Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor Thesis Nr. 2493

Path Planning and Optimization on SLAM-Based Maps

Daniel Estler

Course of Study:	Technische Kybernetik
Examiner:	Prof. Dr. rer. nat. Marc Toussaint
Supervisor:	Ph. D. Vien Ngo
Commenced:	28.05.2016
Completed:	31.10.2016
CR-Classification:	G.1.6,I.2.8,I.2.9

Kurzfassung

Die vorliegende Arbeit zeigt anhand eines praktischen Beispiels wie durch Simultaneous Localization and Mapping (SLAM) entstandene Karten zur Pfadplanung und Pfadoptimierung verwendet werden können. Dazu geht sie auf Ansätze zur Lösung des SLAM Problems ein und beschreibt die Eigenschaften der resultierenden Karten. Es werden außerdem die Prinzipien der Pfadplanung und -optimierung angesprochen. Mithilfe des TurtleBots, als Beispiel für einen mobilen Roboter, wird eine Möglichkeit gezeigt, wie bei der Planung von Pfaden adäquat mit den Eigenschaften von SLAM-basierten Karten umgegangen werden kann. Ein besonderer Augenmerk liegt dabei auf der k-order Markov optimization, die zur Glättung der Pfade verwendet wird.

Abstract

This thesis illustrates based on a practical example how maps that result from simultaneous localization and mapping (SLAM) methods can be used for path planning and path optimization. In order to do so approaches to solve the SLAM problem are discussed and the properties of resulting maps are described. Furthermore the principles of path planning and optimization are addressed. By means of the TurtleBot as example of a mobile robot a possibility to appropriately deal with the properties of SLAM-based maps in path planning will be showed. Special attention will be paid to k-order Markov optimization, which is used to smoothen the paths.

Contents

List of Figures

1 Introduction

1.1 Motivation

Mobile robots are gaining significant relevance and starting to enter our daily live. There are, for example, robotic lawn mower or vacuum cleaner. Besides, they become more popular in industry. Many companies started to use automated guided vehicles or mobile manipulators in their warehouses. Moreover, mobile robots are used in explorations of caves, pyramids, reefs or similar things. Another possible application of mobile robots are military operations where they could be used for transportation, reconnaissance or even fighting.

The main request to a mobile robot obviously is mobility. The robot has to be capable of moving from its current pose, defined by the its position and orientation, to a desired target configuration. In order to achieve this it needs to find a valid collision-free path connecting the corresponding configurations. Moreover there may be other demands to the path like to be as short, as fast or as safe as possible.
If the robot wants to find such a path it needs access to several information. Firstly it needs information about itself such as its own size and its manoeuvrability. Furthermore the environment including the current pose and the target pose have to be known. For this purpose mobile robots usually use a map. Now given this information paths can be found. These paths can be calculated with several path planning algorithms. Planning algorithms usually work in discrete domains, which is why often trajectory optimization methods are used in a second step in order to smoother these paths.

This thesis aims to discuss conditions and requirements for this progress of finding paths in the special case of using maps created via simultaneous localization and mapping (SLAM).
SLAM is one of the fundamental problems in mobile robotics since solving it allows truly autonomous navigation ([DWB06]). SLAM methods can create a map of an unknown environment and localize the robot in this map at once. That means they allow the robot to deal with any unknown environment. Therefore they are very popular and often used in mobile robotics. With SLAM there is no more need to create a map and hand it to the robot in order to navigate it, since the robot develops its own.

This is important for the mentioned exploring robots, but also for domestic robots. Let us, for example, think of a robot serving as domestic helper, that is designed to follow orders like "Bring me a soda from the kitchen". In order to follow that order the robot needs, along with many other abilities, to know, where it is right now, where the kitchen is and how it can get there. Since the robot should be operational in the house of any costumer, it would be great, if the robot could develop a map of the house by itself. SLAM methods make exactly that possible, with is a huge relief, since is no more need to for the distributor to create a map of the house of every costumer, which surely would be a lot of work.
Expect for a few applications like cave exploriations, where a enviorment should only be explored and documented, we want to reuse the SLAM maps in the most cases for path planning. In the introduced example of a domestic robot, this once established map should give us the opportunity to find a path between the robot and the kitchen.

Essentially SLAM methods work with probability-based Bayes filters. As a result SLAM-based maps are not absolute but provide the information about obstacles and the robots pose as probabilities.
Due to this fact SLAM maps have special properties that need to be respected in the approaches used to calculate a path. There already exist different ideas and methods to do so. [Val+13] and [VACP11] for example discuss sample-based path planning with pose SLAM.

This thesis will examine path planning and optimization on grid-based SLAM maps. The used approach to compute paths in this map type is based on a combination of the common Dijkstra algorithm and k-order Markov optimization (KOMO). Whereas Dijkstra surely has been used earlier for path planning on SLAM maps, this thesis introduces how KOMO can be used to smoothen paths on SLAM maps. It aims to point out how these already existing methods can be applied on SLAM maps and evaluates the results by means of the TurtleBot as example for a mobile robot.

1.2 Outline

After this introduction the thesis starts stating the SLAM problem and the operating principles of methods to solve it in chapter 2. This serves as background information for better understanding of section 4.2 and chapter 5.
Thereafter chapter 3 discusses methods to calculate paths. Especially the difference between path planning algorithms and path optimization methods is discussed. Besides it presents why a combination of both is often useful. Additionally the Dijkstra algorithm and the k-order Markov optimization are introduced as examples of these two different

strategies.
Chapter 4 presents the TurtleBot as an example of a mobile robot. It describes its build and software environment. In addition it will discuss why the TurtleBot is suitable to examine path planning and optimization on SLAM-based maps.
Afterwards chapter 5 reports how the introduced path finding methods from chapter 3 can be adjusted to work with the properties of SLAM maps. The TurtleBot example form chapter 4 will be picked up in order to do so.
Finally a discussion and evaluation of the practical example will sum up the thesis in chapter 6. It will also state the limitations of the presented approach and present conceivable ideas to repeal these limitations.

2 Simultaneous Localization and Mapping (SLAM)

2.1 Definition of the SLAM Problem

Simultaneous localization and mapping (SLAM) is a basic problem in robotics. It describes the challenge of a mobile robot to incrementally create a consistent map of the environment and estimate its own location in this evolving map at the same time.
This problem is the key for autonomous navigation of robots. Solving it allows to directly move a robot in an unknown environment. The motivation already illustrated some examples where this ability is indispensable. Because of its great significance the SLAM problem is discussed in a lot of papers. This introduction to the SLAM problem is mainly based on [DWB06]. Moreover, the slides[1] from the "robot mapping" course of the University Freiburg, which give a great introduction to SLAM, were used.

In order to describe the problem we define

- $x_t = (x_x, x_y, x_\theta)^T$ as three-dimensional state vector describing the robots pose at time t,
- y_t as sensor observation at time t,
- m as the map,
- u_{t-1} as control command applied at time $t-1$.

Given this notation the challenge of SLAM is to compute the robots state x_t and the map m depending on previous observations $y_{0:t}$ and control commands $u_{0:t-1}$.
For this to happen a motion model returning the current state x_t depending on the previous state x_{t-1} and the last control command u_{t-1} is needed. The motion model is typically derived from the odometry data of the robot. Since odometry is sensitive to errors like non-round contours of the wheels, belt slip or inaccurate calibration this

[1]The slides can be found at: http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/

model is not deterministic. However, it can describe the belief of the state $p(x_t)$ as conditional probability

$$p(x_t) := P(x_t|x_{t-1}, u_{t-1}). \tag{2.1}$$

Furthermore an observation model returning the current expected sensor output y_t depending on the state x_t and the map m has to be known.
Due to inaccuracies of the sensors, this model is also not deterministic, but can be described as conditional probability

$$p(y_t) := P(y_t|x_t, m). \tag{2.2}$$

These models being non-deterministic means that everything we calculate using them will not be absolute. That implies that the SLAM problem can not be solved absolutely. It is not possible to compute the actual state or map. But what can be estimated is a belief over state and map represented by a probability.
Assuming we know the motion and transition model of a certain robot and all previous states $x_{0:t}$, the belief over the map m could be easily computed as

$$p(m) := P(m|x_{0:t}, y_{0:t}, u_{0:t-1}). \tag{2.3}$$

Vice versa, if the map m and the models are known, the state x_t can be estimated as

$$p(x_t) := P(x_t|m, y_{0:t}, u_{0:t-1}). \tag{2.4}$$

But in the SLAM problem the robot knows neither its location nor the surrounding environment. And it is not just that both are unknown, in fact they depend on each other, which makes SLAM a chicken-egg-problem. The challenge is to compute the belief over the map m and the state x_t simultaneously since they can not be estimated one after each other. Moreover this has to be done carefully because wrong data association can lead to divergence.
In fact there exist two different definitions of the SLAM problem ([HFM15]). The first only seeks to recover the current pose x_t and is formulated as

$$p(x_t, m) := P(x_t, m|y_{0:t}, u_{0:t-1}). \tag{2.5}$$

This description is known as online SLAM.
The other definition, which is called full SLAM, estimates the whole state trajectory $x_{0:t}$ as

$$p(x_{0:t}, m) := P(x_{0:t}, m|y_{0:t}, u_{0:t-1}). \tag{2.6}$$

Both (2.5) and (2.6) may seem impossible to solve at first. But since the solution of the SLAM problem is very significant for mobile robotics, it is a well-studied problem and there where several methods developed to solve it.
These methods will be illustrated in the following sections.

2.2 Bayes Filter as Foundation to Solve the SLAM Problem

Most approaches to solve the SLAM problem are build on the Bayes filter. The Bayes filter is a probabilistic filter based on Bayesian probability used for recursive state estimation ([AMG02]). It applies the Bayesian rule, the Markov assumption and the law of total probability to describe the belief over the state x_t depending on previous observations $y_{0:t}$ and control commands $u_{0:t-1}$ as

$$p(x_t) = P(x_t|y_{0:t}, u_{0:t-1}) \propto \; P(y_t|x_t) \int_{x_{t-1}} P(x_t|x_{t-1}, u_{t-1}) \, p(x_{t-1}) \, dx_{t-1}. \tag{2.7}$$

The Bayes filter can be rewritten as a two step process:

$$\overline{p}(x_t) = \int_{x_{t-1}} P(x_t|x_{t-1}, u_{t-1}) \, p(x_{t-1}) \, dx_{t-1} \tag{2.8}$$

$$p(x_t) \propto \; P(y_t|x_t) \, \overline{p}(x_t). \tag{2.9}$$

In equation (2.8) the prediction step is described. It uses the motion model $P(x_t|x_{t-1}, u_{t-1})$ to estimate a belief for the state x_t. Equation (2.9) uses the observation model $P(y_t|x_t)$ to correct this assumption. Hence this second step is called correction step.

2.3 Approaches to Solve the SLAM Problem

Knowing the SLAM problem cannot be solved deterministically, SLAM methods aim to solve it at least with a high accuracy. This means that they want to get a fast convergence of their belief for the current state x_t (or the state trajectory $x_{0:t}$) and each spot of the map m. A further demand on SLAM methods is consistency and since the motion and transition model are usually non-linear, SLAM methods need to be able to deal with non-linearities.

As described before, SLAM is not only a problem of state estimation like mentioned in equation (2.4) which could be easily solved with a Bayes filter. The probabilities in the equations (2.5) and (2.6), which describe the SLAM problem, represent the belief over both, state and map, and as consequence a basic Bayes filter is not sufficient to solve the SLAM problem. State estimation and map building have to be somehow tied together to find a solution. In order to do so, the main approaches use extended realizations of the Bayes filter.

[GSB05] classifies the different approaches to solve the SLAM problem according

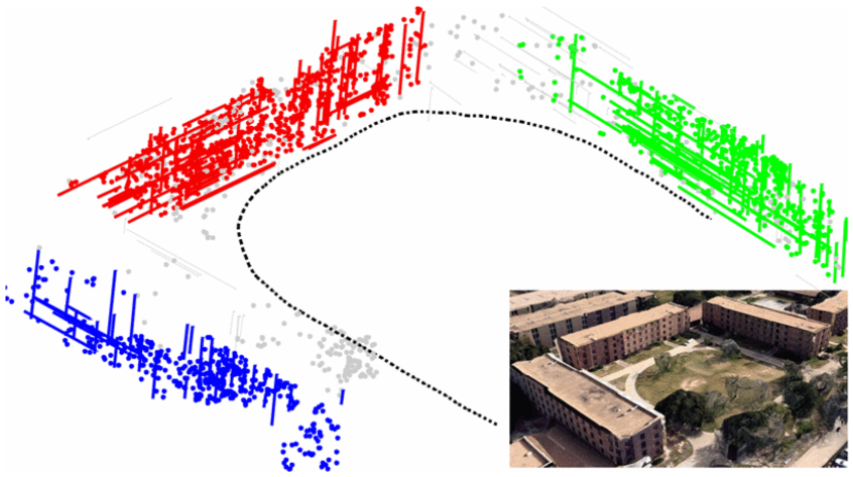


Figure 2.1: A feature-based map using points and lines as landmarks and a google earth™ view of the same site. (©2013 IEEE)

to the type of map they create and the underlying estimation technique.
A popular possibility to represent the mapping result are feature-based maps. They are based on a set of landmarks θ to define the map

$$m = \{\theta_1, \theta_2, ..., \theta_N\}. \tag{2.10}$$

Most commonly, points are used as landmarks, but other features like lines or planes or possible too ([LSY14]). A point-landmark is only defined by its position. For a 2-D feature-map using points as landmarks, each landmark θ is a two-dimensional vector.
Feature-based maps rely on predefined landmarks, which requires them to already know something about the structure of the environment in advance. A beneficial feature of this map type is its compactness. In figure 2.1 an example of a feature-based map is shown.
Another common map type are the grid-based maps. They are based on a rigid grid of cells. These cells are either free if they represent free space or occupied with they represent obstacles. If the state of a certain cell is not certainly known, it can also be considered as unknown. This can be seen in figure 2.2 which shows a grid-based map. The color of the pixels represent the state of the corresponding cell. White pixels are free space, black pixels are occupied and the state of grey pixels is unknown.
Grid maps can describe arbitrary features and provide a detailed representation on the environment. The drawbacks of grid-based mapping approaches are a high computing effort and that they require a huge amount of memory.

As far as the estimation technique is concerned, two main concepts developed.
One of these mainly used techniques is based on a Extended Kalman filter (EKF). The

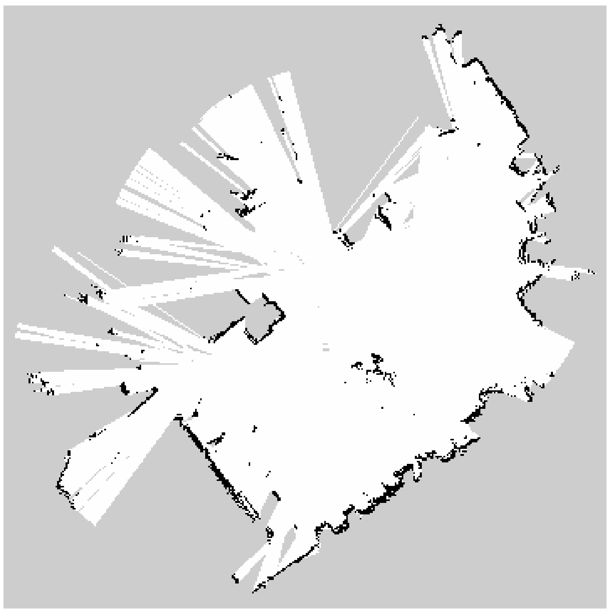


Figure 2.2: Grid-based map of the MLR lab.

Kalman filter is a Bayes filter assuming a Gaussian state and linear Gaussian observation and transition models, which means it is parametric and uni-modal. The EKF is an extension of the Kalman filter for non-linear systems. It linearizes the models about an estimate of the current mean and covariance.
EKF-SLAM methods apply this filter to solve the online SLAM problem as they compute a joint belief over the pose and a map of landmarks. This belief $p_t(x_t, \theta_{1:N})$ is described by a Gaussian. This Gaussian is $3 + 2N$-dimensional as it consists of the three-dimensional x_t and N two-dimensional landmarks θ.
EKF-SLAM methods are feature-based, but nevertheless have a high computational complexity of the order of $O(N^2)$ per step with N beeing the number landmarks ([PTN08]). Also the memory consumption is of the order of $O(N^2)$. Another disadvantage is that EKF-SLAM methods have difficulties with convergence and consistency if their assumptions on motion and transition model are violated ([GSB05]).

The other concept is more robust to non-linearities. It is called Particle SLAM and is based on particle filters. Particle filters are non-parametric recursive Bayes filters. This means that particle filters can deal with arbitrary probability distributions while the Kalman filter and its variants can only handle Gaussian distributions. In order to do this they use a weighted set χ of M particles to describe the distribution with

$$\chi = \{\langle x^i, w^i \rangle\}_{i=1,\dots,M}. \tag{2.11}$$

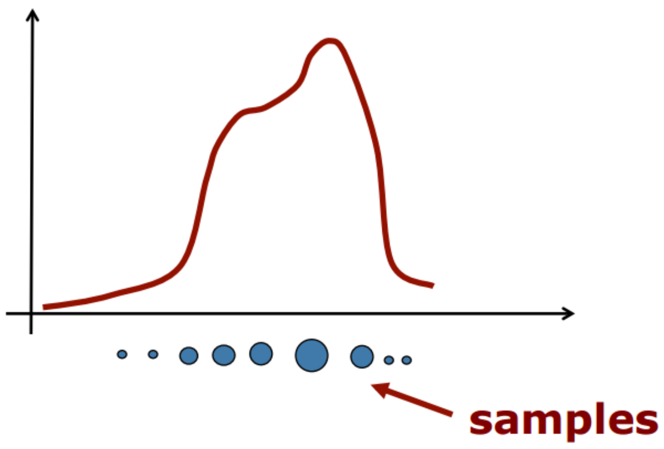


Figure 2.3: An arbitrary probability distribution and its particle representation[2].

These particles are composed of a state hypothesis x^i and a referring importance weight w^i. The importance weights are normalized:

$$\sum_{i=1}^{N} w^i = 1. \tag{2.12}$$

This set of particles represents the posterior

$$p(x) = \sum_{i=1}^{M} w^i \delta(x - x^i) \tag{2.13}$$

using the Dirac delta function δ.

Figure 2.3 illustrates this. Each circle represents a particle and its size corresponds to its importance weight. The graph shows the equivalent distribution $p(x)$.
The prediction step of particle filters is done using a proposal distribution π to sample the particles in the form

$$x_t^i \sim \pi(x_t | y_{0:t}, u_{0:t-1}). \tag{2.14}$$

In the correction step ratio of target and proposal distribution is computed to set the importance weights

$$w_t^i = \frac{p(x_t^i | y_{0:t}, u_{0:t-1})}{\pi(x_t^i | y_{0:t}, u_{0:t-1})}. \tag{2.15}$$

Another step of particle filters is resampling. This means samples with a low importance weight $w^{[i]}$ will get replaced by more likely ones. This is important since the number of

[2]Source: http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam09-particle-filter.pdf

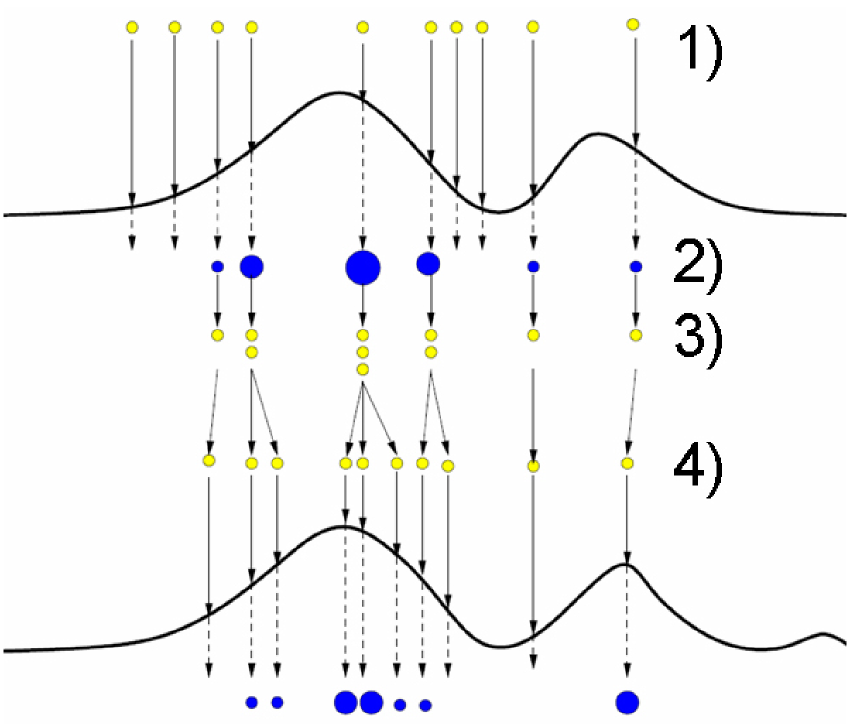


Figure 2.4: A graphical illustration of Monte Carlo localization[3].

particles is limited.
Monte Carlo Localization (MCL), like shown in Figure 2.4, uses the concept of particle filters for state estimation. In this case each particle represents a pose hypothesis. After each movement of the robot the particles get sampled by using the motion model distribution as proposal. In the figure the steps 1) and 4) illustrate the sampling. In the correction step the new observation is used to weight the particles, which can be seen in step 2) of the figure. To this end the observation model is applied as target distribution. Step 3) of the figure shows the resampling of the particles after which the whole progress will start again with a new robot movement. After several iterations the particles will gather on more likely poses of the robot.

Solving SLAM with particle filters works just like Monte Carlo Localization, but particles are not only a weighted state hypothesis anymore, but an estimate over the whole situation including state and map. In a landmark-based SLAM this can be described as

$$x^i = (x_{0:t}, \theta_1, ..., \theta_N)^T. \tag{2.16}$$

[3]Source: http://lia.deis.unibo.it/research/SOMA/MobilityPrediction/images/PFpiccolo.jpg

This high-dimensional state hypothesis certainly requires a much higher number of samples, since the number of possible state hypothesis increases extremely. In general it can be said that the more samples are used, the better are the results. But as time complexity is linearly dependent on the number of particles M, a compromise between speed an acurrancy has to be found.
A popular concept to deal with this conflict is FastSLAM. The core idea of FastSLAM is to exploit dependencies between the different dimensions of the state space $(x_{0:t}, \theta_1, ..., \theta_N)$. Therefore each sample is defined as a path hypothesis. And for each path hypothesis a individual map is computed. This can de done using Rao-Blackwellization ([GSB07]):

$$p(x_{0:t}, \theta_{1:N}|y_{0:t}, u_{0:t-1}) = p(x_{0:t}|y_{0:t}, u_{0:t-1})\ p(\theta_{1:N}|x_{0:t}, y_{0:t}) \tag{2.17}$$

$$= p(x_{0:t}|y_{0:t}, u_{0:t-1}) \prod_{i=1}^{N} p(\theta_i|x_{0:t}, y_{0:t}). \tag{2.18}$$

The second conversion is valid since landmarks are conditionally independent given the robots path if each particle has its own map.
Now Rao-Blackwellization divided the problem in terms that already can be handled. The term $p(x_{0:t}|y_{0:t}, u_{0:t-1})$ is equivalent to Monte Carlo localization. Each belief $p(\theta_i|x_{0:t}, y_{0:t})$ can be computed with a two-dimensional EKF. For each particle the problem is now described as a MCL and N EKFs. The benefit of FastSLAM is that it splits the high dimensional problem in many low-dimensional problems that can be solved efficiently.
Even though with FastSLAM a feature-based particle SLAM method was presented, particle SLAM methods can also be used to create a grid-based map.

3 Path Planning and Optimization

3.1 The Different Concepts of Path Planning

As already stated in the motivation, finding a collision-free path is a basic problem in mobile robotics. We can describe this as the task to find a collision-free trajectory $x \in \mathbb{R}^{T \times n}$ consisting of T robot configurations $x_t \in \mathbb{R}^n$ connecting the current configuration with the desired target configuration. Moreover this trajectory should be short, smooth and respect possible differential constraints of the robot.
Since this is a very important problem a lot of algorithms and methods do so. These algorithms can be divided in two main categories that can be seen in figure 3.1 and will be introduced and compared in the hereafter.

A first option is to use a path finding algorithm in order to find some valid trajectory x connecting start and goal.
There exist many different types of these algorithms. [LaV06] gives an overview over the most important ones. They all somehow need to discretize the environment to compute

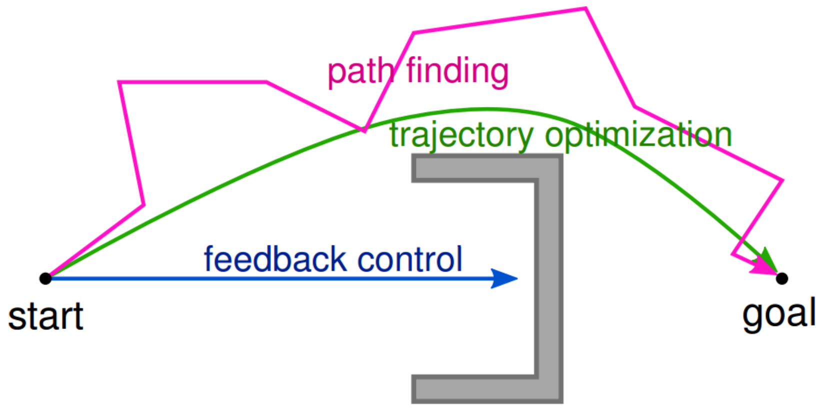


Figure 3.1: Different concepts of path planning[1].

[1]Source: https://ipvs.informatik.uni-stuttgart.de/mlr/marc/teaching/14-Robotics/14-Robotics-script.pdf

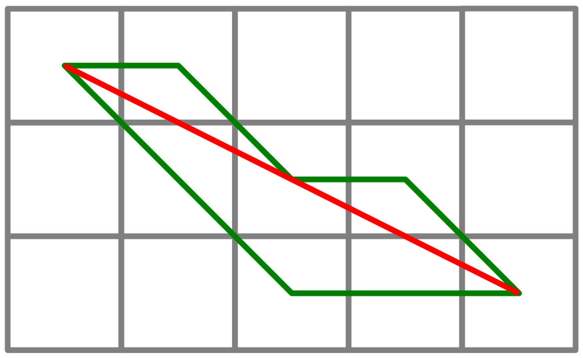


Figure 3.2: Grid-based paths in relation to the desired path

a path. Some of them discretize by using random samples and try to connect them to a collision-free path (e.g. RRT, probabilistic road maps, etc.).
If a grid map, which already is a discretization of the environment, is given, also basic search algorithms like A* or Dijkstra can be directly applied in order to find a path.
Search algorithms are pretty efficient for path finding in discrete domains and can easily find the shortest connection between a start cell and a goal cell. But since the grid has a certain resolution the found path is normally not smooth and not the desired path.
This can be seen in figure 3.2. The green paths are possible results of a search algorithm.
The red path is the desired one.
Moreover, this kind of path finding simplifies the path finding problem to a two-dimensional problem since only positions are connected and the orientation of the robot in each step of the path is not respected.

The other possibility to compute a path is trajectory optimization. The idea of trajectory optimization is to define a cost function f in a sense that the optimal path x is described as

$$x = \min_x f(x). \tag{3.1}$$

That means trajectory optimization needs a well defined cost function is order to return the sought optimum. Usually the path optimization problem is not only defined by the cost function, since in most cases several constraints have to be respected. These constraints can be the differential constraints of non-holonomic systems or they can be used to avoid obstacles. Constrains are usually defined as inequalities or equalities.
Including these constraints the optimization problem for the path x can be described with

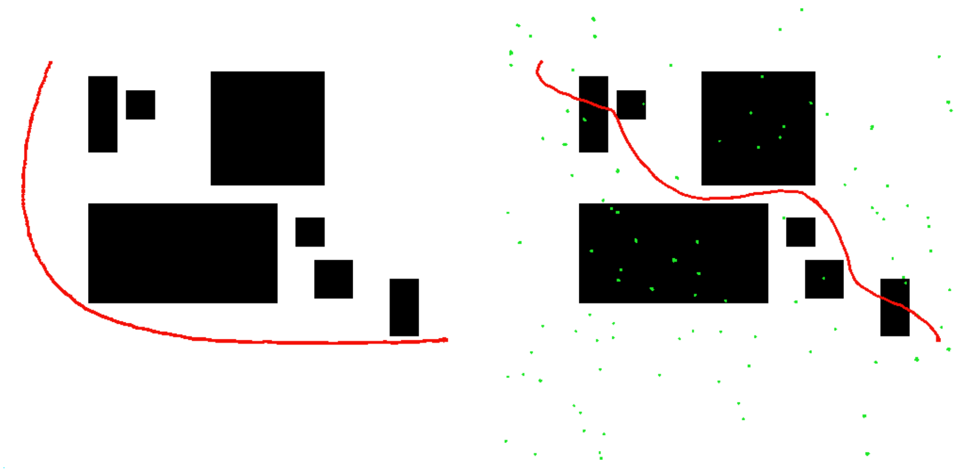
$$\min_x f(x) \;\text{ s.t. }\; g(x) \leq 0,\; h(x) = 0. \tag{3.2}$$

This notation condenses all demands on the path in the following three functions which are

- the scalar cost function $f : \mathbb{R}^{T \times n} \to \mathbb{R}$,
- the function $g : \mathbb{R}^{T \times n} \to \mathbb{R}^{d_g}$ defining d_g inequality constraint functions,
- the function $h : \mathbb{R}^{T \times n} \to \mathbb{R}^{d_h}$ defining d_h equality constraint functions.

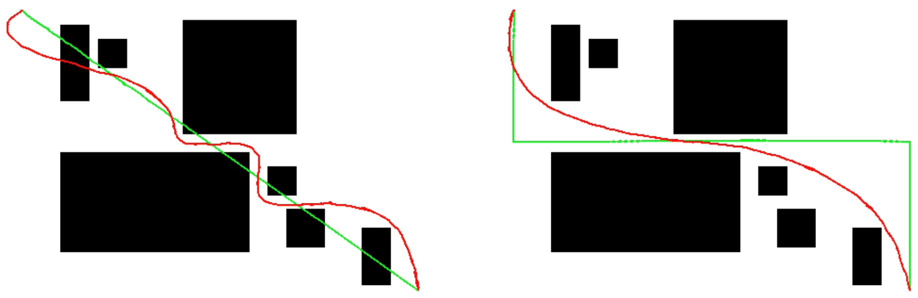
After the definition of the problem a optimization method is applied. This method will optimize a predefined path according to the defined costs and constraints.
A benefit of trajectory optimization is that it can be directly applied to the continuous configurations and needs no discretization. Therefore it is possible to find the optimal path if the optimization problem is well defined. Another advantage is that is easy to integrate the initial and desired orientation in the optimization problem with leads to a path consisting of poses connecting start and goal.

An important point is that it is not either a finding algorithm or trajectory optimization. If anything it is in the most cases very reasonable to combine both.
Like already mentioned before, path finding algorithms return a discrete path with is not smooth and usually does not match the desired optimal path. Because of this it is useful to apply trajectory optimization afterwards in order to smoothen the path.
On the other hand it is not meaningful to only use trajectory optimization. This can be seen in figure 3.3. The four examples show each a optimized trajectory (red) computed with the same constrains, the same method and the same parameters. However the results are quite different. That's because the predefined paths given to the optimizer (green) are different.
In figure 3.3a the optimizer has a no predefined path. It tries to optimize starting with a array of zeros. It can compute a collision-free path, but this for sure is not the shortest path possible.
The optimizer in 3.3b works with a random sample of poses as predefined path. It fails to find a collision-free path.
In a third example shown in figure 3.3c a straight line is predefined as path, but the optimizer returns again a path including collisions.
Finally a pre-calculated collision-free path is committed to the optimizer in example 3.3d. With this input the optimizer manages to find a smooth collision-free path that skirts the obstacles in the same way as the handed path.
As seen in this example it is indispensable to have a good predefined path as input for the optimizer. If the optimizer gets an input that skirts the obstacles in the best way, it will find the global optimum. With other path inputs it is not sure that the optimizer will find the global optimum or even a collision-free path. The reason for not finding the global optimum normally is the step size of the optimization algorithm. If it is not large enough only a local optimum is found. If the path includes collisions, it is usually because the predefined path is cutting obstacles in the middle. In this case the gradient can not say in which direction to avoid the obstacle and no optimum is found.



(a) no input

(b) random



(c) straight line

(d) pre-calculated path

Figure 3.3: Path optimization with different inputs

Another advantage of a good predefined path is that the smaller the difference between the input and the optimum is, the faster the optimization method will reach convergence. That means less iterations and computing power are required to find the optimal path.

3.2 The Dijkstra Algorithm

The Dijkstra algorithm, like described in [CLR00], is a systematic search algorithm that finds the shortest path for each node in a weighted, directed graph $G = (V, E)$ to a given

start node s.
Firstly the algorithm initializes the distance d to each node V of the graph as infinite expect for the start node s who clearly has the distance 0. Besides every predecessor π is initialized as undefined. The algorithm maintains a set S of nodes whose weights for the shortest path to the source s have been already determined and a priority queue Q. While S is empty at the beginning, the queue initially contains all nodes of the graph.

Algorithm 3.1 INITIALIZE-DIJKSTRA(G,s)

```
1: for each node $v \in V[G]$ do
2:     $d[v] \leftarrow \infty$
3:     $\pi[v] \leftarrow$ undefined
4: $d[v] \leftarrow 0$
5: $S \leftarrow \emptyset$
6: $Q \leftarrow V[G]$
```

After the initialization the Dijkstra algorithm always extracts the element with the shortest distance from the start node from Q. It adds this node u to S. For all adjacent nodes of u the distance is updated if it is shorter than before. In this case this node's predecessor will be set as u. $w(u, v)$ is the length of the edge between the nodes u and v.

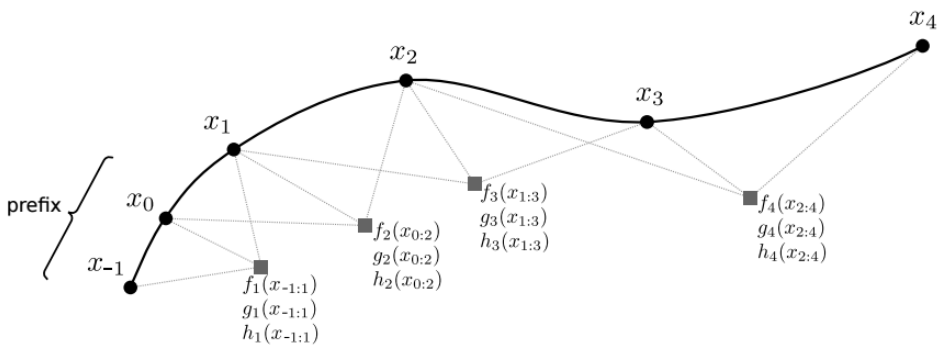
Algorithm 3.2 Dijkstra algorithm

```
1: INITIALIZE-DIJKSTRA$(G, s)$
2: while $Q \neq \emptyset$ do
3:     $u \leftarrow$ EXTRACT-MIN$(Q)$
4:     $S \leftarrow S \cup \{u\}$
5:     for each node $v \in Adj[u]$ do
6:         if $d[v] > d[u] + w(u, v)$ then
7:             $d[v] \leftarrow d[u] + w(u, v)$
8:             $\pi[v] \leftarrow u$
```

After the algorithm finished the shortest path from the start node s to a arbitrary node a is the chain of predecessors beginning with a till s is reached.
In order to use this algorithm to find a path from the start node to a particular node g the algorithm can be terminated after g gets extracted from Q. If this happens on has

$$d[g] \leq d[q] \text{ with } q \in Q \setminus \{g\}.$$

Hence every later extracted u has a longer distance than q and $d[q]$ and $\pi[v]$ will not get updated again.
The running time of the Dijkstra algorithm in standard implementation is $O(V^2)$



x_4
x_2
x_3
x_1
prefix
x_0
x_{-1}
$f_1(x_{-1:1})$ $g_1(x_{-1:1})$ $h_1(x_{-1:1})$
$f_2(x_{0:2})$ $g_2(x_{0:2})$ $h_2(x_{0:2})$
$f_3(x_{1:3})$ $g_3(x_{1:3})$ $h_3(x_{1:3})$
$f_4(x_{2:4})$ $g_4(x_{2:4})$ $h_4(x_{2:4})$

Figure 3.4: Illustration of the formulation used for k-order Markov optimization[2].

3.3 k-Order Markov Optimization (KOMO)

K-order Markov optimization (KOMO) as described in [Tou14] is a possibility to solve the path optimization problem in equation 3.1. Unlike other path optimization methods that are based on a phase space formulation, KOMO uses its specific problem formulation based on configuration space.

This formulation is based on the k-order Markov assumption ([Tou16]). It is assumed that

$$f(x) = \sum_{t=1}^{T} f_t(x_{t-k:t}),\ g(x) = \otimes_{t=1}^{T}(g_t(x_{t-k:t})),\ h(x) = \otimes_{t=1}^{T}(h_t(x_{t-k:t})) \tag{3.3}$$

for a given prefix $x_{1-k:0}$. In this assumption each f_t is scalar, g_t is d_{gt}-dimensional and h_t is d_{ht}-dimensional.
The notation $x_{t-k:t}$ represents the tuple $(x_{t-k}, x_{t-k+1}, ..., x_t)$. The prefix $x_{1-k:0}$ contains the robots configuration before the path. The notation $\otimes$ means that the constraint functions g_t and h_t of each time step t are stacked up full constraint functions g and h for the whole path.

Using assumption 3.3 the problem stated in equation 3.1 can be rewritten to

$$\min_x \sum_{t=1}^{T} f_t(x_{t-k:t}) \text{ s.t. } \forall_{t=1}^{T} : g_t(x_{t-k:t}) \leq 0,\ \forall_{t=1}^{T} : h_t(x_{t-k:t}) = 0. \tag{3.4}$$

[2]Source: https://ipvs.informatik.uni-stuttgart.de/mlr/papers/16-toussaint-Newton.pdf

The Markov assumption implies a structure where $k+1$ consecutive variables are used to define the functions $f_t(x_{t-k:t})$, $g_t(x_{t-k:t})$ and $h_t(x_{t-k:t})$. This can be seen figure 3.4. All features at time t can be gathered in $\phi_t(x_{t-k:t}) \in \mathbb{R}^{1+d_{gt}+d_{ht}}$ with

$$\phi_t(x_{t-k:t}) = \begin{pmatrix} f_t(x_{t-k:t}) \\ g_t(x_{t-k:t}) \\ h_t(x_{t-k:t}). \end{pmatrix} \tag{3.5}$$

The k-order cost vectors $f_t(x_{t-k:t})$ can be used to describe arbitrary costs. Depending on the order these costs are related to positions $(k=0)$, velocities $(k=1)$, accelerations $(k=2)$ or jerks $(k=3)$. In the same way arbitrary inequality and equality constrains g_t and h_t can defined in the different orders.

The in [Tou14] presented KOMO framework was devolved to exploit this formulation. The idea of this framework is to split between definition of the optimization problem and the actual optimization using Newton methods.
In a first step the optimization problem can be defined by the features $\phi_t(x_{t-k:t})$. This can be done in a semantic way referring to the kinematics of the robot.
The framework converts this definition into a general and abstract form exploiting the chain structure of the problem by using fitting matrix packings.
The result of this conversion is a universal formulation including the (approximate) Hessian and other terms needed for Newton methods.
Moreover the framework provides implementations of five different Newton methods to actually solve such a general formulation. These include Gauss-Newton, Augmented Lagrangian and a any-time version of Augmented Lagrangian.

4 The TurtleBot - A Functional and Small Robot

4.1 What is a TurtleBot?

The TurtleBot[1] is a small robot with open-source software. The TurtleBot is efficient, inexpensive and was designed to work in fields like localization and mobility. Its software runs on the Robot Operating System (ROS). ROS is a open-source collection of frameworks for personal robots that is used by many universities and also companies like Intel or Bosch.

Version 2 of the TurtleBot works with a Kobuki base and the Asus Xtion Pro as shown in figure 4.1. The Kobuki base has four wheels. Two of them can actively be controlled as each of it as its own motor. Two of them are passive support wheels. They allow the TurtleBot to drive with a translational velocity up to $0,7\ m/s$ and a rotational velocity up to $180\ deg/s$.
The TurtleBot is non-holonomic as it can only move in direction of its orientation. Even though the TurtleBot is non-holonomic, the two separately controllable wheels make the robot nimble as they allow it to turn on place.
The robots odometry has a accuracy of 2578.33 ticks per wheel revolution which equals 11.7 ticks per millimetre. As a consequence a precise motion model can de derived.
The Asus Xtion Pro is a modern developer solution for motion-sensing. It has infrared sensors, adaptive depth detection technology and color image sensors. Thus it is a useful tool for problems in the fields of mapping and visualization and makes precise SLAM mapping possible.
The software is executed by any ROS-compatible netbook that can be connected with base and camera.

As said before the Robot Operating System (ROS) is a big collection of software frameworks. It provides tools, libraries and conventions for services like hardware abstraction,

[1]More about the TurtleBot can be learned at: http://www.turtlebot.com/

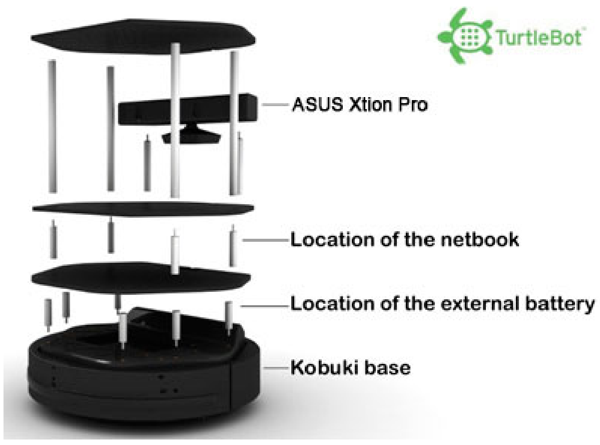


Figure 4.1: Build of a TurtleBot 2[2].

commonly used functionalities, package management and message-passing interfaces. ROS is based on nodes that communicate via topics. Launch files can be written to easily launch multiple nodes from one ore more packages. Besides these launch files can be used to set parameters for the used packages.
Since the TurtleBot is mainly developed for localization and mobility, it is compatible with many ROS packages in these fields , for example the gmapping package, which can be used for grid-based SLAM and will be explained in section 4.2. Moreover the move_base package is provided, which is the centerpiece of navigating the TurtleBot along paths. It can be used to plan and publish paths and will be described later in section 4.3.

With its precise odometry and sensors and its manoeuvrability the TurtleBot is suitable to create precise SLAM-based maps and follow computed paths. Since many frameworks are already implemented in its open-source software and can be reused, the programming effort is reduced which is another benefit. Moreover the Robot Operating System allows to transfer implemented functionalities easily to other personal robots. All these points are reasons why the TurtleBot was chosen to examine the topic of this thesis.

4.2 Gmapping - A Grid-based SLAM Method

The code used to create a SLAM-based map with the TurtleBot is OpenSlam's gmapping[3]. It is already wrapped for ROS in the ROS gmapping package. Gmapping uses a particle

[2]Source: http://www.generationrobots.com/img/products/clearpath/turtlebot-2-EN.jpg
[3]http://openslam.org/gmapping.html

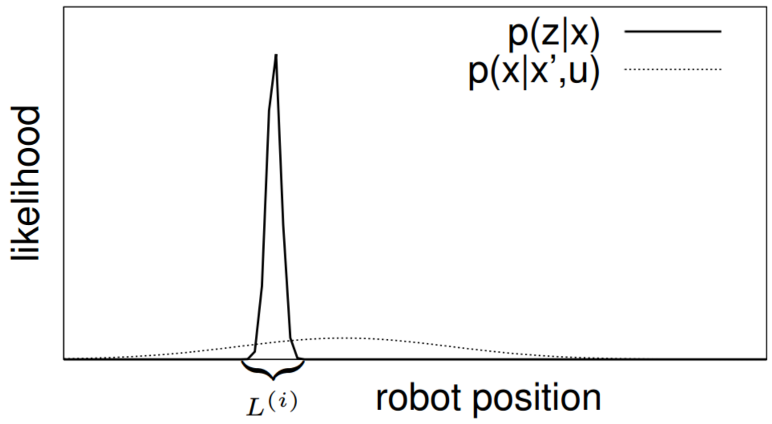


Figure 4.2: The two components of the standard proposal distribution (©2007 IEEE).

SLAM like introduced in chapter 2. In particular it works with a highly efficient Rao-Blackwellized particle SLAM method. This method creates a grid-based map from laser and pose data collected by the TurtleBot.
This section gives a basic introduction to the algorithm behind gmapping. It is based on [GSB05] and [GSB07], where more details about the underlying algorithm can be found.

The idea of the gmapping particle SLAM method is similar to the idea of FastSLAM. But now Rao-Blackwellization is used to compute a grid-based map m. For this purpose

$$p(x_{0:t}, m|y_{0:t}, u_{0:t-1}) = p(x_{0:t}|m, u_{0:t-1})\ p(m|x_{0:t}, y_{0:t}) \tag{4.1}$$

has to be computed. The SLAM posterior gets divided in a path and a map posterior. Each particle is a possible state trajectory $x_{0:t}$ of the robot and each particle maintains its own map. This means Monte Carlo Localization can be applied to compute the term $p(x_{0:t}|m, u_{0:t-1})$. The term $p(m|x_{0:t}, y_{0:t})$ describes mapping with known poses.
But since a grid map has a huge amount of memory, it is even more important to reduce the number of samples. In order to do this a better proposal that allows more precise sampling is needed.
Various methods to compute a better proposal can be found in literature (for example in [Dou98] and [H+03]). Gmapping applies two concepts. The starting point of the first is the proposal distribution. The second concepts attempts to adaptively determine when to resample.
Like already stated in chapter 2, particle filters need a proposal distribution π to sample the particles. π should approximate the true distribution $p(x_t|y_{0:t}, u_{0:t-1})$.
[Dou98] stated the optimal choice for the proposal distribution respecting the variance of particle weights and using the Markov assumption as

$$p(x_t|m^i_{t-1}, x^i_{t-1}, z_t, u_t) = \frac{p(z_t|m^i_{t-1}, x_t)p(x_t|x^i_{t-1}, u_t)}{\int p(z_t|m^i_{t-1}, x')p(x'|x^i_{t-1}, u_t)dx'}. \tag{4.2}$$

Many particle filters use the motion model $p(x_t|x_{t-1}, u_{t-1})$ in the proposal distribution. For a robot working with a laser sensor this often is not the optimal choice. The accuracy of the laser range finder will lead to a extremely peaked likelihood function. For this reason the term $p(z_t|m_t, x_t)$ will dominate the product (4.2) in the area of its peak. This can be seen in figure 4.2. Therefore the gmapping implementation approximates $p(x_t|x^i_{t-1}, u_{t-1})$ by constant k within the intervall L^i with

$$L^i = \{x|p(y_t|m^i_{t-1}, x) > \epsilon\}. \tag{4.3}$$

Under this approximation equation 4.2 can be written as

$$p(x_t|m^i_{t-1}, x^i_{t-1}, z_t, u_t) \approxeq \frac{p(z_t|m^i_{t-1}, x_t)}{\int_{x' \in L^i} p(z_t|m^i_{t-1}, x')dx'}. \tag{4.4}$$

The importance of resampling due to a finite number of particles was already stated earlier. But resampling at each iteration could also lead to depletion of good samples and downgrade the proposal. On this account it is important to find a criterion when to resample. To estimate how well the current particle set represents the true posterior the effective number of particles M_{eff} was introduced by [Liu96]. This value is described by

$$M_{eff} = \frac{1}{\sum\limits_{i=1}^{M}(w^i)^2}. \tag{4.5}$$

The more equal the importance weights are the larger this value becomes. This implies that a high value M_{eff} means that the samples approximate the true distribution well. For a worse approximation the variance of the different importance weights leads to a smaller value M_{eff}.
Given M_{eff} and the total number of particles M gmapping only performs a resampling step if

$$M_{eff} < M/2. \tag{4.6}$$

Using this threshold strongly reduces the risk of replacing good particles, since resampling only takes place, when it is really needed.
Combing both described approaches leads to a highly efficient Rao-Blackwellized SLAM technique, which requires one order of magnitude less particles than other grid-based approaches of that type.

4.3 Planning and Execution of Paths on the TurtleBot

As said before the TurtleBot already provides an environment for planning and execution of paths. Figure 4.3 shows all the nodes of this environment and their connections. The

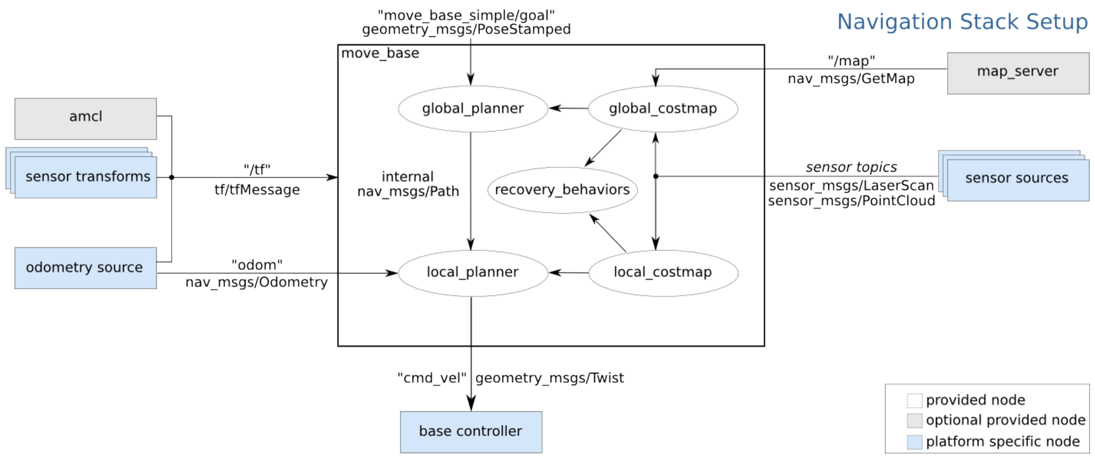


Figure 4.3: High-level view of the move_base node[4].

nodes on the left provide information about the robots position at every time. The nodes on the right side of the figure supply the map and the sensor input.
However, the centrepiece of this navigation environment is the move_base package, where all this data is processed.
Given a goal in the TurtleBot's environment the move_base ROS package provides an implementation that attempts to reach the target with the mobile robot. The move_base node links a global and a local planner and maintains a global and a local costmap in order to accomplish this goal. ROS provides a default implementation for global and local planner. Moreover, there exist interfaces to implement own planners and register them as plugins in order to use them in the move_base package. The benefits of this two-layered architecture consisting of global and local planner will expanded in section 5.5.
The path resulting of the move_base package is published to the base_controller node, which is shown on the bottom of the figure. The base_controller node can execute this path as it has direct access to the TurtleBot's motors.
Whereas the grey and white nodes are equal for all ROS-compatible robots, the blue nodes are specific for the TurtleBot and its properties.
Now if the TurtleBot should find and follow a path from its position to the goal, only a file launching all the nodes from figure 4.3 is required.

In order to apply own methods and algorithms for path planning and optimization, it is only necessary to implement them as new global or local planner plugins. All the other nodes can be reused.

[4]Source: http://wiki.ros.org/move_base?action=AttachFile&do=get&target=overview_tf.png

5 From an Unknown Environment to a Desired Path

5.1 Properties of SLAM Maps

Let us assume the following situation: A mobile robot, in this example the TurtleBot, is placed in a unknown environment. After some grid-based SLAM mapping with a implementation like gmapping introduced in section 4.2, it has an rough idea of at least most parts of the surrounding area. Now the robot shall use this information to find the shortest collision-free path between current configuration and a desired target pose. In order to do so the following properties of grid-based SLAM maps have to be respected. This chapter aims to describe a approach to do using the background information about SLAM, path planning and path optimization and the TurtleBot, which were the content of the previous chapter.

If we recap chapter 2 we know that all popular approaches to solve the SLAM problem have something in common: they work with probabilities. They usually get fast convergence about the landmark position or the state of a grid cell, but nevertheless the maps they create only provides a belief over the environment. This has to be respected in path planning, since the robot should not find a path that is likely collision-free, but one that really is. One the other hand not every spot which is not a 100% free can be avoided. Path planning on SLAM-based maps has to somehow find a compromise in order to deal with this problem.
Another thing to be respected when working with grid-based SLAM maps like in this TurtleBot example is their resolution. The resolution has to be in relation to the accuracy of the motion and transition model. It makes no sense to estimate a map that is more accurate then these models. For this reason the resolution is confined. Another reason for the confined resolution of the map is the computing effort. Since the map was created with a particle SLAM method, it was derived from M map samples. If a higher resolution is favoured, all these map samples need a higher resolution with creates a much higher computing effort. Descriptively spoken, if for example a map with 200 pixels instead of 100 pixels is required, we do not need do compute the state of 100 more pixels, but of $M \times 100$ more pixels. With a default value of $M = 30$ there are 3000 more

pixels that need to be estimated.

For sure there are a lot of ways to deal with these properties. This chapter will show a simple way of how to use a grid-based SLAM map in path planning. In order to do so it will use the Dijkstra-Algorithm and the KOMO framework which were both introcuded in chapter 3. They will be combined and implemented as a global planner plugin for the move_base package (cf. section 4.3).

5.2 Transformation of the Map

The most important step of this approach is to get rid of the probabilities and convert the SLAM-based map into an discrete map. For this purpose two thresholds are defined. Occupancy probabilities above a certain value means these grid cells of the map are occupied. Equally an occupancy probability below a certain value signifies a free spot. Cells with a value between the thresholds are considered unknown. For the gmapping SLAM map on the TurtleBot the thresholds 0.65 and 0.196 were used.
As a result a map with three states accrues: free, occupied and unknown.
Because a path is computed for the center of a robot it is indispensable to know the size of the robot and the distance to the next obstacle or unknown spot for every part of the map. Therefore a costmap with values between 0 and 255 for each cell is derived from the map.
A value of 255 labels the unknown and a value of 254 the occupied cells of the map. Values between 253 and 0 indicate the distance to the next occupied cell. Whereas a cell with the value 253 is a direct neighbour of an occupied cell, a cell with a value of 0 is far away from such a cell. In fact, this scale is designed in way that values below 128 indicate cells that have a greater distance to the obstacle then the radius of the TurtleBot. Meaning that if the center of the robot is exactly on such a cell, it is collision-free. However, if the center of the robot is positioned over a cell with a value of 128 or higher, some part of the robot will collide with an obstacle.
In another transformation the map gets linear rescaled in the sense that 255 corresponds to a new value of 1 and 0 corresponds to a new value of -1. This rescaling step is not obligatory, but it makes things clearer since values ≥ 0 are now indicating cells that are too close to a obstacle to be passed by the center of the robot, whereas value <0 can be included in paths.
A summary of the map transformations described in this chapter is shown in 5.1.

The result of the transformations is a global costmap, which the global planner can use to compute paths. The global planner can include every cell with a value below 0 in

0.65 < op	→ occupied	→ 254	→ 0.9
0.196 < op < 0.65	→ unknown	→ 255	→ 1
op < 0.196	→ free	→ 0-253	→ -1 to 0.9

Figure 5.1: Summary of the map transformations. op is short for occupancy probability.

paths. In the following these cells will be designated as (collision)-free. Cells with a value of 0 and higher will be referred to as occupied or cells in collision range.

5.3 Applying the Dijkstra Algorithm

In the map resulting of the just mentioned transformations the path finding problem can be described as a shortest-path graph problem. For this purpose every collision-free cell of the map is defined as a node of a graph. Cells in collision range are not included in this graph. Hence, every cell has a maximum of eight adjacent cells. The transition costs $u(v,w)$ are defined as 1 for cells that share a edge and $\sqrt{2}$ for cells that share a corner which is equivalent to the distances.
The cell containing the start coordinates is set as start node s and the cell containing as the goal coordinates as goal node g.
Given this concrete problem definition the Dijkstra alogrithm can be directly applied like explained in section 3.2. It returns the shortest possible grid-based connection between the starting and the goal position. This path consists of connected positions, the orientation of the robot is not included.
Due tue the resolution of the map the returned path is usually un-smooth and not really nice drivable (cf. 3.2).
In any case the path calculated with Dijkstra skirts the obstacles in the best way.

5.4 Applying k-Order Markov Optimization

Now given this coarse predefined path, we want to optimize it. Therefore newton methods for k-order Markov optimization problems like introduced in section 3.3 are used to smoothen the path. KOMO also allows us to work with the whole three-dimensional pose of the robot and can compute a appropriate orientation for every step of the path.
For this purpose we need to define the particular problem as KOMO problem. This means the dimensions have to be deterimined and the features $\phi_t(x_{t-k:t})$ have to be defined for each $t = (1, 2, ..., T-1)$. Furthermore the Jacobian of each feature has to be

given in order to apply a newton method.
For the resulting path $x \in \mathbb{R}^{T \times n}$ in the given problem it is $n = 3$ since the configuration of the Turtlebot at each time t is defined trough the 3-dimensional vector $x_t = (x_x, x_y, x_\theta)$. The number of time steps T indicates the length of the path to be optimized. It is meaningful to take the length of the path, which Dijkstra algorithm returned, since this allows to directly optimize the result of Dijkstra algorithm. For sure it would be also possible to use a higher or lower value for T. This needs to convert the predefined path to the respective number of steps. However it is not recommended to use a lower value, since due to the resolution of the map the Dijkstra algorithm already returns a coarse path. A higher value can actually make sense to get a more precise path, but usually this is not necessary.
In order to define the prefix every state x_t with $t < 0$ is set to the starting configuration of the robot. This fixes the start of the path to this configuration and states the presume that the robot is at rest.
As a next step a cost function f_t has to be defined for every time step t. In order to find a short and smooth path, which can be followed easily by the robot we want to minimize squares of accelerations. Consequently f_t at every time step t is described as

$$f_t = |(x_t + x_{t-2} - 2x_{t-1})|^2 \tag{5.1}$$

with the three-dimensional pose vector x_t to penalize accelerations in both directions and the angular acceleration.

Moreover a equality constraint is defined to ensure that the desired target pose x^* is reached. For this reason we define the equality function h_t for the last time step $t = T - 1$ as

$$h_{T-1} = (x_{T-1} - x^*) \tag{5.2}$$

with the three-dimensional vector of the desired target pose x^*.

As a last constraint a inequality is set up that keeps the robot away from obstacles.
Lets consider the map as a function $M(x_x, x_y)$ that returns values from -1 to 1 for integer inputs x_x and x_y.
With this function M a obvious definition of the inequality function g_t is

$$M(x_x(t), x_y(t)) \leq 0 \tag{5.3}$$

in every time step t.
But in fact the implementation uses in every time step t a inequality function g_t which is defined as

$$(M(x_x(t), x_y(t)) + 0.5) \leq 0. \tag{5.4}$$

The term $+0.5$ was added, to stay farer away from obstacles. This definition returned much better optimization results with regards to the resulting path and the number of

needed iterations than (5.3).
For every feature, the Jacobian corresponding is needed in order to apply newton methods. For (5.1) and (5.2) the Jacobians can be derived easily. The Jacobian of (5.4) can not be derived directly, since the map function is discrete and only returns values for integer inputs $x_x(t)$ and $x_y(t)$. For this reason bilinear interpolation was used to compute the Jacobian.

The idea of bilinear interpolation is illustrated in 5.2. In brief bilinear interpolation is interpolation in two-dimensions.
With bilinear interpolation the value $f(P)$ of a function f at the point $P(x, y)$ which is placed inside the four grid points $Q_{11}(x_1, y_1), Q_{12}(x_1, y_2), Q_{21}(x_2, y_1)$ and $Q_{22}(x_2, y_2)$ can be calculated via interpolation of $R_1(x, y_1)$ and $R_2(x, y_2)$ as

$$f(P) \approx \frac{y_2 - y}{y_2 - y_1} f(R_1) + \frac{y - y_1}{y_2 - y_1} f(R_2) \tag{5.5}$$

with R_1 and R_2 also being calculated through interpolation:

$$f(R_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \tag{5.6}$$

$$f(R_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}). \tag{5.7}$$

This concept of bilinear interpolation can be used to estimate a analytic approximation of the map. With this resulting analytic function the gradient of the constraint (5.4) can be computed and the derive the corresponding Jacobian

$$J(m) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ J_{31} & J_{32} & 0 \end{pmatrix} \tag{5.8}$$

with

$$J(31) = ((y_2 - y) * (f(Q_{21}) - f(Q_{11})) + (y - y_1) * (f(Q_{22}) - f(Q_{12}))) \tag{5.9}$$

and

$$J(32) = ((x_2 - x) * (f(Q_{12}) - f(Q_{11})) + (x - x_1) * (f(Q_{22}) - f(Q_{21}))) \tag{5.10}$$

whereby f,x and y have to replaced by M,x_x and x_y.

After defining the optimization problem with the described costs and constraints, newton methods can be applied to optimize the predefined path. The KOMO frameworks allows to easily change the optimization method and its parameters for the defined problem, until the wished result is formed.

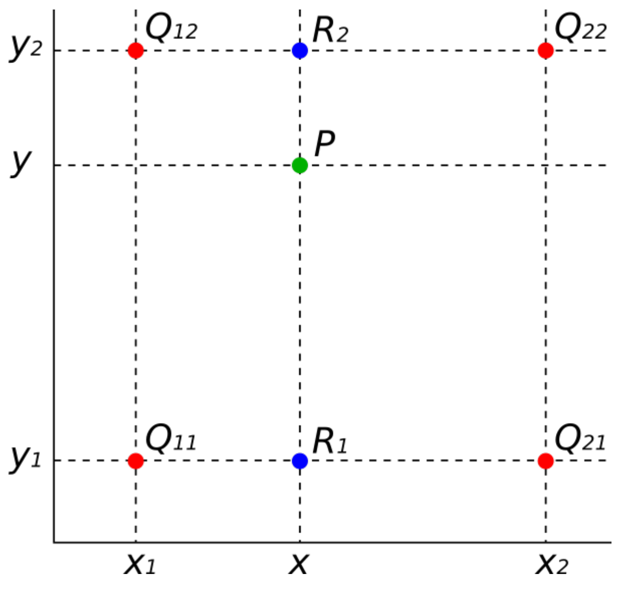


Figure 5.2: Bilinear interpolation[1].

5.5 Dealing with Dynamic Environments and Differential Constraints

If we want a truly autonomous mobile robot, it should be able to not only deal with static but also dynamic environments.
But once the map is established it is a static one. It is an image of the environment at the time the map was drawn. But the environment may have changed since due to different reasons. For instance if the environment is a room, it could have been refurnished. Moreover dynamic objects like humans or other robots could be part of the surrounding area.
Both, the Dijkstra algorithm and the KOMO framework, work with the static map which resulted of the Gmapping. Due to this reason neither the Dijkstra algorithm as path finding algorithm nor the KOMO framework responsible for the path optimization can deal with the problem since they receive no information about dynamic processes in the environment.
This could lead to a situation where a new obstacle is placed in the TurtleBot's computed path. This could for example happen if humans or other robots cross the path. As a result the path is not collision-free anymore, which is a big problem. To make sure the robot is not colliding when following such a path, the earlier mentioned move_base package is based on a two-layered architecture. Beside the described global planner

[1]Source: https://commons.wikimedia.org/wiki/File:BilinearInterpolation.svg

implementation it has the local planner. The global planner uses the static map as imput to plan a path. However, the local planner has a dynamic, local costmap build with current sensor inputs. With this dynamic costmap, the local planner can avoid unexpected obstacles.
Moreover, it connects the poses of the path published by the global planner in a way that respects the differential constraints of the non-holonomic TurtleBot. This is the reason, why the k-order Markov optimization problem in the previous section was defined without a differential constraint.
Further information about the ROS default local planner can be found in the ROS Wiki[2].

[2]http://wiki.ros.org/base_local_planner

6 Discussion

6.1 Results and Possible Alternatives

The introduction stated the aim of this thesis, as examining path planning and optimization on SLAM-based map. In chapter 5 a possible way to do so was described. The described implementation respects the properties of SLAM maps (see section 5.1) and is abled to find a path. But how good are the resulting paths, if this implementation is actually applied on the TurtleBot in order to find paths?

The figures 6.1 and 6.2 show two examples of paths the implementation returned. White spots are collision-free, black spots are in collision-range and grey areas are unknown. The purple line represents the un-smoothed result of the Dijkstra algorithm. The cyan line is the final path after optimizing with KOMO.
One can notice that the purple line is a short, collision-free connection between start and goal, but it does not seem nice to drive. There are reasons for this. One the one hand, Dijkstra does only connect positions and does not care about how drivable a path is, since the orientation is not included. On the other hand, the map resolution is responsible for the typical kinks (cf. figure 3.2).
But this is not a problem, since the KOMO framework is applied to smoothen the path. And the resulting cyan path is all we ever wanted: a short, smooth and collision free connection from the robot's starting pose to a desired pose.

Let us take a closer look on the properties of the implementation and the resulting path. The used grid map has a resolution 576x576 pixels with a size of 0.05 m x 0,05 m. With this resolution the Dijkstra algorithm returns a for a distance of 8 meters a path of about 130 steps. It takes it about 50 to 150 ms to calculate such a path.
For the optimization with the KOMO framework different newton methods were tested. The best results have been reached with augmented Lagrangian and parameters that can be found in the appendix A.1.
With augmented Lagrangian and these parameters the optimization of a 8 meters long path usually just needs about 6 to 8 iterations until it reaches its stoppage criterion. The average time required for the optimization is about 300 to 500 ms.

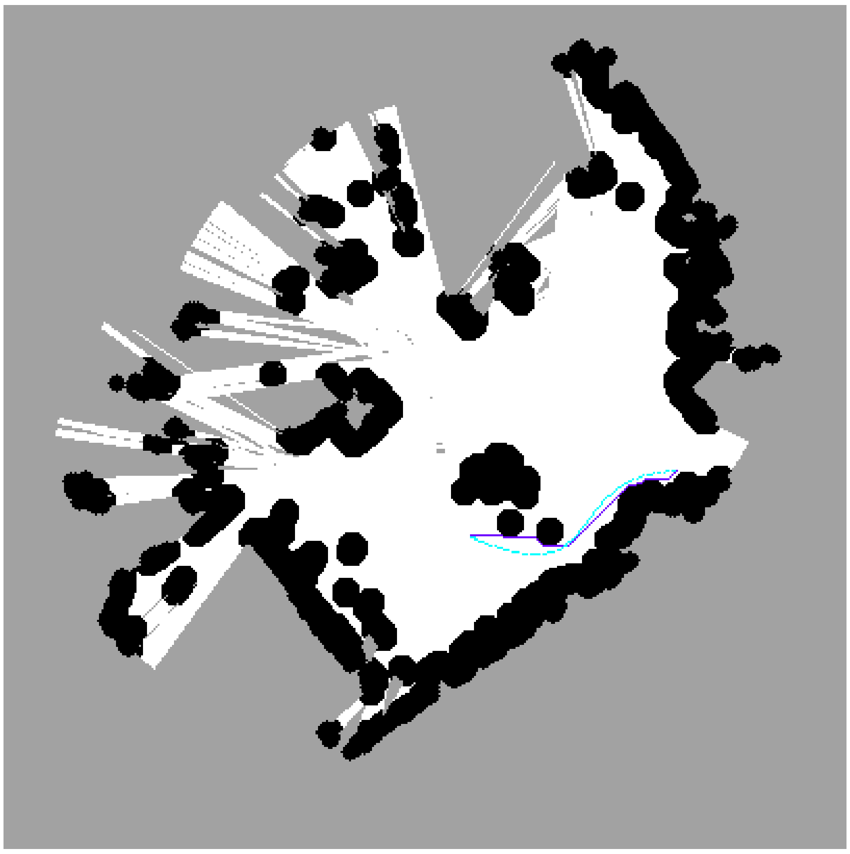


Figure 6.1: Path computed by the described approach - A

Altogether the approach needs a maximum of only 0.7 seconds after it receives a desired target pose until its calculation of a drivable path is finished.

Consequently the results can be summed up in the following way:
The robot is able to fast and efficiently find a short and smooth collision-free path between its configuration and a desired target pose.

For sure this is not the presented way is not the only possibility approach to get this result. Although the combination of Dijkstra algorithm and KOMO framework did very well, other methods are conceivable for path planing and optimization.
As a path finding algorithm the A* algorithm is a very popular option on grid-based maps. A* adds a heuristic function to the Dijkstra algorithm. The resulting path and the computing effort are pretty similar and they should be exchangeable in most situations. Other publications about path planning also work with probabilist roadmaps ([Sha15]), a genetic algorithm ([Sha15]) or potential fields ([YW15]).
Also for the smoothing of a path, a lot possible options beside KOMO exist. [Sha15] for example, discusses the phase-plan method, dynamic programming and discrete

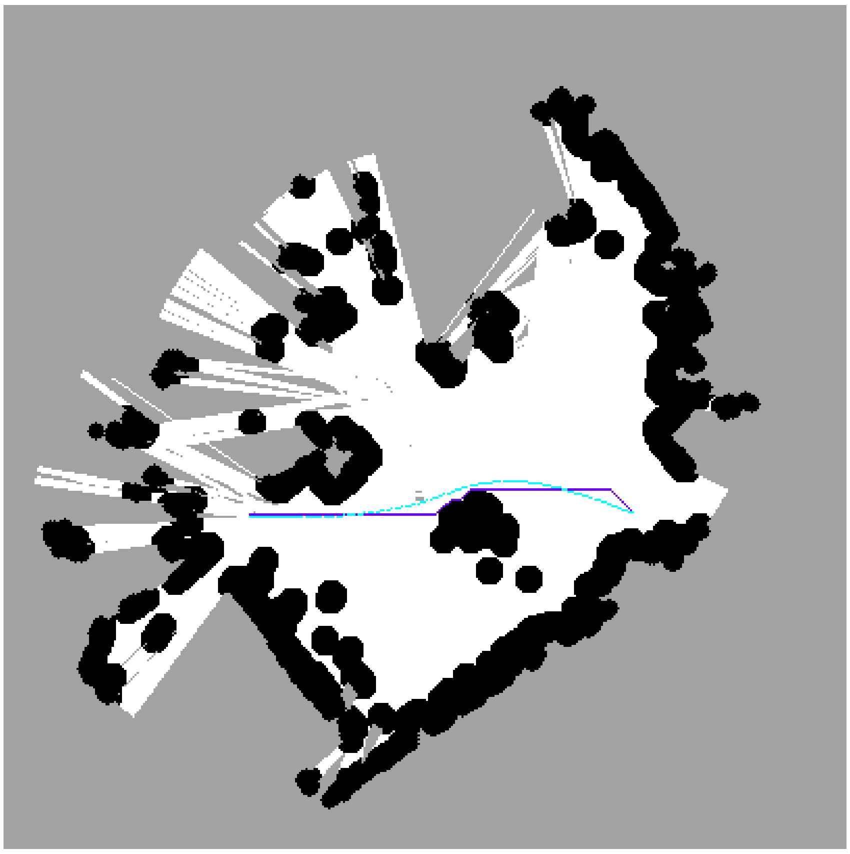


Figure 6.2: Path computed by the described approach - B

mechanics and optimal control as three techniques for trajectory optimization.
[SC14] introduces another interesting approach that includes both, planning and optimization, by using particle swarm optimization.

6.2 Limitations of the Presented Approach

The previous section rated the result of the illustrated implementation pretty good. But we have to keep in mind, that the approach only returns the desired path under different assumptions.
What main assumptions have to be made in order to get the desired, short and smooth path with the presented approach?

(1) Cells that are categorised as "unknown" are occupied.

(2) The map is always up-to-date.

(3) There are no other moving objects in the environment other then the TurtleBot.

Now how do these assumptions limit the approach?
The Assumption (1) limits the approach quite a lot, since there may exist a shorter path than the calculated one, if unknown cells are not considered occupied. The only reason it was not the result of the implementations is that it is including unknown cells. This es even more problematic, if we recap what unknown cells exactly mean: unknown cells are all cells that were not declared as free or occupied in the transformation mentioned in section 5.2. Which means,we may already know that some of the unknown cells are likely free, but the value is just above our threshold.
Assumption (2) and (3) limit the approach to static environments. The mentioned two-layered architecture somehow already removes parts of this limitation since it prevents the robot from colliding with unknown obstacles appearing in the path. In such a case it still guarantees that the desired target configuration is reached (cf. section 5.5). However, if obstacles were removed from the environment since map was drawn and new, shorter paths are possible, the robot is not able to find them.

6.3 Possible Extensions and Adaptations

After stating the assumptions made for the presented example and uncovering its limitations, let us take an outlook on ideas of how to remove the mentioned limitations. A very interesting extension dealing with mentioned limitations is to make adjustments on the used cost-functions including the transitions costs in the Dijkstra algorithm as well as in the KOMO framework. For now, these cost-functions are just designed to find the short, smooth and guaranteed collision-free path according to the map. A idea is to extend these cost-functions with

(a) a term f^1 depending on the occupancy probability

(b) and a term f^2 depending on the date the occupancy probability was updated lastly.

Whereas the term described in (a) makes the assumption (1) unnecessary, term (b) removes the limitations following assumption (2).

Let us be more precisely. For now, unknown cells were ignored in the calculation of paths. If a term f^1 is added, they can be included. This term f^1 should be defined in a way, that cells with a low occupancy probability are more likely included in paths, than cells with a high occupancy probability.
If a term f^1 is included, the result is a trade-off between the length of the path and the probability that it is collision-free.
Publications discussing this idea are for example [HG08] and [Ste94].

If we include a term f^1 we still cannot deal with dynamic environments. The idea of a term f^2 is to keep the map up-to-date. If a the belief over a occupied cell was not updated for a long time, its state may have changed.
This term can be illustrated by somebody driving to work everyday on the same road. Someday the road is blocked by a construction site. The driver will avoid this road for some time and use a drive around. But after several weeks he will likely check again, if the construction site is still blocking his usual way, since it is the best connection of his home and his work.
With defining a term f^2 we can adapt this behaviour on a mobile robot. It allows the robot to sensibly to deal with dynamic environments, that change over time.
If a term f^2 is part of the cost-functions, the resulting path is a compromise length and reliability of the occupancy probabilities.

Repealing assumption (3) is quite hard. It does only require adaptations in the path planning methods but a whole new type of SLAM methods, which abled to detect and track moving objects (DATMO). SLAM methods that include this functionality are called simultaneous localization, mapping and moving object tracking (SLAMMOT) methods and are introduced in [Wan+07].
If the moving objects can be tracked, it is still a difficult task to include them in a meaningful way in the planning and optimization of paths.

A Appendix

A.1 Parameters Used in the KOMO Framework

parameter	**value**
verbose	1
stopTolerance	1e-2
stopFTolerance	1e-1
stopGTolerance	-1.
stopEvals	1000
stopIters	1000
initStep	1.
minStep	-1.
maxStep	10.
damping	0.
stepInc	2.
stepDec	.1
dampingInc	2.
dampingDec	.5
wolfe	.01
nonStrictSteps	0
allowOverstep	true

Bibliography

[AMG02] M. S. Arulampalam, S. Maskell, N. Gordon. “A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking.” In: *IEEE TRANSACTIONS ON SIGNAL PROCESSING* 50 (2002), pp. 174–188 (cit. on p. 7).

[CLR00] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to algorithms*. 24. print. Cambridge, Mass. [u.a.]: MIT Press [u.a.], 2000, XVII, 1028 S. ISBN: 0-262-03141-8 (cit. on p. 16).

[DWB06] H. Durrant-Whyte, T. Bailey. “Simultaneous localization and mapping: part I.” In: *IEEE Robotics Automation Magazine* 13.2 (2006), pp. 99–110 (cit. on pp. 1, 5).

[Dou98] A. Doucet. *On sequential simulation-based methods for bayesian filtering*. Tech. rep. 1998 (cit. on p. 23).

[GSB05] G. Grisettiyz, C. Stachniss, W. Burgard. “Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling.” In: *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*. 2005, pp. 2432–2437 (cit. on pp. 7, 9, 23).

[GSB07] G. Grisetti, C. Stachniss, W. Burgard. “Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters.” In: *IEEE Transactions on Robotics* 23.1 (2007), pp. 34–46 (cit. on pp. 12, 23).

[HFM15] T. S. Ho, Y. C. Fai, E. S. L. Ming. “Simultaneous localization and mapping survey based on filtering techniques.” In: *Control Conference (ASCC), 2015 10th Asian*. 2015, pp. 1–6 (cit. on p. 6).

[HG08] Y. Huang, K. Gupta. “RRT-SLAM for motion planning with motion and map uncertainty for robot exploration.” In: *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2008, pp. 1077–1082 (cit. on p. 38).

[H+03] D. Hähnel, W. Burgard, B. Wegbreit, S. Thrun. “Towards Lazy Data Association in SLAM.” In: *Proceedings of the 11th International Symposium of Robotics Research (ISRR’03)*. Sienna, Italy: Springer, 2003 (cit. on p. 23).

[LSY14] Y. Lu, D. Song, J. Yi. "High level landmark-based visual navigation using unsupervised geometric constraints in local bundle adjustment." In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 1540–1545 (cit. on p. 8).

[LaV06] S. M. LaValle. *Planning Algorithms*. Available at http://planning.cs.uiuc.edu/. Cambridge, U.K.: Cambridge University Press, 2006 (cit. on p. 13).

[Liu96] J. S. Liu. "Metropolized independent sampling with comparisons to rejection sampling and importance sampling." In: *Statistics and Computing* 6.2 (1996), pp. 113–119 (cit. on p. 24).

[PTN08] L. M. Paz, J. D. TardÓs, J. Neira. "Divide and Conquer: EKF SLAM in O(n)." In: *IEEE Transactions on Robotics* 24.5 (2008), pp. 1107–1120 (cit. on p. 9).

[SC14] R. Solea, D. Cernega. "Trajectory planner for mobile robots using particle swarm optimization." In: *System Theory, Control and Computing (ICSTCC), 2014 18th International Conference*. 2014, pp. 111–116 (cit. on p. 37).

[Sha15] Z. Shareef. "Path Planning and Trajectory Optimization of Delta Parallel Robot." Dissertation. Fakultät für Maschinenbau, Universität Paderborn, May 2015 (cit. on p. 36).

[Tou14] M. Toussaint. *KOMO: Newton methods for k-order Markov Constrained Motion Problems*. e-Print arXiv:1407.0414. 2014 (cit. on pp. 18, 19).

[Tou16] M. Toussaint. "A tutorial on Newton methods for constrained trajectory optimization and relations to SLAM, Gaussian Process smoothing, optimal control, and probabilistic inference." In: *Geometric and Numerical Foundations of Movements*. Ed. by J.-P. Laumond. Springer, 2016 (cit. on p. 18).

[VACP11] R. Valencia, J. Andrade-Cetto, J. M. Porta. "Path planning in belief space with pose SLAM." In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. 2011, pp. 78–83 (cit. on p. 2).

[Val+13] R. Valencia, M. Morta, J. Andrade-Cetto, J. M. Porta. "Planning Reliable Paths With Pose SLAM." In: *IEEE Transactions on Robotics* 29.4 (2013), pp. 1050–1059 (cit. on p. 2).

[Wan+07] C.-C. Wang, C. Thorpe, S. Thrun, M. Hebert, H. Durrant-Whyte. "Simultaneous Localization, Mapping and Moving Object Tracking." In: *The International Journal of Robotics Research* 26.9 (2007), pp. 889–916 (cit. on p. 39).

[YW15] Y. Yi, Z. Wang. "Robot localization and path planning based on potential field for map building in static environments." In: *Engineering Review* 35.2 (2015), pp. 171–178 (cit. on p. 36).

[Ste94] A. T. Stentz. “Optimal and Efficient Path Planning for Partially-Known Environments.” In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA ’94)*. Vol. 4. 1994, pp. 3310 –3317 (cit. on p. 38).

All links were last followed on october 22, 2016.

Declaration