AAL-2009-2-049, ALIAS
D6.3
First navigation software module

Due Date of Deliverable	2011-07-30
Actual Submission Date	2011-08-19
Workpackage:	6
Dissemination Level:	Public
Nature:	Report
Approval Status:	Final
Version:	v0.1
Total Number of Pages:	45
Filename:	D6.3-IUT-NavModule-v1.0.pdf
Keyword list:	structure, function, navigation software module

Abstract
This deliverable provides an overview of the structure and functionality of the first navigation module. It describes the overall motion planning approach, realized within this module and experiments with an alternative motion planning approach. It also gives an overview of our person tracking framework as a basis of social acceptable navigation.

History

Version	Date	Reason	RevisedBy
0.1	2011-07-15	created [IUT]	Jens Kessler
0.2	2011-07-30	address comments of internal review [IUT]	Jens Kessler
0.3	2011-08-19	address comments of external review [IUT]	Jens Kessler

Authors

Partner	Name	Phone / Fax / Email
IUT	Jens Kessler	Tel: +49 36 77 694170 Fax: +49 36 77 691665 Email: jens.kessler@tu-ilmenau.de

Table of Contents

1 Executive summary

This document is one of a series of deliverables, which describes the navigational part of the ALIAS project. The navigation within ALIAS focuses on "socially acceptable navigation", which means that the human being and the robot, both as social entities, should react to each other in natural ways, and that especially the robot treats a detected person not only as an obstacle, but applies psychological rules in such cases.

While the first deliverables D6.1 [11] and D6.2 [12] covers the psychological and gerontological background, the state of the art, and methods to implement of all parts of the navigation system, this deliverable provides information on the software structure, the central planning approach and the extension of navigation behavior of the robot. We show the current state of implementation and testing.

This deliverable covers experimental results and details on implementation of the navigation module after the first year of development.

2 Introduction

Since mobile robotics exist, the task of moving a robot is a main challenge of robotics. In industry applications motion patterns are often pre-programmed and remain the same all the time, for example when cars or electronic devices are assembled. In such cases, motion is very exact, but also completely inflexible. This means, the system could not react on changes in the environment and is in that case not "intelligent". It assumes on every action it does identical conditions. Mobile robotics faces the problem on controlling the robot in unknown situation, which are not predictable or countable beforehand, and have to react in a safe, goal driven, and natural way. This describes the challenging task of "motion planning". One the one hand this can be solved by pre-defining drivable paths the robot could drive and label these paths with magnetic sensors, which is often done in industrial service robotics. This approach needs planning effort even when the building is constructed. On the other hand the robot has to be made "intelligent", to allow it to react on all kinds of every-day situations. The latter case is the goal to be achieved by the ALIAS robot. The home environment, the robot works in, is different from all other home environments, so it has to adapt itself to each new environment. Also the safety rules, the tasks and the psychological rules remain the same in each environment.

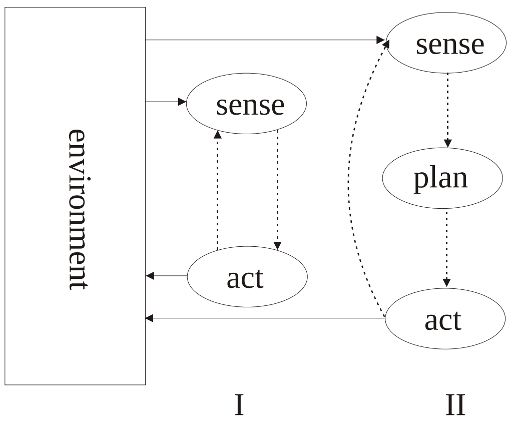


Figure 2.1: The two paradigms of motion planning. I: the direct mapping of sensor input to actions, which is very fast. II: the planning approach, where a planner generates the action of the robot.

In the theory of neuroinformatics two paradigms are discussed to control a robot (see Fig. 2.1). First, the "sense - act" paradigm. Here, the robot gets its sensor information like a camera image, laser sensors, depth information, etc., and chooses by itself a suitable action to fulfill a certain task, like following a person or reaching a certain place. This approach assumes a learned direct response of environmental input. Such reactive mechanisms could be found inside the human's spinal cord and low level "reflex-like" actions of the human brain stem. Also insects are known to navigate with such mechanisms. The benefit (also in technical devices) is, that such mechanisms are very fast to compute.

The second paradigm is the "sense - plan - act" paradigm. In addition to the first paradigm, a planning step is placed to be the central decision process. While in the first approach the environmental knowledge is represented more indirect by training reflexes, here the environment is represented by a dedicated model, like a map, and actions are chosen by evaluating plans inside this "inner world model". Usually every complex task need some sort of planning and this paradigm can be found in almost all complex biological (mobile) entities, ranging from birds towards mammals. The drawback of this approach is that an immediate response of a system is not given, since one has to "think about" a situation to create a sufficient plan.

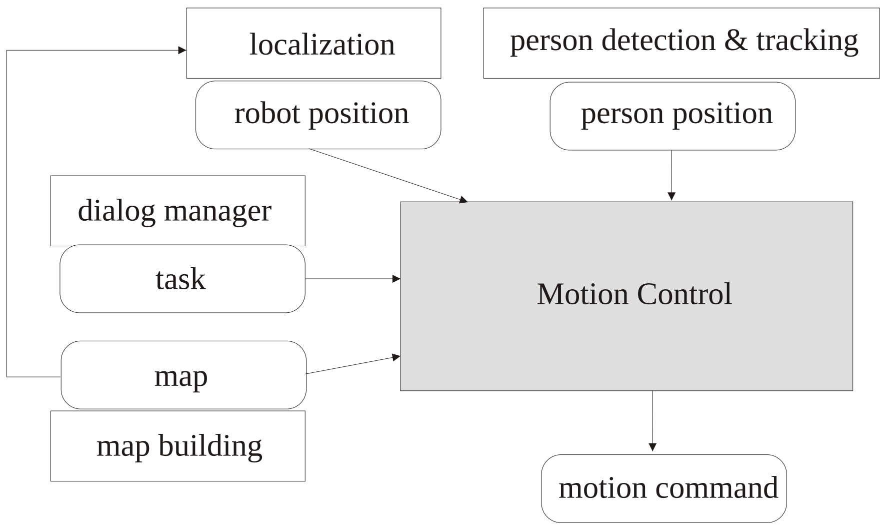


Figure 2.2: Dataflow diagram of the navigation software module.

Both paradigms can be used to create a motion planning system. Within this deliverable we provide approaches and experiments for both paradigms. Classical motion planning has to fulfill three tasks: the robot has to localize itself within the environment, the robot has to move through the environment without harming itself or other parts of the environment, and the robot should drive to a given target. Only the latter parts, the danger-free, goal directed driving are *acting* parts of the software system, while localization deals more with sensing and state estimation. In Fig. 2.2 the structure of the whole navigation software system is shown. Here, additional modules are needed to support the navigation process. The localization was already mentioned as a module which estimates the position of the robot. Despite this function, also an entity is needed, which creates a map of the operating environment and also a module is needed, which gives the motion control system the position and identity of recognized persons. Last but not least, also a task has to be specified, given by the dialog manager.

This deliverable will focus on the person detection & tracking module and on the motion controlling module, which will be described in more detail within this deliverable . The dialog manager is described in detail within the deliverables of work package 3. To give the reader an idea, of what the localization module and the map building module will do, we present a brief description of the used algorithms for both modules:

2.1 Helping software modules

2.1.1 Localization

One of the basic prerequisites to navigate a robot is, that the robot needs to know, *where* it is. This sounds in the first thought as an easy problem, since each human could solve it and also it is known from car-navigation systems, that cars also seem to know their exact position by using GPS positioning systems. But the main problem on this task is: you cannot measure the position! It is possible with GPS localization systems to get a rough idea of the position, but could easily make errors of up to 10 meters. Usually, reliability is within 1m, which is in home environments not sufficient. Also, in indoor-environment GPS is usually not available and has to be installed in every environment.

The position of the robot has to be *estimated*, in quite a way humans would do it: you can measure the distance from distinct points (like doors, edges, etc.) towards the robot and could construct the position geometrically. But these measurement are erroneous. Each time the measurements are done, a slightly different position will be calculated (which is also a proof that the position could not be measured directly). When the robot moves through it's environment after every motion step the position has to be re-estimated. Here two values could help to estimate the position: on the one hand the measured motion of the robot (also called odometry) and on the other hand, and most important, the observations the robot does with its laser scanner.

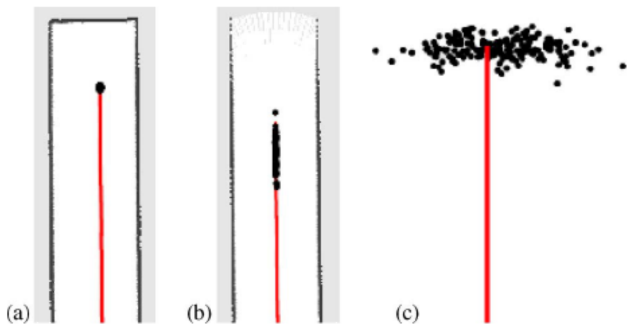


Figure 2.3: The different form of a robot's position probability distribution modeled by a particle filter like shown in [6]. In a) the position could be estimated very sharp while in b) only the relative to the left and right walls could be well estimated. In c) no distance information is available to estimate the position, so only the robot motion is used to sample the distribution.

By using both measurements and taking into account that both will be erroneous, a recursive state estimation problem could be formulated, which results in a *probability density distribution* of the robots position. To model this distribution we use a so called particle filter, which is a common approach within the robot community (see e.g. [14], [20]). Depending on the distinctiveness of the observation, the distributions could lead to different forms. This is shown in Fig. 2.3, where three typical situations are presented. So the exact robot position is *never* known, but the uncertainty is sometimes very small, but could also sometimes be within a few meters. It is a usual approach to take the most likely position within the distribution as the result of the estimation process.

There exist a few classes of localization problems like the ability of the system to localize itself in a global fashion, which means the robot only knows the map of the environment and could estimate its position, as well as a local position tracking, which enables the robot to keep track of its current position with knowledge of the start position. Our system can solve the latter case.

2.1.2 Map building

To enable the robot to operate within the environment, it is necessary to know how the environment looks like. This is usually represented by a map. As described above, it is impossible to measure the position of the robot, so it is naturally also impossible to measure a map, since the map has to be recorded from many positions, which could be only estimated, and to make things worse, the positions could only be estimated with the help of a map. So this is a classical chicken-and-egg problem and is also a very hard estimation problem, called the SLAM (simultaneous localization and mapping) problem.

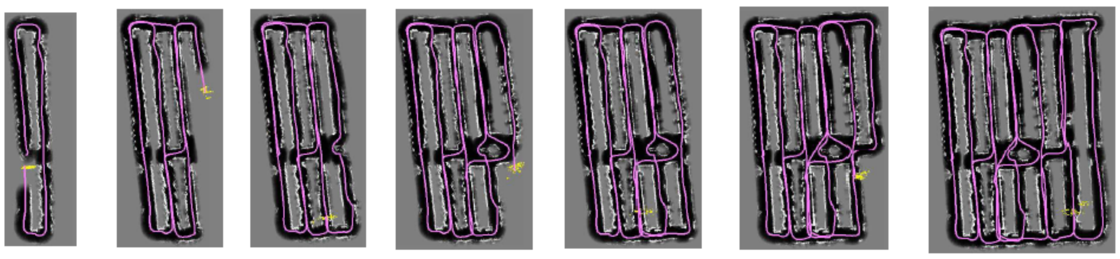


Figure 2.4: Our iterative on-line approach on learning a map. The driven path and the corresponding map is estimated by a particle filter. In this example (from [16]) a home store is shown.

Key idea is always, to collect a series of observations, collect also the driven distances, a assemble from these puzzle pieces of observations and distances the most likely map with the most likely driven path. We also use a particle filter to solve this estimation problem (see [16], [6]). Since it is memory and computational hard to do the map building in a lifelong manner, we have to do this building process in advance for each new environment the robot should operate in.

2.2 Structure of this deliverable

After showing the key concepts of the helper modules, no further details about these modules will be discussed here. So the rest of the deliverable will be structured as follows: In chapter 3 the person tracking module will be described in detail, since this module is a key prerequisite for *social acceptable* navigation. After this chapter is done, in chapter 4 the concept of a "sense-act" motion planner is presented, which implements a reinforcement-learning approach to steer the robot. In the next chapter (chapter 5) a "sense-plan-act" approach is presented (namely the Dynamic Window Approach), which is a well known motion planner and is already part of the navigation system, given from ALIAS partner MetraLabs. We present in the next chapter (6) the addition to this Dynamic Window Approach, to realize an approaching behavior. In the last chapter a short conclusion is drawn and an outlook towards the next years report is given.

3 Tracking a person

3.1 Introduction

The most important basic ability to enable a robot for social acceptable navigation, is to detect a human. This task is quite challenging and whole projects deal only with this topic. This task is also not clearly defined in the projects description of work. That's why we decided to use software, already available and free to use. At the beginning of the project we planned to use as input modalities the laser range finder, a panorama camera on top of the robot and an fish eye camera in front of the robot.

As already described in D6.1 [11], we initially planned to use the Augmented Reality toolbox "ARToolkit", to detect black and white patterns with their corresponding position in space. On that approach we encountered numerous problems: first, the camera has to be calibrated and the camera model is not sufficient to reflect the properties of a fish-eye camera. Only in the mid region of the scene positions are undistorted. Second, the software is not maintained any more and third: the real world appliance (due to lighting conditions) and the need of wearing pattern plates is relatively low.

But the most important point: with the introduction of Microsofts Kinect device, rules of the market of person detection changed. Around this device a very agile, dynamic and enthusiastic community developed with astonishing speed. At the date of writing this deliverable, even Microsoft has published a closed source software library to enable Windows developers using the Kinect device to detect persons, person poses and gestures. This shows the impact of this device, since it was originally designed for a video game play console.

This is also the reason why we decided to discard the idea of using the ARToolbox and instead using the Kinect device with a library to detect persons. So our final configuration is the laser scanner input, the panorama camera (for face detection) and the Kinect device for pose recognition. In this chapter we present the work done to use these input channels to our purposes.

3.2 Laser based leg detection

The used approach was first described from Arras in [1]. In this publication an AdaBoost classifier was trained on a set of 14 predefined features, which are calculated from laser segments. To obtain these segments, the laser scan is segmented by a running slice segmentation. Figure 3.1 shows two examples for scans and their segmentation results.

Sources for this classifier are, however, not available. But the company Willow Garage,

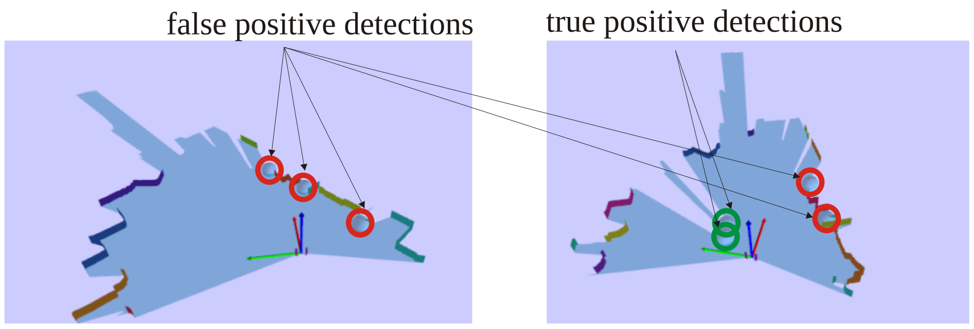


Figure 3.1: Two example scans for a complex situation to detect legs within. Here, the scan (blue fan) is shown plus the corresponding segmentation (colored boundaries). Small circles determine the segments, classified as "legs". It could be seen that in complex scenes pair-like structures are quite normal and detected as false-positives. But it is also possible to detect real legs. This is the first channel of our person detection module.

with its famous framework "Robot Operation System" have created a very similar approach, although no publication are available. They provide an experimentally free person tracker. By investigating the source code, we found the part of leg detection very similar. They use the same 14 features on scan segments, but use a different classifier (here, random forests are used for classification). Additionally Kalman filters are used to track the leg hypothesis and also estimate the speed of a person. We used this source code and ported it to our platform architecture. Since this code is published with the GNU LGPL there are no copyright restrictions in usage.

3.3 Detecting persons in 3D point clouds

The second modality we use to detect persons is the Kinect depth camera. This device is a camera, which could deliver for every pixel in the image also a depth information. The camera could deliver depth images with 30 frames per second. Currently there are two closed source libraries on the market which allow the segmentation of those depth images and which also try to fit a skeleton into these segments. At the one hand the OpenNI framework with the NITE component installed and on the other hand the KinectSDK from Microsoft. An example of the NITE detector is shown in figure 3.2. Both libraries need a lot of resources to detect persons. More details will follow in the experiments section below.

Due to this high demand on processing power we currently develop an easy estimation algorithm of the upper body pose of a person to avoid the processing needs of the skeletation stage. This could only be done in the OpenNI framework. We rely here on the correct segmentation of the depth image. Since segmentation fails when the robot moves, we apply a simple filter for each segment, to classify person-segments and non-person

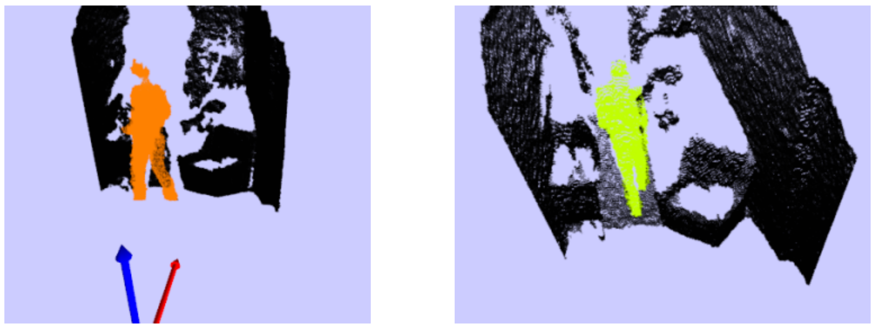


Figure 3.2: Two examples of segmented point cloud by using the OpenNI framework and the NITE segmentation stage. The coloured points represent those points of the point cloud, which belong to a person.

segments. Currently we use a simple threshold heuristic on each segment as a classifier. Future experiments will show if this is sufficient or if more reliable classifiers are needed. Details on the upper body pose estimation will be presented in the next deliverable.

3.4 The person detection module

The person detection module has two sub-modules we already mentioned: the laser based leg-detector, which is able to detect very reliable leg-pairs and their corresponding speeds, but also returns many false leg hypotheses. On the other hand we have the Kinect device, which delivers very accurate person hypotheses and the upper body pose, but sometimes fails to detect persons. Currently both channels are handled separately, which means that hypotheses from both channels form each an independent hypothesis.

Future work, which will be done in the next two month, will combine both channels in a probabilistic Kalman filter framework, to fuse both channels to a more reliable channel of hypotheses.

3.5 Experiments

For both channels we did very basic experiments to find out more about the detection properties of both channels. To do so, we record a set of example situations and run all detectors on them. The first results of the leg detection where disappointing. No legs could be recognized at all! Deep investigations on the algorithm showed, that the classification tree does not fit to our used laser scanner. We had to develop a tool to create training data for a new classification tree. After this was done the leg classifier works as expected. We still encounter some small problems with the used Kalman filters. Prob-

Algorithm	Runtime/step	Resources/%
NITE	26 ms	60
KinectSDK	35 ms	80-100
Laser legs	3 ms	10

Table 3.1: Results on all three detection algorithms for processing time and consumption of processor resources of a dual core system in %. Note that the percentage is for both processors.

lems are, when new filters should be added and when new filters should be removed. But the current impression of quality of the leg detector is good, but could be improved. The resulting leg classifier is very fast.

The tests of the Kinect NITE detector was quite short. Most work had to be done to couple the NITE detector to our environment and to test detection speeds. We did this also briefly with the Windows KinectSDK on the robot. We encountered in both approaches one common problem: if the robot moves, also parts of the static background where segmented and are at least candidates for persons. The KinectSDK ignores these segments and only returns results for valid found skeletons. This could be done very robustly. The NITE framework could also create skeletal information, but needs an initialization pose, which is not always detected. This initialization makes the NITE skeletal processing difficult for real world experiments with non-expert persons, since a feedback has to be given, if initialization succeeds. This is also a point why we decided to NOT use the NITE skeletation process.

Both libraries (NITE and KinectSDK) have in common, that they only work, if a person is fully seen by the camera. Depending on the position, this is the case from 2 meters upwards. Here, the need of combining laser scanner and Kinect could be seen.

Last but not least, we will present some results on the needed processing power for a single detection step. All algorithms run on a dual core 2.66 GHz processor on the robot. Beneath the pure calculation times we also show the percentage of processor consumption for all three approaches. The results are shown in table 3.1.
It could be seen that leg detection needs almost no time and needs only a few resources. The processing of point clouds seems very time consuming and it is easy to occupy two processors for such a task.

3.6 Conclusion

During the development of these task we have encountered two major impacts. First, the appearance of the Kinect sensor, which replaces the visual channel and second the discovery, that processing of point clouds is very time consuming. For our next experiments we have this processing power, but for the final system we have to lessen the processing burden a lot. One idea is to downsample the used points of the point cloud, or to optimize the current algorithms. This may mean that none of the existing point cloud detectors is uses.

In future work we will also focus on fusing all channels for person tracking to improve the classification results. This means not only the usage of the laser channels and the depth image, but also the face detection channel and the face identification channel. The results of these works will be shown in the next deliverable.

4 The Reinforcement Learning Approach

4.1 Main concept

During the introduction we already mentioned, that the reinforcement learning approach belongs to the "sense-act" paradigm. This means, that the observation of the robot (in our case a laser scan) is used to directly derive an action, without an additional planning stage. The reaction of a given situation has to be learned. Reinforcement learning is a unsupervised learning approach, which means, that no teacher is needed to show the robot, what it should learn. The evaluation of the state/action is hereby a punishment or reward (negative or positive values), depending on the fact, if a chosen action was beneficial in a given situation or not. This concept is copied from biological entities, where also learning without a teacher occurs for example when learning to move without hitting an obstacle, learning to open and close drawers. Children use this kind of learning in an extensive way. Pain is usually the signal of punishment, whereas a positive feedback is the successful exploring of new space or a compliment of the parents.

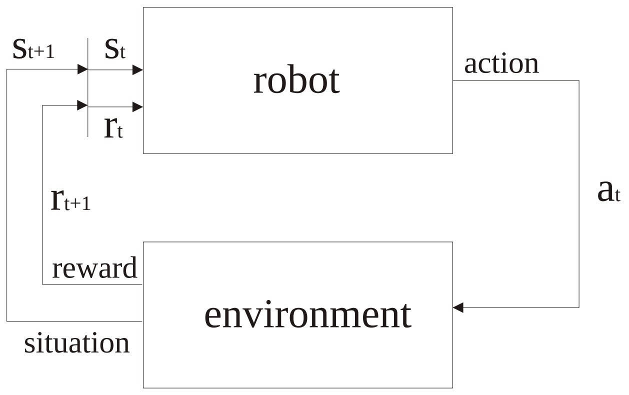


Figure 4.1: The robot motion control module and its environment (from [10]). This figure shows the interaction cycle from the robot and its environment in a reinforcment learning way. The robot chooses from the current state of the environment (and itself) s_t a possible action a_t. This changes the environment and gives the robot as a feedback a new state s_{t+1} and also an reinforcement signal r_t. The reinforcement signal tells the robot if the previous action was a feasible one.

In robotics, the so called reinforcement function r defines the positive and negative feedback of the system. This function evaluates the current state S of the robot or the action A the robot has executed in a particular state.

$$r : S \times A \tag{4.1}$$

The benefit of such an approach is, that unlike classification or approximation problems, no training data has to be collected, and the system is ready to learn by just defining the reinforcement function. The learning system just gets told, what was a good action in a given state and what was a bad action and has to develop a global optimal strategy to maximize the rewards it gets. Note, that the sum of the upcoming N rewards has to be maximized, not only the next reward the system expects to get.

$$R = \sum_{i=0}^{N} r(t_i) \cdot \gamma^i i \in (0..1) \tag{4.2}$$

The parameter γ controls, how long rewards are collected over time, to define the expected long-term reward and is called the discount factor. In this way short viewed decisions as well as long distance decisions could be learned. The practical problem is here, that the robot has to discover which states are good or bad and therefor also has to explore dangerous states. This means, training should always happen in an save environment, like a simulator. The resulting optimal strategy is also called policy. One way to model the policy is to create a function of state-action pairs. This forms the most popular approach of reinforcement learning, the so called Q-learning. The following section will describe this approach.

4.1.1 Q-learning

As stated in [18], many reinforcement approaches assume the knowledge of a state transition function (or an environment model), which gives the information, in which next state the system will switch, if a certain action is executed. In reality, such a function is hard to create or even impossible to construct. The Q-learning approach does not assume the knowledge of such a function. Instead of coding only the benefit of being in state s_t within the policy, Q-learning models the benefit (or the expected long term reinforcement) for being in states_t and executing the action a_t! To do so, the function $Q(s,a)$ is introduced, which simply tries to learn for every possible action in a given state the expected long term reward, defined in equation 4.2. Fig. 4.2 shows such a Q-Function in case of discrete states and actions. In the beginning this Q-function is initialized with constant values or random values, since no knowledge is present, which actions are actually the best. In fact the system has to explore good actions! Here different approaches are known, which lead to different exploration strategies and different results in the Q function. We have investigated this fact in experiments.

The exploration exploitation dilemma

In the beginning of the systems training cycle, a huge number of state-action pairs has to be tried to get an impression of the benefit of such an action. Since the states of the environment are not controllable the only remaining question is, what action to select when being in a certain state?

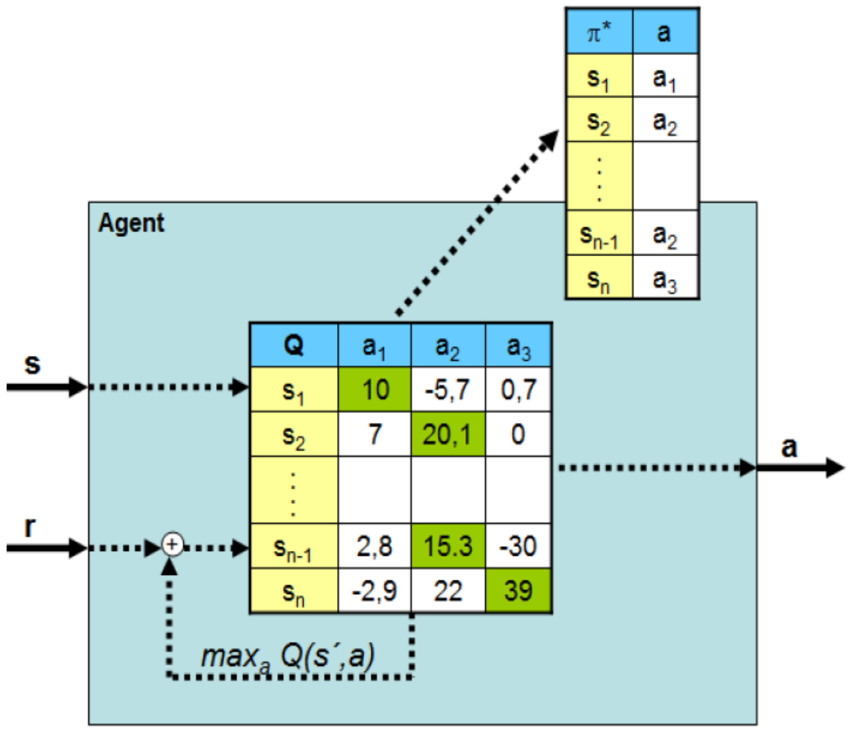


Figure 4.2: Structure of the discrete Q-function with n states and three possible actions. From the Q-function the optimal policy is obtained by choosing for example the maximum Q values.

After learning is finished this task is quite simple. The robot simply choses the action with the highest estimated overall reward, or the maximum value of all $Q(a, s)$ for a given s and all a_i. In the initial phase the information of the best action is not available, since the system has never *experienced* such an action. Here, a random selection of possible actions would be preferable.

In the beginning the system should explore its action space, while with the trained Q function, the system should use gathered knowledge and exploit that knowledge. And the exact point, where to switch both behaviors, formulates the exploration-exploitation dilemma. To overcome this dilemma we use an action selection mechanism that is known as the Bolzmann-action-selection. For a given state s it is defined as follows:

$$P(a_i) = \frac{e^{\frac{Q(s,a_i)}{T(t)}}}{\sum_j = 1^N e^{\frac{Q(s,a_j)}{T(t)}}} \tag{4.3}$$

$T(t)$ is hereby the Bolzmann temperature, when this temperature is very high ($T(t) \to \infty$) the probability of choosing an action a_i is equally distributed $1/N$. When the temperature cools down ($T(t) \to 0$) the highest Q value will become the most probable (with nearly 99%) and the caracteristic of a maximum selection will be realized. For a high temperature the system will have a strong explorative character while in with low temperatures the system will show exploitation characteristics. By choosing $T(t)$ to be an exponential falling function, the exploration-exploitation dilemma can be at handeled. We use this action selection technique in each of the following learning approaches. We also tested each approach experimentally.

Temporal difference learning

The simplest form of Q learning is the TD algorithm (see [18]). Here, only the transition between two successive states is considered: first the Q value from the previous visited state s_{t-1} plus the selected action a_{t-1} and the next entered state s_t. The learning equation is relive simple:

$$\begin{aligned} Q^{new}(s_{t-1}, a_{t-1}) &= Q^{old}(s_{t-1}, a_{t-1}) \\ &\quad +\beta \cdot \left(\left[r(s_t, a_t) + \gamma \cdot max_{a_j} Q(s_t, a_j)\right] - Q^{old}(s_{t-1}, a_{t-1})\right) \end{aligned}$$

The variable β is the systems learning rate and γ is the previous mentioned discount factor, to control the long term or short term characteristic. The new Q value is adjusted with the received reinforcement and the old Q values of the old state-action pair and the maximal expected future reward. This learning approach has a major drawback: by using just two values of Q values the exploration process has to visit a lot of transitions to cover the whole state-action space one time. To let the Q values converge, many visits of state-action pairs are needed, which leads to a very long exploration phase.

Truncated Temporal Differences - TTD

To speed up the convergence it is possible to use more than one state transition, which was invented by Sutton [18]. This allows the algorithm to include memory into the system and respect more than one previous actions. To do so, two additional parameters are introduced:

- m: number of state transitions to remember
- λ: trace decay parameter

The TTD(λ, m) algorithm has to perform m steps until it can update the Q value visited in the first step. The update is done by the following recursive equations:

$$\begin{aligned} Z_{t-1} &= r_{t-1} + \gamma \cdot max_{a_j} Q(s_t, a_j) \\ Z_{t-k} &= r_{t-k} + \gamma \cdot \left[\lambda Z_{t-k+1} + (1-\lambda) \cdot max_{a_j} Q(s_{t-k+1}, a_j)\right] \\ &\quad \forall k \in (2..m) \end{aligned}$$

$$Q^{new}(s_{t-m}, a_{t-m}) = Q^{old}(s_{t-m}, a_{t-m}) + \beta \cdot \left(Z_{t-m} - Q^{old}(s_{t-m}, a_{t-m})\right)$$

By using more rewards and more state transitions, the convergence process could be improved, since each update step has a larger knowledge window. The problem is now the usage of random selected actions, since here positive rewards can lead to higher Q values in a large chain of actions, where the rewarded state action pair has nothing to do with the high reward. Nevertheless, small chains of actions can lead to significant speedups of the exploration phase without negative side effects.

Genetic Q function manipulation

A new idea in speeding up the Q learning approach, is to combine the Q learning strategy with genetic algorithms. The main idea is, to evaluate the policy during usage, no matter how this policy is created. A policy π is defined by assigning *one* action to each state. Usually the policy is created from the Q function. In this approach, presented by [9], we hold a whole set of policies. One is derived from the maximal Q value per state, all others are selected randomly. Each policy could be evaluated and the best policy is used for selecting actions from states.
During the observation of the best policy the reinforcements are traced and situations are detected, where the policy fails. These parts of the policy are optimized by an genetic algorithm. The following steps are executed during learning:

1. Initialize the Q function randomly
2. Generate a pool of random selected policies
3. Evaluate each policy and select the best as active policy
4. Drive with the best policy until a failing situation is detected
5. Upgrade the critical part of *all* policies
6. Evaluate each policy and select the best as active policy
7. Retrain the Q function with the active policy
8. Goto 2.

Recognition of a critical fail of the current policy: First, the robot drives with the (random selected) best policy and simply records the reinforcement signals r_i it receives and the corresponding states s_i the robot visits. When a sequence of constantly bad feedbacks are detected, a critical situation is detected! All states with negative feedback are marked, which is shown in Fig. 4.3.
Since we have created a pool of policies (where just one is active), we select only the parts of the policy where the states where visited during the failing situation and a short period of K states before the sequence of negative feedbacks begun. Obviously the actions chosen within these states are not appropriate and other actions should be chosen. All

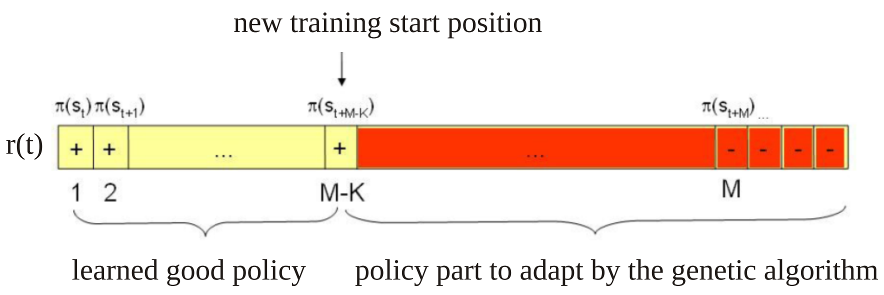


Figure 4.3: Detection of a critical failure of the current policy. Whenever a sequence of bad reinforcements is detected, a critical situation arises.

other states of all policies are defined as well-learned. In our implementation a queue of 40 states are recorded and if a sequence of 10 consecutive negative reinforcements is recognized, a critical situation is detected and the genetic optimization is start upon the marked parts of the policy.

Optimization of the critical situation: When optimizing the pool of policies, the first step is to extract the marked states from the pool of full policies. This is simply done by queuing all marked states in a new sub-policy for each full policy, like shown in Fig. 4.4

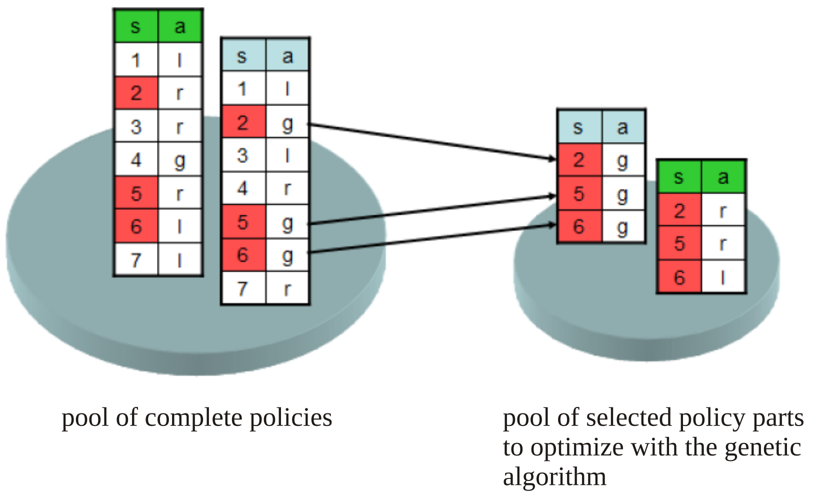

s a
1 l
2 r
3 r
4 g
5 r
6 l
7 l
s a
1 l
2 g
3 l
4 r
5 g
6 g
7 r
s a
2 g
5 g
6 g
s a
2 r
5 r
6 l
pool of complete policies
pool of selected policy parts to optimize with the genetic algorithm

Figure 4.4: Before actual optimization starts, from the pool of full policies a set of sub-policies is created. Here, only marked critical states are used.

As stated before, a genetic optimization algorithm is used to optimize the pool of sub-policies. Genetic optimization is done by using three mechanism of evolution, namely *mutation, recombination* and *selection*. To select a group of best results, we have to evaluate the new created sub-policies by repeatedly "practicing" the situation, where the robot fails. By simply counting the number of positive rewards, a fitness of each sub-policy could be calculated. We want to keep the number of sub-policies constant. When recombining and mutating the policies , the number of policies doubles. When calculating the fitness of each policy, we could rank all policies and throw away the 50% of the worst. For each state in the policy, recombination is done with the Q function and the current policy $\pi(s)$ by defining the likelyhood of changing the current action with any other possible action, which equals very much the Bolzmann action selection:

$$P(s,a_i) = \frac{e^{\frac{Q(s,\pi(s)}{Q(s,a_i)}}}{\sum_j \frac{Q(s,\pi(s)}{Q(s,a_j)}} \tag{4.4}$$

Note that in the equation above probability intervals are defined for recombination with the Q function. A random number will select, which interval is chosen, and in this way also the mutation process is already included in the above equation.

After successful recombination and mutation of all sub-policies of the pool, an evaluation cycle is started to get the fitness of the new created sub-policies. This is done via a simulator, since the critical situation has to be replayed as many times as the number of policies in the pool. This is quite an expensive calculation step. After all policies are evaluated and selected, the progress of improvement of the best policy is investigated. If after 50 evolutionary steps no improvement could be measured, the policy optimization terminates.

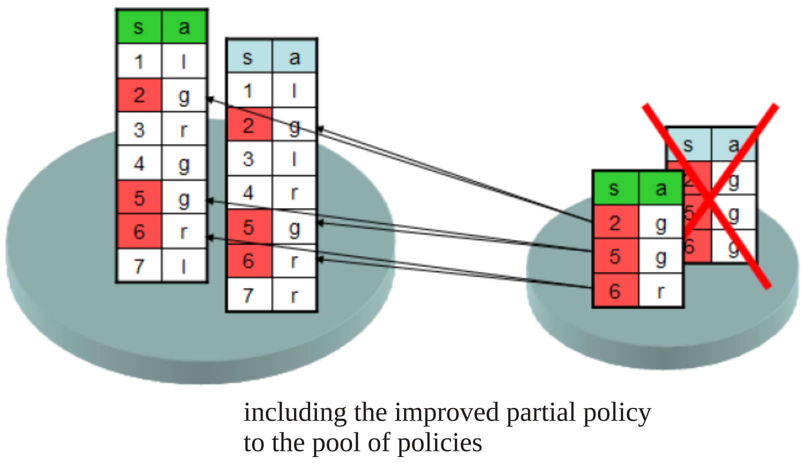


Figure 4.5: After the genetic algorithm terminates, the best sub-policy is selected and projected back into the pool of overall solutions.

Action	turn left				straight	turn right			
V_{rot} in deg/s	0.8	0.6	0.4	0.2	0.0	-0.2	-0.4	-0.6	-0.8
V_{trans} in m/s	0.2								

Table 4.1: The set of defined robot actions

Q function update with new policy The found best sub-policy, created from the genetic optimization process, is now re-established into the overall policy pool (shown in Fig. 4.5). Note, that also the active policy is updated. Now the robot has learned to deal with a new critical situation. The only remaining problem is, that the Q values do not fit to the updated, active policy.

Unfortunately Iglesias [9] does not report, which training algorithm is used to update the Q function. We choose the standard Q learning approach, since only a small part of the Q table has to be updated. To keep the new policy and the Q function synchronous, the robot has to adapt the Q table in an after-training step (by using the simulator again) and simply applying the new active policy within the critical situation, this time with activated standard Q learning. When the Q function converges, the after-training is stopped.

4.2 Experiments

In the last section we presented the theoretical background of Q learning with the focus of three different learning approaches to create a sufficient Q function. But how does these approaches work in real situations? To test these approaches we defined a small navigation task: the robot should follow a path and should not collide with an obstacle. As input data from the environment the robot receives its range data from the laser scanner, the direction of the path and the reinforcement signal. The robot could decide between nine actions (see table 4.1 for details). In all cases the learning process was calculated in a simulated environment. Only the results of the standard Q learning approach where tested on the real robot. All other results do not show a stable simulated behavior.

4.2.1 State estimation

The first problem arises when estimating the state of the robot. The state is defined by all knowledge the robot is given from the environment, here the direction to the path to follow and the scan input. All these values are defined in continuous space, while in our representation the states of our system have to be discrete. The laser range sensor gives an input of 541 ranges, the direction of the path is one extra input, so we have a 542 dimensional space to find descriptive points within. To reduce the dimensionality of the space to discretize, we split the range scan into 16 equally spaced intervals and searched in each interval for the minimal range to represent this interval. So we have a 17-dimensional space to find our discrete states. A classical way to discretize spaces to states is to cluster the given space into a number of clusters. Each cluster represents a discrete state (but contains an infinite number of possible points). We choose to have 220 clusters or states and we choose the neural gas clustering to select them.

Clustering with the neural gas: The neural gas is one candidate of self-organized maps and was developed by Martinetz [15]. It was developed to represent complicated input patterns, which may only be present in sub-spaces of the whole input space. It is defined as a one-layered unsupervised neural network, where every neuron has n input weights according to the input dimension. In our case each neuron has 17 input weights and we have 220 neurons inside the net, each neuron for a cluster. Upon training of the net, the training input patterns $X = (x_1, x_2, .., x_n)$ are shown to the net. The Euclidean distance between the presented input and the neural weights are defining the distance between a neuron and the input. In this way, a nearest neighbor neuron can be found for each presented input. For each input the distance to all neurons can be calculated and the neurons can be sorted by distance, starting with the nearest. According to the rank in the list each neuron gets its activation function $k_i(x_j)$, where the nearest neuron gets the rank 0, the second nearest the rank 1, etc. For each input x_j the net adapts its weights w_i for each neuron as follows:

$$w_i(t+1) = w_i(t) + \eta(t) \cdot [x_j(t) - w_i(t)] \cdot h(k_i(x_j)) \tag{4.5}$$

Here, $\eta(t)$ is the learning rate and decreases over time to guarantee the convergence of the training, while $h(k_i)$ is a neighboring function, defined by a Gaussian, decreasing the influence of the input for higher ranks:

$$h(k_i(x_j)) = e^{\frac{k_i(x_j)}{r}} \tag{4.6}$$

The variable r is the learning radius (in ranks). After the training, the nearest neighbor neuron is the corresponding state to a continuous input.
In our experiments we recorded a set of situation with different obstacle situations and augmented these with different directions towards the target. These data set forms our training data.

4.2.2 The reinforcement function

A second problem in real world applications is the definition of the reinforcement function. After a few unsuccessful trials we found the following definition of such a function. The function consists of two parts: a collision part and a part to deal with the planned path's direction.

$$r(s,a) = collision(s,a) + direction(s,a) \tag{4.7}$$

The collision part is quite easy. If a collision occurred in a given state and after a certain action, a negative reinforcement is given. No reinforcement is given in all other cases:

$$collision(s,a) = \begin{cases} -10, \; if \; collision \\ 0, \; else \end{cases} \tag{4.8}$$

The direction part is a bit more complicated, since the reward does not only depend on the state but also the action, because it is worth a reward if the robot faces with the back to its direction *and* additionally rotates to the correct direction. We also reward a correct heading more than an incorrect heading:

$$direction(s,a) = \begin{cases} 1.3, \; if \; AngleToPath \approx 0 \\ 1, \; else \; if \; abs(OldAngleToPath) > abs(NewAngleToPath) \\ -1, \; else \end{cases} \tag{4.9}$$

4.2.3 Results

Now that the question on state estimation, action selection and reinforcement function is clear, we have used two environments to test all three approaches to for the quality of the Q function. Our training environment is quite simple, mainly an empty room. Additionally we evaluated the resulting Q function, state estimators etc. in an unknown environment, which is a challenge for the state estimator network. The environments are shown as a map in Fig. 4.6.

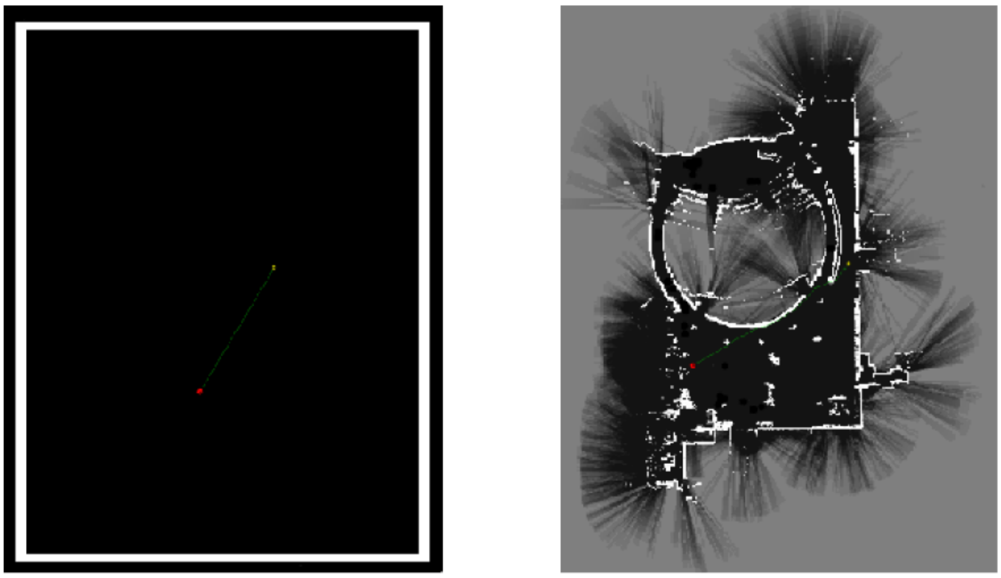


Figure 4.6: The two test environments. On the left is the (empty) training environment, while the unknown evaluation environment is shown on the right.

As mentioned earlier we always used the Bolzmann action selection. This forms the following set of parameters for all training approaches:

	Known Environment	Unknown Environment
Approach	**neg. reward (in %)**	**neg. reward (in %)**
std Q learning	1,675	8,48
TTD	5,15	22,2
genetic	21,11	21,68

Table 4.2: The results of quality measurement of all three learning approaches

- learning rate β = 0.9
- trace decay λ = 0.75
- discount factor γ = 0.7
- start temperature T = 10000
- TTD depth m = 25

After finished training we evaluated each learning approach (Q learning, TTD learning, genetic algorithm) towards its quality. The quality is measured by counting the number of times, the robot receives a negative reinforcement. The results are shown is table 4.2. The interesting fact is, the the standard Q learning is the most robust learner. All other learners work not well within the unknown environment. To clarify this effect we have to look at the resulting Q functions, which are shown in the figures 4.7, 4.8, and 4.9.

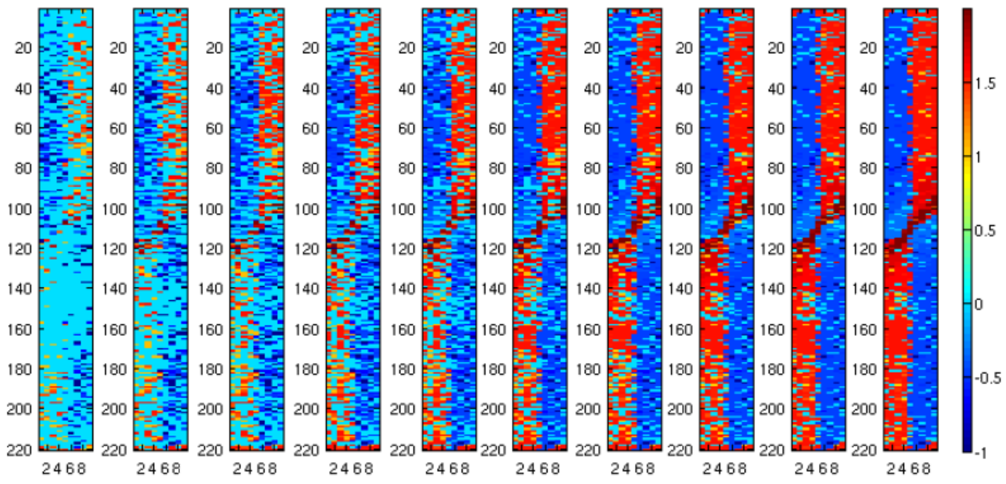


Figure 4.7: Resulting Q function after 10 iterations of the standard Q learning algorithm. The state-action space is explored very densely.

Here the main problem is the sparsely explored state-action space. Only standard Q learning gives reliable results, both other approaches, even after convergence, are not suitable to solve our given task.

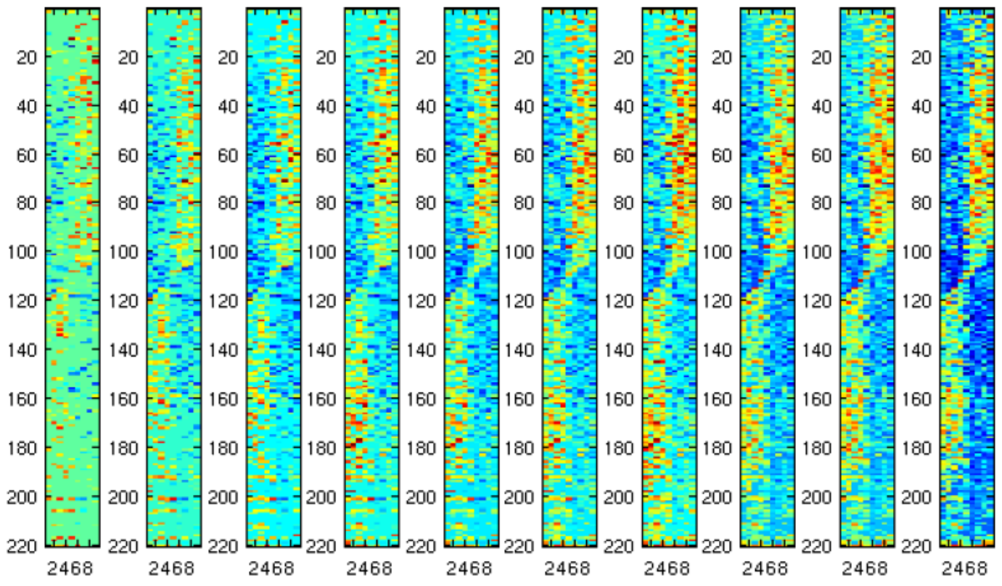


Figure 4.8: Resulting Q function after 10 iterations of the TTD learning algorithm. The state-action space is explored only sparsely.

4.3 Conclusion

After investigating the power of the promising reinforcement approach a mouldy after-taste remains. It has shown that several hidden problems have to be solved to get a system running. First, the state estimation is a huge problem. The collection of training data of each input modality and the parametrization of the clustering system is a source of possible errors, which can lead to complete failing of solving the given task. Also, each task like following a person, driving to a target or observing a person changes the dimension of the input and so a complete new cluster has to be computed.

Second, the exploration phase of the system has to be parametrized correctly to guarantee a fully explored state-action space, and in every day use, this guarantee may be impossible to give anyway. Last but not least, the reinforcement function has to be hand tuned to fit to the given task and to enable the robot to learn that task. Here our trials have shown that even simple approaches to formulate such a function lead to unwanted and unforeseen robot behavior.

Note, that all these points do not consider, which learning approach is used, but show only the problems, defining the prerequisites to start the learning. The configuration of the prerequisites is in itself an expert task for each new problem, which should be solved by such an approach. With such properties, reinforcement learning is not modularizable and is not suitable for our robot system.

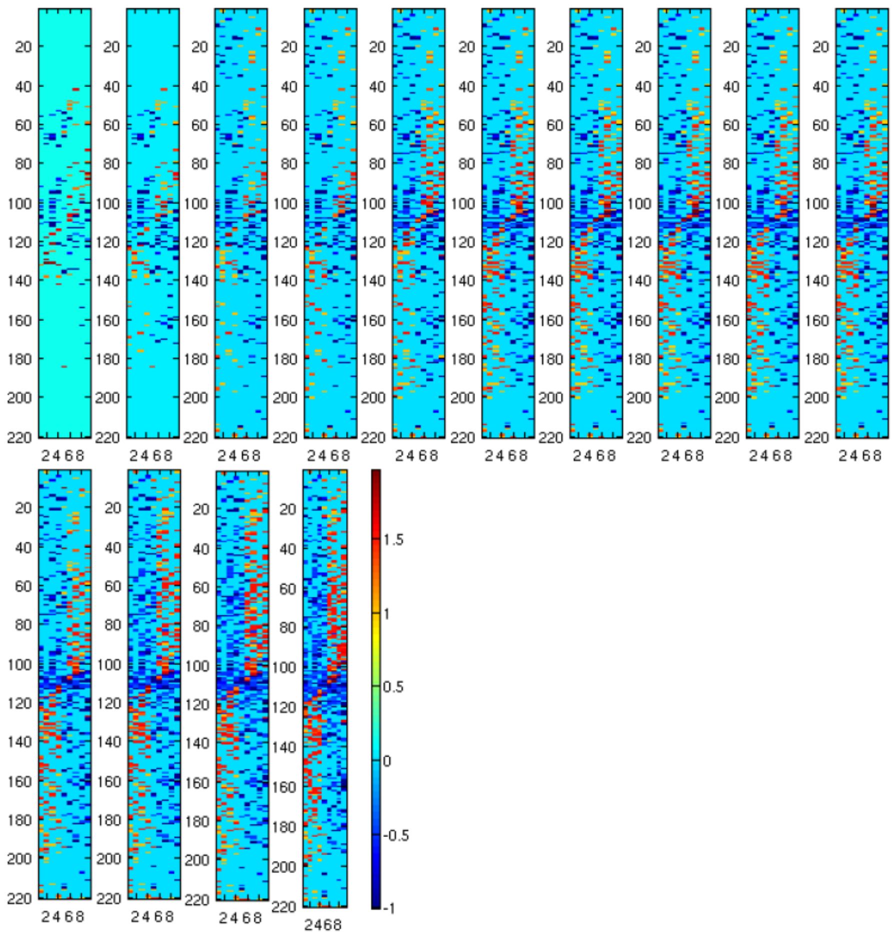


Figure 4.9: Resulting Q function after 14 iterations of the genetic learning algorithm. The state-action space is explored only sparsely and we need a larger number of iterations to get a result comparable to the TTD algorithm.

Also huge security concerns arise when using reinforcement learning, since much randomization is involved in the learning process and so it could not be guaranteed that a good (and collision free!) solution could be found. This is not the case in the next approach, where safety could be guaranteed.

5 The Dynamic Window Approach

5.1 Main concept

One of the most successful motion controlling strategies is the so called "Dynamic Window Approach" [5]. As already mentioned in the introduction we can count this approach as a sense-plan-act approach. It defines a window inside the action space (sometimes also called configuration space), which means rotational speed V_{rot} and translational speed V_{trans} for our robot. Such a configuration is shown in Fig. 5.1. The window is always centered around the current speeds of the robot. It defines a physical plausible region around the current speeds, which is constrained by the maximal available acceleration and maximal deceleration. The maximal speeds for the next time step Δt are defined as follows:

$$
\begin{aligned}
V_{trans}^{max}(t+\Delta t) &= V_{trans}^{curr}(t) + a_{max}^{trans} * \Delta t \\
V_{trans}^{min}(t+\Delta t) &= V_{trans}^{curr}(t) - a_{max}^{trans} * \Delta t \\
\\
V_{rot}^{max}(t+\Delta t) &= V_{t}^{curr}(t) + a_{max}^{rot} * \Delta t \\
V_{trans}^{min}(t+\Delta t) &= V_{trans}^{curr}(t) - a_{max}^{rot} * \Delta t
\end{aligned}
$$

Note that only two parameters (a_{max}^{trans} and a_{max}^{rot}) are needed to configure the dynamic window. The speed V_{trans} is in m/s while the rotational speed is in deg/s. Also the accelerations refer to meter and degree. With this simple approach, the search space of possible actions for the next time step Δt is spanned.

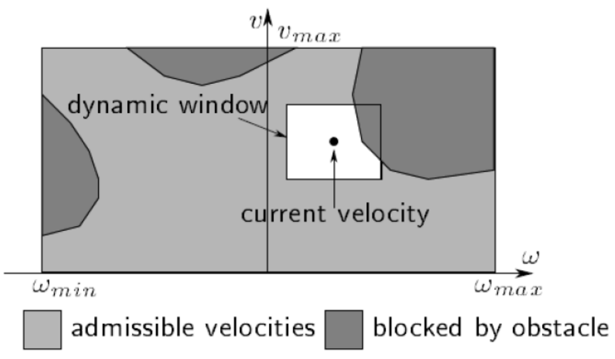


Figure 5.1: Basic idea of the dynamic window approach. In the space of translation and rotation velocities a small window around the current speed values is defined. Within this window the optimal next speed pair is searched to be the next actual speed.

From this window samples are drawn for the next possible actions to take. Normally these samples are aligned in a regular grid within the window. Now the dynamic window approach votes for each sample, whether this speed pair is a good one or not. This voting procedure is done by many so called "objectives". Each objective is asked for each speed pair for its vote. The voting result is at the one hand a single value and on the other hand a vote for the acceptability of this speed pair. If one of the objectives tells the dynamic window, that a certain action is not admissible, the robot is forbidden to execute this action.

This makes the whole concept of the dynamic window highly flexible and modular (see [4]). Evaluations are possible, which deals with questions like: does this action reduce the distance to the target? , do the robot head towards the target after finishing the action? , is the robots speed appropriate?, is the distance to the next obstacle to small?, could collisions occur?. For each of these tasks, a single objective could be formulated. We will discuss a basic set of objectives soon.

The decisions of each objective are expressed by a single scalar value and for each speed pair all values are summed up weighted ($G(V_{rot}, V_{trans}) = \alpha \cdot obj_1(V_{rot}, V_{trans}) + \beta \cdot obj_2(V_{rot}, V_{trans}) + \ldots$). Finally the best possible action within the selected window is chosen over all possible actions and set as the actual action within the defined time window Δt. After executing the action the window is centered towards this action and the process restarts. Key point on action selection is a search for a local optimal solution by construction an evaluation function.

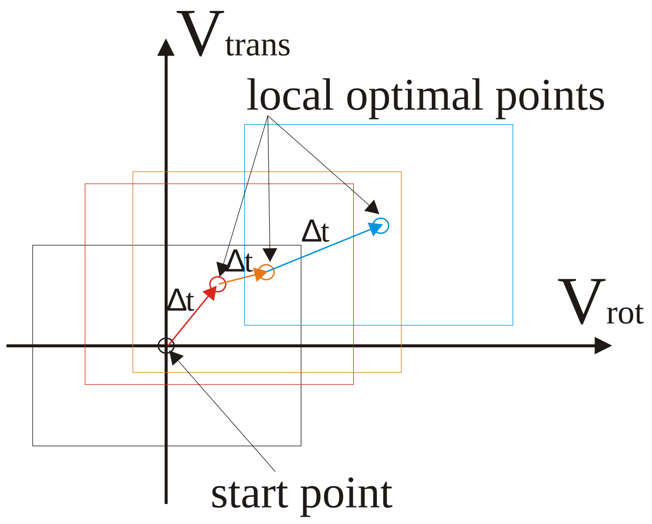


Figure 5.2: A sequence of three window update steps. Per step the optimal action is selected, which is afterwards the active action for the given time interval. Then the next best action is selected.

We already stated in the introduction, that our partner MetraLabs delivers a software framework that is a dynamic window navigator. At this point it should be mentioned, that it is possible in this framework, to include own objectives and specify an active set of objectives during runtime. In such a case the navigation task of the robot can be mapped to a set of objectives. So our goal is here, to specify and deliver sets of objectives for each given navigational task.

5.2 Software structure

The software structure of this quite simple. At startup there exists a set of available objectives, each with an unified interface. By default these objectives are not active. All objectives are managed by the dynamic window module. The dynamic window module knows the list of objectives, the current speeds of the robot, a robot model to predict the trajectories of certain speed configurations and the physical constraints of the robot. Upon a set tasks, the dynamic window will activate a subset of the available objectives and tries to fulfill the tasks with this subset. This knowledge is hard coded during design process and specific for each given task.

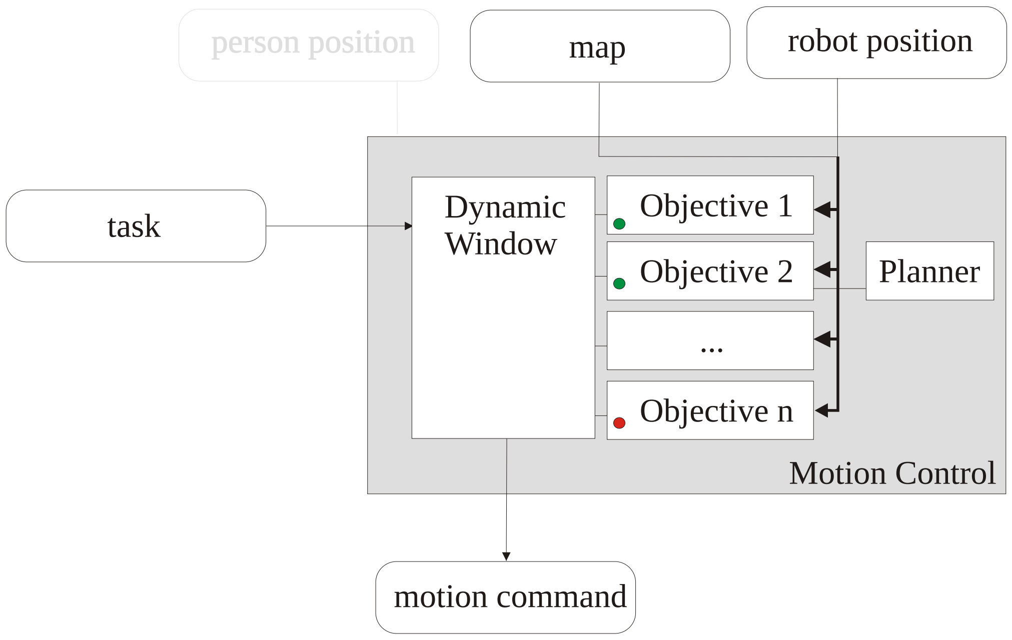


Figure 5.3: The software structure of the dynamic window. The main module receives the task and selects the set of objectives it has to use to fulfill the task.

The input channels the objectives receive are configurable. In a standard scenario, a map of the whole environment is needed for path planning and obstacle avoidance, and the global position of the robot as well. We do not consider person detection here, since this channel is needed by social acceptable navigation only. Note, that objectives could be complex entities. As shown in Fig. 5.3, an objective could even use a planner to create the correct functionality.

5.3 Objectives

Since the navigation framework of MetraLabs is closed source, we will describe here only the standard objectives of the classical dynamic window approach. Only the objective to follow a path is also described briefly. As a whole, the described set of objectives enable the robot to follow a given path.

5.3.1 Collision detection objective

This objective is the main security objective. Here, a trajectory of the given speed pair is projected over the time interval Δt, and if a possible collision inside a given local map is detected, the given speed is marked as "not admissible". A value is not given (or left constant). Note, that a given pair of V_{trans}, V_{rot} will either describe a circle element or a straight line. An example of one trajectory is given in Fig. 5.4, where a collision is detected.

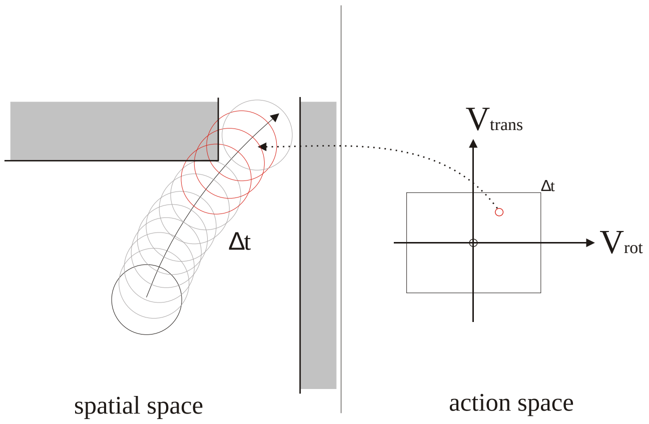


Figure 5.4: A sample trajectory from the current dynamic window configuration. Here, the trajectory is a circle element, defined by the time interval Δt and the collision detection objective detects a possible collision along that path. In this case the objective signals an "not admissible"

5.3.2 Distance objective

The distance objective prefers trajectories with the largest distance to drive until a collision occurs. As stated above, each point pair is either a circle or a straight line. The value returned by the distance objective is the distance along that line, until an obstacle is hit. If the trajectory never hits an obstacle, the value is set to a very huge positive number. An example for one speed pair is shown in Fig. 5.5.

Note, that this time the time interval Δt is not considered. Here, the trajectory is observed for an infinite time.

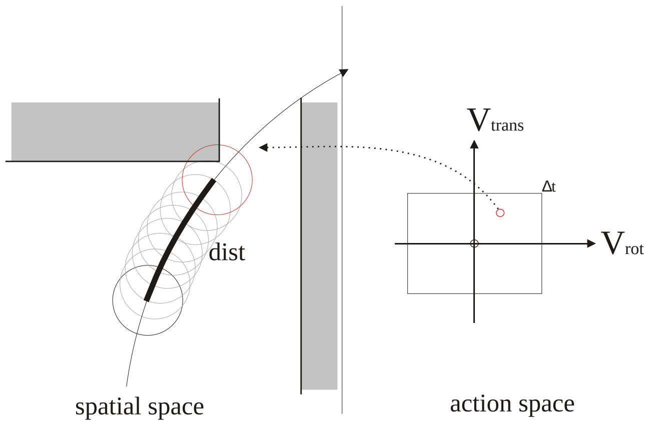


Figure 5.5: A sample trajectory from the current dynamic window configuration to visualize the distance objective. Here the distance is measured until the robot hits an obstacle.

5.3.3 Heading objective

When driving in a reactive manner towards a given goal, this objective forces the robot to head towards the goal. The simple idea is, do predict the trajectory for the given speed pair over the time interval Δt and take the difference angle Θ to the given goal. The returning value is simply $180° - \Theta$.
Again, we visualize the idea of the objective in Fig. 5.6.

5.3.4 Speed objective

The last of the classical objectives is the speed- or velocity objective. This is quite easy, since it only returns a direct linear mapping of the translation speed: $\alpha \cdot V_{trans}$.

5.3.5 Path planning objective

This objective is not a standard objective inside the Dynamic window approach. Instead, it replaces the heading objective and enables the robot to follow a planned path. During path planning, a cost function from the target to all known drivable position is constructed, which the robot should follow by using gradient descent on this cost function.
Here, it is obvious that we simply have to evaluate the predicted position of the robot after the time interval Δt and the given speed pair and return the cost function value at that position. If no planning value is available, since the reached position would be inside an obstacle, we return that action at "not admissible". We present also an example in Fig. 5.7.

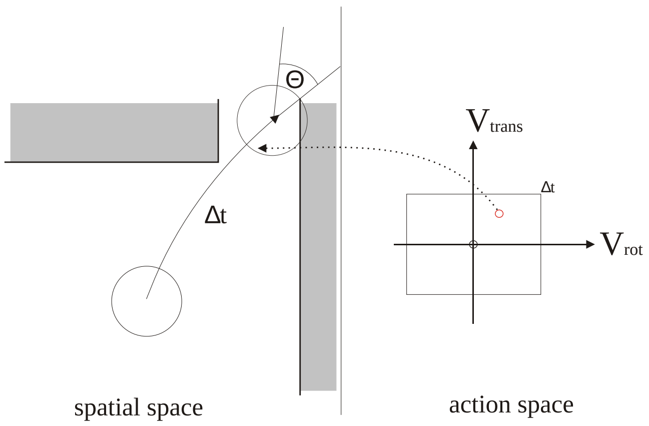


Figure 5.6: A sample trajectory from the current dynamic window configuration to visualize the heading objective. The relative angle of the robot after the given time interval is evaluated here.

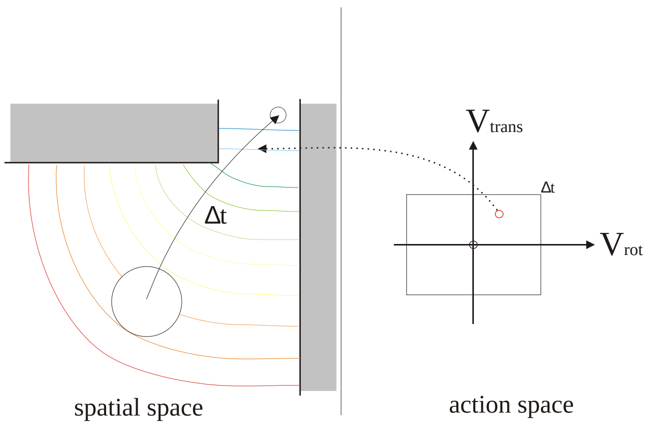


Figure 5.7: A sample trajectory from the current dynamic window configuration to visualize the path planning objective. Here, the value of the planning cost function is returned for each speed pair.

5.4 Experiments

It was already mentioned, that MetraLabs provides an implementation of the Dynamic Window Approach, which supports driving from position A to position B. This implementation was tested extensively by MetraLabs in several real world implementations and has today driven several thousand kilometers of distance. We think it is more than enough to prove the feasibility of this approach.

5.5 Conclusion

To sum up, the driving decision of the robot for an time interval Δt only depends on the evaluation function $G(V_{rot}, V_{trans}) = \alpha_1 \cdot obj_1(V_{rot}, V_{trans}) + \alpha_2 \cdot obj_2(V_{rot}, V_{trans}) + ... + \alpha_n \cdot obj_n(V_{rot}, V_{trans})$. What these objectives model, and how many are used, depends only on the given task. This means, that this approach is able to change the robots driving behavior just by activating different sets of objectives! This makes this approach highly modular.

In comparison to the reinforcement learning approach, all objectives are very distinctive and use almost no random elements. This makes this approach very reliable and predictable. By *always* using the collision detection objective, the system can easily be secured, no matter what other objectives decide for the best action. This cannot be guaranteed by any learning approach.

But there are of course also drawbacks. By simple adding the voting results of all objectives, there is the danger that to many influences create a very bumpy evaluation function. The designers have to be careful to use always a minimal set of criteria to solve a task. By including too many objectives, the system may also become unconfigurable and hard to handle. Also in the standard implementation no moving obstacles are considered during trajectory planning and humans are also not distinguished from normal obstacles. The task of social acceptable navigation will be, to add this extra functionality as a set of objectives to this approach.

6 The "Approach User" objective

In the last chapter the principal structure of the Dynamic Window approach was shown. It was also told, that the standard approach does not support social acceptable navigation and that this has to be extended within the dynamic window approach. This chapter will describe the extension for the first task, to approach a user, previously detected by the person detection module.

6.1 Software structure update

The structure of the dynamic window remains unchanged. Only we add an single objective, which needs additional input from the person detection and which uses a planning module and a model of the personal space of the person. The role of these sub-modules are described in the following parts of this chapter. Note that the following parts are extracted from our publication to appear in the proceedings of the ICIRA this year ([13]);

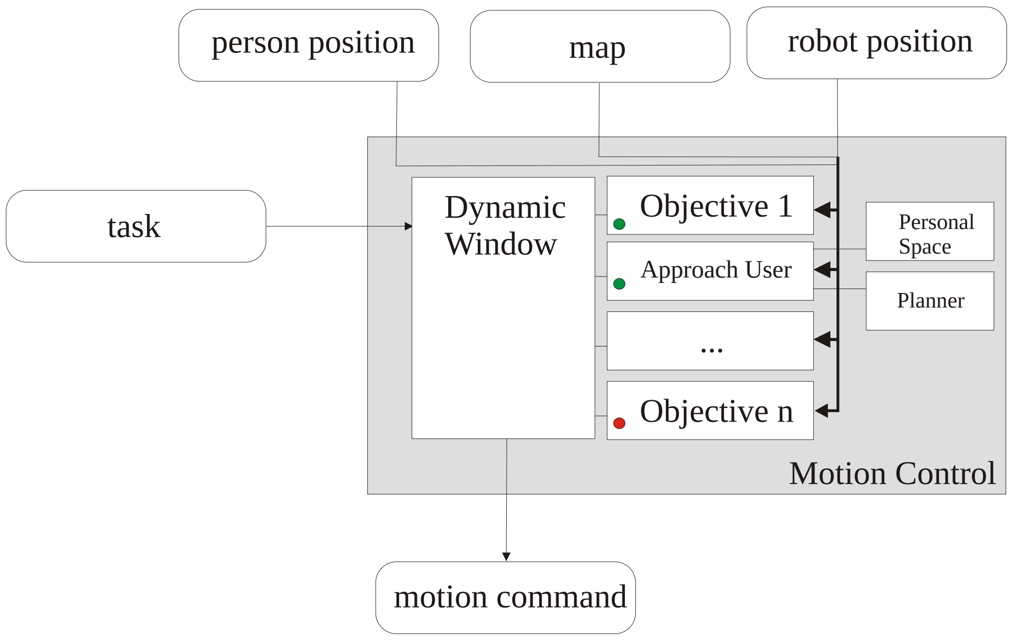


Figure 6.1: The software structure of the dynamic window, now extended with the ability to approach a person.

zone	interval	example situation
close intimate	0.0m - 0.15m	lover or close friend touching
intimate zone	0.15m - 0.45m	lover or close friend talking
personal zone	0.45m - 1.2m	conversion between friends
social zone	1.2m - 3.6m	conversion to non-friend
public zone	from 3.6m	no private interaction

Table 6.1: Psychological definition of the personal space. This space consists of 5 zones, each supporting different activities and different communication intentions.

6.2 The model of the personal space

Psychologists investigated the human-to-human interaction in public areas very carefully since the 70s of the last century. One of the foundations and most important publications is the work of Hall [7],[8], who first introduced the concept of different spaces around a human being to support different modes of interaction. There is a space for non-interaction, public interaction, interactions with friends and also an intimate space for interaction with very close relatives.

By formulating the theory that interaction is also coupled to spatial configurations between interaction partners, many investigations on this matter have taken place, and it could be shown that the configuration depends on many aspects like cultural background, age, sex, social status and person's character.

The model of the personal space is the key component to approach a person. Similar to the work of Dautenhahn [3], we also want the robot to approach a person from the front, but with a slight aberration from the direct front, since most user perceive such a behavior more comfortable. For this purpose, obviously we need the position and viewing direction of the person to calculate the configuration of the personal space model. The space configuration should enable the robot to drive around the person in a comfortable distance and turn towards the person when a "front position" is reached. Like in [19], we model the personal space with a sum of Gaussians. The space relative to the persons upper body direction is separated into two regions: a front-region, which is considered to be within $\pm 45^\circ$ around the persons upper direction, and a back-region, which is the rest (see Fig. 6.2).

In both areas we define a distance function to keep the robot out of the user's personal zone but within his/her social zone while approaching the person. The function is defined relative to the persons upper body direction.

$$a(x, y) = \frac{\alpha}{2\pi\sigma_1} \cdot e^{-\frac{x^2+y^2}{\sigma_1^2}} - \frac{\beta}{2\pi\sigma_2} \cdot e^{-\frac{x^2+y^2}{\sigma_2^2}} \tag{6.1}$$

The variables $\alpha, \beta, \sigma_1, \sigma_2$ describe a classical Difference of Gaussians function and are set in our case (see Fig. 6.2) to $\alpha = 0.6, \beta = 0.3, \sigma_1 = 2m, \sigma_2 = \sqrt{7}m$ to form a

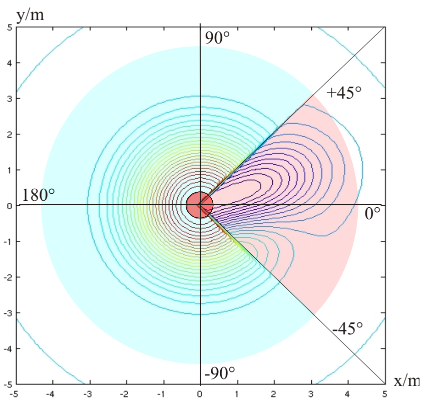


Figure 6.2: Two regions of our personal space model. The front region is within an $\pm 45°$ interval (in red). The back region is the rest (in blue). Note, that the regions are not limited in radial extension, like it is done in the illustration.

minimum cost region in a distance of 3.5 meters around the person. The front region is treated additionally with an "intrusion function" $i(x, y)$. This is also a Gaussian function and is simply added to $a(x, y)$.

$$\begin{aligned} i(x,y) &= \frac{\gamma}{2\pi\sqrt{|\Sigma|}} \cdot e^{-\vec{x}^T \Sigma^{-1} \vec{x}} \\ \Sigma &= \begin{bmatrix} \sigma_x^2 & 0.0 \\ 0.0 & \sigma_y^2 \end{bmatrix} \cdot \begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix} \end{aligned} \tag{6.2}$$

Here the variables σ_x and σ_y define an elliptical region, that is rotated towards the needed approaching direction ϕ, as seen from the persons perspective. The vector $\vec{x}$ is simply a column vector $(x, y)^T$. The variables are set to $\gamma = -0.5$, $\sigma_x^2 = 2.9$ and $\sigma_y^2 = 1.1$. Only ϕ and σ_x need to be set at runtime to regulate the approaching distance and direction. All other parameters are constant and are chosen to reflect the properties of the personal space definition in [7]. So, the final definition of the personal space $p(x, y)$ relatively to the person coordinates $x = 0, y = 0$ and upper body pose towards the x-axis is defined as follows:

$$p(x,y) = \begin{cases} a(x,y) \text{ , if } \langle x,y \rangle \text{ in back-region} \\ a(x,y) + i(x,y) \text{ , if } \langle x,y \rangle \text{ in front-region} \end{cases} \tag{6.3}$$

To compute the personal space in the real world application each point $(\acute{x}, \acute{y})^T$ has to be transformed to the person-centered coordinate system $(x, y)^T$ presented here.

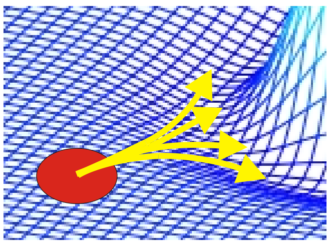


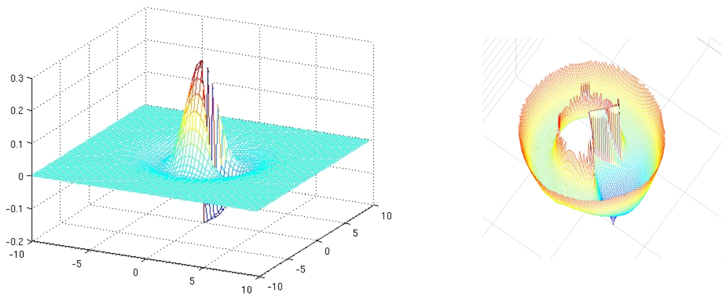
Figure 6.3: The problem of using the personal space directly as an objective function: no distinct speed decision is possible, when the personal space model is used. Here, several actions can lead toward the same minimal value. Also, the robot only seeks local minima within the personal space function.

6.3 Planning with Fast Marching and the Dynamic Window Approach

Up to that point, we have shown how the personal space can be computed, if the upper body pose of a person is known. We also stated, that this space is used within the Dynamic Window Approach (DWA). The basic idea of the DWA is to decide in a local situation, which next action is optimal. The local driving command is only valid for a certain Δt, than the next window configuration is evaluated. If the Dynamic Window uses the personal space directly, it is possible to predict for every speed pair V_{rot}, V_{trans} the trajectory within the interval Δt and simply evaluate the value of the personal space at this point, the robot has reached at that time. This is shown in Fig. 6.3. The minimal value leads to the most supported driving decision. By using the personal space directly, multiple driving decision lead to the same minimal value and a single local optimum can not be guaranteed.

6.3.1 Fast Marching and the cost function

To avoid situations, where no distinct decision is possible, path planning methods are used to create continuous decreasing functions to get to the optimum by gradient descents. An excellent planning technique is the Fast Marching method [17], which origins from the level set methods of single wave fronts and is applied to path planning. The core idea is to code space as a physical medium, where waves can travel with different speeds. For example in obstacles the speed is nearly zero, while in free space the speed can be any feasible speed. By propagating a wave front from the target to the robot, a function of the traveling time of the wave for every point in space is constructed. The benefit is, that also fuzzy values, that are not obstacles or free space, can be considered in this simulation and deform the initial circular waveform. So all we have to do, is to transform the personal space into a physical "speed-space". We know the minimum of $p(x, y)$ and use p_{min} to create a function that is non-negative. High values of the personal space symbolize bad places to drive to, while low values should be preferred. So we define the speed function



a) b)

Figure 6.4: From personal space to the planning function. The personal space function in a) is transformed to create the continuously decreasing planning function b).

$v(x, y)$ as follows:

$$v(x, y) = 1/\left(p(x, y) + p_{min} + \epsilon\right)\right) \tag{6.4}$$

The variable ϵ is used to prevent an infinite speed at the minimum point.

6.3.2 Extracting the target region

To navigate with the Dynamic Window, we use local occupancy maps to represent the surrounding obstacle situation around the robot. In this grid representation, we also have to rasterize the personal space values $p(\acute{x}, \acute{y})$ to merge the costs of the personal space with the costs of obstacles to create an optimal path. Each planning algorithm has to know the target, to which state the system has to drive to. Since we have a rasterized personal space, we are able to easily extract the minimum value $p_{min}(\acute{x}, \acute{y})$. The planning algorithm has to know the target, to which state the robot has to drive to. This target is the origin of the wave and each point $(\acute{x}, \acute{y})$ with $p(\acute{x}, \acute{y}) < p_{min} + \epsilon$ belongs to the target region. Planning is complete when the traveling wave front hits the cell of the current robot position, and now the values of the traveling function can be used directly by the dynamic window to apply a gradient descent. When the robot reaches a small region around the target region the approaching task is done.

6.4 Experiments

A problem on approaching a person is the estimation of the person's position and the associated measurement noise. To test the stability and robustness of the approach, we investigated three scenarios, two in narrow spaces and one in a large room of our lab. At this stage, we use a simulator to avoid the problems of person detection, but with real

Person position		Robot final position
Scenario	σ_{pers} in meter/deg	σ_{rob}
1(I)	$(0.4, 0.1)$	$(0.4, 0.1)$
1(II)	$(0.5, 0.1)$	$(0.4, 0.1)$
2(I)	$(0.2, 0.1)$	$(0.2, 0.2)$
2(II)	$(0.2, 0.2)$	$(0.3, 0.2)$
3(I)	$(0.1, 0.1)$	$(0.1, 0.1)$
3(II)	$(0.1, 0.2)$	$(0.1, 0.1)$

Table 6.2: Variance of the robot's final pose and variance of the wait position of the person

maps to drive within. To investigate the stability of the approaching behavior on a wider range of positions or sensor noise, the position of the person and the robot was chosen randomly to approach in a circle around a marked position. The robot and the person should face towards a given direction each. For each of the three locations, we define two person positions with different viewing angles and performed ten runs for each position. So we have a set of six trials with a sum of 60 single runs. The variance of the final robot position and the person's position are shown in table 6.2.

From the experimental setup we get uncertainties of 0.1 to 0.5 meters in the person's resting position. The question to be answered in our experiments is, how the variance of the robot's target position will increase when approaching a person, by knowing the initial variance of the person's upper body pose. We also want to know, how the trajectories variate on the person's position noise. To do so, we record the trajectory of the robot and calculate the mean and standard deviation of the final robot position. The results are shown in table 6.2 and figure 6.5. The average distance from the person is 0.7 meters, the variance is within the same magnitude as the variance of the person's pose. So measurement noise is not amplified by this method. Figure 6.5 shows the path and the mean person position with variance of all six test cases. Scenario 2 shows, how the upper body pose heavily influences the trajectory of the robot. Scenarios 1 and 3 show, that in narrow spaces the trajectory has to follow the physical restrictions. The personal space has to be intruded, if there is no other chance.

6.5 Conclusion

In this deliverable we presented a method, working within the Dynamic Window Approach, to approach a person by considering his/her personal space. We could demonstrate, by using a planning strategy, that a stable and reliable solution could be achieved. Nevertheless the method of extracting the target region could be improved in future work. We also want to include obstacles into the personal space model, to improve planning quality and focus on the task of real time replanning, when the person changes his/her pose while the robot approaches.

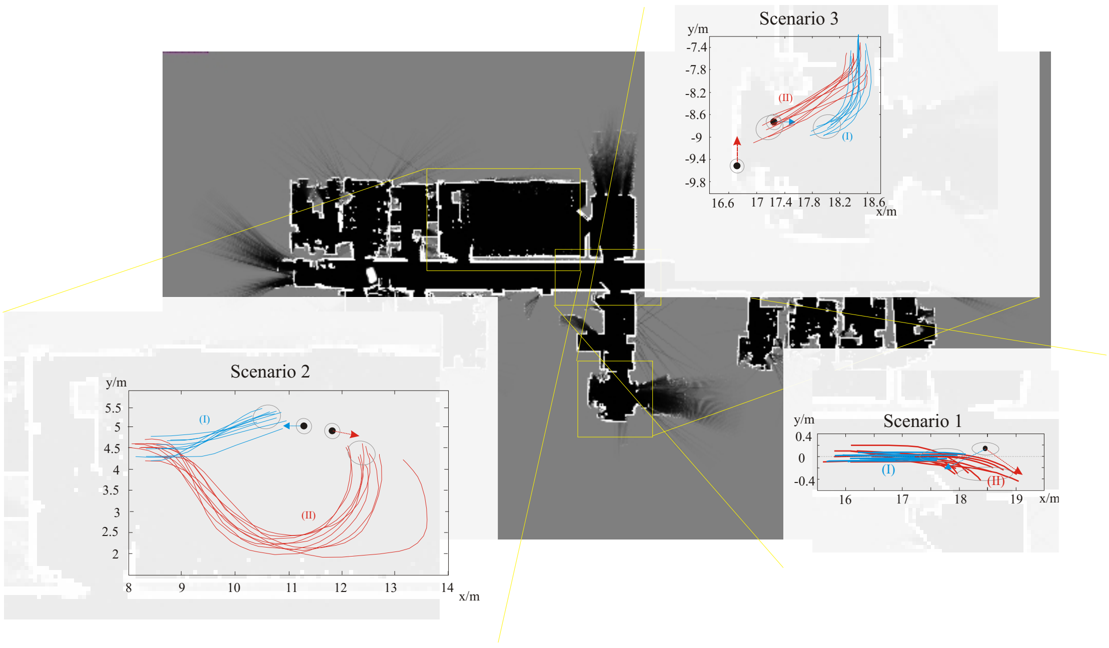


Figure 6.5: Resulting trajectories of the three tested scenarios. Per scenario two different poses are evaluated by the user (I and II). The mean positions of the user are shown as black dots, the mean upper body poses as arrows. In each scenario the blue lines denote the robot's trajectories corresponding to the first person setup, while the red lines show trajectories of the second setup. All scenarios show, how the upper body pose influences the approaching trajectory. Scenario 2 also shows, that the social zone is respected if there is room to navigate.

7 Conclusion and outlook

Within this deliverable we have shown the first version of our navigation software module. We presented the first results of the basic prerequisite, namely the person detection and tracking. Here, a major change occurred by changing the fish-eye camera driven input channel toward the Microsoft Kinect device. We could show first detection results of this channel and the laser-scanner based leg-detector. Future work will focus on efficient upper body pose detection of the Kinect channel and the fusion of all available detection channels (face-detection, point cloud detection, range scan detection).

Second, we present a learning candidate to let the robot learn its driving behavior. This shows up to be a solution, which is very hard to handle (in fact only experts could have done it), and which leads to a specialized solution for all given navigation tasks. A secure and predictable behavior could not be guaranteed, although this approach is the intellectual more interesting one. Within the ALIAS project we will not further investigate this approach.

Finally, we showed the framework of the dynamic window, which is already supported by our project partner MetraLabs. The Dynamic Window Approach could be handled very modular in software structure and the pure reconfiguration of used objectives is sufficient to change the robots driving behavior. The selection of this approach is the more natural choice to use within the ALIAS project.

After this decision was done, we started to develop an objective for approaching a user. First results where shown here and we could successfully place a publication on this issue [13]. We will do more experiments on this issue this year.

Up to now, we have focused on the interaction part of social acceptable navigation. In the next year of the project, we will focus more on the aspects of social acceptable navigation when no interaction is done. This means we will also implement solutions for observing a resting person, to give the robot the ability to recognize speech commands or gestures, while not disturbing the resting person to much. We will also give the robot the ability to react on a moving person, while not interacting with it, and create a polite behavior by making room for a person when both parters move, but do not interact with each other.

Bibliography

[1] K. Arras, O. Mozos, and W. Burgard. Using boosted features for the detection of people in 2d range data. In *Proc. of International Conference on Robotics and Automation (ICRA)*, pages 3402–3407, 2007.

[2] K. Arras, O. Mozos, and W. Burgard. Using boosted features for the detection of people in 2d range data. In *Int. IEEE Conference on Intelligent Robots and Systems (IROS)*, pages 3402–3407, 2007.

[3] K. e. a. Dautenhahn. How may i serve you? a robot companion approaching a seated person in a helping context. In *Proc. HRI 2006*, pages 172–179, 2006.

[4] E. Einhorn and T. Langner. Pilot - modular robot navigation for real-world applications. In *Proc. of 55.th International Scientific Colloquium*, pages 382–387, 2010.

[5] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. 4(1):23–33, 1997.

[6] G. Grisetti, C. Stachniss, and W. Burgard. Improved techniques for grid mapping with rao-blackwellized particle filters. In *IEEE Transactions on Robotics*, pages 34–46, 2006.

[7] E. Hall. *The hidden dimension*. Doubleday, NY, 1966.

[8] E. Hall. Proxemics. 9(2):83+, 1968.

[9] R. Iglesias, M. Rodriguez, and M. Sanchez. Improving reinforcement learnig through a better exploration strategy and an adjustable representation of the environment. In *Proc. of European Conference on mobile Robots (ECMR)*, pages 13–18, 2006.

[10] C. Isik and M. Ciliz. A two-level neural network system for learning control of robot motion. In *IEEE International Symposium on Intelligent Control*, pages 519–522, 1988.

[11] J. Kessler. D6.1 report on different navigation strategies to approach elderly people in a polite manner. *EU Deliverable*, 2010.

[12] J. Kessler. D6.2 collection of several approaching strategies for a robot platform in an ambient assisted living environment. *EU Deliverable*, 2011.

[13] J. Kessler, C. Schroeter, and H.-M. Gross. Approaching a person in a socially acceptable manner using a fast marching planner. In *Proc. of International Conference on Intelligent Robotics and Applications (ICIRA)*, page to appear, 2011.

[14] C. Kwok, D. Fox, and M. Meila. Adaptive real-time particle filters for robot localization. In *Proc. of the IEEE International Conference on Robotics and Automation*, pages 2836–2841, 2003.

[15] T. Martinetz, S. Berkovich, and K. Schulten. ̎Neural-gas ̎ Network for Vector Quantization and its Application to Time-Series Prediction. *IEEE-Transactions on Neural Networks*, 4(4):558–569, 1993.

[16] C. Schroeter, H.-J. Boehme, and H.-M. Gross. Memory-efficient gridmaps in rao-blackwellized particle filters for slam using sonar range sensors. In *Proc. 3rd European Conference on Mobile Robots (ECMR) 2007*, pages 138–143, 2007.

[17] J. Sethian. A fast marching level set method for monotonically advancing fronts. 93(4):1591–1595, 1996.

[18] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[19] M. e. a. Svenstrup. Pose estimation and adaptive robot behaviour for human-robot interaction. In *Proc. ICRA 2009*, page 3571 Ű 3576, 2009.

[20] S. Thrun, D. Fox, and W. Burgard. *Probabilistic robotics*. The MIT Press, 2005.