them for best performance. The odometer- and intersection-based solutions are not tunable as their performance depends solely on the amount of noise introduced into the system.

3.4.1 Odometer and intersection based localizer

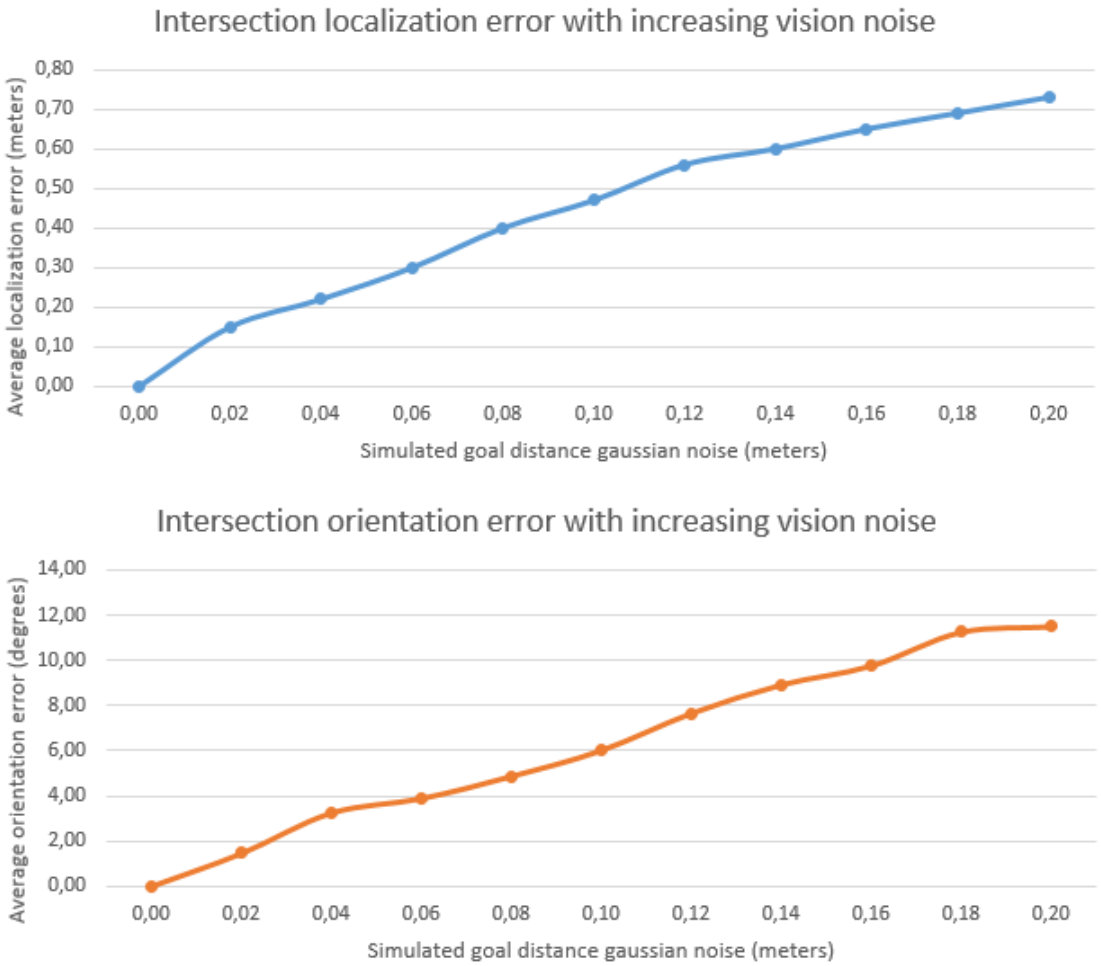
The pure odometer and intersection based localizers do not have any parameters to tune but one may investigate the relationship between the amount of noise in the system and their performance.

For odometer, the noise in the wheel rotational velocities was gradually increased and the localization error monitored both for distance from real location and the difference in orientation. For lower noise values, two runs were averaged and the localizer gave relatively consistent results. As the noise level increased, the variation from run-to-run increased as the earlier the estimation got off track, the further it deviated by the end of the round so five runs were averaged to obtain the result. While this smoothed out the error curves, pure odometer approach still proved unusable for even rather low noise. Graph 1 shows the distance and orientation errors with increasing wheel velocity noise. The relationship seems to be linear – the more noise in the system the more it will deviate from the actual pose.



Graph 1. Odometer error in relation with the amount of noise in the system.

For intersection localizer, the Gaussian noise of observed goal distances was gradually increased. Remember that the applied noise is multiplied with the actual distance to the goal so that the distance to a goal near the robot is determined with more precision than to one further away. This is the observed behavior of the real robot and makes sense as objects further away occupy less vertical pixels than those close by. Graph 2 shows the localization error with increasing observed goal distance noise. While the relationship is more or less linear, it should be noted that with higher noise, the position estimate varied by large amounts from frame to frame in the simulator making this information hard to process for any control algorithms. On a real robot, the goal distance observations are expected to be more stable and depend more on the motion of the robot as slightly uneven terrain and omni-wheels produce vibrations.



Graph 2. Intersection localizer error with increasing goal distance noise.

3.4.2 Kalman filter localizer

The Kalman filter localizer implementation is primarily affected by the following parameters:

Parameter name	Comment
Vision distance noise	Affects the performance of intersection-localizer used as base. Objects further away are sensed with more noise.
Process error	Expected error in the model, smaller value makes the filter rely more on the model and larger on measurements.
Measurement error	Expected error in measurements, smaller value makes the filter rely more on the measurements and larger on model.

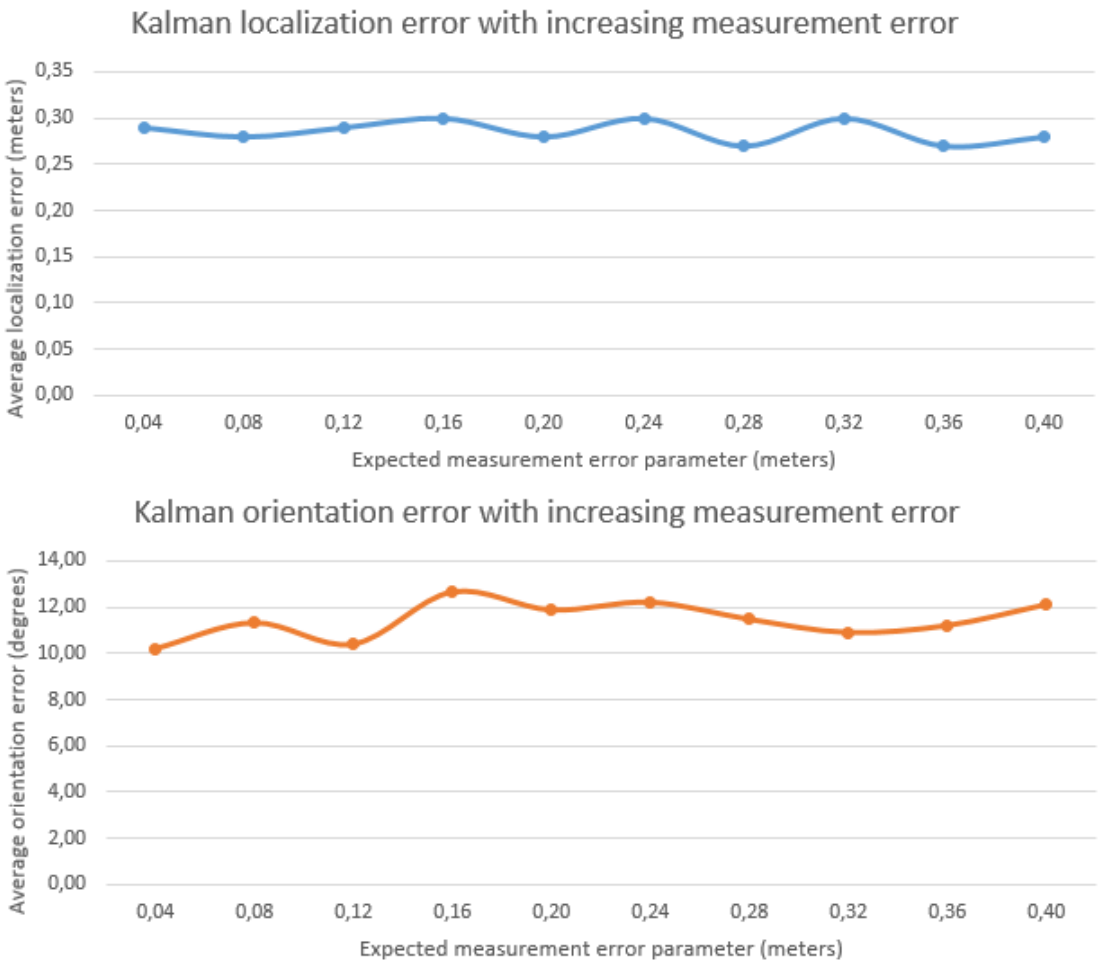
Initial covariance	Defaults to a low value of 0.0001, has little effect as the initial pose of the robot is known.

Graph 3 shows the relationship of Kalman filter accuracy to observed goal distance noise. As expected, the curve has similar shape to intersection filter as the latter is used as observation input. The absolute values here have little meaning as they depend on how the filter is configured as when relying more on the observation data, it will more closely follow the intersection localizer while relying more on motion model (smaller process error and larger measurement error), it will follow the odometer localizer results. This tunable fusion of observation data with model information is what makes the Kalman filter useful for our purpose.

Graph 3. Kalman filter location accuracy based on goal observation distance noise.

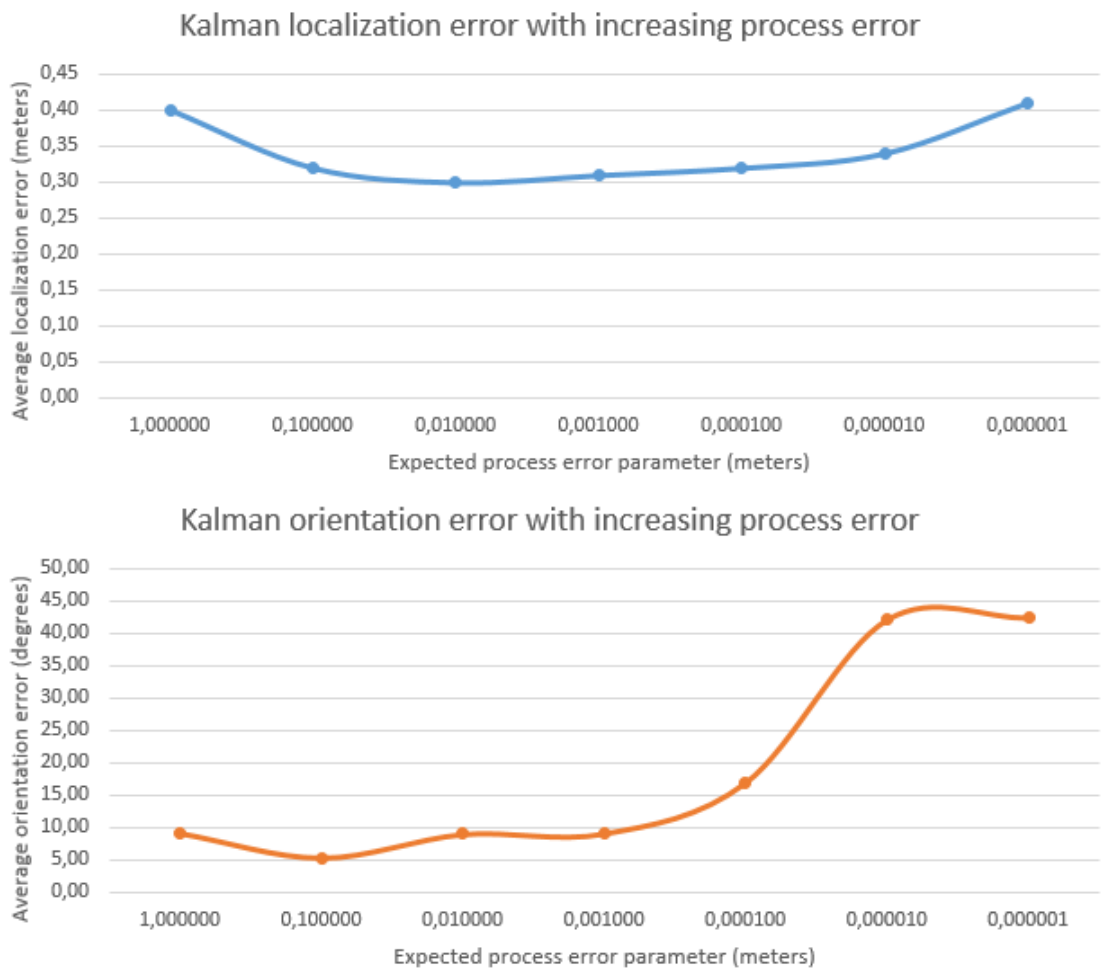
This section studies the effect of process error and measurement error parameters. It is not possible to set the process error to equal zero as this would result in a zero-filled innovation covariance matrix that cannot be inversed. Setting measurement error to zero will make the filter follow measurements exactly making the filter do basically nothing but repeat the input observations from the intersection filter.

Graph 4 shows the Kalman filter performance with increasing measurement error configuration parameter. The distance noise used for these experiments was 0.1m and the wheel rotational velocities noise 0.3 as determined to be realistic to the eye in comparison with the real robot. The process error was set to a low value 0.0001, representing a reasonably low noise in the odometer although the best value for this parameter is not easy to determine. On average, the localization error in both position and orientation is not greatly affected by the measurement error parameter. On the other hand, visually, the effect is noticeable as with smaller expected error, the position estimate varies considerably from frame to frame making it not a good input to any control algorithms. When the expected measurement error is large, the estimate becomes much more stable but the algorithm also becomes more susceptible to longer periods of heading in the wrong direction and larger error as it takes some time for the measurements to propagate and relocate the lost estimate. An optimal value of 0.25 was finally chosen for the simulation giving visually sufficiently stable yet responsive behavior.



Graph 4. Kalman localizer measurement error parameter effect.

The process error parameter has similar effect to measurement error as tuning it makes the filter follow more either the measurement information or the motion model. Graph 5 shows that the parameter has little effect on the average location error and orientation error tends to increase with smaller parameter value. For larger process error parameter values, the filter follows the measurements and the position estimate is not stable. For smaller values, the estimate becomes more stable but the chances of the localizer getting off track is bigger as location error takes more time to be recovered from measurements. A value of 0.0001 was chosen for this parameter which provided the best compromise between short and long term stability.



Graph 5. Kalman filter process error parameter effect.

3.4.3 Particle filter localizer

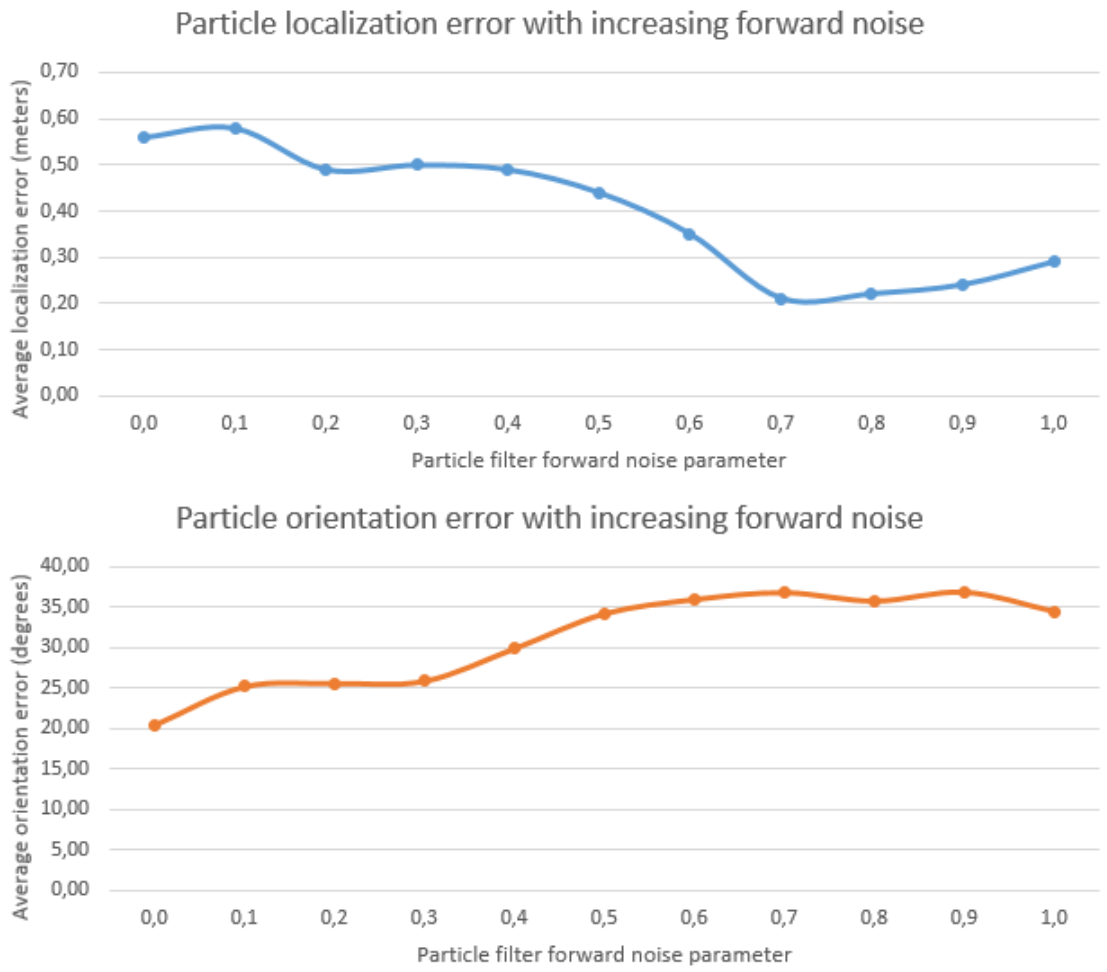
The particle filter has four main configuration parameters – the number of particles, forward movement noise and turn noise and the expected goal distance sense noise.

Parameter name	Comment
Forward noise	Smaller value makes filter more stable, larger makes it less likely to lose track and reacquire it faster should it happen
Turn noise	Smaller value makes filter more stable, larger makes it less likely to lose track and reacquire it faster should it happen
Distance sense noise	Smaller value focuses the particles more near measurements, larger allow relying more on the model

	with noisy sensors
Number of particles	Larger value provides better accuracy at the expense of increased computational resources required

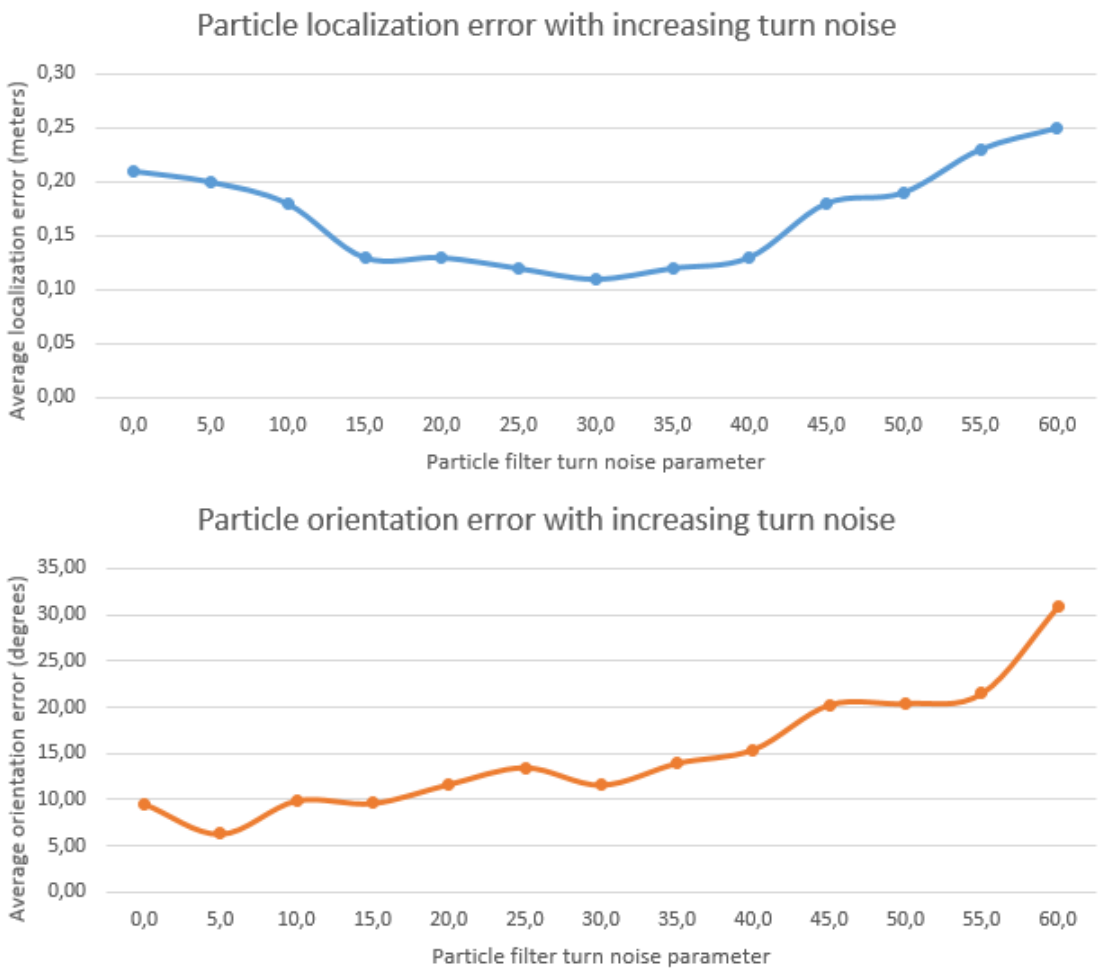
The following experiments use vision noise of 0.1m which is the same as actually generated noise. For number of particles, a value of 1000 is used while tuning the forward and turn noise. These movement noise parameters are not independent so tuning them individually is not straightforward. Turn noise value of 45° was used while testing the forward noise.

Graph 6 shows that the average pose error is mostly flat or slightly decreasing with larger simulated forward noise. It was observed that a low value provides more stable location estimate but is more subject to losing track of real location and takes longer to recover from such an event. Larger generated noise on the other hand makes the estimate unstable but recovers quicker as the particles get spread further apart and thus there are more particles near the real location should the localizer lose track. A value of 0.75 was determined to give the best results and will be used in later experiments.



Graph 6. Particle filter forward noise parameter effect.

The second motion noise parameter to tune is the turn noise. Graph 7 shows the localizer performance with increasing generated turn noise. It was observed that low values are more subject to losing track of the real location as the particles do not spread much and are more likely to all head in the wrong direction. High values introduced large spread which helps resolve losing track issues but produce excess noise, making the estimate unstable and introducing tracking problems. A medium value of 45° proved to work reasonably well in the simulator.



Graph 7. Particle filter turn noise parameter effect.

Next important parameter to consider is the expected distance noise. This sets the relationship between measurement probability and by which degree is the observed value different from expected value. As Figure 26.a demonstrates, a low value causes less particle spread as only those close to the measurement values are likely to survive (get resampled). This makes the filter behavior less stable. A large value shown in Figure 26.b makes the particles spread out making the filter more stable but also more subject to drifting away from true location as particles further away from expected location get resampled back into the set more likely. Large values thus have more variance, sometimes performing well and other times losing track. Graph 8 shows the localization performance with increasing expected distance noise. A medium value of 0.1 was observed to work well which is expected as it is the same as the actual generated goal distance Gaussian noise.

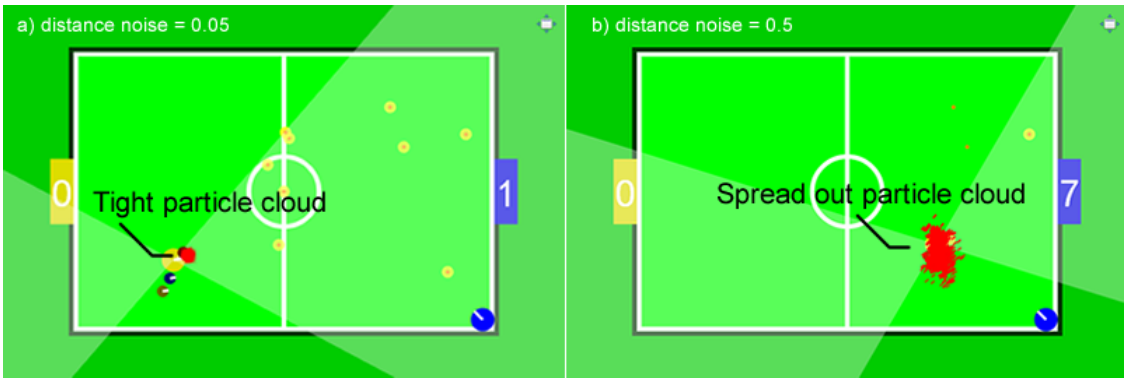
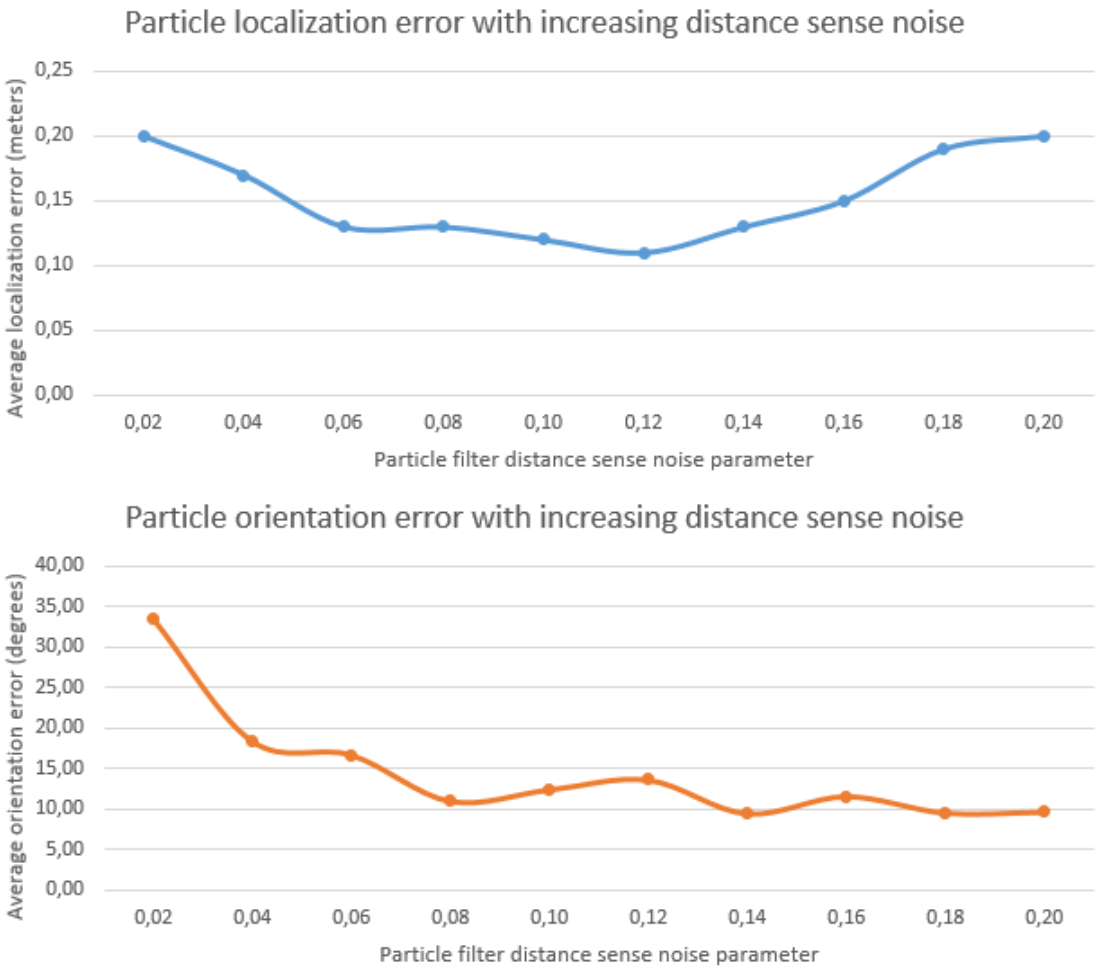


Figure 26. Particle filter distance noise setting affecting particle spread.



Graph 8. Particle filter expected distance noise parameter effect.

An interesting thing to notice is that when the robot only sees a single goal, there is generally a circle of possible locations it could be at. Given enough time for the particles to deviate can produce a situation like shown in Figure 27 (it took about 10 minutes to evolve to such state). Once the robot started moving and saw both goals again, the localizer was able to quickly recover. One could add extra rules not to allow particles to stray outside the playing field.

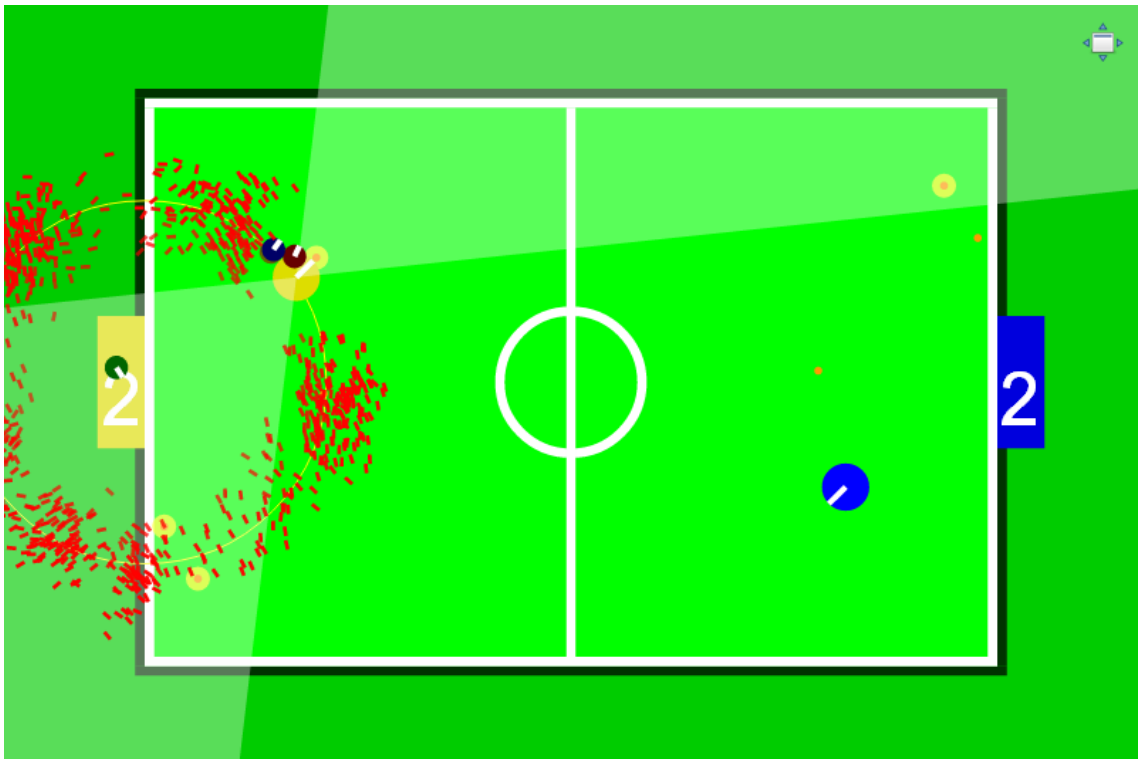
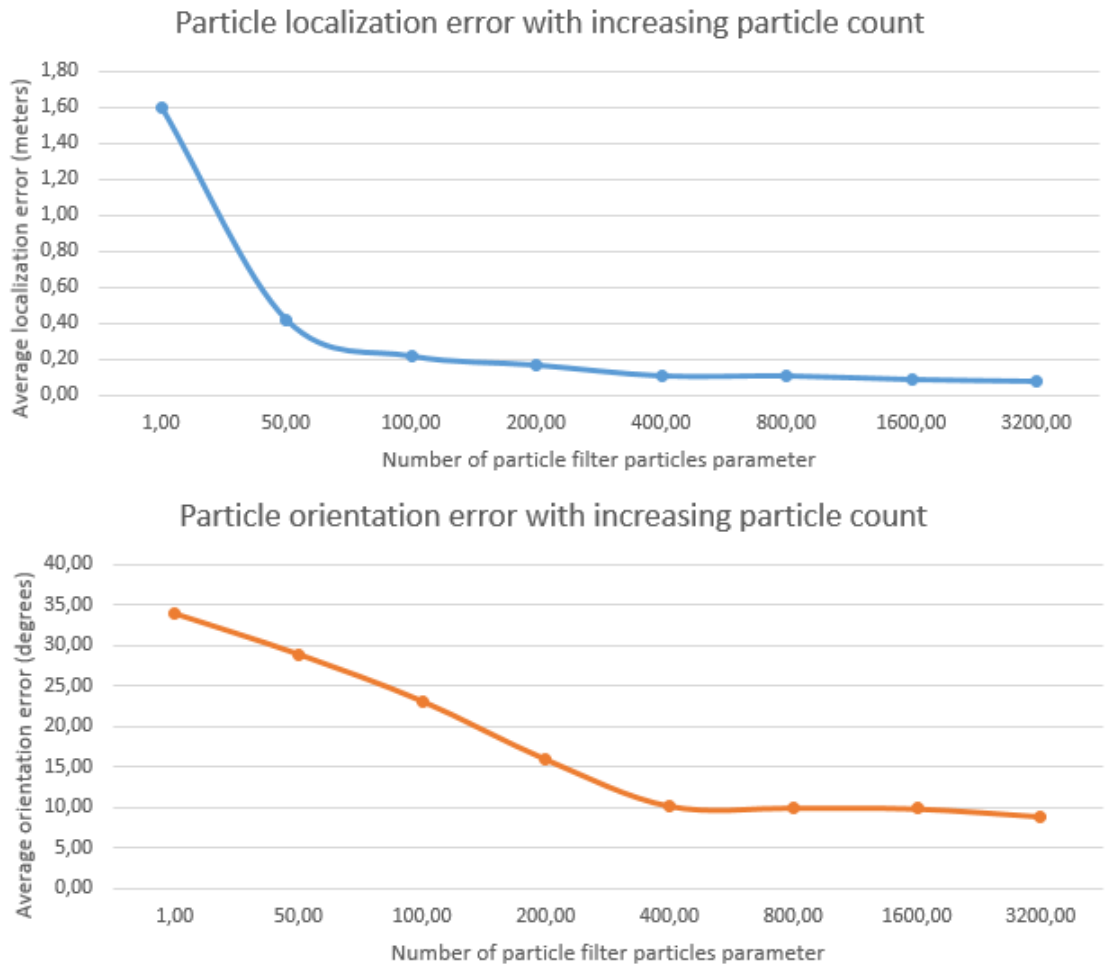


Figure 27. Observing a single goal for prolonged period of time.

The last parameter of the particle filter is the number of particles to use. Previous experiments were carried out using a set of 1000 particles. Graph 9 shows that current application needs about 500 particles to perform well with increasing particle number not providing much performance benefit. It was observed that sometimes as few as 50 particles can perform quite well as long as the localizer does not lose track of the robot pose. If it does then the filter utilizing fewer particles does not recover well. Using many particles on the other hand made the filter more accurate and stable as expected but starting from about 1000 particles, additional ones had very little effect and do not justify the performance penalty so a value of 1000 was chosen as best compromise between accuracy and speed.



Graph 9. Particle filter particle count effect.

3.5 Performance comparison

We have gone over the parameters of all of the implemented localization algorithms and tuned them to produce the best results. While this method was not very precise as most of the parameters tend to affect each other and in addition to average pose error, the produced estimate stability and variance had to be also taken into account, it was found adequate for comparing the algorithms.

The experiment averages the performance of each localization algorithm over ten runs. This provides a good average performance of each algorithm as well as the variance from run to run. The results can be seen in Graph 10 which show the average

position and orientation errors of the implemented algorithms including error bars showing 95% confidence level.

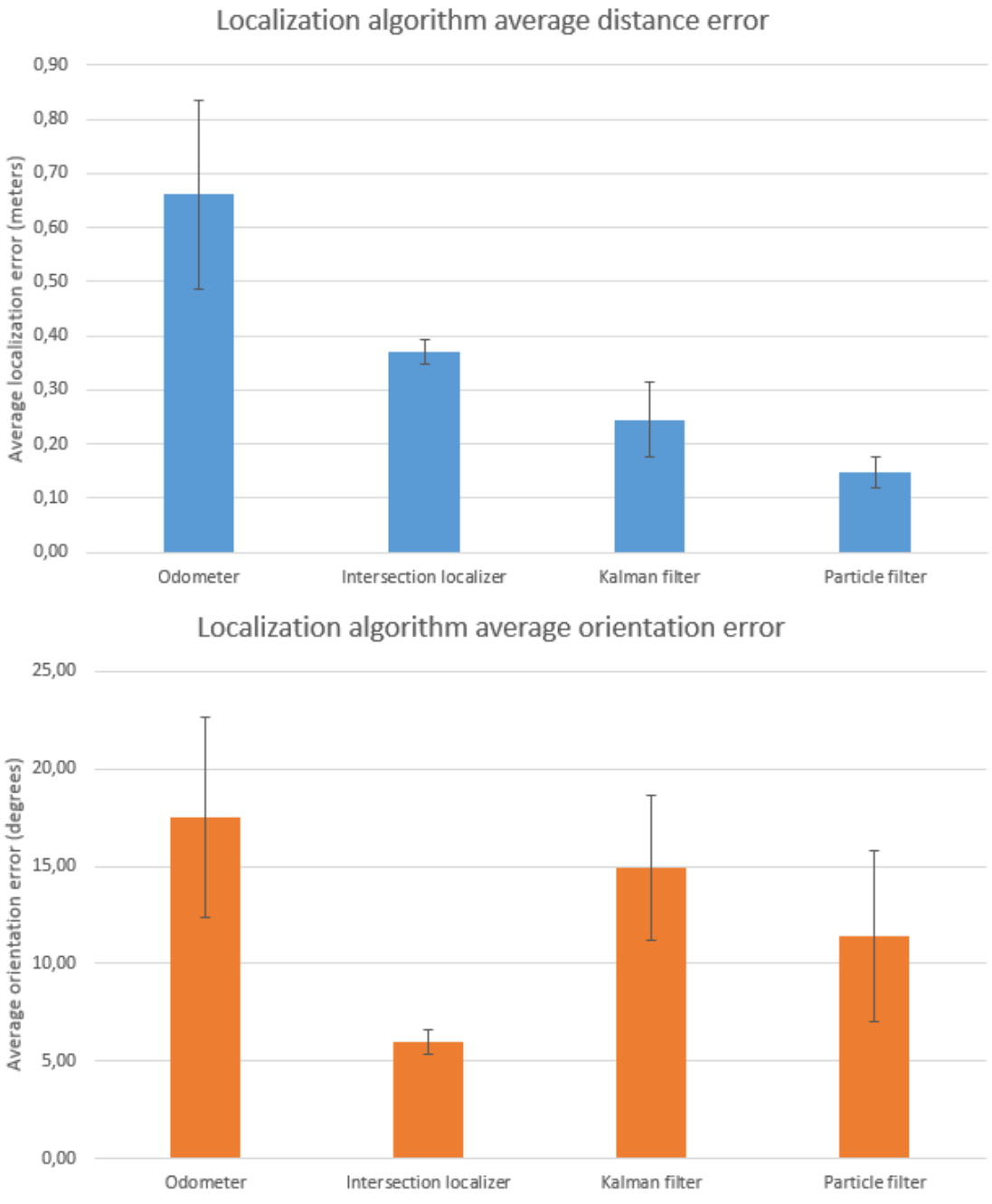
As expected, the pure-odometer solution has the worst performance with average location error over half a meter, rendering this approach unusable for any practical use. The absolute error values have little meaning as this is the result of configurable noise injected into the system, the important aspect is that approximately this level of inaccuracy was observed for the physical robot implementation, making it realistic input to other localization algorithms.

The intersection localizer performance is stable but poor as far as location error is concerned but has the smallest average orientation error. What the shown numbers fail to convey is the large amount of noise in the pose estimate of this method from one frame to another. As the location estimate is very unstable, it would be a poor input to any control algorithms. While the general trigonometry behind this approach is not complex, care must be taken to solve the symmetry ambiguity as well as when averaging angle values. There are also special cases of seeing a single or no goals which need to be handled for optimal performance.

Kalman filter performance is stable and the pose estimate is sufficiently accurate to rely on for building control algorithms on top of. Kalman filter could be implemented in several ways for given problem, but working on lower level of abstraction would likely make the problem non-linear requiring more complex extended Kalman filter or other non-linear variants. As a personal note, the Kalman filter algorithm was found to be the hardest to understand as well as implement. One generally needs third party libraries for working with high-dimensional matrices that support among other operations matrix inversion. This made porting the code from JavaScript simulator to C++ robot more complex.

Particle filter achieved the best performance in estimating the pose of the robot. It provides reasonably stable results but in its current implementation, it is subject to pose estimate error due to the symmetry of the field and distance to goals as shown by Figure 28. This problem could be alleviated by for example injecting a small number of particles at the symmetrical location of the estimate that should be more likely to get resampled as their observed goal angles would match the reality better. The author of this thesis found the particle filter approach the most elegant, easiest to understand and implement without requiring more complex math and this approach is not limited to

linear systems. The algorithm also provides a configurable tradeoff between pose estimate accuracy and required computational resources.



Graph 10. Localization algorithms performance comparison.

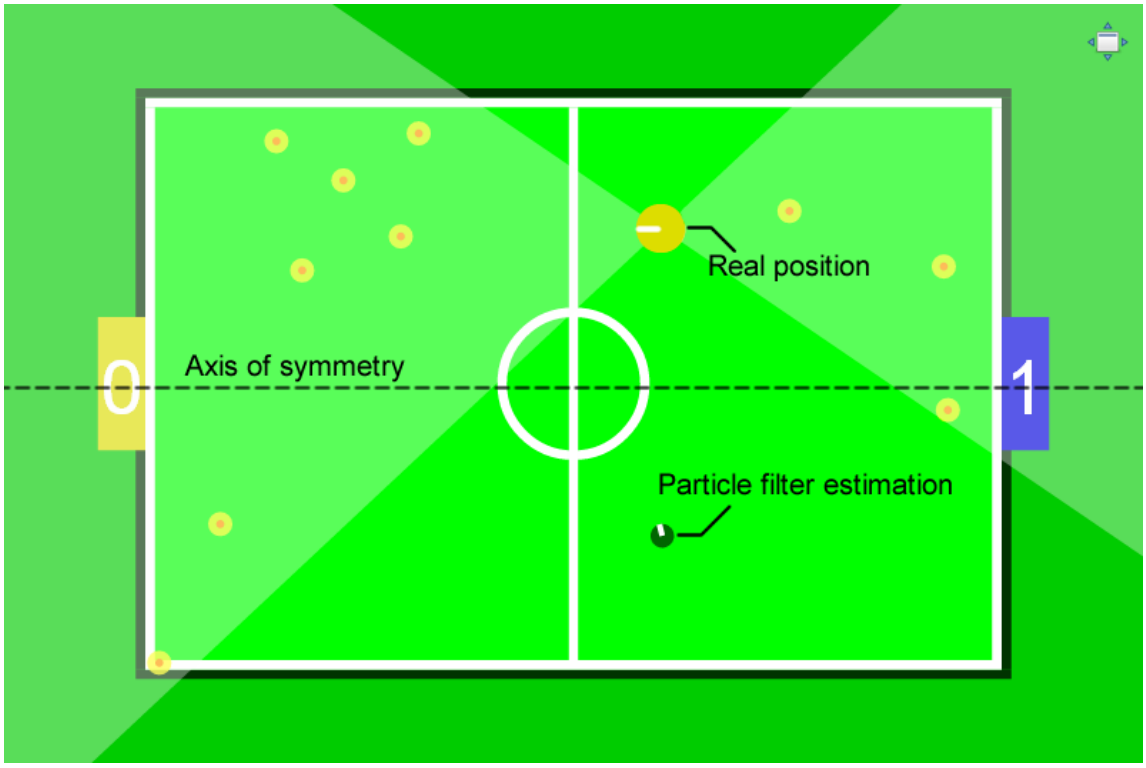


Figure 28. Particle filter failing due to symmetry of the field.

4 Physical robot experiments

4.1 Implementation

This short section describes the algorithm implementation process for the robot and the author's subjective experience. All four localization algorithms described in the thesis and implemented in the simulator were also ported to C++ and tuned to work on the real robot. Porting the Kalman filter proved to be the most difficult as it requires working with n-dimensional matrices. The Armadillo [14] library was chosen for the task. Other simpler libraries considered did not support matrix inversion so one must be careful choosing appropriate library. Porting the other algorithms was more straightforward and they worked practically in the first tests.

A few hard to find bugs were encountered where an algorithm would fail after running for some time or under certain circumstances that were not observed in the simulator. The first one was due to trying to take *asin* for values not confined to the -1..1 range. The second was taking square root of a subtraction which, under noisy data circumstances, could be negative. Both of these were difficult to find as debugging C++ code running in a separate computer on the robot is cumbersome. Considering how much debugging developing the initial algorithms required, stepping through the code and following the progress on the screen, implementing these algorithms directly on the robot would have proved difficult and time-consuming, showing the power of prototyping in the simulator.

4.2 Test setup

Assessing the performance of the localization algorithms on the real robot and soccer field is not as straightforward as with the simulator as there is no simple way to know the actual position of the robot at any given time. As shown in Figure 29, four markers were placed on the field at known locations as reference points.

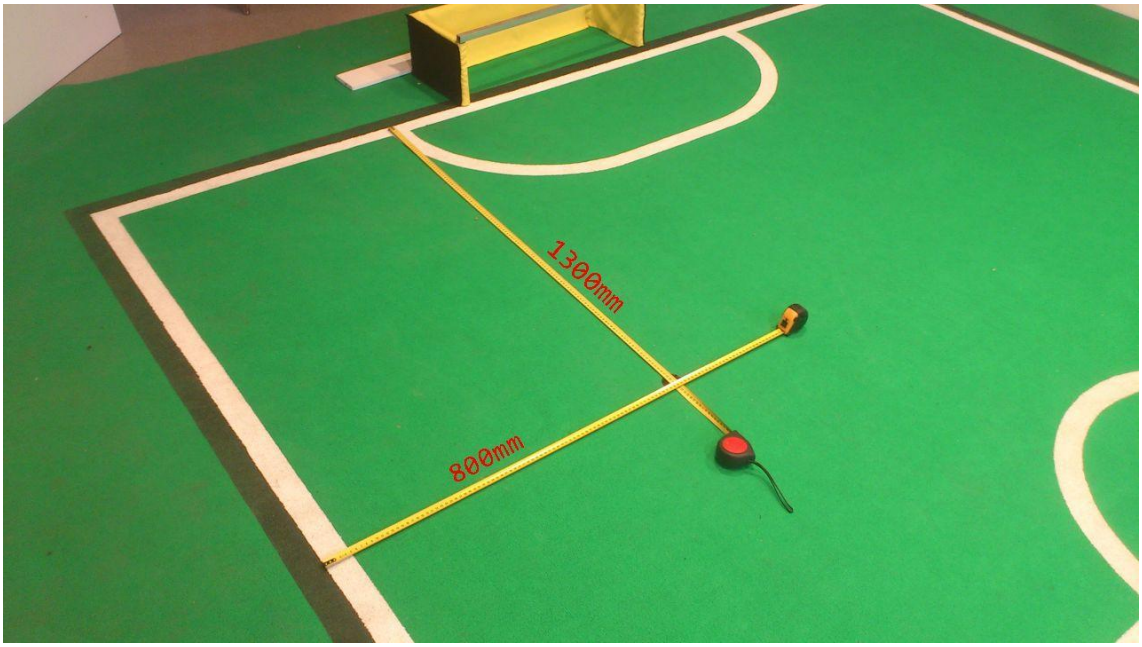


Figure 29. Markers at known positions were placed on the field as reference points.

The robot's computer is connected to the internet over wireless network and the control program sets up a *WebSocket* [15] server that allows for real-time full-duplex communication between the robot and a web-application running in a web-browser. This communication link provides telemetry information about the robot's view of the environment including where each of the localization algorithms believes the robot is currently located. The debugging interface also renders the four markers placed on the field in the same locations as red circles, enabling comparing the predicted and real positions. Figure 30 shows how all four algorithms predicted path of the robot while driving a rectangular trajectory between the four markers. Note that in these tests, the robot is piloted manually using remote control so some of the noise can be attributed to imprecise driving.

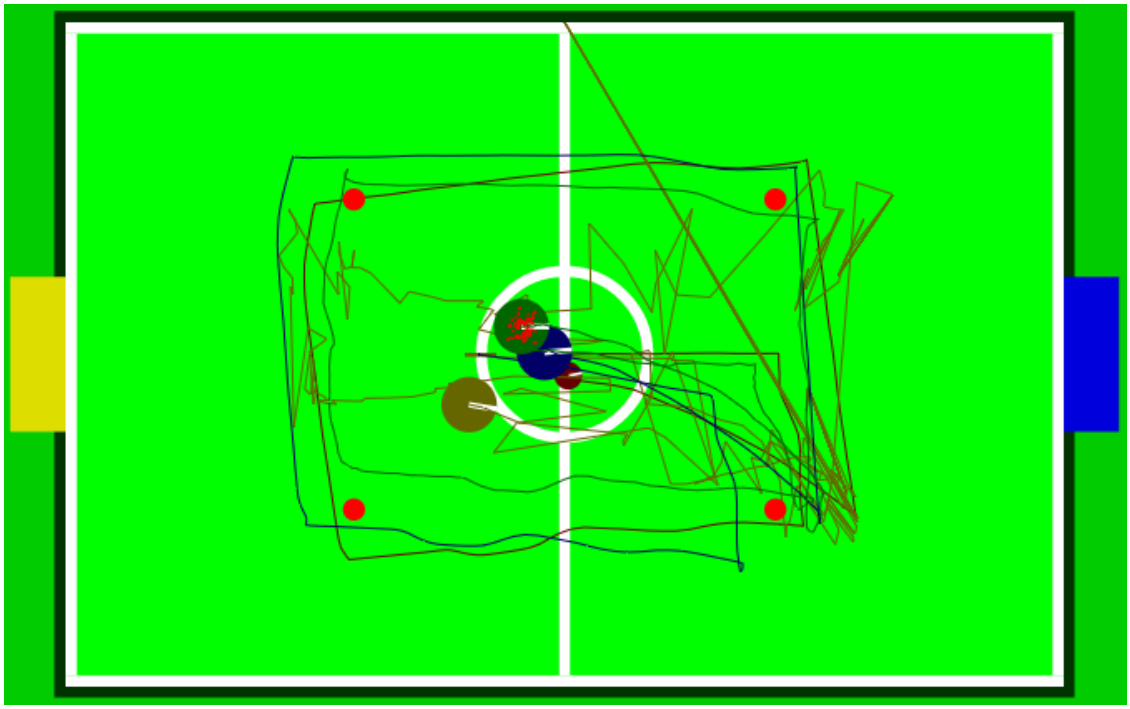


Figure 30. Visualization of all four algorithms while driving a rectangle between the points. Green is particle filter, blue Kalman filter, brown intersection localizer and small dark red is using only odometry.

4.3 Tuning

The algorithms were first tuned to work best on the robot. The same configuration parameters that were used in the simulator actually worked reasonably well on the physical robot, indicating that the simulation is sufficiently accurate for deducing some conclusions. As was actually guessed while developing the simulator, the noise in the odometry system was somewhat poorly modelled as just applying Gaussian noise to wheel speeds is not accurate. The odometer is actually quite precise under low acceleration conditions with little dependency on wheel speed, but slipping of wheels can happen when the robot performs rapid maneuvers. The odometer is also more precise when either only moving straight or rotating on the spot, combining translational and rotational motion causes larger error. Thus a better model for odometry noise would depend on the acceleration of each wheel and the amount of rotational motion. For Kalman filter, the process noise was reduced and for particle filter, the generated forward and turn noise were also reduced. This made the algorithms rely more on odometry input providing less noisy pose predictions.

4.4 Results

To better understand the actual path of the predicted location estimate rather than just comparing poses at fixed locations, the browser telemetry tool renders the path between every predicted pose. Figure 31 shows the predicted path of the robot given by the intersections-based localizer while the robot was driven in a rectangular path between the markers on the field, also rendered as red circles in the tool. As can be seen, the path is very noisy, implying that the observed distances to the two goals vary considerably from measurement to measurement.

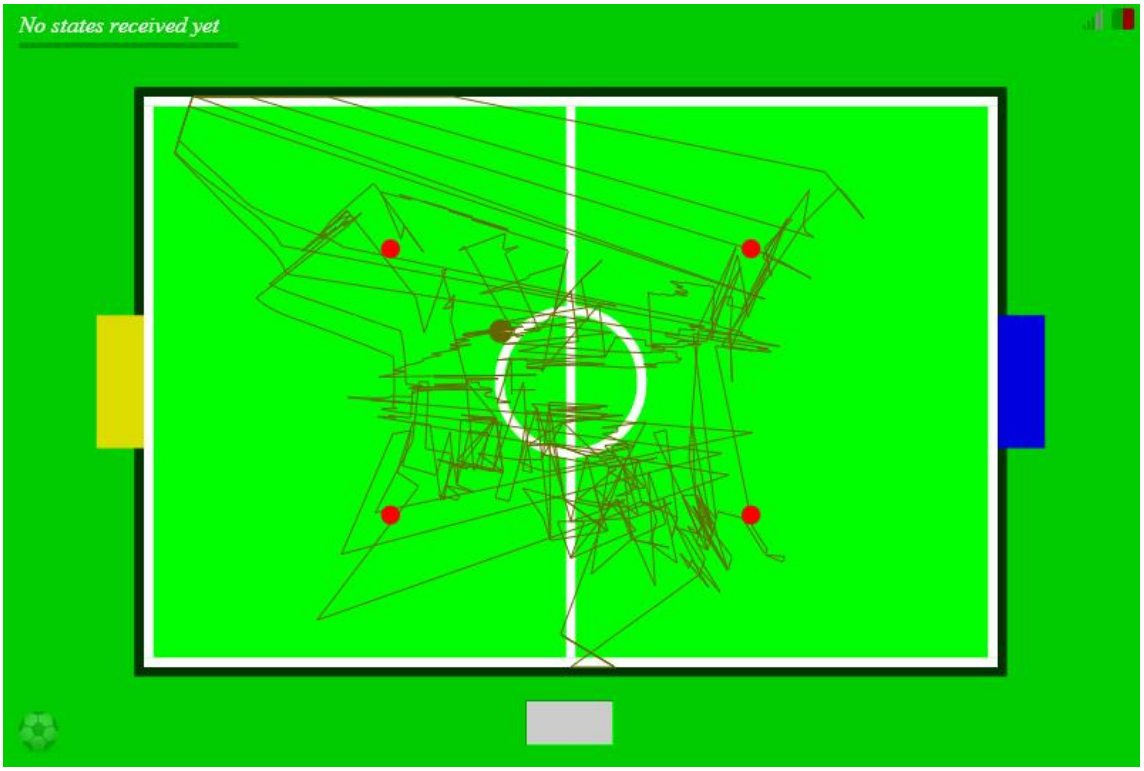


Figure 31. Noisy location estimate of intersection-localizer.

For goal distances, Gaussian noise seems to be quite accurate model although there is often some offset from the real distance. In the simulator, the observed distances fluctuated around the real distance but this is often not the case in the real world as sometimes the observed distance is on average more or less than real distance, depending on robot position, motion, whether the carpet is slightly uneven or whether any of the wheels are on slightly higher white lines of the field. When the robot is situated on the horizontal axis of symmetry (the line between the two goals), this

offset often causes the circles inferred from goal distances to either not intersect at all or intersect more than a bit as shown in Figure 32. This makes the Kalman filter perform not so good near this centerline, particle filter was observed to be less affected.

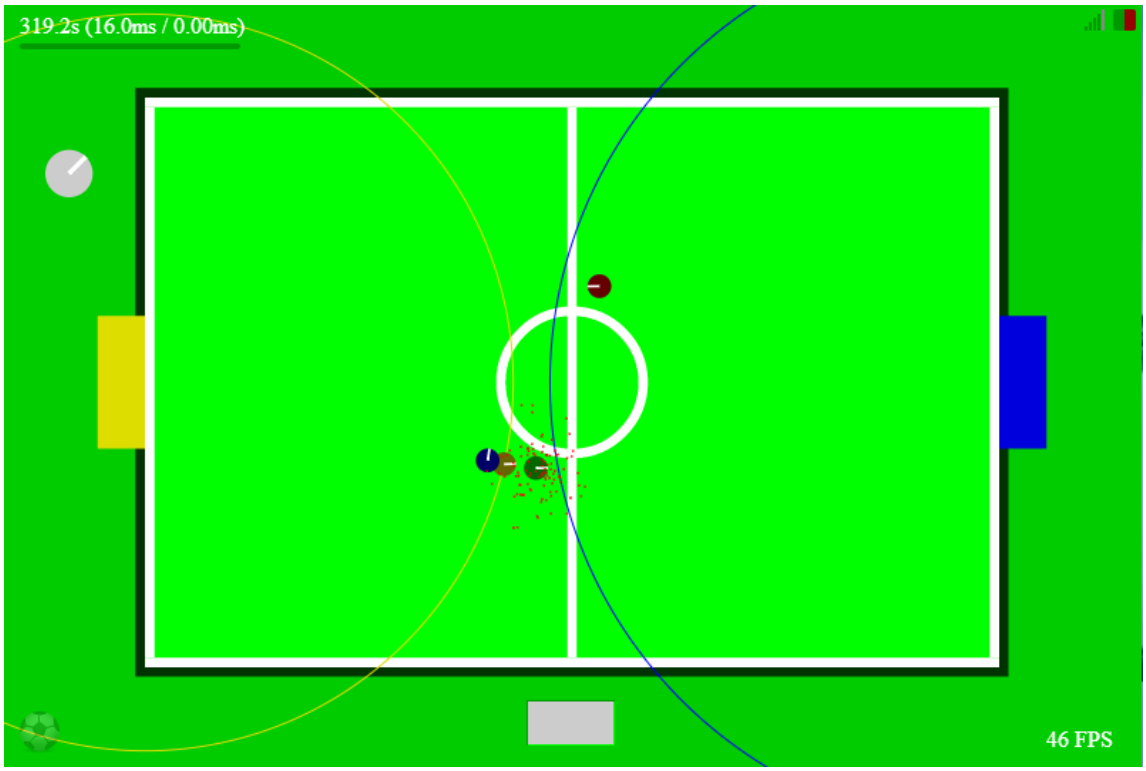
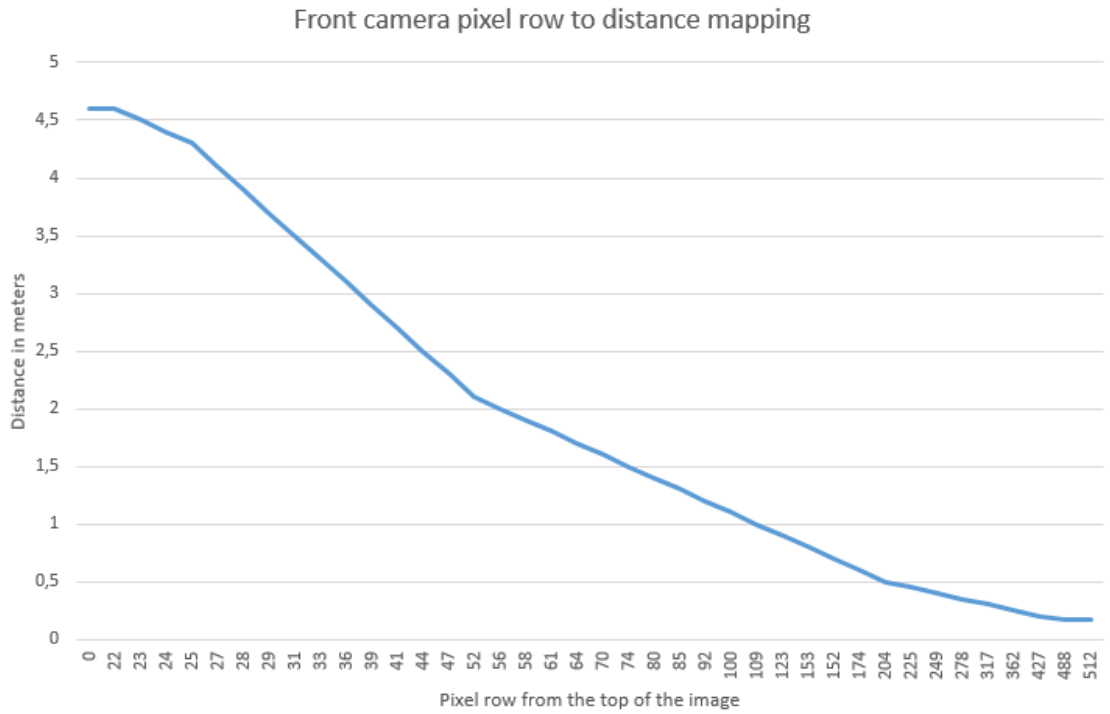


Figure 32. Circles not intersecting, robot's real position is in the center of the field.

The reason why object distance observation becomes less accurate with distance is that the cameras of the robot are located not far up from the ground so the further away the object, the less screen pixels correspond to each unit of distance as shown in Graph 12. For example between 3 and 3.5 meters there is only about 8 pixels difference and the vibrating robot in motion can easily generate that amount of tilt in the camera image.



Graph 12. Mapping between screen pixel row and object distance.

The particle filter proved to produce the best results that are sufficiently accurate, low noise and high stability. For stability conformance, the algorithm was tested while driving around the field for extended period of time (around 10 minutes) continuously performing various maneuvers and rapid movements. Particle filter did not lose track of the location of the robot and was even able to recover the right position after some time when initialized in the wrong position. Figure 33 shows four attempts of driving the robot in a rectangular path between the four markers, starting and ending in the top-left position. Note that the robot was driven manually so some of the noise is due to corrections in the driving. As can be seen, the error of the robot position rarely exceeded about 10 centimeters which can be considered sufficient performance for using as valuable input to control algorithms. Considering the large noise in the goal distance measurement, the algorithm actually performs better than was anticipated.

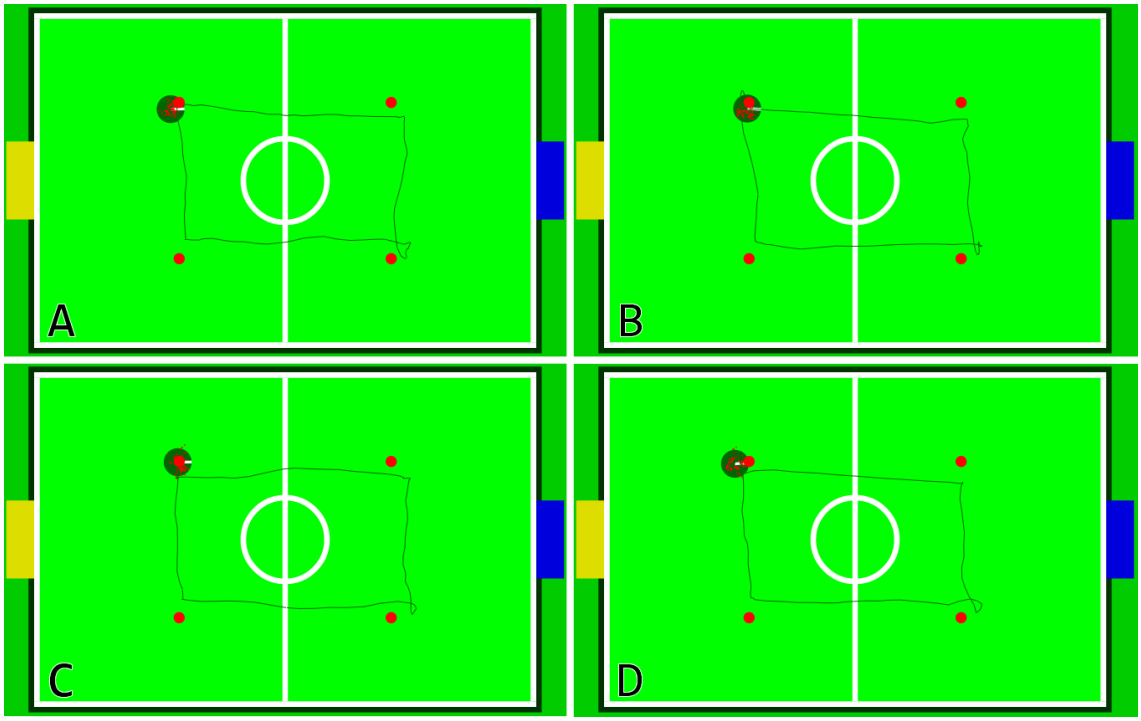


Figure 33. Four runs of particle filter driving rectangular shape between the markers.

As an indication of how particle filter corrects the error introduced into the odometry data, Figure 34 compares the paths of pure-odometer (small dark red circle) and particle filter (big green circle) driving the rectangle in a violent manner, inducing some wheel-slip on purpose. While this makes the particle filter estimation also less accurate, it is able to recover the true pose in the end near the top-left marker while the odometry pose has drifted considerably. Under more calm driving techniques, the odometer does not drift nearly as much.

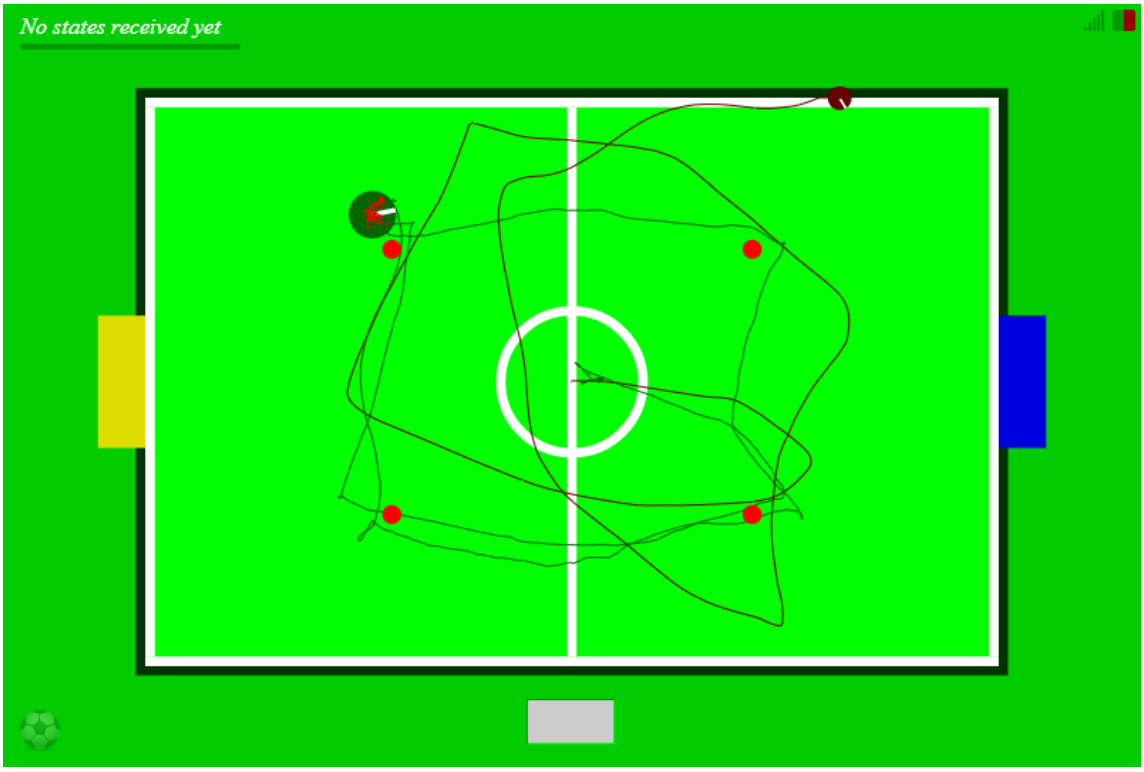


Figure 34. Comparison between particle filter and odometer under rapid maneuvering.

The Kalman filter implementation did not work as good as in the simulator which is due to the rather large noise in goal distance sensing. Its performance suffered especially when executing violent maneuvers such as shown in Figure 35, driving rectangular shape with large accelerations (blue is Kalman, green particle filter). During some tests, the filter lost track of the real pose of the robot and took considerable time to recover (about 20 seconds). For given implementations, the particle filter showed to perform the best.

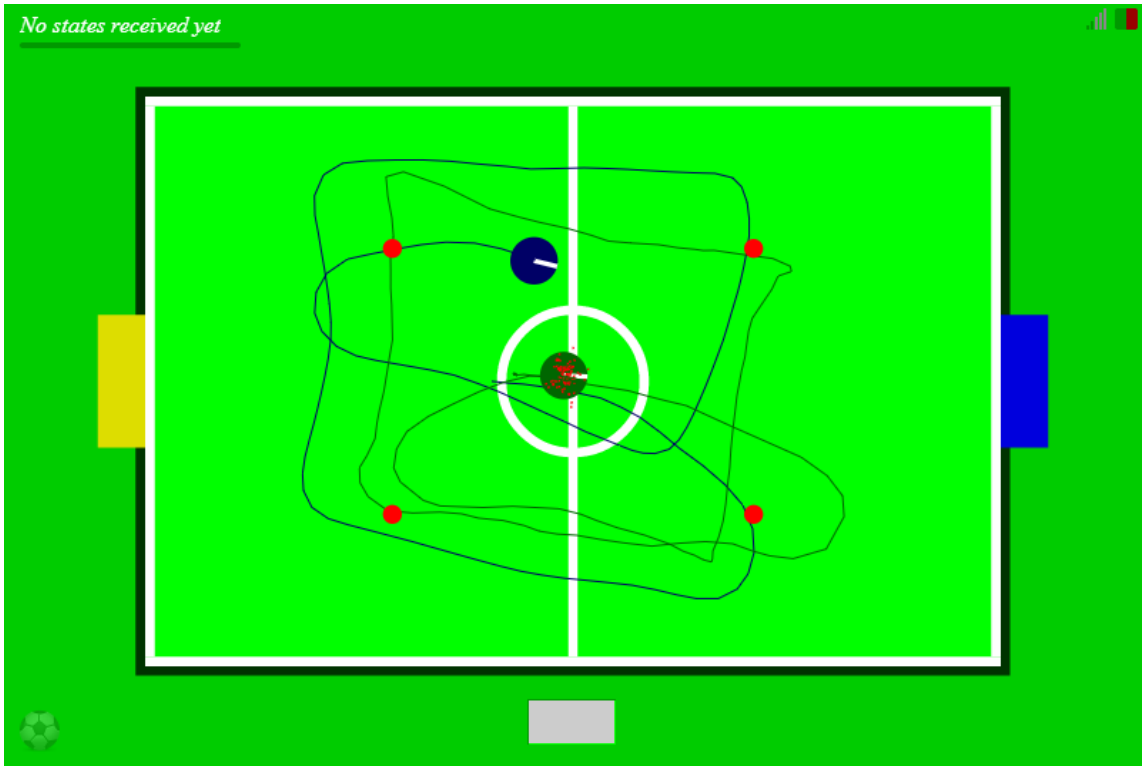


Figure 35. Comparison of particle and kalman filter.

4.5 Future work

While implementing and testing the algorithms on the physical robot, several observations were made that gave ideas for improvements not yet implemented.

The first idea has to do with goal distance and angle calculation. Since this is the main source of information for odometer error compensation, retrieving correct values is critical. The first problem with this approach was discussed before and has to do with the positioning of the cameras, making objects further away having less resolution in the camera image as shown in Graph 12. Ways to alleviate this problem would be to either mount the cameras up higher or increase the resolution of processed images, both of which are planned for next year's competition. Another issue with goal distances is that they are currently calculated from the bottom of the detected goal axis-aligned rectangle, rather than the actual center-point, which introduces some error when looking at a goal at an angle as shown in Figure 36. The error introduced by this is not great but could be fixed by finding the actual bottom center point of the goal.

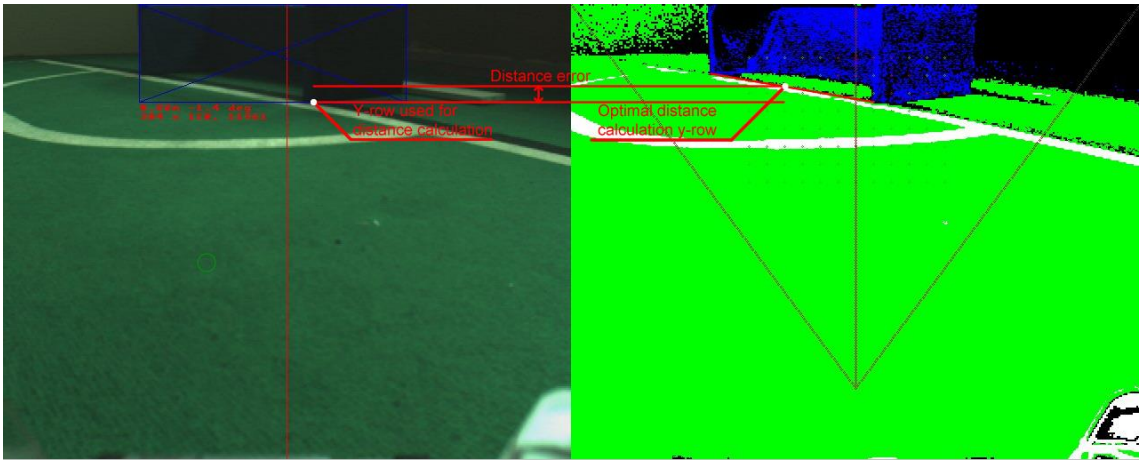


Figure 36. Goal distance calculation error at large angle.

Another issue with distance calculation is the fact that the camera image from the the wide angle lens is distorted. Figure 37 shows how the robot sees the centerline of the field, which is actually perfectly straight but appears to be elliptic. Combined with low pixel resolution of distant objects, an object in the center of the view would appear to be further away than those at the edges of the image with considerable error at distance. This could be one of the reasons that the particle filter was biased towards the centerline of the field in Figure 33.

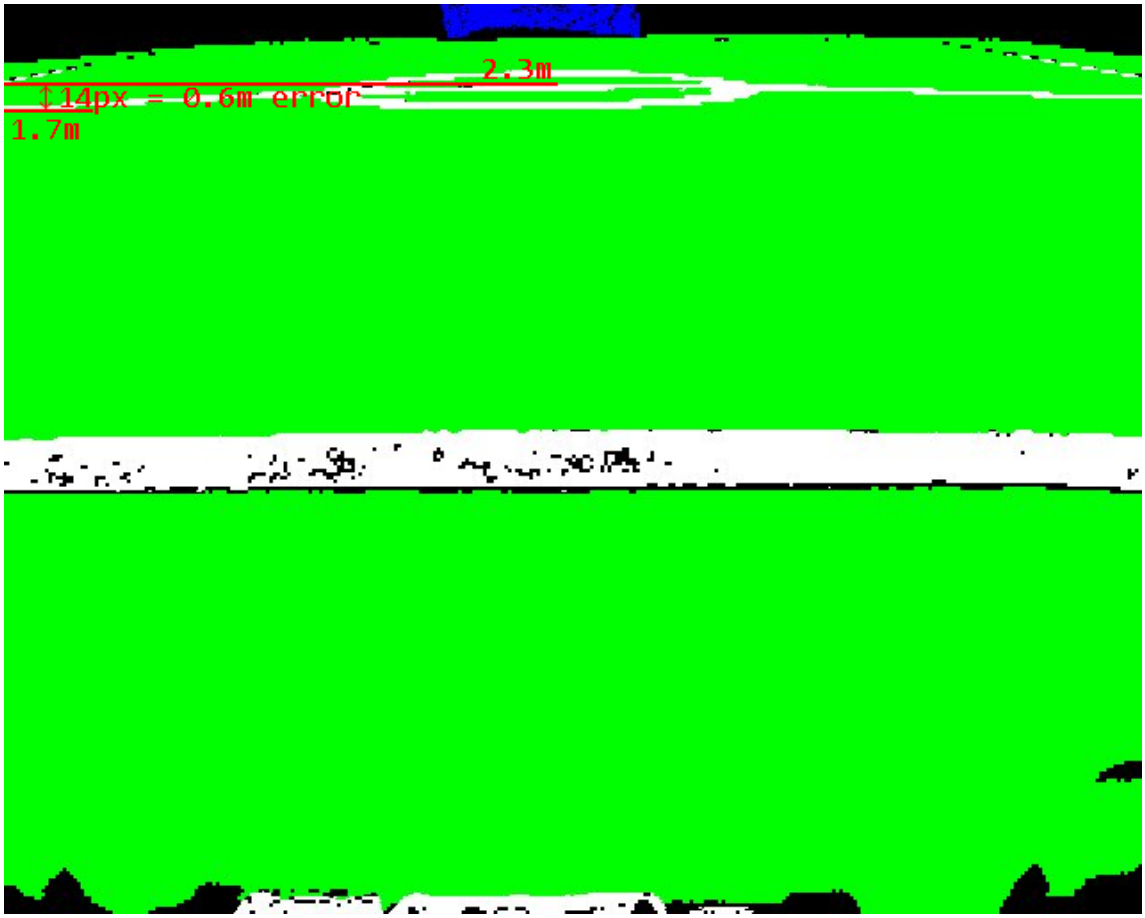


Figure 37. Object distance calculation error due to warped camera image.

One of the main limitations of currently implemented approach is the small number of landmarks as only goals are used for localization. This problem is made worse by the fact that in certain orientations, the robot may see only one or no goals and it is often quite far away from at least one of them, making the distance measurements imprecise. This situation could be improved by extracting additional landmarks from the camera images such as those shown in Figure 38. Panel A shows the center circle with center-line through it. This could be used for both distance calculation, as it has a known position, as well as orientation compensation as the ellipse with a line through it appears different from various angles. Panel B shows a corner of the field. While this is not uniquely detectable (the field has four identical corners), it could still be used as input to the localization algorithms. The same applies for images C, D which show the T-shaped intersection in the centers of the field and line in front of the goals, respectively. The problem with using these landmarks is that they are not trivial to extract from the camera image and require advanced image

processing techniques, but there are still plans to implement at least some of these for the next competition.

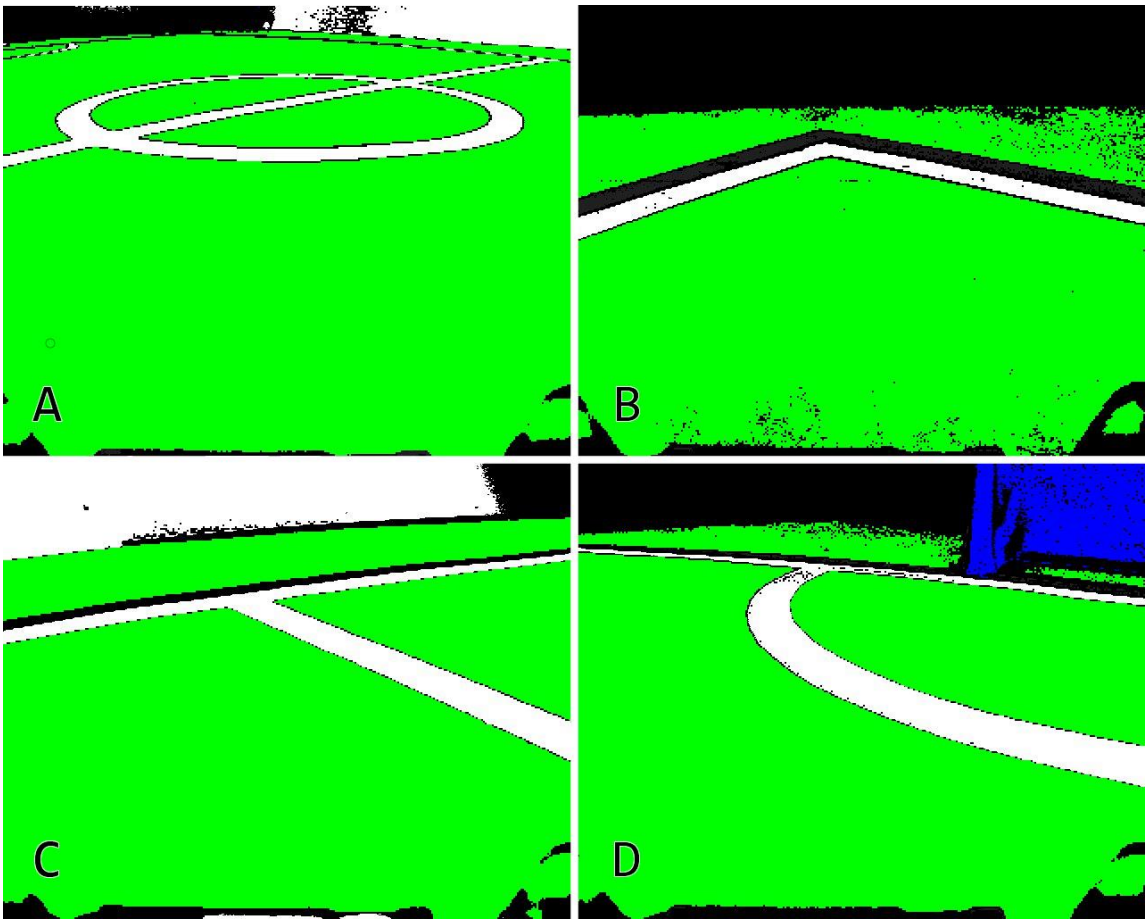


Figure 38. Additional possible landmarks that could be extracted from computer vision.

Both Kalman and particle filter provide configuration parameters that allow tuning the balance between how much to rely on the odometry data versus measurement data. Currently these parameters are statically defined, but expected measurement noise could be a function of landmark distance as error is increased for objects further away. This would mean that the algorithms could rely more on the measurements of nearby object while relying more on the odometer when landmarks are not nearby. Similarly, the odometer is more reliable at lower acceleration rates. The robot control logic could actually limit maximum acceleration of the wheels making the maneuvers smoother and thus less susceptible to slipping.

Summary

Robot localization techniques attempt to solve the problem of estimating mobile robot pose relative to an external frame of reference. Autonomous robots navigating in dynamic natural environments require localization techniques that can explicitly account for the inherent uncertainty in such systems, which cannot be modeled precisely. Often the pose of the robot cannot be directly observed, but instead needs to be inferred from various indirect sensor measurements. Noisy data from several sources of information such as camera images, distance sensors and actuator feedback are fused together to build a probability distribution over all possible states, representing where the robot is believed to be located with some degree of confidence. The state estimation is performed recursively, improving the belief incrementally over several steps of movement and sensory data integration.

The goal of this thesis was to explore and compare various techniques for real-time autonomous robot positioning and to develop a working localization solution for a soccer-playing robot. The first part gave a theoretical overview of the statistics background and Bayes filtering. Two main algorithms were covered in detail – the Kalman filter and the particle filter, both solving the same problem in a slightly different manner. Then the algorithms were applied for the special case of an autonomous soccer-playing robot “Telliskivi II” according to Robotex 2012 professional football league rules. A simulator was made for developing, testing and evaluating the algorithms, which proved to be very useful, because implementing and testing complex algorithms directly on the physical robot would have proven inefficient and time-consuming, as debugging on a mobile robot platform is difficult. The effects of the various algorithm parameters were investigated and tuned to perform best in the simulator. The performance and properties of the algorithms were compared. The algorithms were then ported over to work on the physical Telliskivi II robot and once again tuned and compared.

Pure odometry-based solution proved to be inefficient at long term localization. Noise in the system and unpredictable events such as wheel slip quickly made the estimate drift from the true pose. The intersection-localizer used the two back-to-back cameras on the robot to calculate approximate absolute position on the field using distances to the two goals and interpolating between states referencing odometry information while a single or no goals are visible. As the goal distances measurements are not accurate and vary considerably when the robot is in motion, this solution proved

to be too noisy for using as input to any control algorithms. A Kalman filter solution was developed that uses the noisy estimate from intersection-localizer and fuses this with odometry data to provide a much smoother pose estimate. Another method using the particle filter approach was developed that uses the distances and angles to the landmarks, combined with odometry data. The author found the particle filter to be the most elegant, easiest to understand, implement and capable of dealing with non-linear problems.

The performance of the Kalman filter and particle filter based implementations were quite similar in the simulator with particle filter performing marginally better. On the real robot, particle filter proved to provide superior results with the position error rarely exceeding 0.1 meters from the true pose and the algorithm not losing track of the real location even under violent driving. It was also capable of re-acquiring the correct pose when the robot was kidnapped – i.e. moved to a different location manually. Thus the main goal of the thesis of developing a working localization solution for the soccer-playing robot was achieved.

There are still several ideas that the author plans to investigate to improve the localization performance. It is possible to develop better algorithms and hardware for more accurate distance and angle sensing. In addition, extracting additional landmarks on the field (such as the center circle, corners and intersections) using computer vision, and relying on them in the algorithms, should further increase localization quality.

The Telliskivi II robot came second in 2012 Robotex professional football league competition. Using the localization technique developed in this thesis, the team hopes to build a smarter robot with ambition to win the 2013 competition.

Resümee (eesti keeles)

Roboti lokalisatsioon püüab lahendada probleemi, kuidas määrata mobiilse roboti asukohta keskkonnas välise taustsüsteemi suhtes. Autonoomsed robotid, mis navigeerivad naturaalses ja muutuvas keskkonnas, vajavad lokalisatsioonialgoritme, mis arvestaksid säärastele süsteemidele omase müra ja ebakindlusega. Tihtipeale pole roboti asukoht otseselt vaadeldav, vaid seda tuleb järeldada mitmete sensorite kaudsetest mõõtmistest. Edukad algoritmid kombineerivad andmeid mitmetest sensoritest nagu kaamerad, kaugusandurid ja mootorite tagasiside, milledele kõigile on omane teatud ebatäpsus ja müra. Nende andmete põhjal loovad statistika-põhised lokalisatsioonialgoritmid tõenäosusjaotuse, mis näitab, kus robot arvab end mingi kindlusega asuvat. Positsioneerimine toimub rekursiivselt, integreerides olemasolevale parimale pakkumisele juurde uusi mõõtmisi, nõnda ajas täpsust parandades. Roboti liikumine toob kaasa ebakindluse suurenemist, mõõtmised aga vähendavad seda.

Teesi eesmärk oli uurida ja võrrelda erinevaid lokalisatsioonialgoritme reaalajas autonoomse roboti positsioneerimiseks ning luua toimiv lahendus jalgpalliroboti positsioneerimiseks. Selle esimeses osas anti ülevaade valdkonna statistilisest taustast ja tutvustati *Bayesi filtreerimist*. Põhjalikumalt keskenduti kahele algoritmile – *Kalmani filtrile* ja *osakeste filtrile* (ingl.k *particle filter*). Seejärel uuriti nende algoritmide kasutamist jalgpalliroboti „Telliskivi II“ positsioneerimiseks Robotex 2012 profijalgpalli liiga reeglite kohaselt. Arendati välja simulaator, mille peal sai lokalisatsiooni-algoritme arendada, testida ja võrrelda. Simulaatori loomine õigustas ennast, sest keerukate algoritmide arendamine otse päris roboti peal olnuks keeruline ja aeganõudev, sest koodi testimine ja silumine mobiilse arvuti peal on keerukas. Teesis uuriti algoritmide parameetrite mõju ja häälestati need pakkuma parimat tulemust. Seejärel võrreldi nelja implementeeritud lokalisatsiooni-algoritmide suutlikkust. Samad algoritmid said üle viidud ka pärisroboti peale.

Odomeetria-põhine algoritm kasutab roboti rataste kiiruste andurite tagasisidet, mille järgi arvutatakse roboti dünaamikat arvestades välja selle liikumistrajektoor. Testid kinnitasid hüpoteesi, et selline lähenemine pole pikaajaliseks lokalisatsiooniks sobiv, kuna ebatäpsus ja müra süsteemis kombineeritud ettenägematute sündmustega nagu rataste libisemine, toovad kiiresti kaasa positsiooni hinnangu kõrvalekalde tegelikkusest. Siiski on odomeetria-andmed kasulik sisend teistele algoritmidele. Telliskivi II robotil on kaks teineteisega seljakuti asuvat kaamerat. Teine implementeeritud algoritm kasutas neid hindamaks kaugusi kahe jalgpalliväljaku

väravani, millest sai tuletada roboti asukoha platsil. Nähes vaid ühte või ei ühtegi väravat, tugineti odomeetria-andmetele. Kuna aga väravate kauguste hindamine liikudes vibreeriva roboti peal on ebatäpne, siis selle lähenemise positsiooni-hinnang tõestas end olevat kontrollalgoritmides kasutamiseks liialt mürane. Selle müra silumiseks sai loodud Kalmani filtri põhine lahendus, mis kasutab sisendina eelmist väravate kauguste põhist mürast hinnangut ja odomeetria-andmeid. Nõnda parandatakse odomeetria kõrvalekallet mõõtmiste abil. Viimane neljas meetod põhines osakeste filtril, mis kasutas sisendina samuti väravate kaugusi ja nurkasid nendeni ning ühendas selle info odomeetria-andmetega. Kui Kalmani filter esitas tõenäosusjaotusi normaaljaotuste abil, siis osakeste filtris moodustavad jaotuse suur arv osakesi, millest igaüks esitab üht hüpoteesi roboti asukohast. Neid liigutatakse platsil odomeetria andmete põhjal ning igal sammul jäävad suurema tõenäosusega alles osakesed, mis sobivad paremini mõõtmistulemustega. Autor leidis, et just osakeste filtri lahendus on kõige elegantsem ning lihtsaim mõista ja implementeerida.

Simulaatoris andsid Kalmani- ja osakeste filter sarnaseid tulemusi, viimane toimis natukene paremini. Mõlemad toimisid edukalt ka päris roboti peal, aga osakeste filtri suutlikkus oli suurem ning asukoha viga ei ületanud enamasti 0.1 meetrit. Samuti oli algoritm suuteline õiget asukohta pikka aega järgmina isegi agressiivse, rataste libisemist põhjustava sõidu ajal. Algoritm oli enamasti võimeline ka roboti asukohta mõistliku aja jooksul taas-leidma kui viimane käsitsi algoritmi teadmata uude kohta tõsta. Seega sai täidetud üks teesi põhieesmärke luua toimiv lokalisatsioonilahendus jalgpallirobotile.

Autoril on veel mitmeid ideid kuidas loodud lahendust veelgi paremaks muuta. On võimalik parendada algoritme ja riistvata, et hinnata huvipakkuvate objektide kaugusi ja nurke täpsemini. Lisaks on võimalik suurendada algoritmide täpsust tuvastades platsilt täiendavaid staatilisi objekte nagu platsi jooned, nurgad ja keskmine ring.

Telliskivi II robot tuli 2012 aasta Robotexi võistlusel teiseks ja kasutades antud teesis esitatud ideid ja lahendusi, on meeskonnal plaan luua veelgi targem ja võimekam robot ning esineda võidukalt 2013 aasta üritusel.

References

[1] Leonard, John J., and Hugh F. Durrant-Whyte. "Mobile robot localization by tracking geometric beacons." Robotics and Automation, IEEE Transactions on 7.3 (1991): 376-382.

[2] King-Hele, Desmond. "Erasmus Darwin's improved design for steering carriages—and cars." Notes and Records of the Royal Society of London 56.1 (2002): 41-62.

[3] Dudek, Gregory, and Michael Jenkin. Computational principles of mobile robotics. Cambridge university press, 2010.

[4] Drocourt, Cyril, et al. "Mobile robot localization based on an omnidirectional stereoscopic vision perception system." Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on. Vol. 2. IEEE, 1999.

[5] Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. Probabilistic robotics. Vol. 1. Cambridge: MIT press, 2005.

[6] Lee, Peter M. Bayesian statistics: an introduction. Wiley, 2012.

[7] Azad Kalid. „An Intuitive (and Short) Explanation of Bayes' Theorem". *BetterExplained*. 6. May 2007. Web. 25 February 2013.

[8] Coppersmith, Don, and Shmuel Winograd. "Matrix multiplication via arithmetic progressions." Journal of symbolic computation 9.3 (1990): 251-280.

[9] Gan, Qiang, and Chris J. Harris. "Comparison of two measurement fusion methods for Kalman-filter-based multisensor data fusion." Aerospace and Electronic Systems, IEEE Transactions on 37.1 (2001): 273-279.

[10] Arras, Kai O., Jose A. Castellanos, and Roland Siegwart. "Feature-based multi-hypothesis localization and tracking for mobile robots using geometric constraints." Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on. Vol. 2. IEEE, 2002.

[11] Julier, Simon J., and Jeffrey K. Uhlmann. "Unscented filtering and nonlinear estimation." Proceedings of the IEEE 92.3 (2004): 401-422.

[12] Neyman, Jerzy. "On the two different aspects of the representative method: the method of stratified sampling and the method of purposive selection." Journal of the Royal Statistical Society 97.4 (1934): 558-625.

[13] Kallas Priit. „Robocup Simulator". *Github*. 2013. Web. <https://github.com/kallaspriit/Robocup-Simulator>

[14] Sanderson Conrad. „Armadillo". *Sourceforge*. 2013. Web. <http://arma.sourceforge.net>

[15] Websockets.org. „Websockets". 2013. Web. <http://www.websocket.org>

[16] Kallas Priit. „KalmanJS". *Github*. 2013. Web. <https://github.com/kallaspriit/KalmanJS>

[17] Robotex. Web. 1. February 2013. <http://www.robotex.ee>

[18] Thrun Sebastian. „Artificial Intelligence: How To Build A Robot - Udacity“. Web. 5. February 2013.

[19] MacQueen, James. "Some methods for classification and analysis of multivariate observations." Proceedings of the fifth Berkeley symposium on mathematical statistics and probability. Vol. 1. No. 281-297. 1967.

[20] Guizzo Erico. “How Google’s Self-Driving Car Works”. IEEE Spectrum. 18. October 2013. Web. 06. May 2013.

[21] Rosenblatt, Murray. "Remarks on some nonparametric estimates of a density function." The Annals of Mathematical Statistics (1956): 832-837.

Non-exclusive license to reproduce thesis and make thesis public

I, Priit Kallas (date of birth: 14. January 1988),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

“Probabilistic Localization of a Soccer Robot”, supervised by Konstantin Tretyakov M.Sc.

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **08.05.2013**