Institute for Computer Science
Knowledge-Based Systems

Institute of Cognitive Science

Bachelor's thesis

Probabilistic Robot Localization in Continuous 3D Maps

Sebastian Höffner

November 2014

First supervisor: Prof. Dr. Joachim Hertzberg
Second supervisor: Dr. rer. nat. Thomas Wiemann, M. Sc.

Probabilistic Robot Localization in Continuous 3D Maps

Robot localization in known environments is an important topic, especially in the field of mobile robotics. Whenever a robot is turned on or relocated, it has to (re)locate itself in its environment. Several approaches exist for robots to find their three-dimensional pose in two dimensional raster maps. A popular algorithm for the localization task is the Augmented Monte Carlo Localization (AMCL) which is described in details by Thrun et. al. [TBF05]. A localization is successful when the robot found its current three dimensional pose, that is its two dimensional position and its one dimensional orientation.

Mobile robots have to work not only in two, but also in three dimensional environments. This means they can have six dimensional poses because their position consists of three dimensions and so does their orientation. In this thesis I will derive an algorithm based on AMCL to solve the localization problem in such three dimensional environments and find the six dimensional pose. Therefor I will no longer use raster maps but polygon meshes as continuous maps. For the sensor model I will use a ray tracer, which simulates sensor data comparable to real world distance sensors like laser scanners or depth cameras.

After deriving the algorithm and running some simulations with it, I will take a look out about how the algorithm can be developed further.

Acknowledgements

I would like to thank Prof. Dr. Hertzberg and the people of the working group Knowledge Based Systems for their overall support with my thesis.

Particularly I give thanks to Dr. Thomas Wiemann for his great advice and support. His reviews, critiques and ideas helped me a lot tackling the topic and solving the problems I had to overcome while developing the AMCL6D algorithm.

Additionally many thanks to Lisa Goerke for reading through my thesis and providing lots of valuable feedback and remarks.

Contents

Chapter 1

Introduction

1.1 Kidnapped robot

Let's assume for a second you get kidnapped: Some dark clothed men blindfold you, drag you into their car and drive around in confusing circles before they release you at a completely different place. Luckily you get your hand on a hand radio and when the bad guys leave you to die, you are able to tell your friend (luckily a ham radio operator) where you are, so that she can pick you up: You look around and tell her what you see. The big tree over there, the coffee shop at that corner. Your friend finds several similar locations on her map, so she asks you to go a few more steps and tell her what you see. After telling her about the bakery next to the coffee shop and a small park across the street she is finally able to locate you, jumps into the car and fetches you.

This scenario is quite common for mobile robots. In the annual RoboCup, a scientific competition in which robots play football [Rob08], robots are often relocated by a referee [TBF05]. They then have to realize that they have been relocated and determine their new position to continue efficient play. If there are ambiguous possible positions, they can simply move to get rid of uncertainties, as described in the example above.

A similar situation occurs for every mobile robot which is turned on. If they are supposed to drive around a factory (or a floor in the computer science department) they first have to find out where they are. This is a little bit easier than the first scenario, since a robot doesn't have to find out that it was relocated—it is sufficient to find the current position.

The situation of relocating a robot and thus forcing it to redetermine its position is called *kidnapped robot problem* [TBF05]. It is an extension to the *global localization problem*. The global localization problem just needs to find the initial pose, while the kidnapped robot first has to realize its kidnapping—it might still assume that it is right on its original course while it was carried away.

1.2 Idea and motivation

There are several algorithms which solve localization problems in two dimensional maps locally or globally. A very popular algorithm, which is also capable of solving the kidnapped robot

problem, is Monte Carlo Localization [DFBT99]. One variant, the Augmented Monte Carlo Localization (AMCL), can be found in *Probabilistic Robotics* [TBF05].

For this algorithm, environments are represented as 2D maps. This is useful for many real-life applications, since many robots operate in more or less flat environments. Although it is sufficient for most environments to represent them in a 2D map, environments with altitude differences (e.g. factories with ramps or normal houses with stairs) are hard to represent this way. Also robots which are capable of moving in more than two dimensions, e.g. flying robots, could benefit from more information than just 2D maps. However, there are few approaches to locate robots in 3D environments. Therefore I will derive an algorithm based on the classic AMCL to localize a robot in an arbitrary 3D environment. I will show that the algorithm works up to a certain point but is not stable, thus once it is close to the real pose, it loses track and becomes unstable.

Chapter 2

Robot localization and Bayes filters

2.1 Robot localization

Robot localization is the process of determining a robot's pose in a known environment. A common application for it is to find the starting position for planning algorithms. Thrun et al. distinguish between three different kinds of localization problems: *Local localization*, *global localization*, and the *kidnapped robot problem* [TBF05]. The local localization problem keeps track of the robot's pose from a known initial pose and is considered the easiest problem of the three, since the error rate in position tracking is small and efficient algorithms exist to reduce it even further, like the Kalman filter and its derivatives [TBF05]. The global localization problem deals with finding the pose in a map without knowing the initial pose. It is an extension to the local localization problem and more difficult. But once the robot has found its pose, the problem degenerates into a local localization problem. However, when the robot is relocated by someone, it loses track of its pose. This is the kidnapped robot problem, which is the most difficult of the three localization problems.

All three problems have in common that they need a suitable environment representation, a map of the robot's whereabouts. There is a similar class of problems which are referred to as Simultaneous Localization and Mapping (SLAM) problems. In these scenarios there is no prior map given, which makes the problems even more difficult.

2.2 Bayes filters

A good way to model localization problems are Bayes filters. They make use of Bayes' theorem to calculate the probability of a pose x at time t by accounting for the probability at time $t-1$ and the likelihood of the current pose. This makes it possible to derive a state estimation from the data and a likelihood alone. The Bayes' theorem is as follows:

$$P(X|Y) = \frac{P(Y|X)\,P(X)}{P(Y)} \tag{2.1}$$

In this equation, P denotes the probability density function (PDF) of a random variable.

$1/P(Y)$, commonly called η, is the normalization factor. It is usually derived by marginalization:

$$P(Y) = \sum_i P(Y|x_i) \tag{2.2}$$

In a continuous approach the sum is replaced by an integral, but for the algorithm in this thesis the discrete version is sufficient. $P(X)$ is the prior, a previous assumption. The likelihood, the last factor needed for the calculations, can be found in $P(Y|X)$. One can read it as the *probability of* Y *given* X, that means the probability of event Y when event X was observed. The likelihood is a very important part of the algorithm and will be described in detail in Algorithm 5.5. All together Bayes' theorem is used to calculate the posterior probability of X given Y. That is the probability that event X happens when event Y is observed. In other words this means that the current probability for X given Y is derived by calculating how likely event Y is, given event X regarding X's sole probability.

It should be noted that throughout the rest of this thesis, prior and posterior probabilities will be called prior and posterior beliefs, respectively. This terminology has a simple reason: Although a robot does not have beliefs as we use the word in terms of human thoughts, it fits the context better. Probability might imply that the algorithm would try to *predict* the robot's future position—belief on the other hand describes a theoretical possible *current* state.

By using Bayes' theorem over several time steps, it is possible to implement a Markov chain model. A Markov chain model describes a system in which the Markov assumption holds, which says that the state of a system at time t can be calculated by just knowing the state at time $t-1$ but no other states, thus being independent from the others.

```
1  bayes_filter
2      best_belief = 0
3      best_guess = null
4      while true do
5          foreach x_t do
6              bel(x_t) = η likelihood(x_t) bel(x_{t-1})
7              if bel(x_t) >best_belief then
8                  best_belief = bel(x_t)
9                  best_guess = x_t
10             end
11         end
12     end
13 end
```

Algorithm 2.1: Bayes filter

This simple Algorithm 2.1 is the basic form of a Bayes filter. Its purpose is to find a state estimation by *updating* the beliefs ($\text{bel}(x_t)$) for each state x during every time step t and eventually converging to a value close to the true value. This is done by using the former posterior beliefs as priors for the new posterior beliefs. The state with the highest belief is the likeliest.

2.2.1 A one dimensional example

A popular example for demonstrating Bayes filters is the door example, which is illustrated in Figure 2.1. A robot which can only move in one dimension and only sense whether it is in front of one of many indistinguishable doors or not shall localize itself in a corridor.

At the time of initialization the robot has no information, thus its believe to be at a specific position is uniformly distributed (see Figure 2.1(a), $bel(x)$).

After the initialization the robot uses its sensor to search for a door. It finds out that in fact it is located in front of one, so it applies a high likelihood for all positions which are in front of doors (see Figure 2.1(b), $p(x|z)$ where z corresponds to sensor measurement). Now the prior belief $bel(x)$ and the likelihood $p(z|x)$ are used to calculate the posterior belief $p(x|z)$, which is again called $bel(x)$.

$$p(x|z) = \eta\, p(z|x)\, bel(x) \tag{2.3}$$

A repeated measurement would add little information to the robot's belief. So the best solution to get more information is to move around a little and check for doors again. This can be seen in Figure 2.1(c). Especially noteworthy is that the $bel(x)$ distribution is flattened a little bit. This is done to account for inaccuracies in the movement of robots.

During the next time step Formula 2.3 will be used again, with the new normalized belief and a new measurement. Since the robot again senses a door the robot first assigns high likelihoods for the doors, it then calculates the posterior belief. This leads to a considerable high value for the position close to the second door, we say the pose estimation converges to the real value.

During the next movement the $bel(x)$ distribution flattens again, until the robot senses a new door. With the new information about this landmark the robot will be able to update his beliefs accordingly.

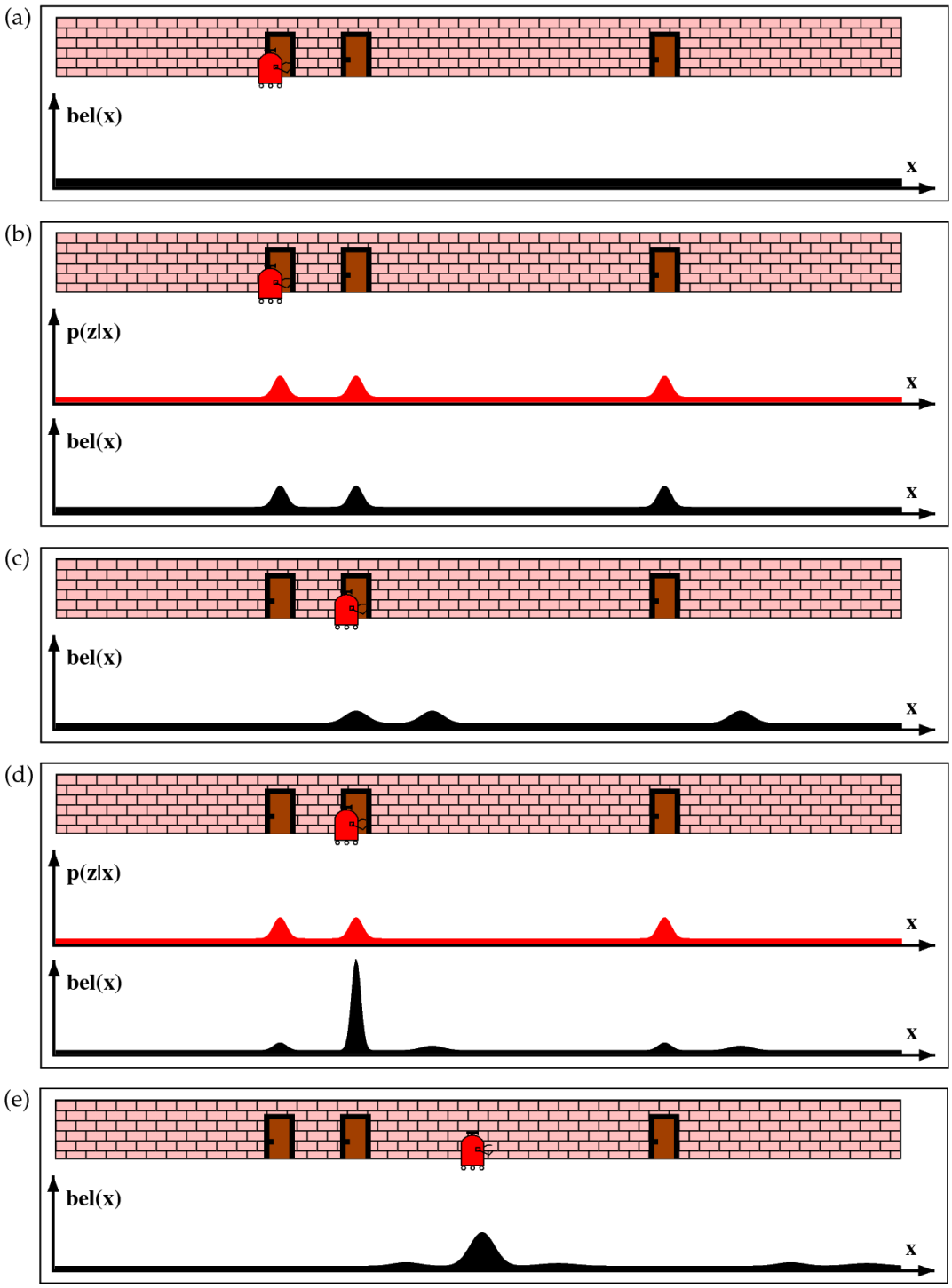

Figure 2.1: Markov localization: A one dimensional corridor with indistinguishable doors. From [TBF05].

Chapter 3

State of the art

The distinction between local localization, global localization and the kidnapped robot problem is often not enough to describe localization problems. To allow for more precise classifications, Thrun et al. also distinguish algorithms in two other categories: Which type of environment is approached and how it is approached [TBF05].

Environments can either be static or dynamic. That means they either stay the same or they change over time and thus makes them harder to navigate. Still most real-life applications in robotics deal with dynamic environments, some examples include opening or closing doors, people walking through the robot's path and many more. Even changes in the environmental lighting can cause the robot's sensor data to yield very different results.

The last metric, the kind of approach chosen, is to distinguish between active and passive approaches. A passive approach is used whenever a robot has some tasks to do and just keeps track of its position while doing them. An active approach controls the robot towards poses which should help in improving the pose hypothesis (the best guess pose), thus reducing the localization error.

Depending on the different characteristics of localization problems there are different algorithms to solve them. In general one can say that algorithms capable of solving kidnapped robot problems are also capable of solving global and local localization problems, while algorithms only designed for local localization problems will not be able to solve global localization nor kidnapped robot problems. It is also the case that algorithms for dynamic environments can solve the problems they are designed for in static environments as well.

3.1 Sample based localization: Particle filters

One popular approach to solve localization problems are particle filters. Instead of representing PDFs as continuous functions as it was done in the 1D example in Section 2.2.1, a particle filter represents a distribution by a set of samples drawn from it. This allows to represent more distributions than continuous models do [TBF05]. The probability of a sample to be drawn is determined by the underlying distribution. Thus a set of samples is just a discrete approximation of the real distribution.

Particle filters are basically an extension to Bayes filters. Each sample has a belief of how

likely it is that it can be drawn from the distribution. A particle filter uses the beliefs of each sample from the last time step as priors to calculate their posterior beliefs of the current time step. Being filters, these algorithms discard samples with low beliefs and replace them with new random samples after each iteration [TBF05]. This way the samples are kept which are the best approximation of the distribution.

3.2 Augmented Monte Carlo Localization

As shown in Figure 2.1, the pose of a mobile robot can be described by a PDF. By using the ability of a particle filter to approximate distributions it is possible to approximate the robot's pose, hence locate the robot. In this case the particle filters implement a motion model to change the samples according to the measured robot motion and a sensor model to provide a simulation of the sensor data, i.e. to determine what sensor data each sample would expect. By comparing each sample's simulated sensor data from the sensor model with the real sensor data it is possible to come up with a likelihood for each sample. After calculating the posterior beliefs for each sample, the unlikeliest are replaced by new random samples. These kind of algorithms are called Monte Carlo Localizations (MCLs).

```
1  Monte_Carlo_Localization ()
2      foreach sample do
3          sample.apply_motion_model()
4          sample.apply_sensor_model()
5      end
6      foreach sample do
7          if sample.belief < θ then
8              sample.replace_sample()
9          end
10     end
11 end
```

Algorithm 3.1: The basic MCL algorithm.

MCL offers different possibilities of adjustments. For instance the motion model and sensor model have to be replaced with suitable variants. Another adjustment is done by Augmented Monte Carlo Localization (AMCL). AMCL adapts the number of samples depending on the overall likelihood of the current samples. That means, that if there are many samples with high beliefs, fewer samples will be generated than in the case that there are only samples with low beliefs. This helps to maintain a stable amount of samples around the likeliest samples without losing the ability to recover from a sudden relocation [TBF05].

Chapter 4

Software, tools and project setup

For the implementation of the algorithm presented in this thesis several tools were needed. A complete list of tools and where to get them can be found in the appendix.

The most important tool was the Robot Operating System (ROS), initially released 2009 by Quigley et al. [QCG+09] The version used for this project was released September 4th, 2013 under the codename *Hydro Medusa*. It runs best under `Ubuntu 12.04 LTS (Precise)`, which was subsequently chosen as the operating system.

ROS provides several advanced tools to write and build software around robots. Small programs and tools are organized in packages and can be run as small individual applications, called *nodes*. These nodes connect to a local ROS master server which handles the communication between them. The communication basically works after the *publisher-subscriber* principle, where some nodes publish certain messages on a certain channel (called *topic*) and other nodes can subscribe to these channels. Another form of communication is the *service-client* principle. In this situation one node offers a service, for which other nodes can send requests and receive answers. Additionally to the basic functions there are also many convenience tools built-in to ROS, for example there exist extendable packages for visualization or dynamic node configuration.

The ray tracing, as described in Chapter 5, is implemented as a ROS service node. Other nodes can request ray traces at specific poses in a predefined polygon map and get the resulting point cloud as an answer. A simple ray tracer implementation is given with the support of Computational Geometry Algorithms Library (CGAL) to perform fast intersection calculations. The resulting point cloud is evaluated with help of Fast Library for Approximate Nearest Neighbors (FLANN).

For the preprocessing of the maps the Las Vegas Reconstruction Toolkit (LVR Toolkit) is employed. The maps are polygon meshes created from points clouds and optimized with the LVR Toolkit as described in [Wie13].

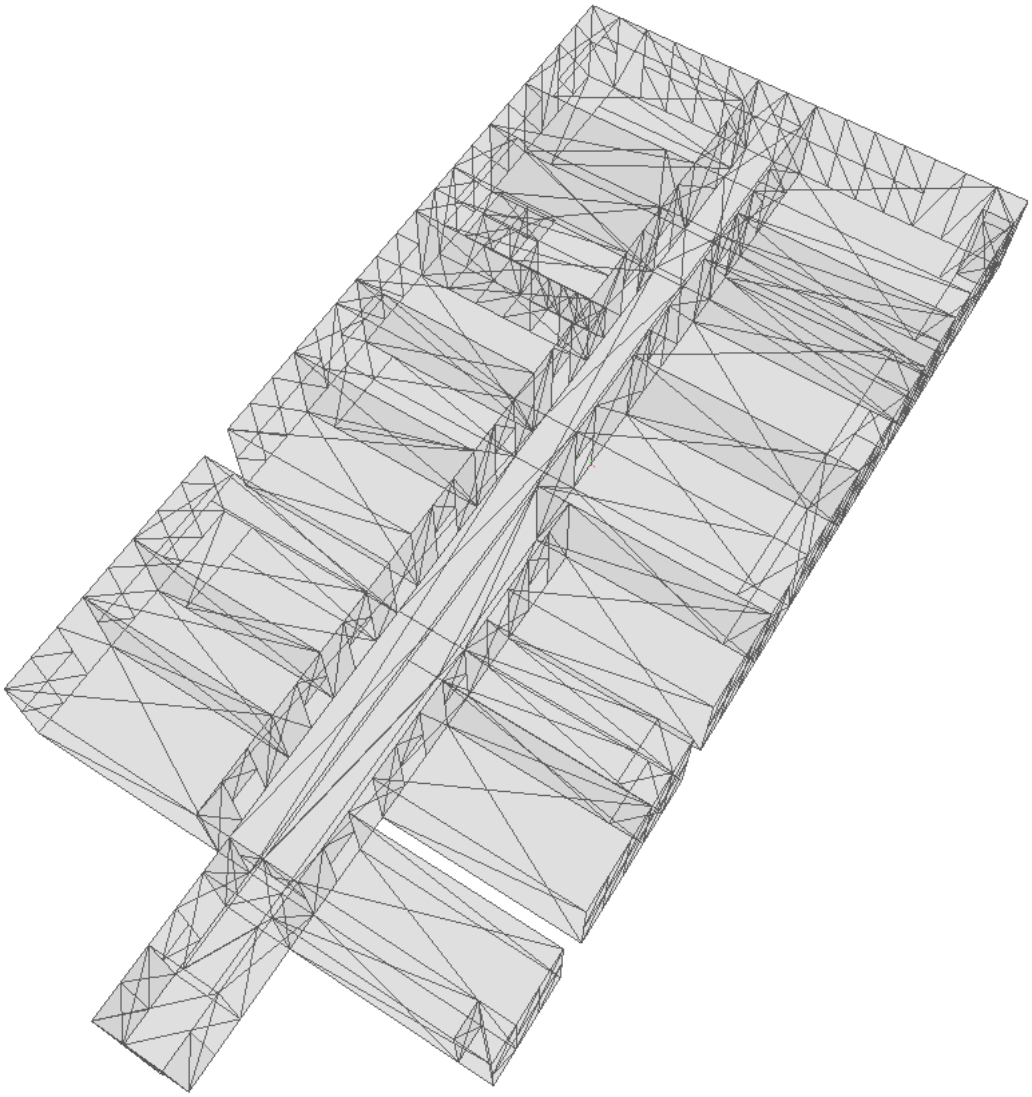

Figure 4.1: An example polygon mesh of an office corridor at the Institute of Computer Science.

Chapter 5

Augmented Monte Carlo Localization in six dimensions

Mobile robots have to work in 3D environments. Such environments can be represented by continuous maps in which robots need to be able to locate themselves. The algorithm Augmented Monte Carlo Localization in six dimensions (AMCL6D) is designed to solve the localization problem in this kind of maps.

AMCL6D is a modification of the AMCL algorithm presented in [TBF05]. Three major components are changed: The motion model, the sensor model, and the evaluation function. The motion model (Algorithm 5.2) supports motion in six dimensions and updates the pose samples accordingly. The sensor model (Algorithm 5.3) is implemented by a ray trace algorithm. This sensor model allows for simulated scan values in the 3D polygon map. Simulated scans are needed for the last changed major component, the evaluation function (Algorithm 5.5). The evaluation function employs a k-Nearest Neighbors (k-NN) algorithm to determine the average distance of the real sensor data to the sensor model. Additionally the sample respawn mechanism was slightly modified. Instead of keeping a varying amount of samples depending on the overall likelihood, a static amount of samples is kept. However, if new samples need to be spawned, some of them are spawned close to samples with high beliefs instead of being spawned randomly.

Before the iterative AMCL6D algorithm starts, the program generates n random samples $X_{t=0}$. They are uniformly distributed across the map, as described in Section 5.4.1. Initially every sample gets assigned with a probability of $1/n$. As soon as some motion is detected, Algorithm 5.1 starts the first iteration and updates the samples X_{t-1} to X_t.

The update procedure consists of several steps. First the motion model (Algorithm 5.2) will be applied to each sample, moving them according to the detected motion. After that, each sample's sensor model needs to be updated. This is done by ray traces as described in Algorithm 5.3. The ray tracer can be adjusted to match any kind of distance sensors, and it has to be adjusted accordingly. The results from the ray traces get evaluated to derive the likelihood for each sample.

The samples' likelihoods and their probabilities are used to calculate their posterior probabilities. Before the last step the samples are sorted according to their posterior probabilities. Their order determines if and how they are respawned: The unlikeliest samples are discarded

```
1  AMCL6D (X_{t-1}, u_t, m_t)
       /* Initialize */
2      X_t = X_{t-1}
3      init θ_rand, θ_close, θ_prob
       /* Update motion and sensor model */
4      foreach x ∈ X_t do
5          x.pose = motion_model(x.pose, u_t)
6          x.raytrace = sensor_model(x.pose)
7          x.likelihood = evaluation(x.raytrace, m_t)
8      end
       /* Find normalization constant η */
9      η = 1 / Σ_{X_t} (x.likelihood · x.probability)
       /* Calculate posterior probabilities */
10     foreach x ∈ X_t do
11         x.probability = η · x.likelihood · x.probability
12     end
       /* Resampling */
13     sort(X_t)
14     foreach x ∈ X_t ∧ x ∉ X_{t,best} do
15         random = rand()
16         if random < 1 − θ_rand then
17             x = resample_random()
18         else if random < 1 − θ_close then
19             x = resample_close(x, X_t)
20         else if x.probability < θ_prob then
21             x = resample_random()
22         end
23     end
24     return X_t
25 end
```

Algorithm 5.1: AMCL6D

and for each of them a new sample is generated.

The algorithm has three input parameters. First the samples from the last time step X_{t-1}, which will be updated to X_t during the algorithm. Each sample $x \in X_t$ has a pose ($x.pose$), a probability ($x.probability$), a likelihood ($x.likelihood$), and a point cloud ($x.raytrace$). The pose and the probability are initialized during the sample generation, while the likelihood and point cloud get assigned later.

To update the samples according to the motion model, the robot motion u_t between times

$t-1$ and t is needed. This is usually supplied by odometry data. The last input parameter is the measured sensor input m_t in form of a point cloud, that is a collection of points which get compared to the sensor model simulation.

Additionally to the input parameters there are a few global parameters to tweak the algorithm, namely θ_{rand}, θ_{close}, and θ_{prob}. They are thresholds used to determine which samples get regenerated and how.

5.1 Motion model

```
1 motion_model (x.pose, u_t)
2 |   noise = generate_noise(Σ)
3 |   x.pose.position+ = u_t.position+noise [0...2]
4 |   q_yaw = new QuaternionAboutAxis(ZAxis, noise [3])
5 |   q_pitch = new QuaternionAboutAxis(YAxis, noise [4])
6 |   q_roll = new QuaternionAboutAxis(XAxis, noise [5])
7 |   x.pose.orientation = q_roll · q_pitch · q_yaw · u_t.orientation^-1 · x.pose.orientation
8 |   return x.pose
9 end
```

Algorithm 5.2: Motion model

Real sensors are subject to noise. Whenever a robot moves or turns around, its odometry data will have some errors. To update the pose samples accordingly, the motion model in Algorithm 5.2 has to describe these uncertainties as well.

Therefor it makes use of a covariance matrix Σ, which stores how the error values of the sensors are related to each other. There are different relations for every robot, so the correct values have to be estimated or determined empirically.

After any arbitrary time step the pose samples have to be updated according to the robot's movement, i.e. the motion model. It adds the difference between the start and goal pose (i.e. the movement) to the current pose and additionally some noise values which are drawn from a multivariate normal distribution. The distribution is around the origin and its variances are taken from the covariance matrix Σ mentioned above.

An illustration can be found at the 2D example in Figure 5.1. The initial pose p_{t-1} at time $t-1$ gets updated with the motion Δp and the noise drawn from the normal distribution. It is more likely for the final pose p_t to be close to the original motion. Since multivariate normal distributions do not have analytical cumulative distribution functions (CDFs) (which could be inverted to sample from them), one has to use some other techniques to sample from them. A very common approach is described by James E. Gentle [Gen05]. It generates samples with the following formula:

$$n = C^T z + \mu \tag{5.1}$$

Here z is a vector of independent and identically distributed (i.i.d.) random values drawn from one dimensional standard normal distributions. The vector z gets right-multiplied to the

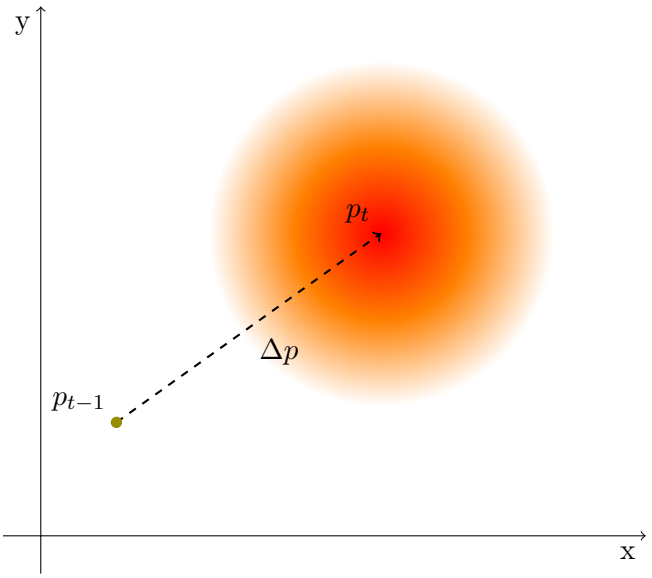

Figure 5.1: The motion model updates the pose sample p by adding the motion Δp and some normal distributed noise (using the identity matrix I as deviations). The sample has a high likelihood to be at the dark red area, though it can be in the outer areas as well.

matrix C^T, which is the Cholesky factorization of the covariance matrix (such that $C^T C = \Sigma$). This adjusts the sampled values to fit the multivariate normal distribution. Lastly, the μ-vector gets added in order to shift the means to the correct positions.

Contrary to the classical AMCL, in AMCL6D not only three but six random values are drawn for each pose sample. The first three values simply get added to the position: $(x_t, y_t, z_t)^T = (x_{t-1}, y_{t-1}, z_{t-1})^T + \Delta p + (n_1, n_2, n_3)^T$, where n_1 describes the first element of the vector n, the sampled noise. Since the robot's covariance matrix uses Euler angles but the orientations are handled by quaternions, the values for the rotation can not simply be added. Instead they are used to create new quaternions q_{yaw}, q_{pitch}, and q_{roll} to express rotations about the yaw, pitch and roll axes respectively (in Algorithm 5.2 the ROS coordinate system is used). These three rotations get multiplied with the updated orientation (that is q_{t-1} rotated by the inverse of the orientation difference Δq^{-1}): $q_t = q_{roll} \cdot q_{pitch} \cdot q_{yaw} \cdot \Delta q^{-1} \cdot q_{t-1}$. Note that quaternion multiplication is non-commutative, so the order is important: The right-most rotation is done first. This order of rotations results in the traditional three body rotation inspired in aviation: First an airplane is able to yaw, when it starts it pitches and in the air it is additionally able to roll.

Putting the position and orientation together the pose sample for time t is updated with the correct noise values.

5.2 Sensor model

The sensor model, illustrated in Algorithm 5.3, is an exchangeable component of AMCL. AMCL6D is working on continuous polygon maps, which require a suiting form of sensor models.

```
1 sensor_model (x.pose)
2 |   raytrace = raytrace_at(x.pose)
3 |   raytrace = normalize_raytrace(raytrace, x.pose)
4 |   return raytrace
5 end
```

Algorithm 5.3: Sensor model

For AMCL6D the sensor model is implemented as a ray tracer. This allows to simulate accurate sensor data in the form of point clouds in continuous maps.

The ray tracer used in AMCL6D calculates the intersections of rays emanating from the robot's position with the map. Since laser scanners, cameras, or other sensors do not only emit a single light ray, the ray tracer for AMCL6D also needs to test many rays.

Since the rays are supposed to simulate a camera, they are cast through an image plane. This helps to determine a second point to construct the rays: The image plane will be divided into small cells, depending on the simulated resolution. One ray is cast through each of these cells and for each of them it has to be checked whether it intersects with the map or not. If it does, the intersection point is stored into a point cloud. The ray tracing is illustrated in Figure 5.2. The image plane is calculated with the aperture angles (θ for the horizontal and ϕ

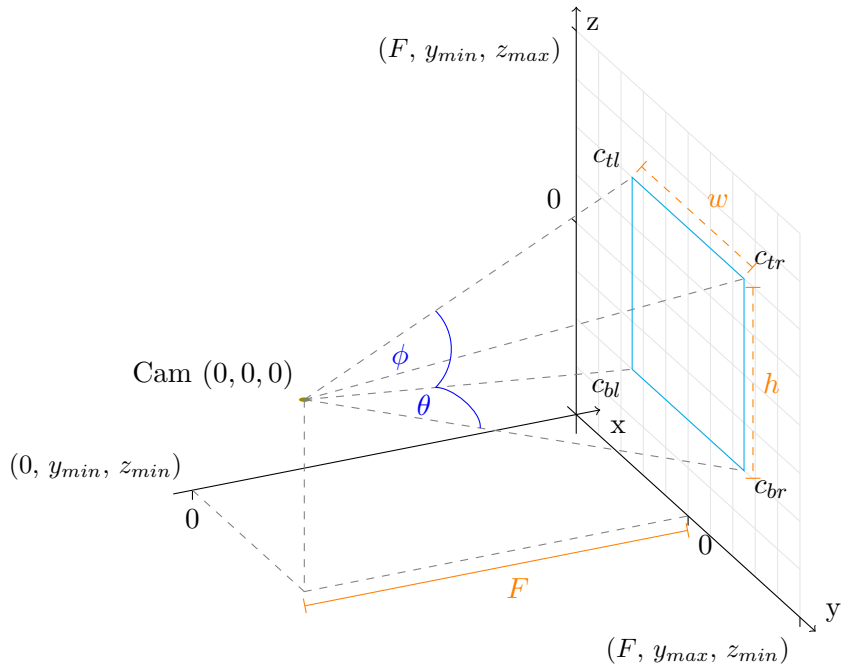

Figure 5.2: Calculation of the virtual image plane for the ray tracing.

for the vertical aperture angle, measured in radians) and the focal length (F) of the simulated sensor. It is described by the intersections of its four edges and gets divided into cells according

to the horizontal and vertical resolution of the sensor. To calculate the intersections in the virtual image plane, the following formulas are used.

$$y_{min} = F \cdot \tan -\frac{\theta}{2} \tag{5.2}$$

$$y_{max} = F \cdot \tan \frac{\theta}{2} \tag{5.3}$$

$$z_{min} = F \cdot \tan -\frac{\phi}{2} \tag{5.4}$$

$$z_{max} = F \cdot \tan \frac{\phi}{2} \tag{5.5}$$

Thus, the image plane's corners are $c_{bl} = (y_{min}, z_{min}), c_{br} = (y_{max}, z_{min}), c_{tl} = (y_{min}, z_{max})$, and $c_{tr} = (y_{max}, z_{max})$. The cell size (*w*idth and *h*eight) is calculated with $w = |c_{br} - c_{bl}|/res_{hor}$ and $h = |c_{tl} - c_{bl}|/res_{ver}$, with res_{hor} being the horizontal and res_{ver} the vertical resolution.

However, algorithmically it is not necessary to calculate all these points or individual cells. $y_{min/max}, z_{min/max}$ and the cell size of the image plane's cells, w and h, are already enough to iterate over the plane and cast a ray for each cell with Algorithm 5.4. The resulting intersections are stored into an array of points which is called point cloud.

```
1  y = y_min
2  z = z_min
3  while y ≤ y_max do
4  |   while z ≤ z_max do
5  |   |   transform (y, z) relative to pose
6  |   |   cast ray from pose.position through (y, z), store intersection
7  |   |   z+ = h
8  |   end
9  |   y+ = w
10 end
```

Algorithm 5.4: Iterating the virtual image plane to cast rays through each cell. The transformation is achieved with the inverse of the matrix representing the pose, as described in Formula 5.6 and Formula 5.7.

To speed up the intersection checks, the CGAL offers the ability to build an Axis-Aligned Bounding Boxes tree (AABB tree) from the map data [ATW14]. The AABB tree stores the mesh faces of the polygon map in axis-aligned bounding boxes. Axis-aligned bounding boxes are cuboids with all their edges parallel to the corresponding axis of the global coordinate system. They AABB tree organizes them in a tree structure, this makes it faster to process the data. The speed up is achieved by checking for an intersection of the ray with a bounding box before calculating the exact intersection with the encased geometry. Additionally the fact that the bounding boxes are axis-aligned helps with the speed up, it allows for only simple predicate calls [ATW14]. Figure 5.4 pictures an example ray trace showing the simulated robot's pose (arrow) and the point cloud which resulted from the ray trace at that pose. To be able to compare

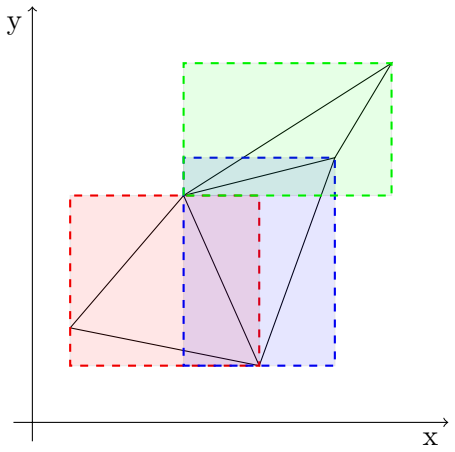

Figure 5.3: Axis-aligned bounding boxes encasing a simple mesh consisting of three triangles.

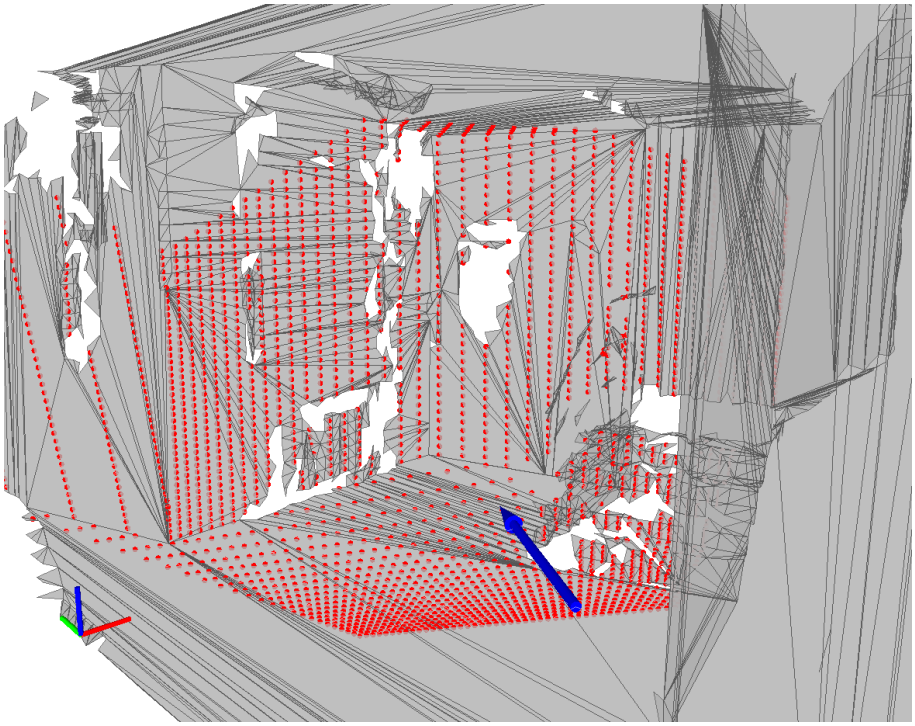

Figure 5.4: A simulated ray trace sample.

the point cloud generated by the ray trace with the point cloud from the sensor, they need to be in the same reference frame. This means they both have to be transformed in a way such that their origins are at the same point. AMCL6D uses the global origin. To transform a pose to the origin, it is possible to use an affine transformation matrix. This affine transformation matrix A can be constructed easily: The orientation of the pose can be expressed by a 3×3 rotation matrix R and the translation by a 3×1 matrix T. These two matrices can be combined and

extended to a 4×4 matrix (Formula 5.6).

$$A = \begin{pmatrix} R_{1,1} & R_{1,2} & R_{1,3} & T_{1,1} \\ R_{2,1} & R_{2,2} & R_{2,3} & T_{2,1} \\ R_{3,1} & R_{3,2} & R_{3,3} & T_{3,1} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.6}$$

A is the matrix which describes the transformation needed from the global origin to the pose. By taking the inverse A^{-1} it is possible to calculate a matrix which describes the opposite: the transformation from the pose back to the origin. This inverted matrix A^{-1} can be applied to each point p in the point cloud result ($\mathbb{P}$) from the ray trace to transform them to positions which are relative to the origin as they were to the pose. Note that the points need to be homogenous, that means they need to be extended with a fourth component which is 1.

$$\forall p \in \mathbb{P} : \begin{pmatrix} p \\ 1 \end{pmatrix} = A^{-1} \begin{pmatrix} p \\ 1 \end{pmatrix} \tag{5.7}$$

The resulting point cloud can be used to evaluate the corresponding sample in the evaluation function.

5.3 Evaluation function

```
1 evaluation (x.raytrace, m_t)
2 |   kd_tree = prepare_kd_tree(x.raytrace)
3 |   average_distance = ε
4 |   foreach p ∈ m_t do
5 |   |   nearest_neighbor = k_nearest_neighbors(p, kd_tree, 1)
6 |   |   average_distance += euclidean_distance(p, nearest_neighbor) / size(m_t)
7 |   end
8 |   return 1/average_distance
9 end
```

Algorithm 5.5: Sample evaluation

The likelihood for each sample gets calculated in the evaluation function (Algorithm 5.5). Most of it is done with the FLANN. First the sample's point cloud gets organized into a k-dimensional tree (kd-tree). In the second step a k-NN search is performed for each point in the sensor measurements to determine each respective closest neighbor.

A kd-tree is a binary tree structure which recursively splits the space and stores each subspace into a child node of the previous one. Thus the root node resembles the whole space and the first split plane, its child nodes contain the subspaces yielded by the split plane. This is done until there is only one point left in a subspace, which becomes a leaf node.

The k-NN algorithm is used to find similarities in high dimensional data [ML14]. Its purpose is to find the k points of a dataset which are closest to a given point. The most naive version of such an algorithm would just iterate over all points to search the k closest points, FLANN takes advantage of the kd-tree to speed up the calculations. In AMCL6D the k-NN algorithm is used as a heuristic to find the distance to the closest point ($k = 1$) in the simulated sensor data for each point in the real sensor data. For each sample the mean distance to the real data is calculated.

By inverting the mean distance between the real sensor data and a sample it is possible to come up with a value between 0 and 1. This value is inverse proportional to the distance, that means if the distance is huge the value becomes really small, but if the distance is small the value gets closer to 1. To avoid division by zero in the unlikely but possible case that the measurements have a distance of 0, the average distance gets initialized with an arbitrary small value $\epsilon \in \mathbb{R} > 0$.

The result of the evaluation function is used as the likelihood for the calculation of a sample's new probability: $x.probability_t = \eta \cdot x.likelihood \cdot x.probability_{t-1}$. η is the normalization constant, which is the marginalization over all posterior probabilities (see Algorithm 5.1, Line 9). After evaluating the samples, they are sorted by their posterior probabilities. This allows for easier regeneration of poses with low posterior beliefs.

5.4 Sample generation and regeneration

One important part of the algorithm is sampling. Not only the sampling of new poses is crucial, it is also important that the noise in the motion model is sampled correctly.

Therefore the algorithm uses three different sampling strategies. The chosen strategy depends on the use case for the sampled values. The first strategy is a uniform sampling to generate the initial poses. The second strategy (described in Section 5.1) is used for sampling noise according to the robot's sensor precision to simulate an accurate motion model. The last strategy is used to determine which poses need to be removed and where to regenerate new samples. This process filters the poses by replacing unlikely pose samples with pose samples biased towards the likeliest ones.

5.4.1 Generating initial pose samples

The generation of new pose samples is done by drawing random numbers from a uniform distribution. For the position there is no more work needed than mapping the sampled values into the correct range. For a random value v from the range S the remapped value v_n from the range T gets calculated by the formula $v_n = T_{min} + v\frac{|S|}{|T|}$, where $|S|$ denotes the length of range S (or T, respectively) and T_{min} denotes the smallest value of T. For the three needed values x, y, and

z this results in the following calculations.

$$x = x_{min} + \mathbf{rand()}\,(x_{max} - x_{min}) \tag{5.8}$$
$$y = y_{min} + \mathbf{rand()}\,(y_{max} - y_{min}) \tag{5.9}$$
$$z = z_{min} + \mathbf{rand()}\,(z_{max} - z_{min}) \tag{5.10}$$
$$v = (x, y, z) \tag{5.11}$$

In this equation **rand**() draws i.i.d. random numbers from $R_s = [0, \ldots, 1]$, the source range. x_{min} and x_{max} (for y and z respectively) describe the target range $R_t = [x_{min}, \ldots, x_{max}]$ in which the samples shall be generated for the respective dimension. In this thesis they correspond to the boundaries of the map in each dimension.

By subtracting the minimum value of the source range R_s from the sampled value, the range the random value is drawn from is shifted to zero—in this case the subtracted value would be 0, so it is left out. The value $x_{max} - x_{min}$ is the ratio between the lengths of the two ranges $|R_t|/|R_s|$. A multiplication of the shifted value and the ratio maps the value to the correct value in a zero shifted target interval with the length of the target range $|R_t|$. By adding the target range's minimal value, x_{min}, the sampled value gets shifted to the correct range.

This procedure is done for each of the three positional coordinates, resulting in the positional vector $v = (x, y, z)$.

Sampling the orientation is done with the *subgroup algorithm* by Ken Shoemake [Sho95]. It generates a unit quaternion from three uniformly i.i.d. random values X_0, X_1, X_2 of the following form:

$$q = \left(\sin 2\pi X_1 \sqrt{1 - X_0},\ \cos 2\pi X_1 \sqrt{1 - X_0},\ \sin 2\pi X_2 \sqrt{X_0},\ \cos 2\pi X_2 \sqrt{X_0}\right) \tag{5.12}$$

The generated vector v and the orientation quaternion q form the sampled robot pose. Initially all poses get the same belief of $1/n$, where n is the number of samples.

5.4.2 Regenerating samples

Sample regeneration is performed as the last step in the AMCL6D algorithm. The method used for resampling individual particles depends on $\theta_{close}, \theta_{random}$, and θ_{prob}. The thresholds θ_{close} and θ_{random} are percentages. Their sum $\theta_{close} + \theta_{random}$ yields the percentage of the particles which will be regenerated. Thus if $\theta_{close} = 0.15$ and $\theta_{random} = 0.45$, the unlikeliest (meaning those with the lowest beliefs) 60% of the samples will be respawned, either *closely* or *randomly*, respectively. Additionally all samples with a belief smaller than θ_{prob} will be respawned *randomly*.

The samples which get respawned *randomly* are definitely respawned, just the method how they are respawned is similar to the initial sample generation. They get a new uniform random pose and a belief of $1/n$.

For the samples which are respawned *closely* another method is used. The new samples shall be close to good samples, so first it has to be determined which pose samples are good. AMCL6D uses a fixed number of best samples, the samples with the highest beliefs. Then for each new sample a random sample from the best samples is chosen and used as a seed to generate a new pose. To generate a sample nearby, the same calculations are used which are already used in

the motion model (see Formula 5.1), only the parameters are changed. The covariance matrix Σ can be adjusted manually, for AMCL6D it is a slightly modified identity matrix.

$$\Sigma = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \alpha & 0 & 0 \\ 0 & 0 & 0 & 0 & \alpha & 0 \\ 0 & 0 & 0 & 0 & 0 & \alpha \end{pmatrix} \tag{5.13}$$

The value α is the change in orientation (in radians) we want to achieve. If for example $\alpha = {}^{5\pi}/_{180}$, this would result in values of $\pm 5^\circ$ relative to the reference sample. Of course the 1s could also be replaced, now they resemble a standard normal distribution for the position.

$$n = C^T z + \mu \tag{5.14}$$

Formula 5.14 is the same as Formula 5.1, used in the motion model. When the Cholesky factorization of Σ is multiplied with a vector of random values and added to a mean vector μ which holds the position of the reference sample and three 0s, the resulting values can be used to construct a sample which is close to the reference. To do so, the position of the new pose has to be set to the three first result values. The orientation needs to be calculated similar to the orientation change in the motion model, by using three quaternions and rotating the orientation of the reference pose accordingly. The resulting pose is somewhere around the reference pose, depending on the choice of Σ this may be closer or further.

Chapter 6

Simulation and results

The ROS package built for AMCL6D also includes a simple tool to simulate a robot pose. This tool was used to simulate and test the algorithm. The algorithm can be visualized using RVIZ, which is illustrated in Figure 6.1. The performance of the algorithm was tested on different maps and with different resolutions. To assure accuracy, for each map there was a random but arbitrary sequence of motions defined which was repeated for each trial. The simulated camera for the ray tracing always had aperture angles of 90° (horizontal) and 60° (vertical).

6.1 Performance in different maps

The performance is measured by taking the distance between the simulated real pose and the best hypothesis after each time step, hence lower values mean better results. For the comparisons in these sections the euclidean distance between the poses' positions was used, as it has greater impact on the localization.

Elevator door

The elevator door map is open on one side and has lots of holes. It is smaller in size than the office map, but has more vertices and faces.

The results in Figure 6.4 show that the algorithm is not capable of localizing the robot within 100 iterations. However it is interesting that the algorithm decreases the distance steadily during the first half of each experiment but after about 50 iterations it spikes far away and has problems to recover.

Office corridor

The office corridor map is about seven times larger than the elevator door map, however it has less vertices and faces. This is due to its very regular structure. In contrast to the other map, the office corridor is complete and without any holes.

The results illustrated in Figure 6.5 are worse than those of the elevator door map. In the office corridor the algorithm was not able to localize the robot, additionally the tendency observed in the first iterations in the elevator door map is not visible. This is most probably

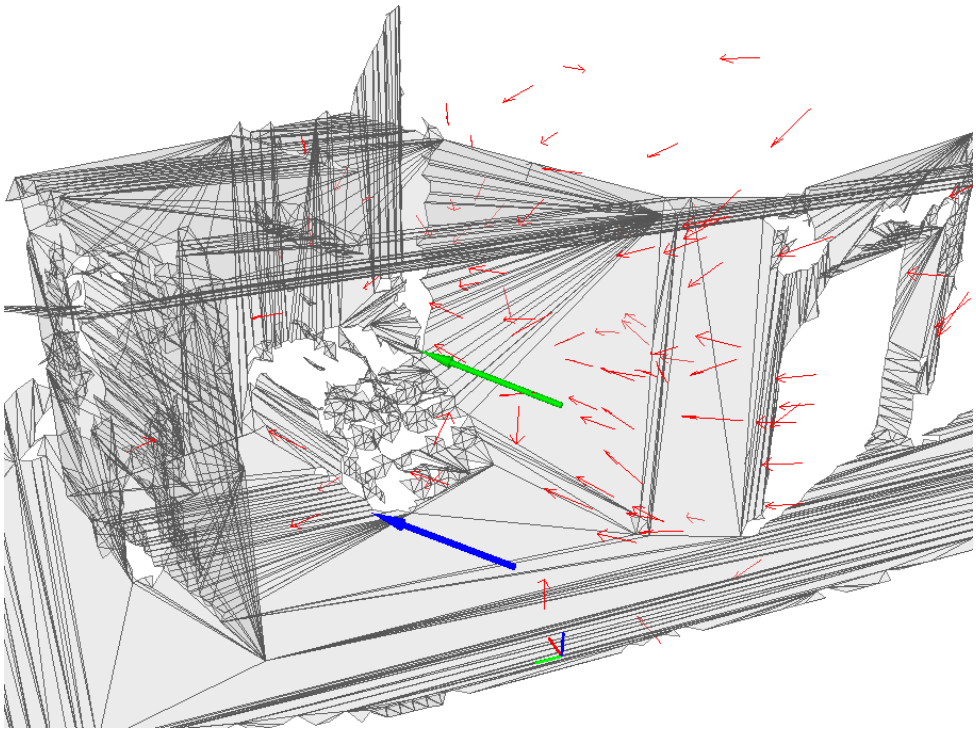

Figure 6.1: The localization of AMCL6D. The small red arrows illustrate the samples, the green arrow the likeliest sample and the blue arrow the real value.

because the sample size of just 100 samples is really small and the map is huger than the other one. However for larger sample sizes the computational time of the ray tracer is not acceptable anymore, as it already was a huge bottleneck with different resolutions (see Section 6.3).

Map and test properties	
Vertex count	3,774
Face count	3,924
Iterations	100
Resolution	65x45
Approx. Dimensions	6x23x4

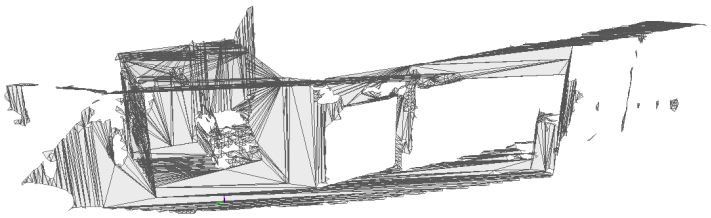

Figure 6.2: Map and test properties: Elevator door

Map and test properties	
Vertex count	1,018
Face count	857
Iterations	100
Resolution	65x45
Approx. Dimensions	15x33x8

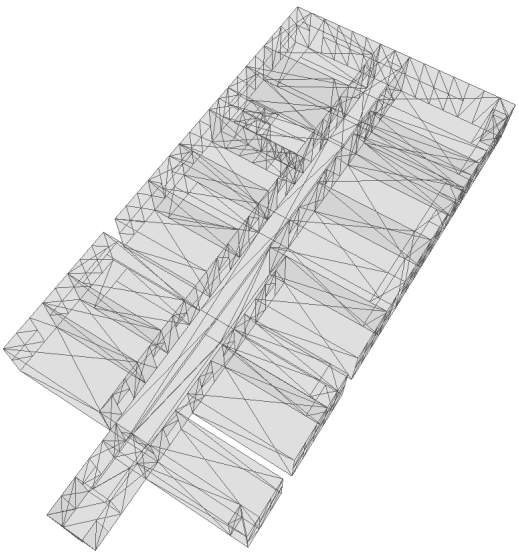

Figure 6.3: Map and test properties: Office corridor

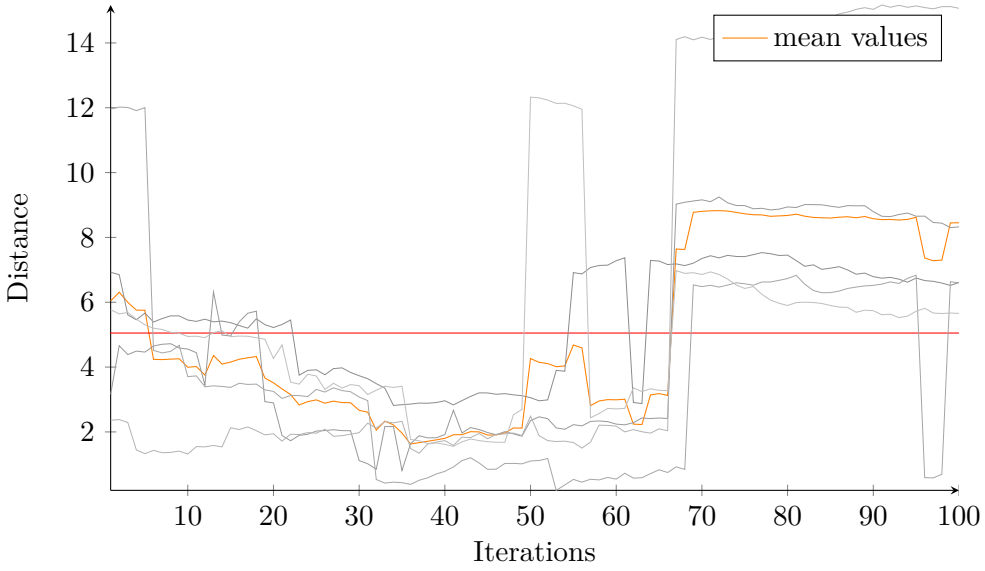

Figure 6.4: Trial results of the elevator door map. The red line shows the over all mean distance.

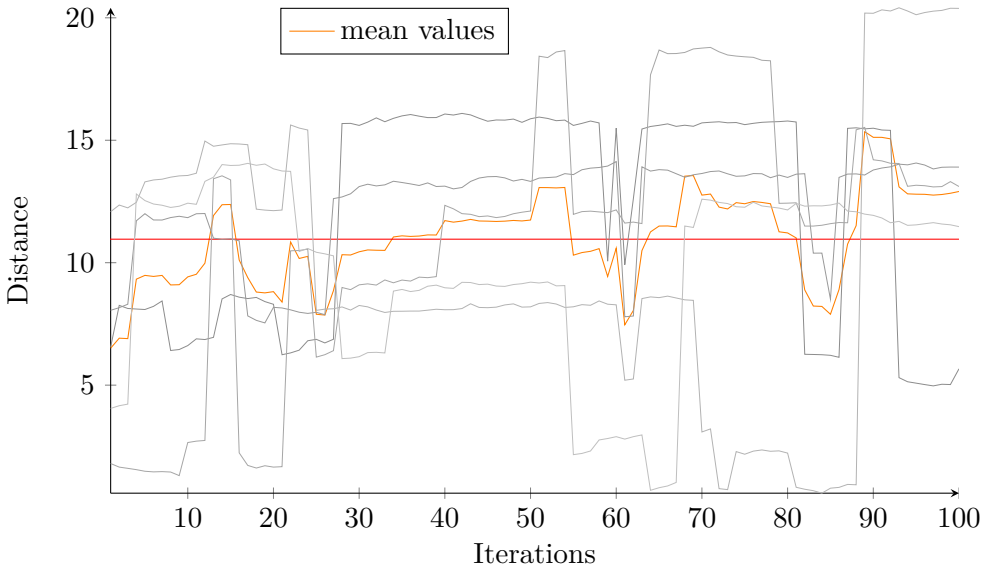

Figure 6.5: Trial results of the office corridor map. The red line shows the over all mean distance.

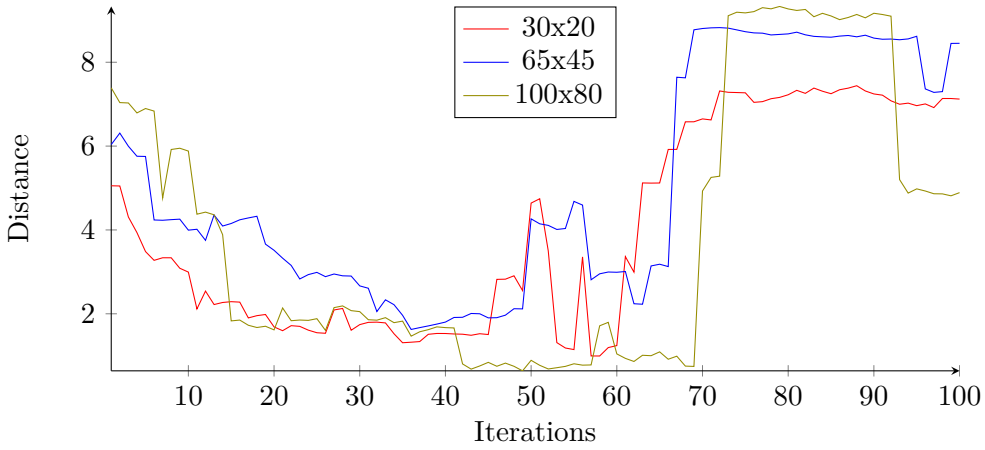

Figure 6.6: Mean results of trials with different resolutions.

6.2 Resolution of ray tracer

To see how much impact the resolution of the sensor model has and if there are better resolutions, tests in the same elevator door map (Figure 6.2) with different resolutions of 100x80 (2 runs), 65x45 (5), 30x20 (5) were compared.

The mean results are illustrated in Figure 6.6. As one can see, all resolutions perform similar and there is none which might be preferable over the others.

6.3 Ray tracing speed

The algorithm performed very slow on the medium and high resolutions. While following the assumption that the ray tracer might be a bottleneck, the computation time of the ray tracer was tracked to find out its portion of time used during execution of the algorithm. The results show that the ray tracer consumes most of the algorithm's running time, with rates varying between 93 and 96 percent. An improved ray tracer might therefor speed up the calculations enormously. It can also be seen in Table 6.1 that therefore smaller numbers of vertices and faces improve the time needed for the ray tracing. Using the mesh optimization software of the LVR Toolkit can thus lead to better results.

6.4 Summary

AMCL6D initially performed well on a small map, independently from the ray tracer's resolution. However, once it got close to the real value it suddenly jumps far away. This makes it less useful in pose tracking than I initially hoped for. Additionally it was found that the ray tracer is really slow and throttles down the whole approach. Maybe replacing the ray tracer allows for more iterations in reasonable time frames and thus for a recovery from the loss of the pose tracking.

Run	Map	Resolution	Total time (s)	RT time (s)	$^{RT}/_{Total}$ (%)
1	Elevator	65x45	2139	2037	95.23
2	Elevator	65x45	2191	2085	95.16
3	Elevator	65x45	1867	1766	94.59
4	Elevator	65x45	2250	2146	95.38
5	Elevator	65x45	2084	1979	94.96
6	Elevator	65x45	2229	2124	95.29
7	Elevator	65x45	1944	1840	94.65
8	Elevator	30x20	481	457	95.01
9	Elevator	30x20	430	407	94.65
10	Elevator	30x20	528	504	95.45
11	Elevator	30x20	407	384	94.35
12	Elevator	30x20	464	443	95.47
13	Elevator	100x80	6166	5890	95.52
14	Elevator	100x80	6012	5730	95.31
15	Office	65x45	787	745	94.66
16	Office	65x45	764	722	94.50
17	Office	65x45	803	762	94.89
18	Office	65x45	786	737	93.77
19	Office	65x45	827	786	95.04

Table 6.1: Ray trace computation time comparisons

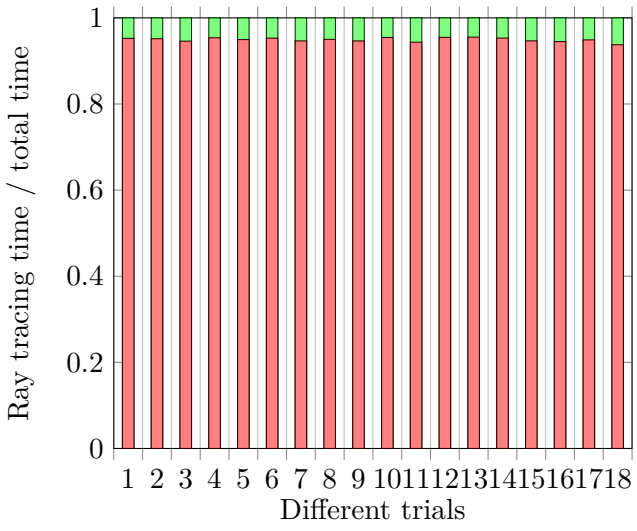

Figure 6.7: Comparison of the total time to ray trace time. The total execution time for each trial is set to 1. The red area is the time consumption by the ray tracer.

Chapter 7

Future improvements and possible changes

Dynamic respawn

One element of AMCL was changed in AMCL6D: The dynamic respawn rate of samples (cf. [TBF05]). Depending on how high the belief for specific poses is, AMCL spawns more or less new pose samples. This allows for a more fine grained control of where to regenerate new samples and also changes the computational time needed for each iteration. If the belief for some pose samples is very high, less samples need to be checked in total until there are some far away from the good samples with like results. The less samples are maintained, the less computational effort is needed. This is especially important for the ray tracer.

Ray tracer and computation times

During the tests and simulation the ray tracer was found to be by far the slowest part of the algorithm. Improvements will most probably be achieved by replacing CGAL with a faster library. Surprisingly it did not make a huge difference in the results which resolutions were used, but a huge difference in computation times. So using a smaller resolution is also a way to reduce computation times.

Evaluation function

Another very important factor is the evaluation function. As in other applications where likelihoods are derived from comparisons with sensor inputs it is important to find the right evaluation to come up with reasonable values. The method used in AMCL6D seemed to be quite reasonable: The average distance between two point clouds would be 0 if they were equal and increases with more points being further away from this baseline. But maybe simply another evaluation function has to be found.

Multiple sensors

If all these changes are done and AMCL6D turns out to work, there are still many things to do. First of course, it should be tested on a real robot and not only in simulations. After that it is possible to extend AMCL6D further, for example by adding a second sensor which is directed to another direction than the first. This could help to reduce ambiguities and thus lead to faster convergence, but also introduces more data to process. This leads to the problem that sometimes better sensors are difficult to handle due to their different kind of data [Smi94]—or in this case more because of the amount of data.

Conclusion

As it turned out, AMCL6D as implemented for this thesis, is not yet very effective. It comes close to the correct poses, but loses track of it and fails to recover from the loss. However, there is still much room left for improvements and changes, and since localization in continuous 3D maps will stay an important topic in the future, it is worth pursuing this or similar approaches.

List of Figures

List of Algorithms

List of Acronyms

AABB tree Axis-Aligned Bounding Boxes tree.

AMCL Augmented Monte Carlo Localization.

AMCL6D Augmented Monte Carlo Localization in six dimensions.

CDF cumulative distribution function.

CGAL Computational Geometry Algorithms Library.

FLANN Fast Library for Approximate Nearest Neighbors.

i.i.d. independent and identically distributed.

k-NN k-Nearest Neighbors.

***k*d-tree** k-dimensional tree.

LVR Toolkit Las Vegas Reconstruction Toolkit.

MCL Monte Carlo Localization.

PDF probability density function.

ROS Robot Operating System.

SLAM Simultaneous Localization and Mapping.

List of tools and software

This is a list of tools and software used to implement AMCL6D. The links were checked before printing the thesis, however they might not exist by the time of reading. You can still try to search the web for the corresponding versions or hope for new versions of each library or tool to work as well.

Tool/ Software	**Version**	**Release**	**URL**
Boost	1.46.1	2011-03-12	`http://www.boost.org/users/history/version_1_46_1.html`
CGAL	3.9-1	2011-10-17	`https://launchpad.net/ubuntu/+source/cgal/3.9-1`
Eigen	3.0.5-1	2012-02-22	`https://launchpad.net/ubuntu/+source/eigen3/3.0.5-1`
FLANN	1.7.1-1	2011-12-20 (v1.7.0)	`http://flann.sourcearchive.com/downloads/1.7.1-1/`
LVR Toolkit	0.7	2013-06-06	`http://www.las-vegas.uni-osnabrueck.de/index.php/download`
OpenMPI	1.4.3	2010-10-05	`http://www.open-mpi.org/software/ompi/v1.4/`
PCL	1.7.0	2013-07-23	`http://casestudies.pointclouds.org/downloads/linux.html`
ROS	Hydro Medusa	2013-09-14	`http://wiki.ros.org/hydro/Installation`
Ubuntu	12.04.5 TLS	2014-08-07	`http://releases.ubuntu.com/12.04/`

University git repositories

Repository	**Description**	**URL**
AMCL6D	AMCL6D code	`software@kure.informatik.uos.de/amcl6d`
meshing.pg2013	RVIZ plugin	`software@kure.informatik.uos.de/meshing.pg2013`
lvr_tools	LVR Toolkit	`software@kure.informatik.uos.de/roslvr`

Bibliography

[ATW14] Pierre Alliez, Stéphane Tayeb, and Camille Wormser. 3d fast intersection and distance computation (aabb tree). In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.5 edition, 2014. Last accessed: October 31st, 2014.

[DFBT99] Frank Dellaert, Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Monte Carlo Localization for Mobile Robots. In *Robotics and Automation*, pages 64–72, Detroit, MI, 1999. IEEE.

[Gen05] James E. Gentle. *Random Number Generation and Monte Carlo Methods*. Statistics and Computing. Springer, New York, second edition, 2005. ISBN 0-387-00178-6.

[ML14] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36, 2014.

[QCG+09] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[Rob08] RoboCup. Robot World Cup Soccer Games and Conferences. `http://www.robocup.org`, 2008.

[Sho95] Ken Shoemake. Uniform Random Rotations. In David Kirk, editor, *Graphics Gems III*, The Graphics Gems Series, pages 124–132. Academic Press, Inc., Palo Alto, California, 1995. ISBN 0-12-059756-X.

[Smi94] Tim Smithers. On Why Better Robots Make it Harder. In *Proceedings of the third international conference on Simulation of adaptive behavior: from animals to animats 3: from animals to animats 3*, pages 64–72. MIT Press, D. Cliff et al., 1994.

[TBF05] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. The MIT Press, Cambridge, Massachusetts, 2005.

[Wie13] Thomas Wiemann. *Automatische Generierung dreidimensionaler Polygonkarten für mobile Roboter*. PhD thesis, Universität Osnabrück, Osnabrück, 2013. urn:nbn:de:gbv:700-2013050710827.

Declaration of Authorship

I hereby certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other university.

Osnabrück, November 2014